State-Slice: New Paradigm of Multi-query Optimization of Window-based Stream Queries[*]

Song Wang, Elke Rundensteiner
Worcester Polytechnic Institute
Worcester, MA, USA.
{songwang|rundenst}@cs.wpi.edu

Samrat Ganguly, Sudeept Bhatnagar
NEC Laboratories America Inc.
Princeton, NJ, USA.
{samrat|sudeept}@nec-labs.com

ABSTRACT

Modern stream applications such as sensor monitoring systems and publish/subscription services necessitate the handling of large numbers of continuous queries specified over high volume data streams. Efficient sharing of computations among multiple continuous queries, especially for the memory- and CPU-intensive window-based operations, is critical. A novel challenge in this scenario is to allow resource sharing among similar queries, even if they employ windows of different lengths. This paper first reviews the existing sharing methods in the literature, and then illustrates the significant performance shortcomings of these methods.

This paper then presents a novel paradigm for the sharing of window join queries. Namely we slice window states of a join operator into fine-grained window slices and form a chain of sliced window joins. By using an elaborate pipelining methodology, the number of joins after state slicing is reduced from quadratic to linear. This novel sharing paradigm enables us to push selections down into the chain and flexibly select subsequences of such sliced window joins for computation sharing among queries with different window sizes. Based on the state-slice sharing paradigm, two algorithms are proposed for the chain buildup. One minimizes the memory consumption while the other minimizes the CPU usage. The algorithms are proven to find the optimal chain with respect to memory or CPU usage for a given query workload. We have implemented the slice-share paradigm within the data stream management system *CAPE*. The experimental results show that our strategy provides the best performance over a diverse range of workload settings among all alternate solutions in the literature.

1. INTRODUCTION

Recent years have witnessed a rapid increase of attention in data stream management systems (DSMS). Continuous query based applications involving a large number of concurrent queries over high volume data streams are emerging in a large variety of scientific and engineering domains. Examples of such applications include environmental monitoring systems [2] that allow multiple continuous queries over sensor data streams, with each query issued for independent monitoring purposes. Another example is the publish-subscribe services [7, 20] that host a large number of subscriptions monitoring published information from data sources. Such systems often process a variety of continuous queries that are similar in flavor on the same input streams.

Processing each such compute-intensive query separately is inefficient and certainly not scalable to the huge number of queries encountered in these applications. One promising approach in the database literature to support large numbers of queries is *computation sharing*. Many papers [8, 18, 10, 13] have highlighted the importance of computation sharing in continuous queries. The previous work, e.g. [8], has focused primarily on sharing of filters with overlapping predicates, which are stateless and have simple semantics. However in practice, stateful operators such as joins and aggregations tend to dominate the usage of critical resources such as memory and CPU in a DSMS. These stateful operators tend to be bounded using window constraints on the otherwise infinite input streams. Efficient sharing of these stateful operators with possibly different window constraints thus becomes paramount, offering the promise of major reductions in resource consumption.

Compared to traditional multi-query optimization, one new challenge in the sharing of stateful operators comes from the preference of in-memory processing of stream queries. Frequent access to hard disk will be too slow when arrival rates are high. Any sharing blind to the window constraints might keep tuples unnecessarily long in the system. A carefully designed sharing paradigm beyond traditional sharing of common sub-expressions is thus needed.

In this paper, we focus on the problem of sharing of window join operators across multiple continuous queries. The window constraints may vary according to the semantics of each query. The sharing solutions employed in existing streaming systems, such as NiagaraCQ [10], CACQ [18] and PSoup [9], focus on exploiting common sub-expressions in queries, that is, they closely follow the traditional multi-query optimization strategies from relational technology [23, 21]. Their shared processing of joins ignores window constraints, even though windows clearly are critical for query semantics.

The intuitive sharing method for joins [13] with different

[*]This work is funded in part by the NSF Computing Research Infrastructure grant CNS 05-51584 and NEC Laboratories America Inc.

window sizes employs the join having the largest window among all given joins, and a routing operator which dispatches the joined result to each output. Such method suffers from significant shortcomings as shown using the motivation example below. The reason is two folds, (1) the per-tuple cost of routing results among multiple queries can be significant; and (2) the selection pull-up (see [10] for detailed discussions of selection pull-up and push-down) for matching query plans may waste large amounts of memory and CPU resources.

Motivation Example: Consider the following two continuous queries in a sensor network expressed using an SQL-like language with window extension [2].

```
Q1: SELECT A.* FROM Temperature A, Humidity B
    WHERE A.LocationId=B.LocationId
    WINDOW 1 min
Q2: SELECT A.* FROM Temperature A, Humidity B
    WHERE A.LocationId=B.LocationId AND
          A.Value>Threshold
    WINDOW 60 min
```

Q_1 and Q_2 join the data streams coming from temperature and humidity sensors by their respective locations. The WINDOW clause indicates the size of the sliding windows of each query. The join operators in Q_1 and Q_2 are identical except for the filter condition and window constraints. The naive shared query plan will join the two streams first with the larger window constraint (60 min). The routing operator then splits the joined results and dispatches them to Q_1 and Q_2 respectively according to the tuples' timestamps and the filter. The routing step of the joined tuples may take a significant chunk of CPU time if the fanout of the routing operator is much greater than one. If the join selectivity is high, the situation may further escalate since such cost is a per-tuple cost on every joined result tuple. Further, the state of the shared join operator requires a huge amount of memory to hold the tuples in the larger window without any early filtering of the input tuples. Suppose the selectivity of the filter in Q_2 is 1%, a simple calculation reveals that the naive shared plan requires a state size that is 60 times larger than the state used by Q_1, or 100 times larger than the state used by Q_2 each by themselves. In the case of high volume data stream inputs, such wasteful memory consumption is unaffordable and renders inefficient computation sharing.

Our Approach: In order to efficiently share computations of window-based join operators, we propose a new paradigm for sharing join queries with different window constraints and filters. The two key ideas of our approach are: *state-slicing* and *pipelining*.

We slice the window states of the shared join operator into fine-grained pieces based on the window constraints of individual queries. Multiple sliced window join operators, with each joining a distinct pair of sliced window states, can be formed. Selections now can be pushed down below any of the sliced window joins to avoid unnecessary computation and memory usage shown above.

However, N^2 joins appear to be needed to provide a complete answer if each of the window states were to be sliced into N pieces. The number of distinct join operators needed would then be too large for a DSMS to hold for a large N. We overcome this hurdle by elegantly pipelining the slices. This enables us to build a chain of only N sliced window joins to compute the complete join result. This also enables us to selectively share a subsequence of such a chain of sliced window join operators among queries with different window constraints.

Based on the state-slice sharing paradigm, two algorithms are proposed for the chain buildup, one that minimizes the memory consumption and the other that minimizes the CPU usage. The algorithms are guaranteed to always find the optimal chain with respect to either memory or CPU cost, for a given query workload. The experimental results show that our strategy provides the best performance over a diverse range of workload settings among alternate solutions in the literature.

Our Contributions:

- We review the existing sharing strategies in the literature, highlighting their memory and CPU consumptions.
- We introduce the concept of a chain of pipelining sliced window join operators, and prove its equivalence to the regular window-based join.
- The memory and CPU costs of the chain of sliced window join operators are evaluated and analytically compared with the existing solutions.
- Based on the insights gained from this analysis, we propose two algorithms to build the chain that minimizes the CPU or the memory cost of the shared query plan, respectively. We prove the optimality of both algorithms.
- The proposed techniques are implemented in an actual DSMS (*CAPE*). Results of performance comparison of our proposed techniques with state-of-the-art sharing strategies are reported.

Organization of Paper: The rest of the paper is organized as follows. Section 2 presents the preliminaries used in this paper. Section 3 shows the motivation example with detailed analytical performance comparisons of alternative sharing strategies of window-based joins. Section 4 describes the proposed chain of sliced window join operators. Sections 5 and 6 present the algorithms to build the chain. Section 7 presents the experimental results. Section 8 contains related work while Section 9 concludes the paper.

2. PRELIMINARIES

A *shared query plan* capturing multi-queries is composed of operators in a directed acyclic graph (DAG). The input streams are unbounded sequences of tuples. Each tuple has an associated timestamp identifying its arrival time at the system. Similar to [6], we assume that the timestamps of the tuples have a global ordering based on the system's clock.

Sliding windows [5] are commonly used constraints to define the stateful operators. See [12] for a survey on window-based join operations in the literature. The size of a window constraint is specified using either a time interval (time-based) or a count on the number of tuples (count-based). In this paper, we present our sharing paradigm using time-based windows. However, our proposed techniques can be applied to count-based window constraints in the same way. We also simplify the discussion of join conditions by using equijoin in this paper, while the proposed solution is applicable to any type of join condition.

The sliding window equijoin between streams A and B, with window sizes W_1 and W_2 respectively over the common attribute C_i can be denoted as $A[W_1] \bowtie_{C_i} B[W_2]$. The semantics [24] for such sliding window joins are that the output of the join consists of all pairs of tuples $a \in A$, $b \in B$, such that $a.C_i = b.C_i$ (we omit C_i in the future and instead concentrate on the sliding window only) and at certain time t, both $a \in A[W_1]$ and $b \in B[W_2]$. That is, either $T_b - T_a < W_1$ or $T_a - T_b < W_2$. T_a and T_b denote the timestamps of tuple a and b respectively in this paper. The timestamp assigned to the joined tuple is $max(T_a, T_b)$. The execution steps for a newly arriving tuple of A are shown in Fig. 1 [1]. Symmetric steps are followed for a B tuple.

```
1.Cross-Purge: Discard expired tuples in window B[W2]
2.Probe: Emit a ⋈ B[W2]
3.Insert: Add a to window A[W1]
```

Figure 1: Execution of Sliding-window join.

For each join operator, the input stream tuples are processed in the order of their timestamps. Main memory is used for the states of the join operators (*state memory*) and queues between operators (*queue memory*).

3. REVIEW OF STRATEGIES FOR SHARING CONTINUOUS QUERIES

Using the example queries Q_1 and Q_2 from Section 1 with generalized window constraints, we review the existing strategies in the literature for sharing continuous queries. Figure 2 shows the query plans for Q_1 and Q_2 without computation sharing. The states in each join operator hold the tuples in the window. We use σ_A to represent the selection operator on stream A.

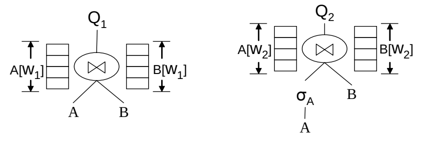


Figure 2: Query Plans for Q_1 and Q_2.

For the following cost analysis, we use the notations of the system settings in Table 1. We define the selectivity of σ_A as: $\frac{number\_of\_outputs}{number\_of\_inputs}$. We define the join selectivity $S_{\bowtie}$ as: $\frac{number\_of\_outputs}{number\_of\_outputs\_from\_Cartesian\_Product}$. We focus on state memory when calculating the memory usage. To estimate the CPU cost, we consider the cost for value comparison of two tuples and the timestamp comparison. We assume that comparisons are equally expensive and dominate the CPU cost. We thus use the count of comparisons per time unit as the metric for estimated CPU costs. In this paper, we calculate the CPU cost using the nested-loop join algorithm. Calculation using the hash-based join algorithm can be done similarly using an adjusted cost model [14].

Symbol	Explanation
λ_A	Arrival Rate of Stream A (Tuples/Sec.)
λ_B	Arrival Rate of Stream B (Tuples/Sec.)
W_1	Window Size for Q_1 (Sec.)
W_2	Window Size for Q_2 (Sec.)
M_t	Tuple Size (KB)
S_σ	Selectivity of σ_A
$S_{\bowtie}$	Join Selectivity

Table 1: System Settings Used in Section 3.

Without loss of generality, we let $0 < W_1 < W_2$. For simplicity, in the following computation, we set $\lambda_A = \lambda_B$, denoted as λ. The analysis can be extended similarly for unbalanced input stream rates.

[1]In this paper we only consider cross-purge, while self-purge is also applicable.

3.1 Naive Sharing with Selection Pull-up

The PullUp or Filtered PullUp approaches proposed in [10] for sharing continuous query plans containing joins and selections can be applied to the sharing of joins with different window sizes. That is, we need to introduce a router operator to dispatch the joined results to the respective query outputs. The intuition behind such sharing lies in that the answer of the join for Q_1 (with the smaller window) is contained in the join for Q_2 (with the larger window). The shared query plan for Q_1 and Q_2 is shown in Fig. 3.

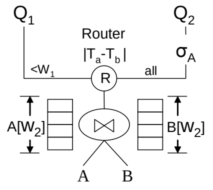


Figure 3: Selection Pull-up.

By performing the sliding window join first with the larger window size among the queries Q_1 and Q_2, computation sharing is achieved. The router then checks the timestamps of each joined tuple with the window constraints of registered CQs and dispatches them correspondingly. The compare operation happens in the probing step of the join operator, the checking step of the router and the filtering step of the selection. We can calculate the state memory consumption C_m (m stands for memory) and the CPU cost C_p (p stands for processor) as:

$$\begin{cases} C_m = & 2\lambda W_2 M_t \\ C_p = & 2\lambda^2 W_2 + 2\lambda + 2\lambda^2 W_2 S_{\bowtie} + 2\lambda^2 W_2 S_{\bowtie} \end{cases} \quad (1)$$

The first item of C_p denotes the join probing costs; the second the cross-purge cost; the third the routing cost; and the fourth the selection cost. The routing cost is the same as the selection cost since each of them perform one comparison per result tuple.

As pointed out in [18], the selection pull-up approach suffers from unnecessary join probing costs. With strong differences of the windows the situation deteriorates, especially when the selection is used in continuous queries with large windows. In such cases, the states may hold tuples unnecessarily long and thus waste huge amounts of memory.

Another shortcoming for the selection pull-up sharing strategy is the routing cost of each joined result. The routing cost is proportional to the join selectivity $S_{\bowtie}$. This cost is also related to the fanout of the router operator, which corresponds to the number of queries the router serves. Similar to [10], a router having a large fanout could be implemented as a range join between the joined tuple stream and a static profile table, with each entry holding a window size. Then the routing cost is proportional to the fanout of the router, which may be much larger than one.

3.2 Stream Partition with Selection Push-down

To avoid unnecessary join computations in the shared query plan using selection pull-up, we employ the selection push-down approach proposed in [10]. Selection push-down can be achieved using multiple join operators, each processing part of the input data streams. We then need a split operator to partition the input stream A by the condition in the σ_A operator. Thus the stream A into different join operators are disjoint. We also need an order-preserving (on tuple timestamps) union operator [1] to merge the joined results coming from the multiple joins. Such sharing paradigm applied to Q_1 and Q_2 will result in the shared query plan as shown in Figure 4.

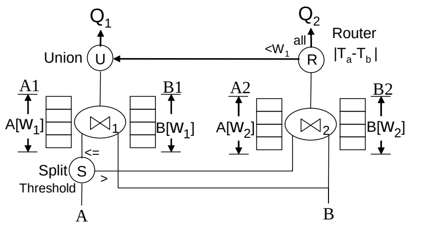


Figure 4: Selection Push-down.

The compare operation happens during the splitting of the streams, the merging of the tuples in the union operator, the routing step of the router and the probing of the joins. We can calculate the state memory consumption C_m and the CPU cost C_p for the selection push-down paradigm as:

$$\left\{ \begin{array}{rl} C_m = & (2 - S_\sigma)\lambda W_1 M_t + (1 + S_\sigma)\lambda W_2 M_t \\ C_p = & \lambda + 2(1 - S_\sigma)\lambda^2 W_1 + 2S_\sigma \lambda^2 W_2 + \\ & 3\lambda + 2S_\sigma \lambda^2 W_2 S_{\bowtie} + 2\lambda^2 W_1 S_{\bowtie} \end{array} \right. \qquad (2)$$

The first item of C_m refers to the state memory in operator $\bowtie_1$; the second the state memory in operator $\bowtie_2$. The first item of C_p corresponds to the splitting cost; the second to the join probing cost of $\bowtie_1$; the third to the join probing cost of $\bowtie_2$; the fourth to the cross-purge cost; the fifth to the routing cost; the sixth to the union cost. Since the outputs of $\bowtie_1$ and $\bowtie_2$ are sorted, the union cost corresponds to a one-time merge sort on timestamps.

Different from the sharing of identical file scans for multiple join operators in [10], the state memory B_1 cannot be saved since B_2 may not contain B_1 at all times. The reason is that the sliding windows of B_1 and B_2 may not move forward simultaneously, unless the DSMS employs a synchronized operator scheduling strategy.

Stream sharing with selection push-down tends to require much more joins (mn, m and n are the number of partitions of stream A and B respectively) than the naive sharing. With the asynchronous nature of these joins as discussed above, extra memory is consumed for the state memory. Such memory waste might be significant.

Obviously, the CPU cost C_p of a shared query plan generated by the selection push-down sharing is much smaller than the CPU cost of using the naive sharing with selection pull-up. However this sharing strategy still suffers from similar routing costs as the selection pull-up approach. Such cost can be significant, as already discussed for the selection pull-up case.

4. STATE-SLICE SHARING PARADIGM

As discussed in Section 3, existing sharing paradigms suffer from one or more of the following cost factors: (1) expensive routing step; (2) state memory waste among asynchronous parallel joins; and (3) unnecessary join probings without selection push-down. Our proposed state-slice sharing successfully avoids all three types of costs.

4.1 State-Sliced One-Way Window Join

A one-way sliding window join [14] of streams A and B is denoted as $A[W] \ltimes B$ (or $B \rtimes A[W]$), where stream A has a sliding window of size W. The output of the join consists of all pairs of tuples $a \in A$, $b \in B$, such that $T_b - T_a < W$, and tuple pair (a, b) satisfies the join condition.

Definition 1. *A sliced one-way window join on streams A and B is denoted as $A[W^{start}, W^{end}] \overset{s}{\ltimes} B$ (or $B \overset{s}{\rtimes} A[W^{start}, W^{end}]$), where stream A has a sliding window of range: $W^{end} - W^{start}$. The start and end window are W^{start} and W^{end} respectively. The output of the join consists of all pairs of tuples $a \in A$, $b \in B$, such that $W^{start} \leq T_b - T_a < W^{end}$, and (a, b) satisfies the join condition.*

We can consider the sliced one-way sliding window join as a generalized form of the regular one-way window join. That is $A[W] \ltimes B = A[0, W] \overset{s}{\ltimes} B$. Figure 5 shows an example of a sliced one-way window join. This join has one output queue for the joined results, two output queues (optional) for purged A tuples and propagated B tuples. These purged tuples will be used by another sliced window join as input streams, which will be explained later in this section.

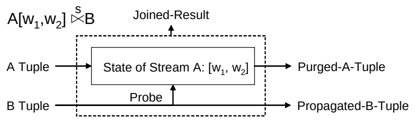


Figure 5: Sliced One-Way Window Join.

The execution steps to be followed for the sliced window join $A[W^{start}, W^{end}] \overset{s}{\ltimes} B$ are shown in Fig. 6.

When a new tuple a arrives on A
1. *Insert:* Add a into sliding window $A[W^{start}, W^{end}]$

When a new tuple b arrives on B
1. *Cross-Purge:* Update $A[W^{start}, W^{end}]$ to purge expired A tuples, i.e. if $a' \in A[W^{start}, W^{end}]$ and $(T_b - T_{a'}) > W^{end}$, move a' into *Purged-A-Tuple* queue (if exists) or discard (if not exists)
2. *Probe:* Emit result pairs (a, b) according to Def. 1 for b and $a \in A[W^{start}, W^{end}]$ to *Joined-Result* queue
3. *Propagate:* Add b into *Propagated-B-Tuple* queue (if exists) or discard (if not exists)

Figure 6: Execution of $A[W^{start}, W^{end}] \overset{s}{\ltimes} B$.

The semantics of the state-sliced window join require the checking of both the upper and lower bounds of the timestamps in every tuple probing step. In Fig. 6, the newly arriving tuple b will first purge the state of stream A with W^{end}, before probing is attempted. Then the probing can be conducted without checking of the upper bound of the window constraint W^{end}. The checking of the lower bound of the window W^{end} can also be omitted in the probing since we use the sliced window join operators in a pipelining chain manner, as discussed below.

Definition 2. *A chain of sliced one-way window joins is a sequence of pipelined N sliced one-way window joins, denoted as $A[0, W_1] \overset{s}{\ltimes} B$, $A[W_1, W_2] \overset{s}{\ltimes} B$, ..., $A[W_{N-1}, W_N] \overset{s}{\ltimes} B$. The start window of the first join in a chain is 0. For any adjacent two joins, J_i and J_{i+1}, the start window of J_{i+1} equals the end window of prior J_i ($0 \leq i < N$) in the chain. J_i and J_{i+1} are connected by both the Purged-A-Tuple output queue of J_i as the input A stream of J_{i+1}, and the Propagated-B-Tuple output queue of J_i as the input B stream of J_{i+1}.*

Fig. 7 shows a chain of state-sliced window joins having two one-way joins J_1 and J_2. We assume the input stream tuples to J_2, no matter from stream A or from stream B, are processed strictly in the order of their global time-stamps. Thus we use one logical queue between J_1 and J_2. This does not prevent us from using physical queues for individual input streams.

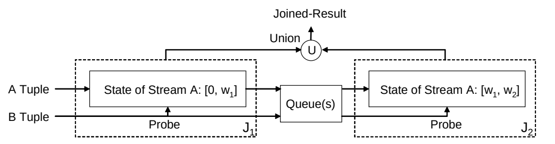


Figure 7: Chain of 1-way Sliced Window Joins.

Table 2 depicts an example execution of this chain. We assume that one single tuple (an a or a b) will only arrive at the start of each second, $w_1 = 2sec$, $w_2 = 4sec$ and every a tuple will match every b tuple (Cartesian Product semantics). During every second, an operator will be selected to run. Each running of the operator will process one input tuple. The content of the states in J_1 and J_2, and the content in the queue between J_1 and J_2 after each running of the operator are shown in Table 2.

T	Arr.	OP	$A :: [0,2]$	Queue	$A :: [2,4]$	Output
1	a_1	J_1	$[a_1]$	[]	[]	
2	a_2	J_1	$[a_2,a_1]$	[]	[]	
3	a_3	J_1	$[a_3,a_2,a_1]$	[]	[]	
4	b_1	J_1	$[a_3,a_2]$	$[b_1,a_1]$	[]	(a_2,b_1), (a_3,b_1)
5	b_2	J_1	$[a_3]$	$[b_2,a_2,b_1,a_1]$	[]	(a_3,b_2)
6		J_2	$[a_3]$	$[b_2,a_2,b_1]$	$[a_1]$	
7		J_2	$[a_3]$	$[b_2,a_2]$	$[a_1]$	(a_1,b_1)
8	a_4	J_1	$[a_4,a_3]$	$[b_2,a_2]$	$[a_1]$	
9		J_2	$[a_4]$	$[a_3,b_2]$	$[a_2,a_1]$	
10		J_2	$[a_4]$	$[a_3]$	$[a_2,a_1]$	(a_1,b_2), (a_2,b_2)

Table 2: Execution of the Chain: J_1, J_2.

Execution in Table 2 follows the steps in Fig. 6. For example at the 4th second, first a_1 will be purged out of J_1 and inserted into the queue by the arriving b_1, since $T_{b_1} - T_{a_1} \geq 2sec$. Then b_1 will purge the state of J_1 and output the joined result. Lastly, b_1 is inserted into the queue.

Note that the union of the join results of J_1: $A[0, w1] \overset{s}{\ltimes} B$ and J_2: $A[w1, w2] \overset{s}{\ltimes} B$ is equivalent to the results of a regular sliding window join: $A[w2] \ltimes B$. The order among the joined results is restored by the merge union operator.

To prove that the chain of sliced joins provides the complete join answer, we first introduce the following lemma.

LEMMA 1. *For any sliced one-way sliding window join $A[W_{i-1}, W_i] \overset{s}{\ltimes} B$ in a chain, at the time that one b tuple finishes the cross-purge step, but not yet begins the probe step, we have: (1) $\forall a \in A :: [W_{i-1}, W_i] \Rightarrow W_{i-1} \leq T_b - Ta < W_i$; and (2) $\forall a$ tuple in the input steam A, $W_{i-1} \leq T_b - Ta < W_i \Rightarrow a \in A :: [W_{i-1}, W_i]$. Here $A :: [W_{i-1}, W_i]$ denotes the state of stream A.*

Proof: (1). In the cross-purge step (Fig. 6), the arriving b will purge any tuple a with $T_b - T_a \geq W_i$. Thus $\forall a_i \in A :: [W_{i-1}, W_i], T_b - Ta_i < W_i$. For the first sliced window join in the chain, $W_{i-1} = 0$. We have $0 \leq T_b - Ta$. For other joins in the chain, there must exist a tuple $a_m \in A :: [W_{i-1}, W_i]$ that has the maximum timestamp among all the a tuples in $A :: [W_{i-1}, W_i]$. Tuple a_m must have been purged by b' of stream B from the state of the previous join operator in the chain. If $b' = b$, then we have $T_b - T_{a_m} \geq W_{i-1}$, since W_{i-1} is the upper window bound of the previous join operator. If $b' \neq b$, then $T_{b'} - T_{a_m} > W_{i-1}$, since $T_b > T_{b'}$. We still have $T_b - T_{a_m} > W_{i-1}$. Since $T_{a_m} \geq T_{a_k}$, for $\forall a_k \in A :: [W_{i-1}, W_i]$, we have $W_{i-1} \leq T_b - Ta_k$, for $\forall a_k \in A :: [W_{i-1}, W_i]$).

(2). We use a proof by contradiction. If $a \notin A :: [W_{i-1}, W_i]$, then first we assume $a \in A :: [W_{j-1}, W_j], j < i$. Given $W_{i-1} \leq T_b - T_a$, we know $W_j \leq T_b - T_a$. Then a cannot be inside the state $A :: [W_{j-1}, W_j]$ since a would have been purged by b when it is processed by the join operator $A[W_{j-1}, W_j] \overset{s}{\ltimes} B$. We got a contradiction. Similarly a cannot be inside any state $A :: [W_{k-1}, W_k], k > i$. □

THEOREM 1. *The union of the join results of all the sliced one-way window joins in a chain $A[0, W_1] \overset{s}{\ltimes} B$, ..., $A[W_{N-1}, W_N] \overset{s}{\ltimes} B$ is equivalent to the results of a regular one-way sliding window join $A[W_N] \ltimes B$.*

Proof: "⇐". Lemma 1(1) shows that the sliced joins in a chain will not generate a result tuple (a, b) with $T_a - T_b > W$. That is, $\forall (a, b) \in \bigcup_{1 \leq i \leq N} A[W_{i-1}, W_i] \overset{s}{\ltimes} B \Rightarrow (a, b) \in A[W] \ltimes B$.
"⇒". We need to show: $\forall (a, b) \in A[W] \ltimes B \Rightarrow \exists i, s.t.(a, b) \in A[W_{i-1}, W_i] \overset{s}{\ltimes} B$. Without loss of generality, $\forall (a, b) \in A[W] \ltimes B$, there exists unique i, such that $W_{i-1} \leq T_b - T_a < W_i$, since $W_0 \leq T_b - T_a < W_N$. We want to show that $(a, b) \in A[W_{i-1}, W_i] \overset{s}{\ltimes} B$. The execution steps in Fig. 6 guarantee that the tuple b will be processed by $A[W_{i-1}, W_i] \overset{s}{\ltimes} B$ at a certain time. Lemma 1(2) shows that tuple a would be inside the state of $A[W_{i-1}, W_i]$ at that same time. Then $(a, b) \in A[W_{i-1}, W_i] \overset{s}{\ltimes} B$. Since i is unique, there is no duplicated probing between tuples a and b. □

From Lemma 1, we see that the state of the regular one-way sliding window join $A[W] \ltimes B$ is distributed among different sliced one-way joins in a chain. These sliced states are disjoint with each other in the chain, since the tuples in the state are purged from the state of the previous join. This property is independent from operator scheduling, be it synchronous or even asynchronous.

4.2 State-Sliced Binary Window Join

Similar to Definition 1, we can define the binary sliding window join. The definition of the chain of sliced binary joins is similar to Definition 2 and is thus omitted for space reasons. Fig. 8 shows an example of a chain of state-sliced binary window joins.

DEFINITION 3. *A sliced binary window join of streams A and B is denoted as $A[W_A^{start}, W_A^{end}] \overset{s}{\bowtie} B[W_B^{start}, W_B^{end}]$, where stream A has a sliding window of range: $W_A^{end} - W_A^{start}$ and stream B has a window of range $W_B^{end} - W_B^{start}$. The join result consists of all pairs of tuples $a \in A$, $b \in B$, such that either $W_A^{start} \leq T_b - T_a < W_A^{end}$ or $W_B^{start} \leq T_a - T_b < W_B^{end}$, and (a, b) satisfies the join condition.*

The execution steps for sliced binary window joins can be viewed as a combination of two one-way sliced window joins. Each input tuple from stream A or B will be captured as two reference copies, before the tuple is processed by the first binary sliced window join[2]. One reference is annotated as

[2]The copies can be made by the first binary sliced join.

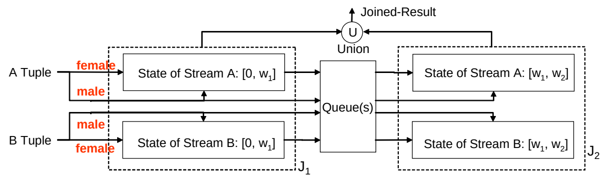

Joined-Result
Union
A Tuple
female
male
B Tuple
male
female
State of Stream A: [0, w1]
State of Stream B: [0, w1]
Queue(s)
State of Stream A: [w1, w2]
State of Stream B: [w1, w2]
J1
J2

Figure 8: Chain of Binary Sliced Window Joins.

the *male* tuple (denoted as a^m) and the other as the *female* tuple (denoted as a^f).

The execution steps to be followed for the processing of a stream A tuple by $A[W^{start}, W^{end}] \overset{s}{\bowtie} B[W^{start}, W^{end}]$ are shown in Fig. 9. The execution procedure for the tuples arriving from stream B can be similarly defined.

```
When a new tuple $a^m$ arrives
1.*Cross-Purge:* Update $B[W^{start}, W^{end}]$ to purge expired $B$ tuples, i.e. if $b^f \in B[W^{start}, W^{end}]$ and $(T_{a^m} - T_{b^f}) > W^{end}$, move $b^f$ into the queue (if exists) towards next join operator or discard (if not exists)
2.*Probe:* Emit $a^m$ join with $b^f \in B[W^{start}, W^{end}]$ to *Joined-Result* queue
3.*Propagate:* Add $a^m$ into the queue (if exists) towards next join operator or discard (if not exists)

When a new tuple $a^f$ arrives
1.*Insert:* Add $a^f$ into the sliding window $A[W^{start}, W^{end}]$
```

Figure 9: Execution of Binary Sliced Window Join.

Intuitively the male tuples of stream B and female tuples of stream A are used to generate join tuples equivalent to a one-way join: $A[W^{start}, W^{end}] \overset{s}{\ltimes} B$. The male tuples of stream A and female tuples of stream B are used to generate join tuples equivalent to the other one-way join: $A \overset{s}{\rtimes} B[W^{start}, W^{end}]$.

Note that using two copies of a tuple will not require doubled system resources since: (1) the combined workload (in Fig. 9) to process a pair of female and male tuples equals the processing of one tuple in a regular join operator, since one tuple takes care of purging/probing and the other filling up the states; (2) the state of the binary sliced window join will only hold the female tuple; and (3) assuming a simplified queue (M/M/1), doubled arrival rate (from the two copies) and doubled service rate (from above (1)) still would not change the average queue size, if the system is stable. In our implementation, we use a copy-of-reference instead of a copy-of-object, aiming to reduce the potential extra queue memory during bursts of arrivals. Discussion of scheduling strategies and their effects on queues is beyond the scope of this paper.

THEOREM 2. *The union of the join results of the sliced binary window joins in a chain* $A[0, W_1] \overset{s}{\bowtie} B[0, W_1]$, ..., $A[W_{N-1}, W_N] \overset{s}{\bowtie} B[W_{N-1}, W_N]$ *is equivalent to the results of a regular sliding window join* $A[W_N] \bowtie B[W_N]$.

Using Theorem 1, we can prove Theorem 2. Since we can treat a binary sliced window join as two parallel one-way sliced window joins, the proof is fairly straightforward. It is omitted here for space reasons.

We now show how the proposed state-slice sharing can be applied to the running example in Section 3 to share the computation between the two queries. The shared plan is depicted in Figure 10. This shared query plan includes a chain of two sliced sliding window join operators $\overset{s}{\bowtie}_1$ and $\overset{s}{\bowtie}_2$. The purged tuples from the states of $\overset{s}{\bowtie}_1$ are sent to $\overset{s}{\bowtie}_2$ as input tuples. The selection operator σ_A filters the input stream A tuples for $\overset{s}{\bowtie}_2$. The selection operator σ'_A filters the joined results of $\overset{s}{\bowtie}_1$ for Q_2. The predicates in σ_A and σ'_A are both $A.value > Threshold$.

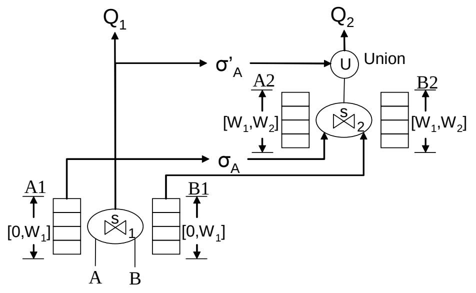


Figure 10: State-Slice Sharing for $Q1$ and $Q2$.

4.3 Discussion and Analysis

Compared to alternative sharing approaches discussed in Section 3, the state-slice sharing paradigm offers the following benefits:

- Selection can be pushed down into the middle of the join chain. Thus unnecessary probings in the join operators are avoided.
- The routing cost is saved. Instead a pre-determined route is embedded in the query plan.
- States of the sliced window joins in a chain are disjoint with each other. Thus no state memory is wasted.

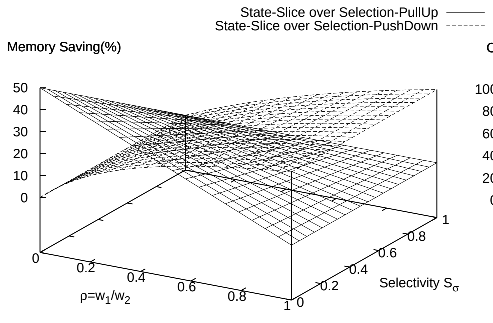
Using the same settings as in Section 3, we now calculate the state memory consumption C_m and the CPU cost C_p for the state-slice sharing paradigm as follows:

$$\begin{cases} C_m = & 2\lambda W_1 M_t + (1+S_\sigma)\lambda(W_2 - W_1)M_t \\ C_p = & 2\lambda^2 W_1 + \lambda + 2\lambda^2 S_\sigma (W_2 - W_1) + \\ & 4\lambda + 2\lambda + 2\lambda^2 S_{\bowtie} W_1 \end{cases} \tag{3}$$

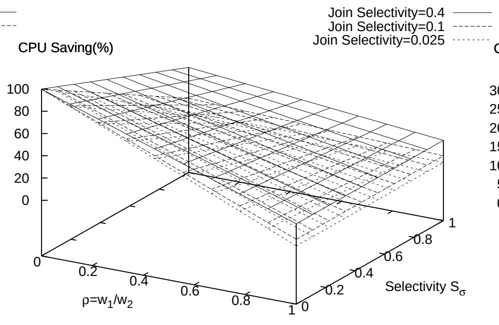
The first item of C_m corresponds to the state memory in $\overset{s}{\bowtie}_1$; the second to the state memory in $\overset{s}{\bowtie}_2$. The first item of C_p is the join probing cost of $\overset{s}{\bowtie}_1$; the second the filter cost of σ_A; the third the join probing cost of $\overset{s}{\bowtie}_2$; the fourth the cross-purge cost; while the fifth the union cost; the sixth the filter cost of σ'_A. The union cost in C_p is proportional to the input rates of streams A and B. The reason is that the male tuple of the last sliced join $\overset{s}{\bowtie}_2$ acts as punctuation [26] for the union operator. For example, the male tuple a_1^f is sent to the union operator after it finishes probing the state of stream B in $\overset{s}{\bowtie}_2$, indicating that no more joined tuples with timestamps smaller than a_1^f will be generated in the future. Such punctuations are used by the union operator for the sorting of joined tuples from multiple join operators [26].

Comparing the memory and CPU costs for the different sharing solutions, namely naive sharing with selection pull-up (Eq. 1), stream partition with selection push-down (Eq. 2) and state-slice chain (Eq. 3), the savings of using the state slicing sharing are:

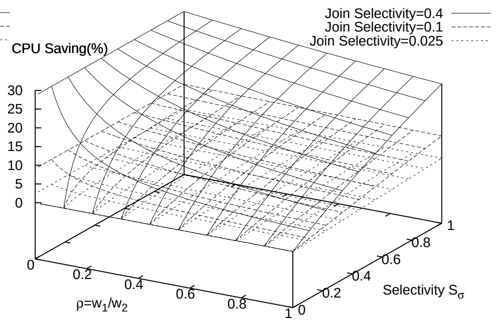
$$\begin{cases} \frac{C_m^{(1)} - C_m^{(3)}}{C_m^{(1)}} = & \frac{(1-\rho)(1-S_\sigma)}{2} \\ \frac{C_m^{(2)} - C_m^{(3)}}{C_m^{(2)}} = & \frac{\rho}{1+2\rho+(1-\rho)S_\sigma} \\ \frac{C_p^{(1)} - C_p^{(3)}}{C_p^{(1)}} = & \frac{(1-\rho)(1-S_\sigma)+(2-\rho)S_{\bowtie}}{1+2S_{\bowtie}} \\ \frac{C_p^{(2)} - C_p^{(3)}}{C_p^{(2)}} = & \frac{S_\sigma S_{\bowtie}}{\rho(1-S_\sigma)+S_\sigma+S_\sigma S_{\bowtie}+\rho S_{\bowtie}} \end{cases} \tag{4}$$



(a)



(b)



(c)

Figure 11: (a) Memory Comparison; (b) CPU Comparison: State-Slice vs. Selection PullUp; (c) CPU Comparison: State-Slice vs. Selection PushDown.

with $C_m^{(i)}$ denoting C_m, $C_p^{(i)}$ denoting C_p in Equation i ($i = 1, 2, 3$); and window ratio $\rho = \frac{W_1}{W_2}$, $0 < \rho < 1$.

The memory and CPU savings under various settings calculated from Equation 4 are depicted in Fig. 11. Compared to sharing alternatives in Section 3, state-slice sharing achieves significant savings. As a base case, when there is no selection in the query plans (i.e., $S_\sigma = 1$), state-slice sharing will consume the same amount of memory as the selection pullup while the CPU saving is proportional to the join selectivity $S_{\bowtie}$. When selection exists, state-slice sharing can save about 20%-30% memory, 10%-40% CPU over the alternatives on average. For the extreme settings, the memory savings can reach about 50% and the CPU savings about 100% (Fig. 11(a), 11(b)). The actual savings are sensitive to these parameters. Moreover, from Eq. 4 we can see that all the savings are positive. This means that the state-sliced sharing paradigm achieves the lowest memory and CPU costs under all these settings. Note that we omit λ in Eq. 4 for CPU cost comparison, since its effect is small when the number of queries is only 2. The CPU savings will increase with increasing λ, especially when the number of queries is large.

5. STATE-SLICE: BUILD THE CHAIN

In this section, we discuss how to build an optimal shared query plan with a chain of sliced window joins. Consider a DSMS with N registered continuous queries, where each query performs a sliding window join $A[w_i] \bowtie B[w_i]$ ($1 \leq i \leq N$) over data streams A and B. The shared query plan is a DAG with multiple roots, one for each of the queries.

Given a set of continuous queries, the queries are first sorted by their window lengths in ascending order. We propose two algorithms for building the state-slicing chain in that order (Section 5.1 and 5.2). The choice between them depends on the availability of the CPU and memory in the system. The chain can also first be built using one of the algorithms and migrated towards the other by merging or splitting the slices at runtime (Section 5.3).

5.1 Memory-Optimal State-Slicing

Without loss of generality, we assume that $w_i < w_{i+1}$ ($1 \leq i < N$). Let's consider a chain of the N sliced joins: $J_1, J_2, ..., J_N$, with J_i as $A[w_{i-1}, w_i] \overset{s}{\bowtie} B[w_{i-1}, w_i]$ ($1 \leq i \leq N, w_0 = 0$). A union operator U_i is added to collect joined results from $J_1, ..., J_i$ for query Q_i ($1 < i \leq N$), as shown in Fig. 12. We call this chain the *memory-optimal state-slice sharing* (Mem-Opt).

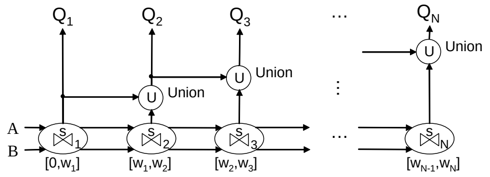


Figure 12: Mem-Opt State-Slice Sharing.

The correctness of Mem-Opt state-slice sharing is proven in Theorem 3 by using Theorem 2. We have the following equivalence for i ($1 \leq i \leq N$):

$$Q_i: \ A[w_i] \bowtie B[w_i] = \bigcup_{1 \leq j \leq i} A[W_{j-1}, W_j] \overset{s}{\bowtie} B[W_{j-1}, W_j]$$

THEOREM 3. *The total state memory used by a Mem-Opt chain of sliced joins $J_1, J_2, ..., J_N$, with J_i as $A[w_{i-1}, w_i] \overset{s}{\bowtie} B[w_{i-1}, w_i]$ ($1 \leq i \leq N, w_0 = 0$) is equal to the state memory used by the regular sliding window join: $A[w_N] \bowtie B[w_N]$.*

Proof: From Lemma 1, the maximum timestamp difference of tuples (e.g., A tuples) in the state of J_i is $(w_i - w_{i-1})$, when continuous tuples from the other stream (e.g., B tuples) are processed. Assume the arrival rate of streams A and B is denoted by λ_A and λ_B respectively. Then we have:

$$\begin{aligned} & \sum_{1 \leq i \leq N} Mem_{J_i} \\ = \ & (\lambda_A + \lambda_B)[(w_1 - w_0) + (w_2 - w_1) + ... + (w_N - w_{N-1})] \\ = \ & (\lambda_A + \lambda_B)w_N \end{aligned}$$

□

$(\lambda_A + \lambda_B)w_N$ is the minimal amount of state memory that is required to generate the full joined result for Q_N. Thus the Mem-Opt chain consumes the minimal state memory.

Let's again use the count of comparisons per time unit as the metric for estimated CPU costs. Comparing the execution (Fig. 9) of a sliced window join with the execution (Fig. 1) of a regular window join, we notice that the probing cost of the chain of sliced joins: $J_1, J_2, ..., J_N$ is equivalent to the probing cost of the regular window join: $A[w_N] \bowtie B[w_N]$.

Comparing to the alternative sharing paradigms in Section 3, we notice that the Mem-Opt chain may not always win since it requires CPU cost for: (1) $(N-1)$ more times of purging for each tuple in the streams A and B; (2) extra system overhead for running more operators; and (3) CPU cost for $(N-1)$ union operators. In the case that the selectivity of the join $S_{\bowtie}$ is rather small, the routing cost in the selection pull-up sharing may be less than the extra cost of the Mem-Opt chain. In short, the Mem-Opt chain may not be the CPU-optimal solution for all settings.

5.2 CPU-Optimal State-Slicing

We hence now discuss how to find the CPU-Optimal state-slice sharing (*CPU-Opt*) which will yield minimal CPU costs. We notice that the Mem-Opt state-slice sharing may result in a large number of sliced joins with very small window ranges each. In such cases, the extra per tuple purge cost and the system overhead for holding more operators may not be neglectable.

In Fig 13(b), the state-sliced joins from J_i to J_j are merged into a larger sliced join with the window range being the summation of the window ranges of J_i and J_j. A routing operator then is added to split the joined results to the associated queries. Such merging of concatenated sliced joins can be done iteratively until all the sliced joins are merged together. In the extreme case, the totally merged join results in a shared query plan, which is equal to that formed by using the selection pull-up sharing method shown in Section 3. The CPU cost may decrease after the merging.

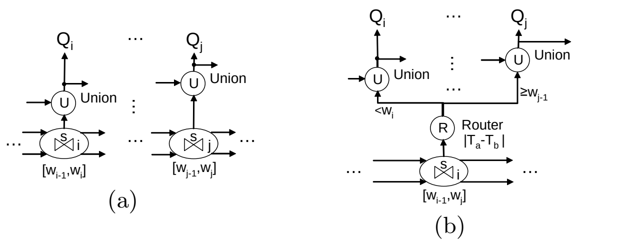


Figure 13: Merging Two Sliced Joins.

Both the shared query plans in Fig. 13 have the same join probing costs and union costs. Using the symbols defined in Section 3 and C_{sys} denoting the system overhead factor, we can calculate the difference of partial CPU cost $C_p^{(a)}$ in Fig 13(a) and $C_p^{(b)}$ in Fig 13(b) as:

$$\begin{aligned} C_p^{(a)} - C_p^{(b)} = \quad & (\lambda_A + \lambda_B)(j-i) - 2\lambda_A\lambda_B(w_j - w_{i-1})\sigma_{\bowtie}(j-i) + \\ & C_{sys}(j-i+1)(\lambda_A+\lambda_B) \end{aligned}$$

The difference of CPU costs in these scenarios comes from the purge cost (the first item), the routing cost (the second item) and the system overhead (the third item). The system overhead mainly includes the cost for moving tuples in/out of the queues and the context change cost of operator scheduling. The system overhead is proportional to the data input rates and number of operators.

Considering a chain of N sliced joins, all possible merging of sliced joins can be represented by edges in a directed graph $G = \{V, E\}$, where V is a set of $N+1$ nodes and E is a set of $\frac{N(N+1)}{2}$ edges. Let $\forall v_i \in V (0 \le i \le N)$ represent the window w_i of Q_i $(w_0 = 0)$. Let the edge $e_{i,j}$ from node v_i to node v_j $(i < j)$ represent a sliced join with start-window as w_i and end-window as w_j. Then each path from the node v_0 to node v_N represents a variation of the merged state-slice sharing, as shown in Fig. 14.

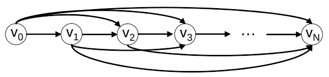


Figure 14: Directed Graph of State-Slice Sharing.

Similar to the above calculation of $C_p^{(a)}$ and $C_p^{(b)}$, we can calculate the CPU cost of the merged sliced window joins represented by every edge. We denote the CPU cost $e_{i,j}$ of the sliced join as the length of the edge $l_{i,j}$. We have the following lemma.

LEMMA 2. *The calculations of CPU costs $l_{i,j}$ and $l_{m,n}$ are independent if $0 \le i < j \le m < n \le N$.*

Based on Lemma 2, we can apply the *principle of optimality* [4] here and transform the optimal state-slice problem to the problem of finding the shortest path from v_0 to v_N in an acyclic directed graph. Using the well-known *Dijkstra's algorithm* [11], we can find the CPU-Opt query plan in $O(N^2)$, with N being the number of the distinct window constraints in the system. Even when we incorporate the calculation of the CPU cost of the $\frac{N(N+1)}{2}$ edges, the total time for getting the CPU optimal state-sliced sharing is still $O(N^2)$.

In case the queries do not have selections, the CPU-Opt chain will consume the same amount of memory as the Mem-Opt chain. With selections, the CPU-Opt chain may consume more memory. See Section 6 for more discussion.

5.3 Online Migration of the State-Slicing Chain

Online migration of the shared query plan is important for efficient processing of stream queries. The state-slicing chain may need maintenance when: (1) queries enter or leave the system, (2) queries update predicates or window constraints, and (3) runtime statistic collection invokes plan adaptation.

The chain migration is achieved by two primitive operation: merging and splitting of the sliced join. For example when query Q_i $(i < N)$ leaves the system, the corresponding sliced join $A[w_{i-1}, w_i] \overset{s}{\bowtie} B[w_{i-1}, w_i]$ could be merged with the next sliced join in the chain. Or if the corresponding sliced join had been merged with others in the CPU-Opt chain, splitting of the merged join may be invoked first.

Online splitting of the sliced join J_i can be achieved by: (1) stopping the system execution for J_i; (2) updating the end window of J_i to w'_i; (3) inserting a new sliced join J'_i with window $[w'_i, w_i]$ to the right of J_i and connecting the query plan; and (4) resuming the system. The queue between J_i and J'_i is empty right after the insertion. The execution of J_i will purge tuples, due to its new smaller window, into the queue between J_i and J'_i and eventually fill up the states of J'_i correctly.

Online merging of two adjacent sliced joins J_i and J_{i+1} requires the queues between these two joins empty. This can be achieved by scheduling the execution of J_{i+1} after stopping the scheduling of J_i. Once the queue between J_i and J_{i+1} is empty, we can simply (1) concatenate the corresponding states of J_i and J_{i+1}; (2) update the end window of J_i to w_{i+1}; (3) remove J_{i+1} from the chain; and (4) resume the system.

The overhead for chain migration corresponds to constant system cost for operator insertion/deletion. The system suspending time during join splitting is neglectable, while during join merging it is bounded by the execution time needed to empty the queue in-between. No extra processing costs arise in either case.

6. PUSH SELECTIONS INTO CHAIN

When the N continuous queries each have selections on the input streams, we aim to push the selections down into the chain of sliced joins. For clarity of discussion, we focus on the selection push-down for predicates on one input stream. Predicates on multiple streams can be pushed down similarly. We denote the selection predicate on the input stream A of query Q_i as σ_i and the condition of σ_i as $cond_i$.

6.1 Mem-Opt Chain with Selection Push-down

The selections can be pushed down into the chain of sliced joins as shown in Fig. 15. The predicate of the selection σ'_i corresponds to the disjunction of the selection predicates from σ_i to σ_N. That is:

$$cond'_i = cond_i \vee cond_{i+1} \vee \cdots \vee cond_N$$

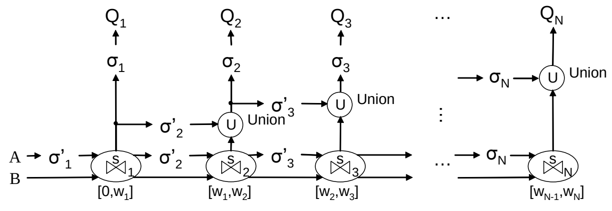


Figure 15: Selection Push-down for Mem-Opt State-Slice Sharing.

Logically each tuple may be evaluated against the same selection predicate for multiple times. In the actual execution, we can evaluate the predicates ($cond_i$, $1 \leq i \leq N$) in the decreasing order of i for each tuple. As soon as a predicate (e.g. $cond_k$) is satisfied, stop further evaluating and attach k to the tuple. Thus this tuple can survive until the kth slice join and no further. Such idea is similar to the tuple lineage proposed in [18]. We omit the detailed discussion since it is orthogonal to our state-slice concept.

Similar to Theorem 3, we have the following theorem.

THEOREM 4. *The Mem-Opt state-slice sharing with selection push-down consumes the minimal state memory for a given workload.*

Intuitively the total state memory consumption is minimal since that: (1) each join probe performed by $\bowtie_i$ in Fig. 15 is required at least by one of the queries: Q_i, Q_{i+1}, ..., Q_N; (2) any input tuple that won't contribute to the joined results will be filtered out immediately; and (3) the contents in the state memory of all sliced joins are pairwise disjoint with each other.

6.2 CPU-Opt Chain with Selection Push-down

The merging of adjacent sliced joins with selection push-down can be achieved following the scheme shown in Fig. 16. Merging sliced joins having selection between them will cost extra state memory usage due to selection pull-up. The tuples, which would be filtered out by the selection before, will now stay unnecessarily long in the state memory. Also, the consequent join probing cost will increase accordingly. Continuous merging of the sliced joins will result in the selection pull-up sharing approach discussed in Section 3.

Similarly to the CPU optimization in Section 5.2, the Dijkstra's algorithm can be used to find the CPU-Opt sharing plan with minimized CPU cost in $O(N^2)$. Such CPU-Opt sharing plan may not be Mem-Opt.

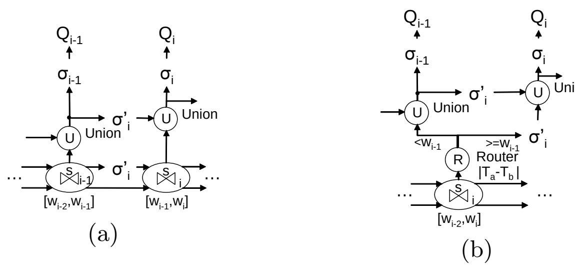


Figure 16: Merging Sliced Joins with Selections.

7. PERFORMANCE STUDY

We have implemented the proposed state-slice sharing paradigm in a DSMS system (*CAPE*) [22]. Experiments have been conducted to thoroughly test the ability of the sharing paradigm under various system resource settings. We compare the CPU and memory usages for the same set of continuous queries using different sharing approaches.

7.1 Experimental System Overview

The *CAPE* is implemented in Java. All experiments are conducted on a machine running windows XP with a 2.8GHz processor and 1GB main memory. The DSMS includes a synthetic data stream generator, a query processor and several result receivers. The query processor employs round-robin scheduling for executing the operators. The query processor has a monitor thread that collects the runtime statistics of each operator. In all the experiments, the stream generator will run for 90 seconds. All the experiments start with empty states for all operators.

We measure the runtime memory usage in terms of the number of tuples staying in the states of the joins. We measure the CPU cost of the query plans in terms of the average service rate ($\frac{Total\_Throughput}{Running\_Time}$).

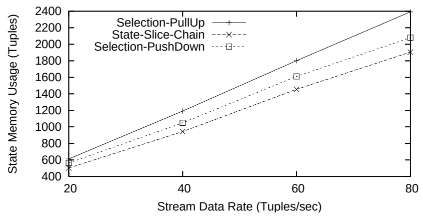
The tuples in the data streams are generated according to the Poisson arrival pattern. The stream input rate is changed by setting the mean inter-arrival time between two tuples.

7.2 State-Slice vs. Other Sharing Strategies

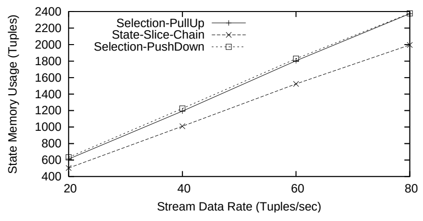
Eq. 4 analytically compares the performance of state-slice sharing with other sharing alternatives. The experiments in this section aim to verify these benefits empirically.

We use three queries and the Mem-Opt chain buildup in these experiments. The queries are: Q_1 ($A[W_1] \bowtie B[W_1]$), Q_2 ($\sigma(A[W_2]) \bowtie B[W_2]$) and Q_3 ($\sigma(A[W_3]) \bowtie B[W_3]$). Apparently these three queries can share partial computations among each other. Using the Mem-Opt state-slice sharing, the shared query plan has a chain of three sliced joins with window constraints as $[0, W_1]$, $[W_1, W_2]$ and $[W_2, W_3]$. The joined results are unioned and sent to each data receiver respectively. We compare the state-slice sharing with the naive sharing with selection pull-up and the stream partition with selection push-down (see Section 3). Using the naive sharing approach with selection pull-up, the shared plan will have one regular sliding window join: $A[W_3] \bowtie B[W_3]$. Using the stream partition with selection push-down, the shared plan will have two regular joins: $A[W_1] \bowtie B[W_1]$ and $A[W_3] \bowtie B[W_3]$. The input stream A is partitioned by σ and sent to these two joins.

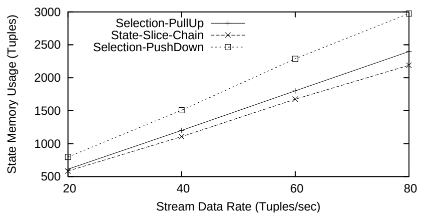
We vary the parameters as shown in Table 3. All the settings are moderate instead of extreme. Experiments with



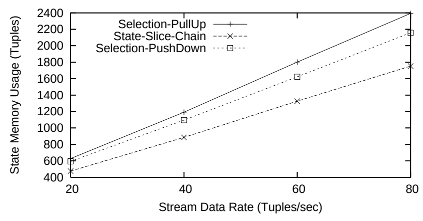
(a) Mostly-Small, $S_{\bowtie} = 0.1$, $S_{\sigma} = 0.5$



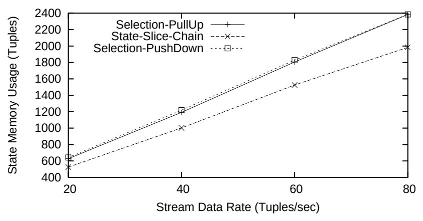
(b) Uniform, $S_{\bowtie} = 0.1$, $S_{\sigma} = 0.5$



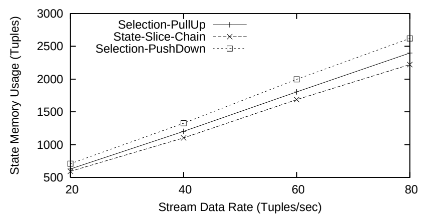
(c) Mostly-Large, $S_{\bowtie} = 0.1$, $S_{\sigma} = 0.5$



(d) Uniform, $S_{\bowtie} = 0.025$, $S_{\sigma} = 0.2$



(e) Uniform, $S_{\bowtie} = 0.025$, $S_{\sigma} = 0.5$



(f) Uniform, $S_{\bowtie} = 0.025$, $S_{\sigma} = 0.8$

Figure 17: Memory Comparison with Various Parameters

all the combination of these settings are conducted. The input rates of the streams vary from 20 tuples/sec. to 80 tuples/sec in all the experiments.

Window Distribution(Sec.)	Mostly-Small: 5, 10, 30	Uniform: 10, 20, 30	Mostly-Large: 20, 25, 30
S_{σ}	Low(0.2)	Middle(0.5)	High(0.8)
$S_{\bowtie}$	Low(0.025)	Middle(0.1)	High(0.4)

Table 3: System Settings Used in Section 7.2.

The results showing memory consumption comparisons are depicted in Fig. 17. Fig. 17(a), 17(b) and 17(c) show that the memory usage is sensitive to the window distributions. Fig. 17(d), 17(e) and 17(f) illustrate the effect of S_{σ} on the memory usage. Comparing Fig. 17(b) and 17(e), we can see that $S_{\bowtie}$ does not affect the memory usage since the number of joined tuples is unrelated to the state memory of the join. Overall, the state-slice sharing always achieves the minimal memory consumption, with the memory savings ranging from 20% to 30%.

Fig. 18 shows the comparison of the service rate under various settings. Fig. 18(a), 18(b) and 18(c) show the change of service rate under different window distributions. Fig. 18(d), 18(e) and 18(f) illustrate the effect of $S_{\bowtie}$ on the service rate. Overall, the state-slice sharing always achieves the maximum service rate.

From Fig. 18 we can see that with increasing data input rate, more performance improvements can be expected from the state-slice sharing. One reason is that the number of joined tuples is proportional to $\lambda_A * \lambda_B$. Thus the routing cost increases quadratically. On the contrary, the extra purging cost in the state-slice sharing is proportional to $\lambda_A + \lambda_B$. Thus the purging cost only increases linearly. Then the state-slice sharing is more scalable with the data input rates. Under the scenario of large join selectivities and high-volume input streams, the performance improvement of using state-slice sharing can reach 40%.

7.3 State-slice: Mem-Opt vs. CPU-Opt

In this set of experiments, we focus on the performance comparison between the Mem-Opt and the CPU-Opt chains under different system settings. We use similar queries as in Section 7.2 with the selections removed. We also use the service rate to measure the CPU consumptions. The CPU-Opt chain is built from the Mem-Opt chain by merging some of the slice joins according to the algorithm discussed in Section 5.2. The experiments are conducted using different numbers of queries (12, 24, 36) and various window distributions. The window distributions for the 12 queries are shown in Table 4. The window distributions for other number of queries are set accordingly. We set the join selectivity to be 0.025. The input rates of the streams vary from 20 tuples/sec to 80 tuples/sec in all experiments. The service rate comparisons are shown in Fig. 19.

Uniform(Sec.)	2.5, 5, 7.5, 10, 12.5, 15, 17.5, 20, 22.5, 25, 27.5, 30
Mostly-Small(Sec.)	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30
Small-Large(Sec.)	1, 2, 3, 4, 5, 6, 25, 26, 27, 28, 29, 30

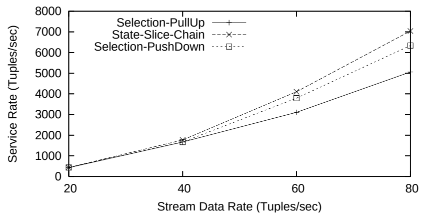
Table 4: Window Distributions Used for 12 Queries.

In Fig. 19(a), the CPU-Opt chain is actually the same as the Mem-Opt chain. However, for skewed windows distribution, the CPU-Opt chain has fewer operators than the Mem-Opt chain. In Fig. 19(b), all the small windows are merged together in the CPU-Opt chain. In Fig. 19(c), the CPU-Opt chain will have only 2 sliced joins, after merging all the small windows and all the large windows. The more skewed the windows are, the more performance improvement can be expected. The benefit of CPU-Opt over Mem-Opt chain also increases along with the number of queries, as shown in Fig. 19(d) and Fig. 19(e). The average service rate improvement is 20%-30%.

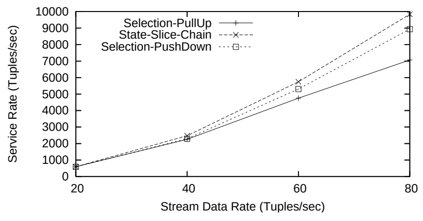
8. RELATED WORK

The problem of sharing the work between multiple queries is not new. For traditional relational databases, multiple-query optimization [23] seeks to exhaustively find an optimal shared query plan. Recent work, such as [21, 19], provides heuristics for reducing the search space for the optimally shared query plan for a set of SQL queries. These works differ from our work since we focus on the computation sharing for window-based continuous queries. The traditional SQL queries do not have window semantics.

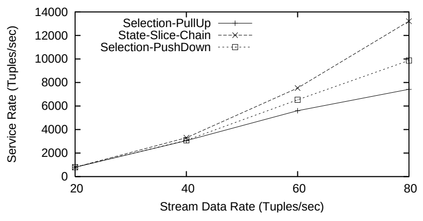
Many papers [8, 18, 10, 13, 15] in the literature have highlighted the importance of computation sharing in continuous queries. The sharing solutions employed in existing systems, such as NiagaraCQ [10], CACQ [18] and PSoup [9], focus on exploiting common subexpressions in queries. Their shared



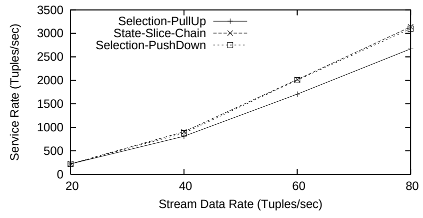
(a) Mostly-Small, $S_{\bowtie} = 0.1$, $S_{\sigma} = 0.5$



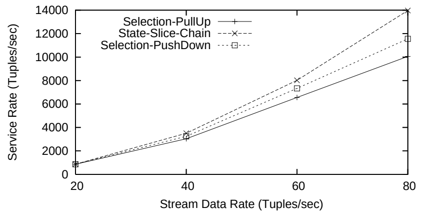
(b) Uniform, $S_{\bowtie} = 0.1$, $S_{\sigma} = 0.5$



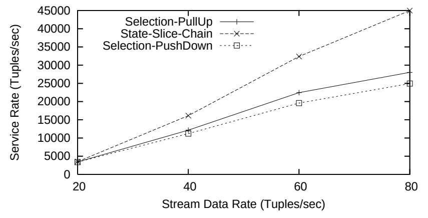
(c) Mostly-Large, $S_{\bowtie} = 0.1$, $S_{\sigma} = 0.5$



(d) Uniform, $S_{\bowtie} = 0.025$, $S_{\sigma} = 0.8$

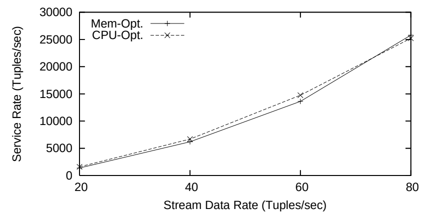


(e) Uniform, $S_{\bowtie} = 0.1$, $S_{\sigma} = 0.8$

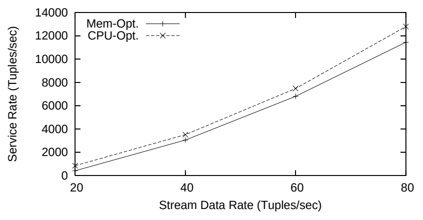


(f) Uniform, $S_{\bowtie} = 0.4$, $S_{\sigma} = 0.8$

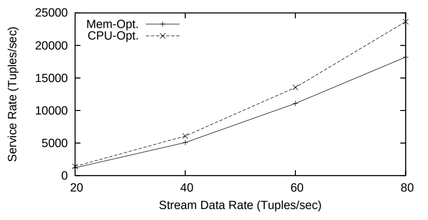
Figure 18: Service Rate Comparison with Various Parameters



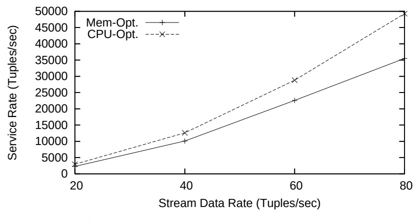
(a) Uniform, 12 Queries



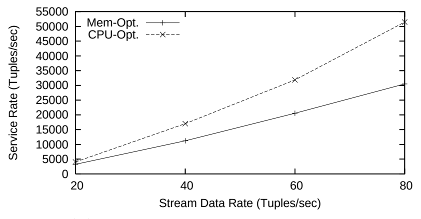
(b) Mostly-Small, 12 Queries



(c) Small-Large, 12 Queries



(d) Small-Large, 24 Queries



(e) Small-Large, 36 Queries

Figure 19: Service Rate Comparison of Mem-Opt. Chain vs. CPU-Opt. Chain

processing of joins simply ignores window constraints which are critical for window-based continuous queries.

Some previously proposed techniques are complementary to our state-slice concept, and thus can be applied to our sharing paradigm as below. The lineage of the tuples proposed in [18] can be used to avoid repeated evaluation of the same selections on a tuple in a chain of sliced joins. The precision sharing in the TULIP [15] can be used in our paradigm for selections on multiple input streams. The grouping of similar queries in [10] can be used for sharing one sliced window join among multiple continuous queries.

Recent papers [3, 28, 16] have focused on sharing computation for stateful aggregations. The work in [3], addressing operator-level sharing of multiple aggregations, has considered the effect of different windows constraints on a single stream. The work in [28] discusses shared computations among aggregations with fine-grained *phantoms*, which is the smallest unit for sharing the aggregations. The work in [16] discusses runtime aggregation sharing with different periodic windows and arbitrary predicates. However, efficient sharing of window-based join operators has thus far been ignored in the literature.

In [13] the authors propose various strategies for intra-operator scheduling for shared sliding window joins with different window sizes. Using a cost analysis, the strategies are compared in terms of average response time and query throughput. Our focus instead is on how we can minimize the memory and CPU cost for shared sliding window joins. The intra-operator scheduling strategies proposed in [13] can naturally be applied for inter-operator scheduling of our sliced joins.

Load-shedding [25] and spilling data to disk [27, 17] are alternate solutions for tackling continuous query processing with insufficient memory resources. Approximated query processing [24] is another general direction for handling memory overflow. Different from these, we minimize the actual resources required by multiple queries for accurate processing. These works are orthogonal to our work and can be applied together with our state-slice sharing.

9. CONCLUSION AND FUTURE WORK

Window-based joins are stateful operators that dominate the memory and CPU consumptions in a DSMS. Efficient sharing of window-based joins is a key technique for achieving scalability of a DSMS with high query workloads. We present a new paradigm for efficiently sharing of window-based continuous queries in a DSMS. By slicing a sliding window join into a chain of pipelining sliced joins, our novel paradigm results in a shared query plan supporting the selection push-down, without using an explosive number of operators. Based on the state-slice sharing, two algorithms are proposed for the chain buildup, which achieve either optimal memory consumption or optimal CPU usage.

One interesting direction is to extend the state-slice concept to distributed systems, because the properties of the pipelining sliced joins fit nicely in the asynchronous distributed system. Also, when the queries are too many to fit into memory, combining query indexing with state-slicing is an interesting open challenge.

10. ACKNOWLEDGMENTS

We thank Songting Chen and the anonymous reviewers for their insightful comments. We are grateful to Yali Zhu, Luping Ding, Bin Liu, Rimma V. Nehme, Mariana Jbantova and other DSRG members for their efforts building the CAPE system.

11. REFERENCES

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

[2] M. H. Ali, W. G. Aref, R. Bose, A. K. Elmagarmid, A. Helal, I. Kamel, and M. F. Mokbel. NILE-PDT: A phenomenon detection and tracking framework for data stream management systems. In *VLDB*, pages 1295–1298, 2005.

[3] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, Aug/Sep 2004.

[4] M. J. Atallah. Algorithms and theory of computation handbook, 1999.

[5] B. Babcock, S. Babu, R. Motwani, and J. Widom. Models and issues in data streams. In *PODS*, pages 1–16, June 2002.

[6] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.

[7] P. Bizarro, S. Babu, D. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *VLDB*, pages 757–768, 2005.

[8] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, August 2002.

[9] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, pages 203–214, August 2002.

[10] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, pages 345–356, 2002.

[11] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numerische Mathematik*, volume 1, pages 269–271. 1959.

[12] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[13] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, Sep 2003.

[14] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, March 2003.

[15] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *VLDB*, pages 972–986, 2004.

[16] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[17] B. Liu, Y. Zhu, and E. A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD*, pages 347–358, 2006.

[18] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60, June 2002.

[19] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, pages 307–318, 2001.

[20] O. Papaemmanouil and U. Çetintemel. Semcast: Semantic multicast for content-based data dissemination. In *ICDE*, pages 242–253, 2005.

[21] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[22] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB Demo*, pages 1353–1356, 2004.

[23] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[24] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.

[25] N. Tatbul, U. etintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[26] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, May/June 2003.

[27] T. Urhan and M. Franklin. XJoin: A reactively scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[28] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310, 2005.