Optimization of Threshold Functions over Streams

Walter Cai
University of Washington
walter@cs.washington.edu

Philip A. Bernstein
Microsoft Research
Phil.Bernstein@microsoft.com

Wentao Wu
Microsoft Research
Wento.Wu@microsoft.com

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

ABSTRACT

A common stream processing application is alerting, where the data stream management system (DSMS) continuously evaluates a threshold function over incoming streams. If the threshold is crossed, the DSMS raises an alarm. The threshold function is often calculated over two or more streams, such as combining temperature and humidity readings to determine if moisture will form on a machine and therefore cause it to malfunction. This requires taking a temporal join across the input streams. We show that for the broad class of functions called *quasiconvex* functions, the DSMS needs to retain very few tuples per-data-stream for any given time interval and still never miss an alarm. This surprising result yields a large memory savings during normal operation. That savings is also important if one stream fails, since the DSMS would otherwise have to cache all tuples in other streams until the failed stream recovers. We prove our algorithm is optimal and provide experimental evidence that validates its substantial memory savings.

PVLDB Reference Format:
Walter Cai, Philip A. Bernstein, Wentao Wu, and Badrish Chandramouli. Optimization of Threshold Functions over Streams. PVLDB, 14(6): 878 - 889, 2021.
doi:10.14778/3447689.3447693

1 INTRODUCTION

Joins over two or more data streams are ubiquitous in many applications. Most existing data stream management systems (DSMS's), such as Apache Spark [8] and Apache Flink [14], all support stream joins. Typical join processing techniques are the classic *symmetric hash join* algorithm and its variants [21], which need to buffer intermediate join state in main memory. This raises challenges when processing stream joins with memory constraints, e.g., Internet of Things (IoT) applications that run on *edge* devices such as Raspberry Pi or cell phones, or when the join state is very large [7].

In this paper, we study one common class of stream queries in practice, namely applying a threshold function over a stream join to compute an alert. Our main observation is that if the threshold function satisfies the mathematical property of *quasiconvexity*, then we can drastically reduce the memory required for the join. Our technique omits tuples from the input streams without affecting the query result.

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447693

Consider machinery on a manufacturing assembly line. A common requirement is that machines remain dry, since water would interfere with their operation. To detect when equipment is in danger of collecting moisture, the maintenance staff monitors three sensors: relative humidity h, air temperature a, and surface temperature s. The sensors emit measurements for these values in three separate data streams H, A and S. Values from these streams are then combined using the Magnus Formula [32]:

$$\phi = \ln\left(\frac{h}{100}\right) + \frac{18.678a}{257.14 + a} - s \qquad (1)$$

If ϕ exceeds 0, the surface temperature of the machinery has dropped below the dew point of the air. This implies water will condense at a higher rate than it evaporates, causing water to collect on the surface of the machinery. This should trigger an alarm.

The Magnus formula should be applied to triples of sensor readings that are within a time window of each other, that is, tuples in the result of a temporal join. For example, if the window size is ten seconds and stream elements $s \in S$, $h \in H$, and $a \in A$ are within ten seconds of each other, then the function should be applied to the triple of readings $[s, h, a]$ to determine if they exceed the dew point threshold.

Our primary observation is that for a broad class of threshold functions we do not need to keep all tuples. Many tuples can simply be omitted without any knowledge of tuples from other streams and without ever missing an alarm.

During normal operation, the memory savings from omitting these tuples is significant, because the factory may have thousands of machines, each with its own sensors. The savings is also important during abnormal operation. For example, suppose the humidity sensor output significantly lags the machines' surface-temperature sensor output. Then the DSMS needs to cache all of the surface temperature readings following the most recent humidity reading until their corresponding humidity readings arrive. This costs memory. Also, when the humidity readings finally arrive, the DSMS's catch-up processing risks a delay in triggering the alarm. Even worse, if the cached surface temperature readings overflow memory, the system might crash and alarms that should have been raised are permanently lost.

If tuples are only needed to trigger the alarm, then our technique can be used to filter them out on an edge device, thereby avoiding the expense of sending them to a datacenter. However, sometimes it may be necessary to capture all tuples in persistent storage for later analysis. In this case, even though all tuples must be read in,

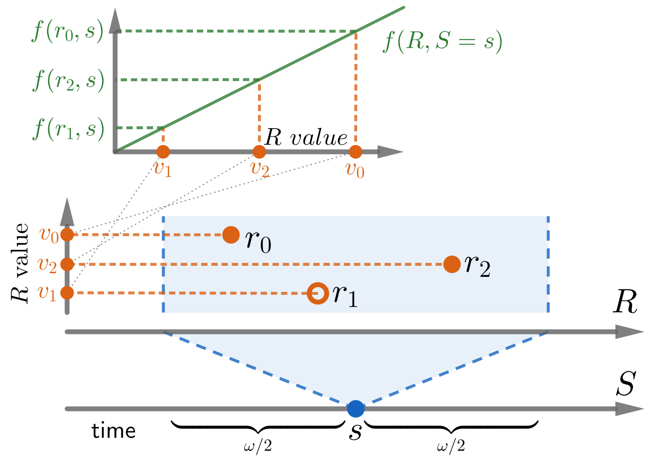


Figure 1: Simple tuple omission depiction.

most of them can be streamed immediately to storage. Only the small number that are needed for the threshold calculation will consume main memory for a non-negligible time period.

To illustrate this intuition, consider a simple case where we have only two streams $R \bowtie S$ and the threshold formula is $f(r, s) = r + s$. Assume tuples $r \in R$ and $s \in S$ join if their timestamps are within a time window of length $\omega/2$ from each other. This is known as an interval join and is explained in detail in Section 2.1. Suppose R sees a sequence of tuples (t, v), where t is a timestamp and v is the value of a sensor reading.

$$r_0 = (t_0, v_0), \quad r_1 = (t_1, v_1), \quad r_2 = (t_2, v_2)$$

Assume $t_0 < t_1 < t_2$, $\{r_0, r_1, r_2\}$ are within ω of each other (i.e., $t_2 - t_0 < \omega$), $v_1 < v_0$, and $v_1 < v_2$. (See the bottom of Figure 1.) Observe that if r_1 joins with a tuple s from stream S, then s must join with at least one of r_0 and r_2. Since r_0 carries a higher value, if $f(v_1, s) = v_1 + s > \mathcal{T}$, then $f(v_0, s) = v_0 + s > v_1 + s > \mathcal{T}$. (See the top of Figure 1.) Similarly for r_2 and $f(v_2, s)$. Since at least one of $r_0 \bowtie s$ and $r_2 \bowtie s$ will be passed to the threshold function f, r_1 is redundant. That is, if we omit r_1, an alarm is still raised and the monitoring system still accomplishes its goal. The remainder of this paper formalizes and generalizes this simple intuition.

Our contributions are as follows

(1) We show that for a large class of threshold functions over the result of a temporal join of two streams, the system needs to retain very few tuples per-data-stream in each time interval yet never miss a alarm.
(2) We prove that our technique is optimal in the sense that it deletes as many tuples as possible.
(3) We generalize our technique to multi-stream joins and show for which join topologies it does and does not work.
(4) We provide experimental evidence that validates the technique's substantial memory savings.

The paper is arranged as follows. Section 2 describes necessary background to understand our omission policy. Section 3 describes our omission algorithm in greater detail as well as its limitations. Section 4 generalizes our omission algorithm to the multi-stream join environment. Section 5 explores the question of automatically checking if a threshold function is amenable to our omission algorithm. Section 6 reports on experiments that evaluate our method against synthetic and real world workloads. Section 7 discusses past work and how it relates to our contribution.

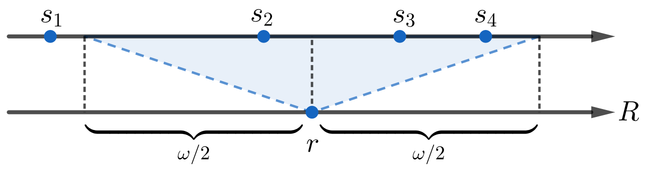


Figure 2: Sliding join between R and S. Tuple r defines time interval $[t_r - \omega/2, t_r + \omega/2]$. Tuples s_2, s_3, s_4 join with r since they live inside this interval. Tuple s_1 falls outside the interval and is therefore not a joining partner.

2 BACKGROUND

In this section, we discuss necessary background to understanding our contributions.

2.1 Temporal Joins

There are two semantic considerations when defining a time-based join between two or more streams. The first is how to assign timestamps to tuples. Using *event time* semantics, the event source assigns a timestamp to each base stream tuple that remains with the tuple as it passes through the DSMS. Using *processing time* semantics, the DSMS assigns a timestamp to each stream tuple when it arrives, that is, when it is "processed". Event time semantics are preferred as event-time timestamps are not susceptible to network failures or latency concerns. The timestamps are also a more accurate representation of the base tuple data. Processing time semantics are generally easier to handle as tuples can never arrive out of order since the clock at the DSMS is the ground truth. In this work we use event time semantics as it is more representative of current systems.

The second consideration is how we define the join predicate. Again there are generally two approaches: *sliding* window and *hopping* window. In sliding window semantics (a.k.a. interval join), tuples r and s join if their event times fall within $\omega/2$ of one another. See Figure 2. In hopping window join semantics, the windows have a specific length ω and move forward in steps of size δ. That is, if the first window is $[0, \omega]$, then the second is $[\delta, \delta + \omega]$, the third is $[2\delta, 2\delta + \omega]$, and so on. Tuples r and s join if there exists a window W such that $t_r, t_s \in W$. Note that tuples may fall within multiple windows if $\delta < \omega$.

Both sliding and hopping semantics define a collection of time windows where tuples join if there exists a window in that collection that contains both. Sliding windows define a superset of the windows defined by hopping windows. In fact, assuming some nontrivial and bounded event time space, hopping defines a finite number of windows whereas sliding defines an infinite number. Thus, while we focus on sliding semantics, it should be clear that our methods are easily applicable to hopping semantics as well. However, the discrete nature of hopping yields more obvious and straightforward omission techniques. For the rest of the paper we use 'interval join' to mean a 'sliding window' temporal join.

2.2 Quasiconvex Functions

We say a function $f : \mathbb{R} \to \mathbb{R}$ is quasiconvex iff for all $x_1, x_2 \in \mathbb{R}$, for all $\lambda \in [0,1]$

$$f\big(\lambda x_1 + (1-\lambda)x_2\big) \leq \max\big\{f(x_1), f(x_2)\big\}.$$

That is, for all pairs of points x_1, x_2, when f is evaluated on a point x in between x_1 and x_2, $f(x)$ cannot exceed both $f(x_1)$ and $f(x_2)$. We provide an example in Figure 3. We can generalize the notion of quasiconvexity to a function over multiple variables, $f : \mathbb{R}^n \to \mathbb{R}$, where values $x_1, x_2 \in \mathbb{R}^n$ are vector valued.

We do not require that a function be globally quasiconvex. We only require it to be quasiconvex with respect to any streaming input on which we apply our omission policy. A multivariate function $f : (x_1, x_2, \ldots, x_n) \to \mathbb{R}$ is quasiconvex with respect to x_1 if for any values $\bar{x}_{i\neq 1}$ the univariate function $g(x_1) = f(x_1, \bar{x}_2, \ldots, \bar{x}_n)$ is quasiconvex.

Many classes of functions are quasiconvex. Examples include convex functions, linear functions, monotonic functions (including step functions, such as floor and ceiling), positive quadratic, logarithmic, exponential, simple inequality boolean triggers, and the application of trained linear or logistic regressors. The question of how to automatically determine if a function is quasiconvex is addressed in Section 5.

3 METHOD

The basic form of our problem is as follows: We are given two streams R and S. Each tuple of R takes the form $r = (t_r, v_r)$ where t_r is the event time and v_r is the value. Similarly, each tuple of S takes the form $s = (t_s, v_s)$. We are also given a function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ where f ingests values taken from an interval join $R \bowtie S$. Our goal is to trigger an alarm whenever there exists some tuples $r \in R$ and $s \in S$ that join and $f(v_r, v_s) > \mathcal{T}$ for a given threshold $\mathcal{T}$. Going forward, we will often abuse notation and write $f(r, s)$ instead of $f(v_r, v_s)$.

Our primary observation is this: if f is quasiconvex w.r.t. S, and there exist tuples $s_1, s_2, s_3 \in S$ each of which joins with some tuple $r \in R$, then we may omit the tuple of S with middle value. That is, if $v_{s_1} \leq v_{s_2} \leq v_{s_3}$ then as far as tuple r is concerned, we may omit s_2. The reason is that quasiconvexity guarantees that if $f(s_2, r) > \mathcal{T}$, then at least one of $f(s_1, r), f(s_3, r)$ also exceeds $\mathcal{T}$ and an alarm will be raised even with s_2 omitted. Stated formally,

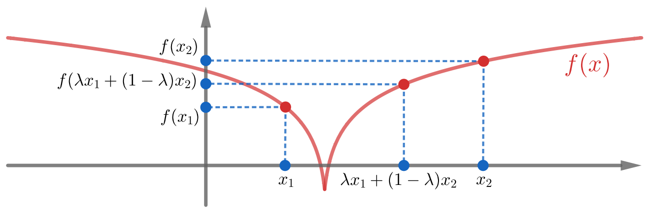


Figure 3: Example quasiconvex function.

if f is quasiconvex w.r.t. S, then

$$f(s_2, r) > \mathcal{T} \implies \Big(f(s_1, r) \geq f(s_2, r)\Big) \vee \Big(f(s_3, r) \geq f(s_2, r)\Big).$$

This observation generalizes to a sufficient condition for omitting tuples from the maintained state of S independent of any specific tuple from R: for all tuples $s \in S$, if there exists a pair of tuples that occurs before and after s, have values *greater* than v_s, and occur within ω of each other, and similarly there exists a pair of tuples that occurs before and after s, have values *less* than v_s, and occur within ω of each other, then we may omit s from S's cache. We call this tuple s *doubly bracketed*. This suggests a policy for reducing the amount of memory by deleting tuples that are doubly bracketed.

This observation can be specialized for monotonic functions. In this scenario, a singly bracketed tuple may safely be omitted. More specifically, if the function is monotonic increasing (decreasing) we may omit any tuple that is bracketed by two tuples with greater (lesser) value.

Is this omission criterion "optimal"? That is, is the ability to omit future tuples unaffected by omitting a tuple s? The answer is yes. Before proving it Section 3.3, we define some useful terminology and present the algorithm for omitting tuples.

3.1 Terminology

Intuitively a tuple s is above bracketed if there exist tuples before and after s whose values are greater than v_s. We formalize this and related concepts in the following definitions.

Definition 3.1 (Above/Below Bracketed)**.** A tuple s is *above* (resp. *below*) *bracketed* iff there exist tuples s_e, s_ℓ s.t.

(1) $t_{s_e} < t_s < t_{s_\ell}$
(2) $v_{s_e}, v_{s_\ell} > v_s$ resp. $(v_{s_e}, v_{s_\ell} < v_s)$
(3) $t_{s_\ell} - t_{s_e} \leq \omega$

(I.e., e and l are shorthands for "earlier" and "later".) Tuples s_e, s_ℓ, are called an *above bracket* (resp. *below bracket*). We refer to a quartet of tuples that form an above and below bracket of the same tuple as simply a *bracket*.

Definition 3.2 (Maximal/Minimal)**.** A tuple s is *maximal* (resp. *minimal*) iff there exists an interval I of length ω containing s where for all tuples $s' \in I$, $v_s \geq v_{s'}$ (resp. $v_s \leq v_{s'}$).

Definition 3.3 (Maximally/Minimally Bracketed)**.** A tuple s is *maximally* (resp. *minimally*) *bracketed* iff there exist tuples s_e, s_ℓ s.t.

(1) $t_{s_e} < t_s < t_{s_\ell}$
(2) $v_{s_e}, v_{s_\ell} > v_s$ (resp. $v_{s_e}, v_{s_\ell} < v_s$)
(3) $t_{s_\ell} - t_{s_e} \leq \omega$
(4) s_ℓ and s_e are maximal (resp. minimal)

An above bracketed tuple cannot be maximal because any window containing s would also have to contain s_e and/or s_ℓ, implying that s is not maximal in that window. Equivalently, any maximal tuple cannot be above bracketed. Similarly, no below bracketed tuple can be minimal.

3.2 Greedy Algorithm

Algorithm 1 presents a greedy approach to omitting tuples using the above reasoning. As tuples arrive from the input stream, they are inserted into a generic tuple store ($\mathcal{TS}$). In most cases, the store will sort the entries on event time. If we regard the $\mathcal{TS}$ as sorted left-to-right with newest (highest timestamp) elements on the right, we can use left and right as synonyms for earlier and later.

We assume $\mathcal{TS}$ provides a generic $\mathcal{TS}$.search(t) function that provides a pointer to the element in $\mathcal{TS}$ with largest timestamp $\leq t$. If $\mathcal{TS}$ allows duplicate timestamps, then we arbitrarily break ties by the stored value and $\mathcal{TS}$.search(t) returns the "first" such tuple.

Each tuple is stored in $\mathcal{TS}$ with four additional attributes: lb_s, la_s, rb_s, ra_s. They denote the time difference between the given tuple s and the chronologically closest known tuple on its left with value below, left with value above, right with value below, and right with value above, respectively. These values are initialized to ∞ at line 5 in the pseudocode.

When a new tuple s arrives, we probe the tree using the tuple's timestamp. Assuming $\mathcal{TS}$ allows duplicate timestamps (i.e., not a red-black tree), we first check any tuples that may have the same timestamp (lines 7 - 16). We then traverse left (lines 19 - 28) and right (lines 31 - 40) searching for above and below bracketing pairs. If s completes a bracket for a tuple in $\mathcal{TS}$, then the newly bracketed tuple is dropped from $\mathcal{TS}$. If the probe reveals a bracket of s that already exists in $\mathcal{TS}$, we omit s.

The pseudocode assumes the use of doubly-closed interval boundary semantics. For doubly-open interval boundary semantics, swap $\leq$ /$\geq$ for $<$ /$>$ (or vice-versa) at lines 19, 31 in Algorithm 1, and line 2 in Algorithm 2. We discuss mixed boundary interval join semantics in Section 3.4.

3.3 Global Optimality of Greedy Algorithm

Algorithm 1 omits a tuple when it finds an above bracketing pair and a below bracketing pair of neighboring tuples. We want to verify that doing so will not harm our chances of omitting other tuples later. First, we will consider above bracketing tuples when deciding whether to omit the tuple. It will become clear that a symmetric argument holds for below bracketing pairs and that the conditions may be considered separately. We simply evaluate the conjunction of both conditions to decide on omission.

Theorem 3.1. Algorithm 1 leads to a globally optimal state.

It is sufficient to prove that if a tuple is above bracketed, then it is also maximally bracketed, because any above bracketed tuple must have an above bracketing pair of tuples both of which are maximal. Since a maximal tuple cannot be above bracketed, it will never be omitted. This implies that the maximal tuples will always be available to above bracket the original tuple in question, and thereby justify its omission.

Lemma 3.2. *A tuple is above bracketed iff it is maximally bracketed.*

The reader may skip the proof to Lemma 3.2 without having it detract from understanding the rest of the paper.

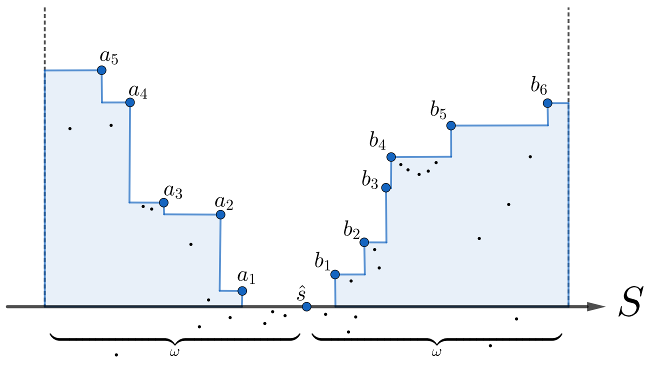
Proof. Assume we are using closed interval semantics. The proof for open interval semantics is nearly identical. In the proof, we will refer to above bracketed tuples simply as bracketed. If two tuples have equal value, we arbitrarily choose the later one to have

Algorithm 1 Maintains a minimal collection of necessary tuples from stream S.

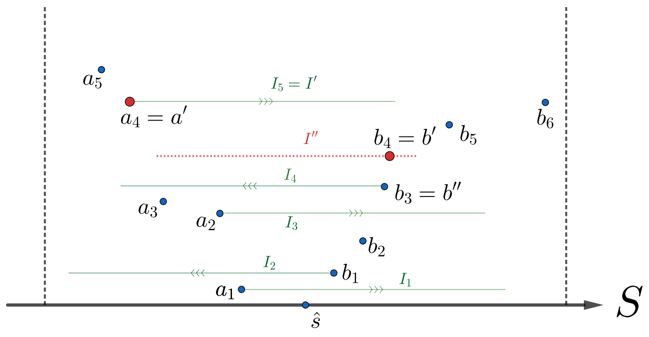
```
 1: procedure GREEDY($S$)
 2:     $\mathcal{TS} \leftarrow$ Store[t, $v, la, lb, ra, rb$]()          ▷ empty tuple store
 3:     while $S$.has_next() do
 4:         $s \leftarrow S$.next()
 5:         $la_s, lb_s, ra_s, rb_s \leftarrow \infty$
 6:         $s' \leftarrow \mathcal{TS}$.search($t_s$)
 7:         while $t_s = t_{s'}$ do
 8:             if $v_{s'} = v_s$ then
 9:                 continue              ▷ $s$ is duplicate of $s'$: omit $s$.
10:             else if $v_s < v_{s'}$ then
11:                 $la_s, ra_s, lb_{s'}, rb_{s'} \leftarrow 0$
12:                 if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
13:             else
14:                 $lb_s, rb_s, la_{s'}, ra_{s'} \leftarrow 0$
15:                 if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
16:             $s' \leftarrow$ succ($s'$)
17:         $s' \leftarrow \mathcal{TS}$.search($t_s$)
18:         if $t_s = t_{s'}$ then $s' \leftarrow$ prev($s'$)
19:         while $(la_s = \infty \lor lb_s = \infty) \land t_s - t_{s'} \leq \omega$ do
20:             if $v_s < v_{s'}$ then
21:                 $la_s \leftarrow \min(la_s, t_s - t_{s'})$
22:                 $rb_{s'} \leftarrow \min(rb_{s'}, t_s - t_{s'})$
23:                 if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
24:             else if $v_s > v_{s'}$ then
25:                 $lb_s \leftarrow \min(lb_s, t_s - t_{s'})$
26:                 $ra_{s'} \leftarrow \min(ra_{s'}, t_s - t_{s'})$
27:                 if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
28:             $s' \leftarrow$ prev($s'$)
29:         $s' \leftarrow \mathcal{TS}$.search($t_s$)
30:         while $t_s = t_{s'}$ do $s' \leftarrow$ succ($s'$)
31:         while $(ra_s = \infty \lor rb_s = \infty) \land t_{s'} - t_s \leq \omega$ do
32:             if $v_s < v_{s'}$ then
33:                 $ra_s \leftarrow \min(ra_s, t_{s'} - t_s)$
34:                 $lb_{s'} \leftarrow \min(lb_{s'}, t_{s'} - t_s)$
35:                 if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
36:             else if $v_s > v_{s'}$ then
37:                 $rb_s \leftarrow \min(rb_s, t_{s'} - t_s)$
38:                 $la_{s'} \leftarrow \min(la_{s'}, t_{s'} - t_s)$
39:                 if Bracketed($s'$) then $\mathcal{TS}$.remove($s'$)
40:             $s' \leftarrow$ succ($s'$)
41:         if $\neg$Bracketed($s$) then
42:             $\mathcal{TS}$.insert($t_s, v_s, la_s, lb_s, ra_s, rb_s$)
```

Algorithm 2 Checks if tuples s is bracketed

```
1: procedure BRACKETED($s$)
2:     if $la_s + ra_s \leq \omega \land lb_s + rb_s \leq \omega$ then
3:         return True
4:     else
5:         return False
```



(a) $\{a_i\}$ **and** $\{b_j\}$ **sequences.**



(b) $\{I_k\}$ **sequence.**

Figure 4: Visualization of constructive proof.

the "higher" value. Trivially, maximally bracketed implies bracketed. It remains to prove that bracketed implies maximally bracketed. The proof is constructive. Suppose there exists a bracketed tuple $\hat{s}$ with bracketing pair s_1, s_2. Define two sequences of tuples $a_0, a_1, a_2, \ldots$, and $b_0, b_1, b_2, \ldots$ that extend backwards and forwards in time from $\hat{s}$. Let $a_0 = b_0 = \hat{s}$. For $i > 0$ let a_i be recursively defined as the latest element earlier than a_{i-1}, and later than $t_{\hat{s}} - \omega$, with value greater than a_{i-1}. More formally, for $i > 0$:

$$a_i = \underset{s \in S}{\arg\max} \left\{ t_s | t_{\hat{s}} - \omega \leq t_s < t_{a_{i-1}}, v_s > v_{a_{i-1}} \right\}$$

Similarly for b_j where $j > 0$:

$$b_j = \underset{s \in S}{\arg\min} \left\{ t_s | t_{b_{j-1}} < t_s \leq t_{\hat{s}} + \omega, v_s > v_{b_{j-1}} \right\}$$

A visualization of these sequences is in Figure 4a. Other tuples may appear in the time-line but not all tuples end up as elements of $\{a_i\}$ or $\{b_j\}$. Although the bracketing pair s_1, s_2 might not actually be elements of the $\{a_i\}, \{b_j\}$ sequences, the existence of s_1, s_2 guarantees that the sequences are nonempty and $t_{b_1} - t_{a_1} \leq \omega$.

We construct a sequence of intervals and associated heights starting with whichever of a_1 or b_1 has a lesser value. WLOG assume it is a_1. The first interval under consideration is $I_1 = [t_{a_1}, t_{a_1} + \omega]$ with initial height v_{a_1}. Intervals (and heights) are recursively defined as follows: If there exist values from the opposite tuple sequence that fall inside the interval I_k, and whose value exceeds h_k, then we select the closest (with respect to time) such tuple as the new starting point for I_{k+1} and set the new height h_{k+1} as the value of said tuple. If the opposite tuple sequence is $\{a_i\}$, we choose the latest such a_i. If the opposite tuple sequence is $\{b_j\}$, we choose the earliest such b_j. By construction of the $\{a_i\}, \{b_j\}$, the tuple that is time-wise closest value to $\hat{s}$ on the opposite side of $\hat{s}$ must be a member of either $\{a_i\}$ or $\{b_j\}$. See Figure 4b.

Note that h_k is a strictly increasing sequence of values with upper bound $\max(\max_i(a_i), \max_j(b_j))$. Thus there must exist some final interval I' (with height h') where there does not exist tuples on the opposite side of $\hat{s}$ that exceed h'. WLOG assume I' is anchored on an element a' of $\{a_i\}$. In Figure 4b, this interval is I_5 with anchor $a' = a_4$ and height $h' = v_{a_4}$. By construction of the $\{I_k\}$ sequence, the interval contains $\hat{s}$ and must contain at least one element b' of $\{b_j\}$. This is because the previous interval was anchored at a point b where a' is within ω of b. (In Figure 4b, this tuple is $b' = b_4$.) By construction, a' and b' form a bracketing pair on $\hat{s}$. Furthermore, a' is maximal on the interval I'. It remains to prove that b' is also maximal. Consider the interval $I'' = [t_{a'} + \delta, t_{a'} + \omega + \delta]$ where we choose δ sufficiently small that no tuples appear in the intervals $(t_{a'}, t_{a'} + \delta]$ and $(t_{a'} + \omega, t_{a'} + \omega + \delta]$. This is only guaranteed possible when using doubly-closed, or doubly-open join interval boundary semantics. When using mixed interval boundaries (left-closed-right-open, left-open-right-closed), the existence of such a δ is not guaranteed. Section 3.4 presents a counterexample to Theorem 3.1 when using mixed boundaries along with further discussion.

We wish to show that b' is maximal in I''. By construction, there cannot exist any tuples to the right of $\hat{s}$ whose value is greater than $v_{b'}$. Assume towards contradiction that there exists some a'' that lies inside I'' and whose value exceeds $v_{b'}$. This would imply that a'' lives in the most recent interval anchored at a tuple from $\{b_j\}$. Call this tuple b''. We have $v_{a''} > v_{b'} \implies v_{a''} > v_{b''}$. We know such an interval exists in $\{I_k\}$ since we began anchoring at the lower of a_1 and b_1 and hence at least two intervals (at least one anchored from both $\{a_i\}$ and $\{b_j\}$) are in the sequence $\{I_k\}$. In Figure 4b, this tuple is $b'' = b_3$. This is a contradiction because it would imply that a'' should have been chosen to anchor I' instead of a'. Recall the anchors for the $\{I_k\}$ based on *closest* (w.r.t. time) higher value in the opposite sequence, not simply the highest. Therefore, no such a'' can exist. In more detail, no element to the left of $\hat{s}$ in the interval I'' may exceed $v_{b'}$ and thus b' is maximal in I''. Hence a' and b' are both maximal in some interval and form a maximal bracket on $\hat{s}$. □

3.4 Mixed Boundary Interval Semantics

We present a counter example to Theorem 3.1 when using mixed boundary interval semantics. Let the join interval be over a left-closed-right-open interval with $\omega = 3$. Our workload consists of 5 tuples (see Figure 5):

$$s_0 = (0, 3), s_1 = (1, 1), s_2 = (2, 0), s_3 = (3, 2), s_4 = (4, 4)$$

We wish to justify the omission of s_2. The interval $I = [0.5, 3.5)$ and pair s_1 and s_3 imply s_2 is above bracketed. It remains to show that s_2 is not maximally bracketed. Clearly, with value 1, tuple s_1 is not maximal: any interval containing s_1 must contain either s_0 or s_3. Therefore any maximal bracketing of s_2 must involve s_0.

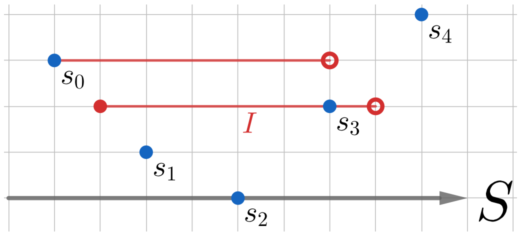


Figure 5: Mixed boundary counter example.

However, no interval exists that contains both s_0 and the closest tuple right of s_2, namely s_3. Therefore no maximal bracket exists and the statement of Theorem 3.1 fails. This workload also yields a counter example for left-open-right-closed interval join semantics.

We emphasize that it is still "safe" to deploy our omission policy here. It just lacks a guarantee of global optimality. However, the practitioner should be confident that the retained tuples are extremely close to optimal, although not easily provably so. We may simply pretend the query is operating under doubly-open interval boundary semantics and omit tuples based on this assumption. For example, if the semantics are left-closed-right-open with interval length ω, we execute our omission strategy based on left-open-right-open interval of length ω. This might lead to slightly more tuples being retained than necessary, but the system will not suffer any false negatives while still omitting many tuples.

3.5 Further Discussion

We wish to highlight a few additional aspects of our greedy approach. First, our algorithm is robust to out-of-order streams. This is true both in terms of delay differences between two streams and out-of-order behavior within the same stream. In the different streams scenario two tuples r, s from different streams arrive out of order. That is, we have $t_r < t_s$ but s arrives ahead of r due to differences in stream latency. It is less obvious how tuples from the same stream may arrive out of order, but it is certainly possible, e.g., if a single logical stream is the union of multiple physical streams.

Second, our omission algorithm may be pushed down to the emitter. This saves network bandwidth, not just memory usage. Furthermore, in the event of a network failure the memory required to cache results before delivering them to the central processing node would be significantly reduced. This extends the state savings benefits across the entire system.

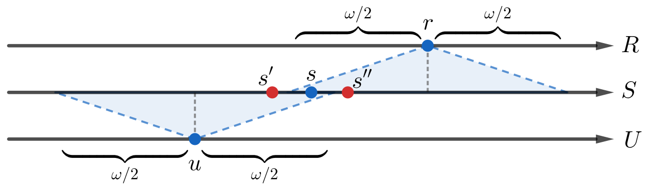
Finally, while we focus on joins, our omission policy may be applied in a setting where a single stream provides input parameters to the function. However, if the threshold function is quasiconvex, the calculation of the function is likely to be relatively simple, in which case it is efficient simply to execute the function at the edge node and skip any kind of state management. Still, if the execution of the function is an external service or otherwise "expensive", then our method may again be useful.

4 GENERALIZATION TO MULTIJOINS

This section generalizes our strategy to multijoin settings. In the two stream scenario, Algorithm 1 may be applied to a single stream independent of the join partner and relying only on knowledge of

```
SELECT f(R.v, S.v, U.v) FROM R, S, U
WHERE  |R.t - S.t| <= 1 AND |U.t - S.t| <= 1;
```

(a) Multistream query SQL.



(b) Multistream interval intersection.

Figure 6: Multistream Query.

the interval size ω. This is because the options for join predicates are highly limited. With more streams the choice of join topology is more complex and the omission policy must applied more carefully.

Consider the chain joins defined by the SQL query in Figure 6a. Streams R and U directly join to S but not to each other. This relationship is depicted in Figure 6b. It is not safe to omit a bracketed tuple from the state of stream S using Algorithm 1. For example, suppose tuple s is bracketed by s' and s'' and thus omitted. Tuple s' fails to join with r while tuple s'' fails to join with u. The omission of s has led to no joined output tuples being delivered to the threshold function.

However, it is still possible to apply the omission policy to tuples from streams R and U. This is because they do not have to worry about the intersections of multiple intervals.

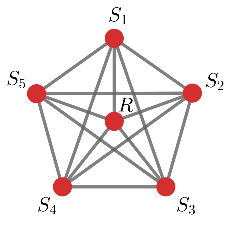
In general, we may omit tuples from any *peripheral* streams in the join topology, that is, any stream that is temporally joined directly with only one other stream. Conversely, a stream is *internal* if it is temporally joined to more than one other stream.

This suggests that join topologies heavily influence the applicability of our omission policy. For instance, a chain join only allows tuples from the two end streams to be omitted while all internal streams must be cached in full (see Figure 7b). However, such a chain join in the context of streams seems unlikely, since the timestamps from contributing base tuples might stretch unnaturally over the event-time space. An alternative is a star join topology where one stream is chosen to be the internal center of the star while all other streams are peripheral (see Figure 7c). The star query allows the application of our omission policy on all but the central stream, potentially a major savings.

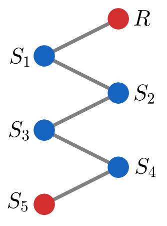
Arguably the most natural join topology is where every pair of tuples contributing to a join output must be within ω of one another[1], that is, a clique query (see Figure 7a). One might expect a clique to be the worst case join topology for our omission policy, since there are no peripheral streams. Surprisingly however, in a clique query our omission algorithm may be applied to every stream!

We define the neighborhood of a stream S in a join topology to be set of all of streams with which S joins directly. For any query topology, our omission policy may be applied to every stream whose

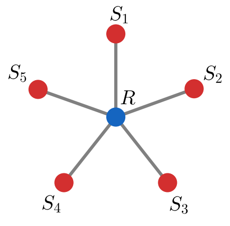
[1]We switch from pairwise distance $\omega/2$ to ω in the multijoin scenario. The two stream scenario is a special case where we are allowed to omit tuples using intervals whose length is twice the pairwise distance bound.



(a) Clique.



(b) Chain.



(c) Star.

Figure 7: Applicability of our omission policy across different join topologies. Streams highlighted in red may omit tuples safely.

neighborhood is a clique. That is, for any stream S, for any pair of distinct streams S', S'' where S joins with S' and S'' directly, then S' and S'' must join directly as well in order to omit tuples from S. We formalize this statement in the following Theorem and Corollary.

Theorem 4.1. Algorithm 1 may be applied safely to every stream whose neighborhood in the join topology is a clique.

Corollary 4.1.1. Algorithm 1 may be applied safely to every stream in a clique query.

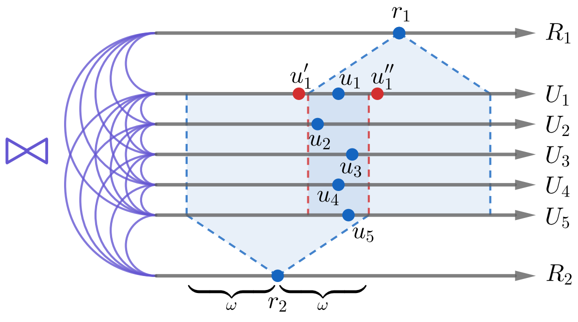
In order to prove this we start with a short lemma.

Lemma 4.2. *Consider a clique query $Q = (\bowtie_i S_i)$, where all joins are interval joins over an interval of length ω. For every output $(\bowtie_i s_i)$ of Q, there is an interval I of length ω that contains the timestamps of all tuples in $(\bowtie_i s_i)$.*

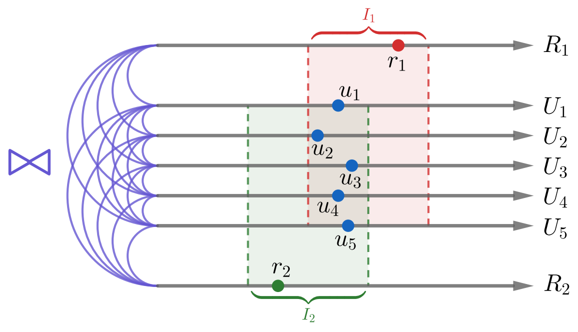
Proof. Let t_i be the timestamp of s_i for all i. Let $t_{\min} = \min_i t_i$, the minimum over all contributing tuples' timestamps, and similarly, $t_{\max} = \max_i t_i$. By definition of the clique, $t_{\max} - t_{\min} < \omega$ and $t_{\min} \leq t_i \leq t_{\max}$ for all i. Thus, $[t_{\min}, t_{\max}]$ satisfies the definition of I. □

We now prove Theorem 4.1.

Proof. We will show that the omission of one tuple cannot lead to a false negative and generalize inductively. Consider a query $(\bowtie_i S_i) \bowtie (\bowtie_j R_j)$ where subquery $(\bowtie_i S_i)$ is a clique (the R_j need not form a clique). Let the neighborhood of stream S_1 be the clique $(\bowtie_i S_i)$. Thus, S_1 does not join directly with any R_j. Pick an arbitrary join output $(\bowtie_i s_i) \bowtie (\bowtie_j r_j)$. For each i, let t_i be the timestamp of tuple s_i. By Lemma 4.2, there exists an interval I of length ω where $t_i \in I$ for all i. Assume there exists a bracket on s_1. For both the above and below bracketing pair, at least one tuple from the pair falls inside I. Therefore at least one of the four bracketing tuples (call it s_1') may *replace* s_1 and $(s_1' \bowtie (\bowtie_{i \neq 1} s_i)) \bowtie (\bowtie_j r_j)$ will still raise the alarm. Moreover, we know that no join predicate between pairs of streams other than S_1 has been violated since all other base tuples have been held constant. This logic may be applied inductively to any chain of omissions and subsequent replacements within the same stream and agnostic of omissions and replacements in other streams. Therefore, we need not worry about a replacement tuple being replaced itself, so this will not cause false negatives and the omission is safe. □



(a) Omitting u_1 may lead to false negative.



(b) Omitting r_1 or r_2 is safe.

Figure 8: Near-clique join missing predicate $(R1, R2)$.

In a clique query the neighborhood of each stream is the entire query. Thus Corollary 4.1.1 follows trivially. In Section 4.1 we discuss a scenario where a single join predicate is dropped from a clique. Furthermore, note that the neighborhood of any peripheral stream is automatically a clique subquery as the clique of size 2 only has one edge. The only difference is that cliques of size 2 allow us to use intervals of double length in our omission policy (See Footnote 1).

4.1 Near Clique

Consider the near clique query in Figure 8. Except for streams R_1 and R_2, all pairs of streams share a join predicate. Figure 8a shows how a false negative can occur when omitting u_1. The join tuple $x = r_1 \bowtie u_1 \bowtie \cdots \bowtie u_5 \bowtie r_2$ satisfies all join predicates and thus should be passed to the threshold function. Assume x would trigger the alarm. Now suppose u_1 is bracketed by tuples u_1' and u_1'' and hence omitted. Observe that u_1' joins with r_2 but not with r_1. Conversely, u_1'' joins with r_1 but not with r_2. Thus, in this scenario the omission of u_1 due to the bracket u_1', u_1'' can lead to a false negative.

On the other hand, omitting r_1 is safe, because the neighborhood of stream R_1 is $\{R_1, U_1, \ldots, U_5\}$, which forms a clique. As depicted in Figure 8b any bracket on r_1 will necessarily leave at least one replacement in the interval I_1. Thus we may apply our omission policy to R_1. A similar argument holds for tuple r_2.

4.2 Multiquery Workloads

The consideration of mulitijoin streaming queries begs the question of how to apply our omission policy in a multiquery setting. In this scenario, the central processing node is executing multiple queries and a stream may be an input to some or all of them.

Suppose each query is amenable to our omission policy individually. That is, it includes a threshold function applied to the join of some stream inputs. Consider some stream S. We will explain how each query that takes S as input enforces a minimal interval size restriction on S. Recall that the set of tuples that are maintained for interval size ω is the set of all tuples for which there exists some interval I of length ω in which the tuple is minimal/maximal. I.e. the set of all minimal and/or maximal tuples. Let this subset of tuples be S_{ω}. For any tuple, if there exists an interval I' of length $\omega' > \omega$ in which the tuple is minimal/maximal, then there must exist some interval $I \subset I'$ which has length ω and in which the tuple is still minimal/maximal. Therefore, the set of tuples that are maintained for interval size ω is a superset of those tuples that are maintained for any larger interval size $\omega' > \omega$. More formally, $\omega' > \omega \implies S_{\omega'} \subseteq S_{\omega}$. Thus, the shorter the interval size, the more tuples we have to maintain to guarantee no false negatives. This implies that if we have two queries both operating on some stream S, it suffices to maintain the necessary tuples to whichever query is applying the smaller interval size, i.e. the more restrictive query. By the argument above, all necessary tuples for the less restrictive query are maintained as a byproduct of maintaining tuples for the more restrictive query.

If the query is a simple join of two streams and computes a quasiconvex function with respect to both inputs, then both streams must maintain min and max values for any interval of length 2ω where ω is the pairwise distance bound to guarantee no false negatives. If the query is a clique join of three or more streams, then all streams must maintain min and max values for any interval of length ω (see footnote 1). If the query is a non-clique multijoin, then any stream whose neighborhood is a clique must again only maintain min and max values for any interval of length ω, where ω might differ from one query to the next. Finally, for any stream S, if there is a query Q whose threshold function is not quasiconvex with respect to S or if S's neighborhood in Q's join graph is not a clique, then we will need to maintain every tuple from S. That is, the query enforces a length 0 interval size restriction.

For each stream, it suffices to maintain a subset of tuples corresponding to the narrowest interval size restriction. This is disappointing since the existence of less restrictive queries in the workload will not help us to reduce overall maintained state; the DSMS will still need to maintain the necessary tuples for the most restrictive query. However, this is also a "best case scenario" as maintaining the bare minimum amount of state to satisfy the most restrictive query will be sufficient for all other queries and would be required to evaluate that strict query anyway. That is, the less restrictive queries do not incur any extra work beyond what is required for the most restrictive query.

5 CERTIFYING QUASICONVEXITY

Ensuring a function is linear or monotonic step is simple but requires a restrictive user interface. On the other hand, general verification of quasiconvexity is difficult. Simple functions may be rewritten in innumerable complex but still equivalent ways. Even normal convexity is difficult to verify since there is no generic base formulation with which all convex functions may be written. In fact, checking the convexity of just polynomials is NP-hard [6].

One imprecise solution is to evaluate the function at every point in a discretization of the domain and check to see if quasiconvexity holds for those points. For the motivating example, this involves evaluating Equation 1 for each point (s, h, a) in some discretization of $[-273.15, T] \times [0, 1] \times [-273.15, T]$ where we choose T to be a reasonable upper bound on a temperature reading.

A further refinement is automatically detecting if the threshold function is a (positive) linear combination of smaller *sub-functions*. Each sub-function may then be certified against the cross product of only those streams that appear in that sub-function. For the motivating example, the sub-functions are:

$$\phi_1 = -s, \quad \phi_2 = \ln\left(\frac{h}{100}\right), \quad \phi_3 = \frac{18.678a}{257.14 + a}$$

This technique may be used to cut down on the exponential size of the discretization: the exponent of the complexity of the evaluation drops from the number of stream inputs to the number of streams in the sub-function with the most stream inputs.

Even if one sub-function fails, our omission policy can still be applied to some streams. It only discounts streams that were inputs to the failed sub-function. For example, if ϕ_1 fails, but ϕ_2 and ϕ_3 succeeds, then we can still safely omit tuples from H and A.

If the discretization is not sufficiently fine, then this test might miss unsafe functions. In this scenario, small dips/spikes might appear between discretization steps creating the illusion of quasiconvex behavior. However, the expense of using a finer discretization can be affordable, since this analysis need only be run once to test the applicability of a function.

6 EVALUATION

In this section, we demonstrate the effectiveness of our omission policy empirically. We first briefly describe our implementation, then compare different tuple stores, and finally investigate raw tuple retention.

6.1 Implementation Details

Many DSMS's assume that the data is timestamp-ordered after ingestion [14, 17]. In these cases, the first step in a query plan is a stateful *incremental sort* operator that sorts the input during ingestion [18]. With this in mind, we prototyped our omission policy as an online sorting algorithm that also omits bracketed and thus unnecessary tuples. We call it a *threshold sorter*. This design allows us to plug the component easily into an existing DSMS. The simplicity of the bracketing condition implies that the threshold sorter requires only a few hundred lines of code to implement, not including the skip-list implementation [25].

6.2 Data Sets

We evaluate our method against both synthetic and real world data. The synthetic data streams are comprised of (timestamp, value) pairs where the timestamps are chosen uniformly at random from time range $[0, T)$. The pairs are sorted into chronological order. Values are drawn iid from a uniform distribution over some range. We refer to such a dataset as "synth-uniform" or simply s-unif. Alternatively, we may choose values non-independently. We simulate samples from a Wiener Process where the time discretization corresponds to the already chosen random timestamps. We refer to such a dataset as "synth-wiener" or simply s-wiener.

Disordered behavior may be introduced to a stream by adding random noise to the timestamp to simulate disparity between event time and processing time. We use Gaussian noise and vary the standard deviation parameter to increase or decrease the degree of disorder. The degree of disorder can be measured using the "inversion" rate measure: the fraction of all pairs of tuples in the stream that are disordered (i.e., *inverted*) [27]. Formally, given some sequence of values $x_0, x_1, \ldots, x_n$, the inversion rate of the sequence is defined as:

$$\frac{|\{i < j : x_i > x_j\}|}{|\{i < j\}|}$$

6.3 Tuple Store Comparison

We compare the performance of Algorithm 1's greedy tuple retention policy using three different tuple store data structures: linked list (LL), skip-list (SL)[28], and red-black trees (RB)[13]. All data structures sort on timestamp with tuple values used to break ties.

LL offers worst-case linear search/insertion/deletion. However, if the stream is in-order, then the algorithm needs to follow only a single pointer to the tail of the linked list. By contrast, SL and RB both offer logarithmic expected worst case complexity for search/insertion/deletion.

Like LL, SL offers convenient lateral traversal over neighbors. While RB also offers lateral traversal, there is potentially "wasted work" in that some nodes in the tree represent tuples that are not within the interval ω of the new tuple's timestamp but need to be traversed to reach tuples that are within the interval. (See the loops starting at lines 19 and 31 in Algorithm 1.) Moreover, RB does not support duplicate timestamps. Thus, the timestamps in synthetic streams are drawn without replacement from the timestamp range. In summary, we should expect SL to outperform LL and RB in the presence of disordered tuples and underperform LL slightly if the stream is perfectly in order.

The following experiments measure the time needed to process each dataset without any data arrival latency. That is, each implementation ingests the streams sequentially and without any waiting between sucessive tuples. We report the total time spent on the stream. In each experiment the stream consists of 10^6 tuples with timestamps drawn from the range $[0, 10^7)$ (approximately 1 tuple every 10 time units). The interval size is kept at a constant 100 units for all experiments. With these parameters, approximately 40% of all tuples are preserved after a full pass. Each experiment yields the average runtime of 5 trials. Table 1 contains results and empirical inversion rates.

Stream	sigma	inv rate	LL	SL	RB
s-unif	0 (in order)	$< 10^{-6}$	0.727	1.184	4.565
s-wiener	0 (in order)	$< 10^{-6}$	0.753	1.247	4.824
s-unif	1	$< 10^{-6}$	0.754	1.243	4.620
s-wiener	1	$< 10^{-6}$	0.747	1.200	4.754
s-unif	10	$< 10^{-6}$	0.768	1.232	4.821
s-wiener	10	$< 10^{-6}$	0.787	1.254	5.058
s-unif	100	$1.2 \cdot 10^{-5}$	0.793	1.277	4.634
s-wiener	100	$1.2 \cdot 10^{-5}$	0.853	1.344	4.984
s-unif	1000	0.000111	1.031	1.373	4.801
s-wiener	1000	0.000102	1.113	1.440	4.861
s-unif	10000	0.001078	5.103	1.645	4.910
s-wiener	10000	0.001181	5.279	1.666	4.994
s-unif	100000	0.011324	84.341	2.594	6.036
s-wiener	100000	0.011216	80.047	2.522	6.131

Table 1: Stream processing times in seconds for linked-list (LL), skip-list (SL), and red-black tree (RB) tuple store implementations on synthetic datasets with varying Gaussian noise standard deviation (sigma). Stream size 10^6, timestamp range $[0, 10^7 - 1)$, interval length 100.

While LL performs best for in-order streams, as the degree of disorder increases LL performance suffers a sharp decline. In contrast, while SL lags behind LL due to higher bookkeeping costs of the underlying tuple-storage data structure, the logarithmic probe complexity leads to graceful performance degradation as the stream becomes increasingly disordered. RB follows a similar performance trend as SL with respect to degree of disorder in the stream but consistently lags behind SL because it traverses more nodes than is strictly necessary. Thus, if the stream is guaranteed to be in order, LL is the obvious simple choice. However, if the stream might be disordered, SL is the more resilient data store.

Another consideration is trimming data. Often streaming query engines will drop data that is no longer relevant. This is the case in normal operation when failures and network partitions are not experienced. The engine assumes that no tuple will arrive with a timestamp earlier than some punctuation. While SL and RB will both find that cutoff point in logarithmic time, RB will have to re-balance the tree after trimming. LL will again take linear time to find the cutoff point. Both SL and LL can simply drop all tuples left of the punctuation, a constant time operation. In this scenario SL has a clear theoretical advantage over LL and RB.

6.4 Tuple Retention Savings

While the ability to deploy the algorithm is dependent on the join topology and threshold function, the execution is entirely agnostic of external streams. Thus to demonstrate how effectively our omission policy reduces the state, it suffices to demonstrate it on a single stream. We evaluate the effectiveness of our approach on multiple streams as described below.

- **DEBS 2012 Grand Challenge** (DEBS) is a dataset consisting of monitoring data from manufacturing equipment [24]. We demonstrate the effectiveness of our omission policy by pairing the given timestamps with the value of column `mf01`:

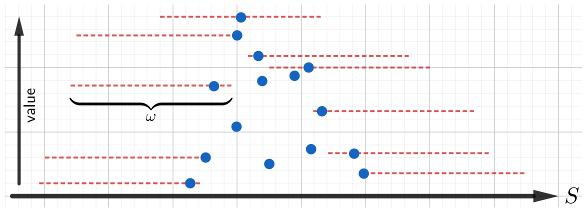


Figure 9: Retained tuples in the presence of significant time gaps. Each retained tuple is paired with an interval in which that tuple is maximal thus forcing the inclusion of the tuple. Of the 15 total depicted tuples, we are forced to retain 10.

the electrical power main phase sensor reading. The stream consists of just over 32 million tuples.

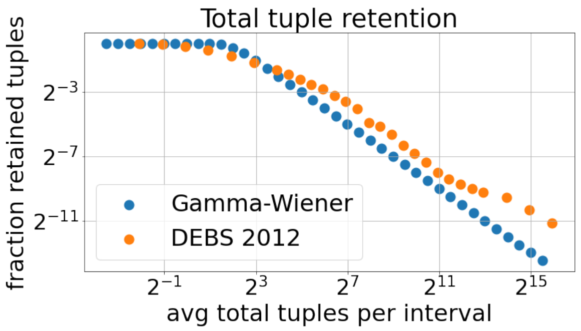
- **Gamma-Wiener** is a synthetic dataset consisting of 10 million tuples. Timestamps start at 0 with each subsequent timestamp adding an independent draw from a Gamma distribution. Values are taken from a Wiener process with the discretization chosen with respect to the timestamps.

For each stream, we assume the value is an input parameter to a quasiconvex threshold function. Recall that the specific nature of the stream or any joining stream is irrelevant given that the threshold function is quasiconvex. We wish to demonstrate that the performance of our omission policy depends mostly on the number of tuples that appear inside any given interval. As we vary interval length, we expect the proportion of omitted tuples to vary approximately linearly with respect to the average number of tuples per interval.

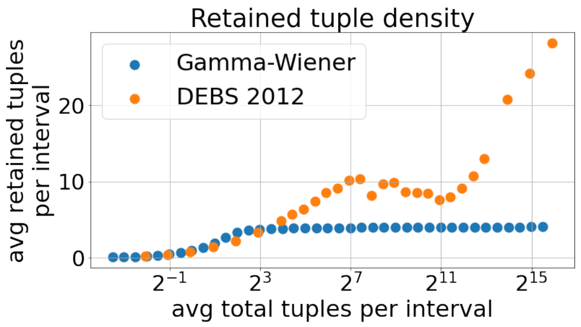
Longer intervals are semantically more inclusive: one would expect the true join output to grow exponentially with respect to the interval length, where the exponent is the number of streams being joined. However, longer intervals also lead to more base tuples being omitted as the range of tuples that may comprise a bracket grows. In the following experiments we omit tuples if they are both above and below bracketed. See Figure 10 for results.

Results for Gamma-Wiener are noticeably uniform, because large time gaps and longer sustained shifts in the value parameter are less likely. For very narrow intervals, the true number of tuples per interval is exceedingly small. Hence the fraction of retained tuples over total tuples is very high (lefthand side of Figure 10a), while the average number of tuples that are retained per interval is very low: there just are not enough tuples per interval. However, once the total tuples per interval grows to about four, the fraction of retained tuples drops dramatically. This trend is most easily understood in Figure 10b where the average number of retained tuples per interval for Gamma-Wiener flat-lines at four. Given that we omit a tuple when it is above and below bracketed, this average of four tuples per interval is not surprising. Each omitted tuple requires one above and one below bracket. If one draws a loose connection between an omitted tuple and an interval it represents, the four points in the above and below bracket are the four retained tuples per interval.

Results are less uniform for DEBS. Unsurprisingly, when intervals are exceedingly narrow and on average few tuples appear for any given interval, the fraction of total retained tuples stays close to one as there are rarely enough close neighbors to form a bracket.



(a) Total tuple retention.



(b) Average per interval tuple retention.

Figure 10: Tuple retention varying interval length.

Notice that DEBS begins omitting tuples at a lower average tuple density than Gamma-Wiener. This is because the timestamp distribution for DEBS is highly skewed with large gaps between clusters of timestamps.

Clustering tuples increases the probability of forming brackets and omitting tuples. While this timestamp skew helps at lower density, it actually does the opposite at higher average tuple densities. Observe in Figure 10b that DEBS seems to level off around 8 retained tuples per interval. This is partially because the large timestamp gaps create boundary effects that force us to retain disproportionate numbers of tuples just before and just after these gaps. Simply put, the lack of tuples in a time gap often eliminates the ability to construct brackets forcing us to retain more tuples. Thus, disproportionately many tuples are retained near the boundaries of time gaps. An example scenario is in Figure 9.

As we increase interval length, often these gaps may be bridged and the time gap boundary retained tuples decrease proportional to the total tuples per interval. This is characterized as the slight downward trend in average retained tuples per interval for DEBS between 2^7 and 2^{11} tuples per interval. However, as intervals grow but still fail to bridge larger gaps, many time gap boundary retained tuples remain. Thus the number retained tuples per interval also increases proportionately. This explains the spike in average retained tuples per interval for DEBS on the right hand side of Figure 10b. However, at this point the average tuples per interval is very large making this scenario unlikely in practice.

7 RELATED WORK

Although we are unaware of any existing work on our specific problem of threshold queries over streaming joins, we have found similar scenarios in the literature where our proposed techniques may have potential applications.

7.1 Load Shedding

One closely related area of research is load shedding, where tuples are dropped (sometimes at random) to prevent overloading and hence increased latency [12, 19, 34]. One can view our omission policy as a theoretically optimal form of load shedding.

Dropping tuples randomly leads to an obvious trade-off in query answer degradation. Past work has often focused on how to implement this degradation gracefully, generally in response to "bursty" stream behavior overwhelming the system. In contrast, our omission policy does not depend heavily on the tuple input rate except in the corner case where there exist large time gaps. In this sense, our method is robust to these bursty scenarios; we never need to tweak any sampling rate.

Our omission policy and traditional load shedding are not mutually exclusive. A simple method of combining the two is a simple two phase omission policy. If the system is still overwhelmed after omitting bracketed tuples, tuples may be dropped randomly to improve performance further. This hybrid approach is a possible topic of future work.

7.2 Local Geometric Constraint Thresholding

Another line of related research focuses on multiparameter threshold functions over distributed data [29]. Unlike our work, it preaggregates the data with respect to each object and compute node and does not explicitly join on timestamps. For instance, vector $x_{j,i}$ corresponds to some subset of the data corresponding to object j and living at node i. The full vector describing object j is The full vector describing object j is $x_j = \sum_i x_{j,i}$. The threshold function is applied to x_j. In order to avoid a full distributed aggregation to find objects that trigger this threshold the authors describe an algorithm for generating bounds local to each node using geometric constraints of the parameter vector space. These local geometric constraints were first highlighted by Sharfman et al. [30, 31]. They generalize this method from exclusively monotonic to functions that may be expressed as the difference of monotonic functions—a class of functions that subsumes the set of globally quasiconvex functions. Giatrakos et al. expand on this local geometric constraint theme by incorporating predictors to further reduce the number of system synchronizations that need to be performed [20]. Their approach relies on individual nodes reliably predicting the "drift" in vector values in neighboring nodes since the most recent synchronization. This introduces a trade-off space between system load and query answer quality similar to many load shedding algorithms.

7.3 Analytic Functions over Data Streams

Most commercial streaming and time-series database systems now support *analytic functions* over data streams or time series. For example, TimescaleDB [5], a time-series database built on top of PostgreSQL, offers simple analytic functions such as `first()`, `last()`, and so on. The recently launched Amazon Timestream [1], a serverless time-series database hosted on Amazon AWS, supports more advanced functions such as computing the cosine similarity of two vectors. The query language of InfluxDB [4] provides an even richer set of functions for time-series analysis, such as `holt_winters()` [3]. Although the query language reference manuals of these systems do not provide specific examples, it is straightforward to write queries that apply thresholds on top of such analytic functions.

7.4 Time-Series Analysis

Thresholding has been a common technique in time-series analysis for various applications. For example, *threshold models*, a popular class of nonlinear models in time-series analysis, have been around for decades [35]. The basic idea is to use different models for different parts of a time series that are above or below a threshold. Another example is *threshold-based data mining* [9, 10], which uses threshold queries for similarity search over time-series data. The idea is to truncate time series using a threshold, and then measure similarity between (two) time series using a distance function that only considers the intervals above the threshold. One may express such distance computation using timestamp-based, streaming joins.

7.5 Streaming Joins

Joins over two or more streams have become increasingly popular in real-world applications. As a result, stream processing systems, such as Apache Spark Structured Streaming [8], Apache Flink [14], and Microsoft's Azure Stream Analytics [2], recently started supporting streaming joins. One common technique implemented by existing systems is the *symmetric hash join* algorithm (and an analogous symmetric nested-loop join algorithm) [21, 22, 26, 38], which has to buffer join states in main memory. This raises challenges in applications where the size of join states can be extremely large. Specialized systems, such as Google's Photon [7], have been built to deal with such cases. Our technique provides another novel perspective on reducing the amount of join state to be kept.

7.6 Stream Query Optimization

There has also been quite a bit of work on the query optimization side of stream join processing. For example, Viglas and Naughton proposed cost models for both nested-loop join and symmetric hash join in the streaming context with the goal of maximizing the query output rate, which can be easily integrated into classic query optimization frameworks such as ones that are based on dynamic programming [37]. Ayad and Naughton [11] further proposed a query optimization framework for conjunctive queries over data streams that considers resource constraints. There are various other optimization techniques for stream query processing in general, such as operator separation, fusion, and reordering (see [23] for a survey). It would be interesting future work to consider the interaction between our technique and existing query optimization techniques. For instance, if an input stream of a join is the output stream of a subquery, then we can perhaps push down our tuple omission strategy into input streams of that subquery.

7.7 Stream Memory Management

There has been work on systems-oriented techniques for memory management, for example, offloading operator state in SEEP [15] and Google Cloud Dataflow [16]. These techniques are orthogonal to our approach, which exploits query semantics for reducing memory. Memory reduction techniques have been applied in the context of specific operators such as aggregates over sliding windows [33, 36]; our work uses a similar flavor in the context of state reduction for join queries.

8 CONCLUSION

In this work we showed how to omit tuples when evaluating a threshold function over the joins of streams. We defined bracketing, a simple local condition that is sufficient to omit tuples from a stream. We also provided a complementary greedy algorithm for exploiting brackets and showed the algorithm may be safely pushed down to the stream emitter so that the benefits of reduced state may be enjoyed across the entire data stream management system. We proved the greedy approach is globally optimal and generalized it to multi-stream joins. Finally, we demonstrated the effectiveness of our approach empirically by evaluating it against synthetic and real world datasets.

This initial contribution opens the door to future work. The question of automatically detecting and/or enforcing the applicability of a threshold function is left open. While we provided a heuristic pseudo-certification approach, a stronger threshold function analysis technique would enable the application of our omission policy to be safely abstracted away from the user. While our prototype implementation demonstrates the potential of our omission policy, stronger proof of its value would come from integrating it into a production system and evaluating it there.

ACKNOWLEDGMENTS

We would like to thank Dan Suciu for pointing out that quasiconvexity is the exact superset of convexity we were searching for, Elio Damaggio and Alexandre Mineev for originally pointing out this problem space, and the reviewers for their thoughtful critiques that led to many improvements of the final draft.

REFERENCES

[1] 2020. Amazon Timestream. docs.aws.amazon.com/timestream/index.html.
[2] 2020. Azure Stream Analytics. azure.microsoft.com/en-us/services/stream-analytics/.
[3] 2020. Holt-Winters' Seasonal Method. otexts.com/fpp2/holt-winters.html.
[4] 2020. InfluxDB. docs.influxdata.com/influxdb/v1.8/.
[5] 2020. TimescaleDB. docs.timescale.com/latest/api.
[6] Amir Ahmadi, Alex Olshevsky, Pablo Parrilo, and John Tsitsiklis. 2010. NP-hardness of Deciding Convexity of Quartic Polynomials and Related Problems. *Mathematical Programming* 137 (12 2010).
[7] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. 2013. Photon: fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*. 577–588.
[8] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD*. 601–613.
[9] J. Aßfalg, H.-P. Kriegel, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. 2006. Similarity Search on Time Series Based on Threshold Queries. In *EDBT*. 276–294.
[10] J. Aßfalg, H.-P. Kriegel, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. 2008. T-Time: Threshold-Based Data Mining on Time Series. In *ICDE*. 1620–1623.
[11] Ahmed Ayad and Jeffrey F. Naughton. 2004. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *SIGMOD*. 419–430.
[12] Brian Babcock, Mayur Datar, and Rajeev Motwani. 2004. Load Shedding for Aggregation Queries over Data Streams. In *ICDE 2004*. IEEE.
[13] Rudolf Bayer. 1972. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Inf.* 1, 4 (Dec. 1972), 290–306.
[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.
[15] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD 2013*. 725–736.
[16] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: easy, efficient data-parallel pipelines. *ACM Sigplan Notices* 45, 6 (2010), 363–375.
[17] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 401–412.
[18] B. Chandramouli, J. Goldstein, and Y. Li. 2018. Impatience Is a Virtue: Revisiting Disorder in High-Performance Log Analytics. *ICDE* (2018), 677–688.
[19] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. 2003. Approximate Join Processing over Data Streams. In *SIGMOD 2003* (San Diego, California). 40–51.
[20] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, and A. Schuster. 2012. Prediction-Based Geometric Monitoring over Distributed Data Streams. In *SIGMOD 2012*. 265–276.
[21] Lukasz Golab and M. Tamer Özsu. 2003. Issues in data stream management. *SIGMOD Rec.* 32, 2 (2003), 5–14.
[22] Lukasz Golab and M. Tamer Özsu. 2003. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *VLDB*. 500–511.
[23] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. 2013. A catalog of stream processing optimizations. *ACM Comput. Surv.* 46, 4 (2013).
[24] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. 2012. The DEBS 2012 Grand Challenge. In *DEBS 2012* (Berlin, Germany). 393–398.
[25] David Jeske. 2020. *BDSkipList.* https://www.codeproject.com/Articles/138241/Bi-Directional-SkipList-in-C last accessed 2020-09-30.
[26] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. 2003. Evaluating Window Joins over Unbounded Streams. In *ICDE*. 341–352.
[27] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms.* Addison Wesley Longman Publishing Co., Inc., USA.
[28] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.
[29] Guy Sagy, Daniel Keren, Izchak Sharfman, and Assaf Schuster. 2010. Distributed Threshold Querying of General Functions by a Difference of Monotonic Representation. *Proc. VLDB Endow.* 4, 2 (Nov. 2010), 46–57.
[30] Izchak Sharfman, Assaf Schuster, and Daniel Keren. 2007. A Geometric Approach to Monitoring Threshold Functions over Distributed Data Streams. *ACM Trans. Database Syst.* 32, 4 (Nov. 2007), 23–es.
[31] Izchak Sharfman, Assaf Schuster, and Daniel Keren. 2008. Shape Sensitive Geometric Monitoring. In *PODS 2008* (Vancouver, Canada) *(PODS '08)*. 301–310.
[32] D. Sonntag. 1990. Important new values of the physical constants of 1986, vapour pressure formulations based on the ITS-90, and psychrometer formulae.
[33] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. *VLDB* 8, 7 (2015), 702–713.
[34] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load Shedding in a Data Stream Manager. In *VLDB 2003* (Berlin, Germany). VLDB Endowment, 309–320.
[35] Howell Tong. 2011. Threshold models in time series analysis–30 years on. *Statistics and its Interface* 4, 2 (2011), 107–118.
[36] J. Traub, P. M. Grulich, A. R. Cuellar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. 2018. Scotty: Efficient window aggregation for out-of-order stream processing. In *ICDE 2018*. IEEE, 1300–1303.
[37] Stratis Viglas and Jeffrey F. Naughton. 2002. Rate-based query optimization for streaming information sources. In *SIGMOD*. 37–48.
[38] Annita N. Wilschut and Peter M. G. Apers. 1991. Dataflow Query Execution in a Parallel Main-Memory Environment. In *PDIS*. 68–77.