Factor Windows: Cost-based Query Rewriting for Optimizing Correlated Window Aggregates

†Wentao Wu, †Philip A. Bernstein, †Alex Raizman, ‡Christina Pavlopoulou*
†*Microsoft Corporation*, ‡*University of California, Riverside*
†{wentao.wu, philbe, alexr}@microsoft.com, ‡cpavl001@ucr.edu

***Abstract*—Window aggregates are ubiquitous in stream processing. In Azure Stream Analytics (ASA), a stream processing service hosted by Microsoft's Azure cloud, we see many customer queries that contain aggregate functions (such as `MIN` and `MAX`) over multiple correlated windows (e.g., tumbling windows of length five minutes and ten minutes) defined on the *same* event stream. In this paper, we present a cost-based optimization framework for optimizing such queries by sharing computation among multiple windows. In particular, we introduce the notion of *factor windows*, which are auxiliary windows that are not in the input query but may nevertheless help reduce the overall computation cost, and our cost-based optimizer can produce rewritten query plans that have lower costs than the original query plan by utilizing factor windows. Since our optimization techniques are at the level of query (plan) rewriting, they can be implemented on any stream processing system that supports a declarative, SQL-like query language without changing the underlying query execution engine. We formalize the shared computation problem, present the optimization techniques in detail, and report evaluation results over both synthetic and real datasets. Our results show that, compared to the original query plans, the rewritten plans output by our cost-based optimizer can yield significantly higher (up to 16.8×) throughput.**

I. Introduction

Near-real-time querying of data streams is required by many applications, such as algorithmic stock trading, fraud detection, process monitoring, and RFID event processing. The importance of this technology has been growing due to the surge of demand for Internet of Things (IoT) and edge computing applications, leading to a variety of systems from both the open-source community (e.g., Apache Storm [46], Apache Spark Streaming [10], [52], Apache Flink [18]) and the commercial world (e.g., Amazon Kinesis [1], Microsoft Azure Stream Analytics [4], Google Cloud Dataflow [6]). Although imperative programming/query interfaces, such as the functional expressions used in Trill [23] (see Figure 1(b) for an example), remain available in these stream processing systems, declarative SQL-like query interfaces are becoming increasingly popular. For example, Apache Spark recently introduced *structured streaming*, a declarative streaming query API based on Spark SQL [10]. Azure Stream Analysics (ASA), Microsoft's cloud-based stream processing service, also differentiates itself with a SQL interface.

Declarative query interfaces allow users of stream processing systems to focus on *what* task is to be completed, rather than the details of *how* to execute it. When it comes to the question of efficient query execution, they rely on powerful query optimizers. In the traditional world of database management systems, the success of declarative query languages heavily depends on *cost-based query optimization*, which has been an active area for research since 1970's [43]. Unfortunately, in spite of the increasing popularity of declarative query interfaces in stream processing systems, cost-based query optimization of such systems remains underdeveloped — most systems, if not all, rely on *rule-based* query optimizers.

In this paper, we focus on cost-based optimization techniques for window aggregates, a ubiquitous category of streaming queries, in declarative stream processing systems. In our experience with ASA, users often want to perform the same aggregate function over the same data stream but with windows of different sizes. They do this for a variety of reasons, such as learning about or debugging a stream by exploring its behavior over different time periods, reporting near real-time behavior of a stream over small windows as well as much longer windows (e.g., an hour vs. a week), and simultaneously supporting different users whose dashboards display stream behavior over different window sizes. For example, Microsoft's Azure IoT Central service [3] hosts thousands of concurrently running dashboard queries that are window aggregates over event streams generated by various IoT devices from multiple users. It is very common in this scenario to see multiple (e.g., 5 to 10) queries over the same event stream but with varying window sizes, issued by downstream user applications that collect various telemetries from the same device reading.

A straightforward implementation would evaluate the aggregate function over each window separately. Although this implementation is relatively simple, it potentially wastes CPU cycles. We start with an example to illustrate this inefficiency.

Example 1 (Multi-window Aggregate Query)**.** *Figure 1(a) presents a query with a single aggregate function,* `MIN`*, over multiple windows. It returns the minimum temperature reported by each device every 20, 30, and 40 minutes. Figure 1(b) presents its execution plan in ASA, which is a Trill [23] expression that runs the aggregate over each window separately and then takes a union of the results. The plan is shown graphically on the left side of Figure 2(a).*

This execution plan is clearly inefficient. For example, the `MIN` function over the 40-minute tumbling window can be computed from two consecutive tuples output by the 20-

*This work was done when Christina Pavlopoulou was at Microsoft.

```
SELECT DeviceID, System.Window().Id, Min(T) AS MinTemp,
FROM Input TIMESTAMP BY EntryTime
GROUP BY DeviceID, Windows(
      Window('20 min', TumblingWindow(minute, 20),
      Window('30 min', TumblingWindow(minute, 30),
      Window('40 min', TumblingWindow(minute, 40))
```

(a) ASA query

```
Input.Multicast(s => s
   .Tumbling(minute, 20).GroupAggregate('20 min', w => w.Min(e => e.T))
.Union(s
   .Tumbling(minute, 30).GroupAggregate('30 min', w => w.Min(e => e.T)))
.Union(s
   .Tumbling(minute, 40).GroupAggregate('40 min', w => w.Min(e => e.T))))
```

(b) Translated Trill [23] expression

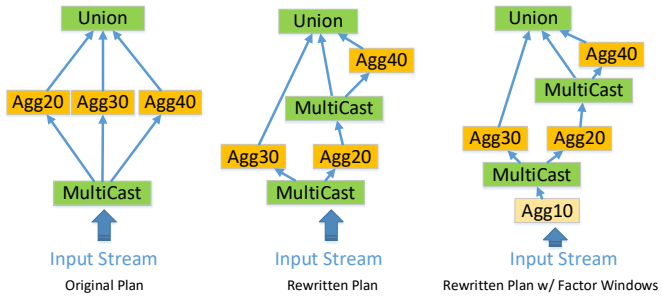
Fig. 1. An ASA aggregation query over multiple windows.

minute tumbling window, instead of computing it directly from the input stream. Such overlapping windows present an opportunity for optimization.

Our cost-based optimization technique exploits this opportunity by finding the cheapest way of computing the window aggregates in terms of the overall CPU overhead. It produces the revised query plan shown graphically in the middle of Figure 2(a). Instead of computing the aggregate function over the three windows separately, the revised plan organizes the windows into a hierarchical structure. As a result, downstream windows use sub-aggregates from their upstream windows as inputs. For instance, aggregates of the 40-minute window are computed from sub-aggregates that are outputs of the 20-minute window. This revised plan's graph is translated into a Trill [23] expression shown in Figure 2(b).

In addition to exploiting shared computation among the *existing* windows in the input query, we can further explore other *auxiliary* windows that are not in the input query but may nevertheless help reduce the overall computation cost. For this purpose, we introduce the notion of *factor window*. As an example, for the ASA query in Figure 1(a), we can insert a 10-minute tumbling window as a factor window, which leads to the revised query plan on the right side of Figure 2(a). The corresponding Trill expression is given in Figure 2(c). Based on our experimental evaluation, query plans with factor windows can yield significantly higher throughput [33] than both the original plans and plans without using factor windows, on both synthetic and real datasets.

Comparison to Window Slicing: One prominent line of work on optimizing window aggregates is *window slicing* (e.g. [29], [30], [35], [36], [47], [48]), which chops the entire window into smaller chunks and then computes the aggregate over the whole window by aggregating sub-aggregates over the small chunks. Unlike window slicing, we do not proactively chop a window. Instead, we exploit the internal overlapping relationships between correlated windows, which are ignored by window slicing techniques. Recently, Traub et al. proposed Scotty, a "general stream slicing" framework that extends the scope of window types and aggregate functions where window slicing can be applied [47], [48]. Scotty offers a *non-intrusive* implementation approach by writing "connectors" to existing



(a) Query plan rewriting

```
Input.Multicast(s1 => s1
  .Tumbling(minute, 20).GroupAggregate('20 min', w => w.Min(e => e.T))
     .Multicast(s2 => s2
     .Union(s2.Tumbling(minute, 40).GroupAggregate('40 min', w => w.Min(e => e.T))))
  .Union(s1.Tumbling(minute, 30).GroupAggregate('30 min', w => w.Min(e => e.T))))
```

(b) Translated Trill expression of the rewritten plan

```
Input.Tumbling(minute, 10).GroupAggregate('10 min', w => w.Min(e => e.T))
  .Multicast(s1 =>
     s1.Tumbling(minute, 20).GroupAggregate('20 min', w => w.Min(e => e.T))
     .Multicast(s2 => s2
        .Union(s2.Tumbling(minute, 40).GroupAggregate('40 min', w => w.Min(e => e.T))))
     .Union(s1.Tumbling(minute, 30).GroupAggregate('30 min', w => w.Min(e => e.T))))
```

(c) Translated Trill expression of the rewritten plan with factor windows

Fig. 2. Rewritten query plans by cost-based optimization.

stream processing engines such as Apache Flink. Our approach shares the same non-intrusive aspiration, though it operates by *query rewriting*. One advantage of our approach is that it does not assume any extra support from the underlying stream processing engine, such as the "*user-defined operator*" feature required by Scotty. For example, Scotty currently does not support Trill, and it is unclear how to write a Scotty "connector" for Trill. Moreover, our approach does not require engine-specific implementation beyond the support of a SQL-like query interface. For example, Scotty needs to handle checkpoints and state backends for Apache Flink [5]. In our experimental evaluation, we compared our cost-based optimization approach with Scotty (Section V-F). For the types of windows and aggregate functions that are supported by both our approach and Scotty, we observe that our approach achieved similar and often much better throughput. On the other hand, Scotty supports more types of windows and aggregate functions. We leave the problem of extending our approach to these cases as future work.

Summary of contributions and Paper Organization: To summarize, this paper makes the following contributions:

- We introduce the *window coverage graph* (WCG), a formal model and data structure that captures the overlapping relationships between windows (Section II).
- We propose a cost-based optimization framework using the WCG model, to minimize the computation cost of multi-window aggregate queries, as well as related query rewritings on the optimal, min-cost WCG (Section III).
- We extend the cost-based optimization framework by considering *factor windows*, which are auxiliary windows

that are not present in the query but can further reduce the overall computation cost (Section IV).
- We evaluate our proposed optimizations using both synthetic and real streaming datasets, with a focus on comparing the throughput of the original query plans and the optimized plans, without and with factor windows. Our results demonstrate that the optimized plans, especially the ones with factor windows, can outperform the original plans by having up to 16.8× throughput (Section V).

II. Overlaps Between Windows

We start with a formal study of the overlapping relationships between windows. We then propose *window coverage graph*, a formal model and data structure that captures overlapping relationships for a given set of windows.

A. Preliminaries

We follow the convention in the literature to represent a *window* W using two parameters [33]:

- r – the *range* of W that represents its duration;
- s – the *slide* of W that represents the gap between its two consecutive firings.

Throughout this paper, we assume that s and r are integers and use the same time unit (e.g., second, minute, hour). We assume $0 < s \leq r$ and write $W\langle r, s\rangle$. We call W a *hopping window* if $s < r$, or a *tumbling window* if $s = r$.

A *window set* $\mathcal{W} = \{W_1, ..., W_n\}$ represents a *set* of windows with no duplicates. An aggregate function f defined over a window set $\mathcal{W}$ computes a result for each $W \in \mathcal{W}$ and takes a union of the results, i.e., $f(\mathcal{W}) = \cup_{W \in \mathcal{W}} f(W)$.

1) The Interval Representation of a Window: As an alternative to the "range-slide" based representation, we can use a sequence of *intervals* to represent the lifetime of a window [13]. Without loss of generality, we assume the intervals are *left-closed* and *right-open* and define the *interval representation* of a window $W\langle r, s\rangle$ as $W = \{[m \cdot s, m \cdot s + r)\}$, where $m \geq 0$ is an integer. For example, the interval representation of window $W(10, 2)$ is $\{[0, 10), [2, 12), ...\}$.

B. Window Coverage and Partitioning

Now consider two windows $W_1\langle r_1, s_1\rangle$ and $W_2\langle r_2, s_2\rangle$. Using their interval representations, we also have $W_1 = \{[m_1 \cdot s_1, m_1 \cdot s_1 + r_1)\}$ and $W_2 = \{[m_2 \cdot s_2, m_2 \cdot s_2 + r_2)\}$, where $m_1 \geq 0$ and $m_2 \geq 0$ are integers.

Definition 1 (Window Coverage)**.** *We say that W_1 is* covered by *W_2, denoted $W_1 \leq W_2$, if $r_1 > r_2$ and for any interval $I = [a, b)$ in W_1 there exist intervals $I_a = [a, x)$ **and** $I_b = [y, b)$ in W_2 such that $a < y$ and $x < b$. (Note that, if W_1 is covered by W_2, then these two intervals are unique.) As a special case, a window is covered by itself.*

Example 2 (Window Coverage)**.** *Consider $W_1\langle r_1 = 10, s_1 = 2\rangle$ and $W_2\langle r_2 = 8, s_2 = 2\rangle$. Figure 3 plots the first two intervals of W_1 ($\{[0, 10), [2, 12)\}$) and the first three intervals of W_2 ($\{[0, 8), [2, 10), [4, 12)\}$). The first interval of W_1 is covered by the 1st and 2nd intervals of W_2, and the second interval of W_1 is covered by the 2nd and 3nd intervals of W_2.*

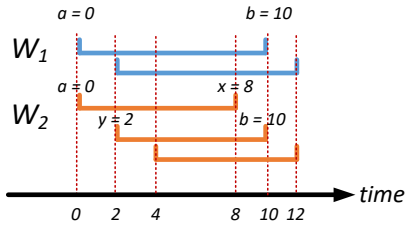


Fig. 3. An example of window coverage.

The following theorem provides sufficient and necessary conditions for the window coverage relation (proofs are available in the full version [51] of this paper):

Theorem 1. *W_1 is covered by W_2 if and only if (1) s_1 is a multiple of s_2 and (2) $\delta_r = r_1 - r_2$ is a multiple of s_2.*

Example 3 (Window Coverage Theorem)**.** *Consider again the windows of Example 2: $W_1\langle r_1 = 10, s_1 = 2\rangle$ and $W_2\langle r_2 = 8, s_2 = 2\rangle$. We have $s_1/s_2 = 1$, so s_1 is a multiple of s_2, and $(r_1 - r_2)/s_2 = 1$, so $r_1 - r_2$ is a multiple of s_2. By Theorem 1, W_1 is covered by W_2.*

1) A Partial Order: The window coverage relation defines a *partial order* over windows, as characterized by:

Theorem 2. *The window coverage relation is* reflexive, anti-symmetric, *and* transitive.

2) Interval Coverage: Suppose that $W_1 \leq W_2$. For any interval $I = [a, b)$ in W_1, let $I_a = [a, x)$ and $I_b = [y, b)$ be the two intervals in W_2 specified by Definition 1.

Definition 2 (Covering Interval Set)**.** *Let the set of intervals "between" I_a and I_b in W_2 be $\mathcal{I}_{a,b} = \{[u, v) : a \leq u$ **and** $v \leq b\}$. We call $\mathcal{I}_{a,b}$ the* covering (interval) set *of I.*

Clearly, $I_a, I_b \in \mathcal{I}_{a,b}$. The cardinality $|\mathcal{I}_{a,b}|$ is independent of the choice of a and b. We call it the *covering multiplier* of W_2 with respect to W_1, denoted $M(W_1, W_2)$. An analytic form for the covering multiplier is given by:

Theorem 3. *If the window $W_1\langle r_1, s_1\rangle$ is covered by the window $W_2\langle r_2, s_2\rangle$, then $M(W_1, W_2) = 1 + (r_1 - r_2)/s_2$.*

We now introduce the more general notion of "interval coverage" based on the above discussion.

Definition 3 (Interval Coverage)**.** *We say that an interval I is* covered by *a set of intervals $\mathcal{I}$ if $I = \cup_{J \in \mathcal{I}} J$.*

Example 4 (Interval Coverage)**.** *In Figure 3, for the first interval in W_1 the covering set consists of the first and second intervals in W_2, and for the second interval in W_1 consists of the second and third intervals in W_2.*

3) Interval/Window Partitioning: A special case of interval coverage is when the intervals in the covering set are disjoint.

Definition 4 (Interval Partitioning)**.** *If an interval I is covered by a set of intervals $\mathcal{I}$ such that the intervals in $\mathcal{I}$ are mutually exclusive, then I is* partitioned by *$\mathcal{I}$.*

(a) Window partitioning (b) Window coverage

Fig. 4. A comparison of window partitioning with general window coverage.

We can further define "window partitioning" accordingly, which is a special case of window coverage:

Definition 5 (Window Partitioning)**.** *We say that W_1 is* partitioned by *W_2, if W_1 is covered by W_2* ***and*** *each interval in W_1 is* partitioned by *its covering set in W_2.*

Figure 4 illustrates the difference between window partitioning and general window coverage. Here each interval of W_1 is covered by two intervals of W_2, i.e., $M(W_1, W_2) = 2$. We now provide rigorous conditions for window partitioning:

Theorem 4. *W_1 is partitioned by W_2 if and only if (1) s_1 is a multiple of s_2, (2) r_1 is a multiple of s_2, and (3) $r_2 = s_2$ (i.e., W_2 is a tumbling window).*

Example 5 (Window Partitioning)**.** *In Example 2, $s_1/s_2 = 1$ and $r_1/s_2 = 5$. So conditions (1) and (2) in Theorem 4 hold. However, condition (3) is violated since $r_2 \neq s_2$ (i.e., W_2 is not tumbling). As a result, W_1 cannot be partitioned by W_2.*

C. Window Coverage Graph (WCG)

We define the *window coverage graph* $\mathcal{G} = (\mathcal{W}, \mathcal{E})$ for a given window set $\mathcal{W}$ based on the partial order introduced by the window coverage relation. For every $W_1, W_2 \in \mathcal{W}$ such that $W_1 \leq W_2$, we add an edge $e = (W_2, W_1)$ to the edge set $\mathcal{E}$. The time complexity of constructing the WCG is $O(|\mathcal{W}|^2)$, given that checking the window coverage relationship takes only constant time (Theorems 1 and 4).

III. Aggregates over WCG

We now study the problem of evaluating aggregate functions over a window set that is modeled by its WCG. We first revisit a classic taxonomy of aggregate functions in the new context of window set and WCG. We then present a cost-based framework for the WCG, with the goal of minimizing the overall computation cost. We further present query rewriting techniques with respect to an optimal WCG.

A. A Taxonomy of Aggregate Functions

Let f be a given aggregate function, e.g., `MIN`, `MAX`, `AVG`, and so on. Gray et al. classified f into three categories [27]:

- *Distributive* – f is distributive if there is some function g s.t., for a table T, $f(T) = g(\{f(T_1), ..., f(T_n)\})$, where $\mathcal{T} = \{T_1, ..., T_n\}$ is a *disjoint* partition of T. Typical examples include `MIN`, `MAX`, `COUNT`, and `SUM`. In fact, $f = g$ for `MIN`, `MAX`, and `SUM` but for `COUNT` g should be `SUM`.
- *Algebraic* – f is algebraic if there are functions g and h s.t. $f(T) = h(\{g(T_1), g(T_2), ..., g(T_n)\})$. Typical examples are `AVG` and `STDEV`. For `AVG`, g records the sum and count for each subset T_i $(1 \leq i \leq n)$ and h computes the average for T_i by dividing the sum by the count.
- *Holistic* – f is holistic if there is no constant bound on the size of storage needed to describe a sub-aggregate. Typical examples include `MEDIAN` and `RANK`.

Only *distributive* or *algebraic* aggregate functions can be computed by aggregating sub-aggregates [14], [47]. Although recent work [14], [47] on window slicing "supports" holistic aggregate functions, the corresponding window slices contain all input events rather than sub-aggregates. Therefore, for holistic aggregate functions, we currently fall back to the default execution plan where each window is processed independently. We leave the exploration of better support for holistic aggregate functions as interesting future work.

One important prerequisite in this taxonomy is that $\mathcal{T} = \{T_1, ..., T_n\}$ is a *partition* of T. In our scenario, it means that if we want to evaluate f over a window W_1 by aggregating sub-aggregates that have been computed over another window W_2, then W_1 has to be *partitioned by* W_2.

Theorem 5. *Given that window W_1 is partitioned by window W_2, if the aggregate function f is either distributive or algebraic, then f over W_1 can be computed by aggregating sub-aggregates over W_2.*

If W_1 is only covered (but not partitioned) by W_2, then the type of aggregate function f that can be computed using Theorem 5 must be further restricted, such that f remains distributive or algebraic even if the T_i's in $\mathcal{T}$ can overlap. The aggregate functions `MIN` and `MAX` retain such properties, as stated by the following theorem:

Theorem 6. *The aggregate functions `MIN` and `MAX` are distributive even if $\mathcal{T}$ is not disjoint.*

B. A Cost-based Optimization Framework

Given a streaming query Q that contains an aggregate function f over a window set $\mathcal{W}$, our goal is to *minimize* the total computation overhead of evaluating Q. A naive approach to evaluate Q is to compute f over each window of $\mathcal{W}$ one by one. Clearly, this will do redundant computation if the windows in $\mathcal{W}$ "overlap." To minimize computation one needs to maximize the amount of computation that is shared among overlapping windows. We present a cost-based optimization framework that does this by exploiting the window coverage relationships captured by the WCG of $\mathcal{W}$.

1) Cost Modeling: Let $\mathcal{W} = \{W_1, ..., W_n\}$ be a window set. Given the WCG $\mathcal{G} = (\mathcal{W}, \mathcal{E})$, we assign a *weight* c_i to each vertex (i.e., window) W_i in $\mathcal{W}$ that represents its computation cost with respect to the (given) aggregate function f. The total computation cost is simply the *sum* of these weights, i.e., $C = \sum_{i=1}^{n} c_i$. Our goal is to minimize C.

We assume that the cost of computing f is proportional to the number of events processed. We further assume a steady input event rate $\eta \geq 1$. Let $R = \text{lcm}(r_1, ..., r_n)$ be the *least common multiple* of the ranges of the windows $W_1\langle r_1, s_1\rangle$, ..., $W_n\langle r_n, s_n\rangle$ in $\mathcal{W}$. For each window W_i, *the cost c_i of computing f over W_i for events in a period of length R* depends on two quantities:

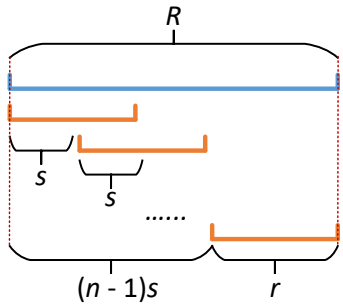


Fig. 5. Illustration of the recurrence count.

- *Recurrence count* n_i – the number of intervals (i.e., instances) of W_i occurring during the period of R;
- *Instance cost* μ_i – the cost of evaluating an instance of W_i.

Clearly, $c_i = n_i \cdot \mu_i$. We next analyze the two quantities.

Recurrence count: For each window W_i, let $m_i = R/r_i$ be its *multiplicity*. The *recurrence count* n_i can be written as

$$n_i = 1 + (m_i - 1)\frac{r_i}{s_i}. \quad (1)$$

Figure 5 illustrates how we obtained the above formula for n_i. Essentially, we have $R = (n_i - 1) \cdot s_i + r_i$, which yields

$$n_i = 1 + \frac{R - r_i}{s_i} = 1 + \left(\frac{R}{r_i} - 1\right)\frac{r_i}{s_i} = 1 + (m_i - 1)\frac{r_i}{s_i}.$$

If W_i is a tumbling window, then $n_i = m_i$. In this paper we assume that r_i is a multiple of s_i so that n_i is an integer. [1]

Instance cost: Clearly, without any computation sharing, the instance cost of W_i is $\mu_i = \eta \cdot r_i$. Sharing computation, however can reduce the computation cost. Consider $W_1\langle r_1, s_1\rangle$ and $W_2\langle r_2, s_2\rangle$. We have the following observation:

Observation 1. *If W_1 is covered by (perhaps multiple) W_2's, then the instance cost of W_1 can be reduced to*

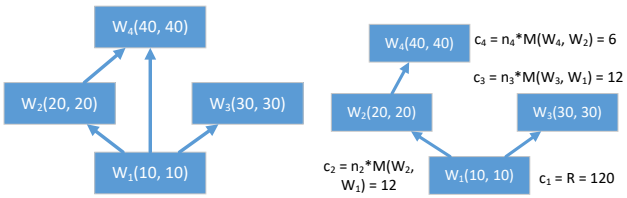
$$\mu_1 = \min_{W_2 \text{ s.t. } W_1 \leq W_2} \{M(W_1, W_2)\}.$$

2) Cost Minimization: Algorithm 1 presents our procedure for finding the minimum overall cost based on the WCG, cost model, and Observation 1. It starts by constructing the WCG $\mathcal{G}$ with respect to the given window set $\mathcal{W}$ and aggregate function f (line 1) – we need f to know whether to use "covered by" or "partitioned by" when constructing WCG.[2] We then process the windows one by one (lines 2 to 5).

For each window W_i, at line 3 we initialize its cost with $c_i = n_i \cdot (\eta \cdot r_i)$. (The initial cost is $c_i = m_i \cdot (\eta \cdot r_i) = \eta \cdot R$ if W_i is a tumbling window.) We then iterate over incoming edges (W', W_i), revising the cost c_i w.r.t. Observation 1 (lines 4 to 5). Finally, we remove all edges that do not correspond to the one that led to the minimum cost (lines 6 to 7). The result is graph $\mathcal{G}_{\min}$, called the *min-cost WCG* hereafter, which captures all minimum cost information. It is the input to the query rewriting algorithm that we will discuss in Section III-C.

[1] If we want n_i to be an integer when r_i is not a multiple of s_i, $m_i - 1$ must be a multiple of s_i. Thus, $m_i - 1 = l_i \cdot s_i$ where l_i is an integer, which yields $R = r_i(1 + l_i \cdot s_i)$, for all $1 \leq i \leq n$. Therefore, all n_i's are integers only if there exist integers l_1, ..., l_n such that $r_1(1 + l_1 \cdot s_1) = \cdots = r_n(1 + l_n \cdot s_n)$. We leave the case when n_i's may not be integers for future work.

[2] In our current implementation, we use "covered by" semantics when f is `MIN` or `MAX`, and "partitioned by" when f is `COUNT`, `SUM`, and `AVG`, which are part of the SQL standard. Future work could expand these two lists with other aggregate functions.



(a) Initial WCG (b) Min-cost WCG

Fig. 6. WCG and min-cost WCG for Example 6.

Algorithm 1: Find the min-cost WCG.

Input: $\mathcal{W} = \{W_i\}_{i=1}^n$, a window set; f, aggregate function.
Output: $\mathcal{G}_{\min}$, the min-cost WCG w.r.t. $\mathcal{W}$ and f.
1 Construct the WCG $\mathcal{G} = (\mathcal{W}, \mathcal{E})$ w.r.t. "covered by" or "partitioned by" as determined by f;
2 **foreach** $W_i \in \mathcal{W}$ **do**
3 Initialize its cost $c_i \leftarrow n_i \cdot (\eta \cdot r_i)$;
4 **foreach** $W' \in \mathcal{W}$ *s.t.* $(W', W_i) \in \mathcal{E}$ **do**
5 Revise cost $c_i \leftarrow \min\{c_i, n_i \cdot M(W_i, W')\}$;
6 **foreach** $W_i \in \mathcal{W}$ **do**
7 Remove all incoming edges that do not correspond to (the final value of) c_i;
8 **return** *the result graph* $\mathcal{G}_{\min}$;

Example 6. *Consider a query that contains four tumbling windows: $W_1\langle 10, 10\rangle$, $W_2\langle 20, 20\rangle$, $W_3\langle 30, 30\rangle$, and $W_4\langle 40, 40\rangle$. It does not matter which aggregate function f we choose here, since "covered by" and "partitioned by" semantics coincide when all windows in a window set are tumbling windows.*

Assuming an incoming event ingestion rate $\eta = 1$, the total cost of computing the four windows is $C = 4\eta R = 4R = 480$, where $R = \text{lcm}\{10, 20, 30, 40\} = 120$.

Figure 6 shows the initial WCG (Figure 6(a)) and the final min-cost WCG (Figure 6(b)) by running Algorithm 1, when exploiting the overlaps between the windows. The total cost is therefore reduced to $C' = c'_1 + c'_2 + c'_3 + c'_4 = 120 + 12 + 12 + 6 = 150$, a 62.5% reduction from the initial cost $C = 480$.

Limitations: Since our cost-based optimization framework exploits the coverage relationships between windows, it cannot improve the execution plan if such opportunities are not present. For example, consider a set of tumbling windows where all ranges are "mutually prime," e.g., $W_1(15, 15)$, $W_2(17, 17)$, and $W_3(19, 19)$. In such cases, our cost model cannot lead to plans that improve over the default plan where each window is evaluated independently.

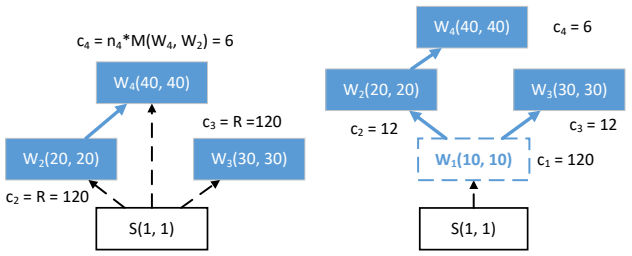
C. Query Rewriting

To leverage the benefits of shared window computation, we rewrite the original ASA query plan with respect to the min-cost WCG $\mathcal{G}_{\min}$ based on the following observation:

Theorem 7. *$\mathcal{G}_{\min}$ is a forest, i.e., a collection of trees.*

The proof follows directly from noticing that each window in $G_{\min}$ has at most one incoming edge (due to lines 6 to 7).

Figure 2 shows how we revise the query execution plan in Example 1. Figure 2(a) presents the original plan and the



(a) Initial WCG (b) Min-cost WCG

Fig. 7. Min-cost WCGs for Example 1 with and without using factor windows.

revised plan based on the min-cost WCG. Figure 2(b) presents the translated Trill expression [23]. We include a formal description of this query rewriting procedure in [51]. Translation to query execution plans expressed by other streaming API's, such as the Apache Flink DataStream API [2], is similar.

IV. Factor Windows

We have been confining our discussion to sharing computation over windows in the given window set. One can add auxiliary windows that are not in the window set but may nevertheless help reduce the overall computation cost. We call them *factor windows*.

Definition 6. *Given a window set $\mathcal{W}$, a window W is called a* factor window *with respect to $\mathcal{W}$ if $W \notin \mathcal{W}$ and there exists some window $W' \in \mathcal{W}$ such that $W' \leq W$.*

Note that we do not expose the results of factor windows to users, as they are not part of the user query.

Example 7. *Suppose we modify the query in Example 6 by removing the tumbling window $W_1(10, 10)$. The resulting query Q contains three tumbling windows $W_2(20, 20)$, $W_3(30, 30)$, and $W_4(40, 40)$. The cost of directly computing them is $C = 3R = 360$, as here $R = \text{lcm}\{20, 30, 40\} = 120$ remains the same.*

If we apply Algorithm 1 over Q, we get the min-cost WCG presented in Figure 7(a). As a result, the overall cost is $C' = c'_2 + c'_3 + c'_4 = 120 + 120 + 6 = 246$, a reduction of 31.7% from the baseline cost $C = 360$.

If we allow factor windows and apply Algorithm 3 over Q, then we get the min-cost WCG in Figure 7(b). The window $W_1(10, 10)$ is "added back" as a factor window, which participates in evaluating Q but does not expose its result to users. As in Example 6, the overall cost now is $C'' = 150$, which is 58.3% less than the baseline cost $C = 360$ and 39% less than the cost $C' = 246$ without using factor windows.

A. Impact of Factor Window

One natural question to ask is: When does a factor window help? In the following, we provide a formal analysis.

Augmented WCG: For the WCG $\mathcal{G} = (\mathcal{W}, \mathcal{E})$ induced by the given window set $\mathcal{W}$ and aggregate function f, we add a virtual tumbling window $S\langle r = 1, s = 1\rangle$ into $\mathcal{W}$, and add an edge (S, W) into $\mathcal{E}$ for each $W \in \mathcal{W}$ that has no incoming edges (i.e., W is not covered by any other window). However, if such an S already exists in $\mathcal{W}$, we do not add another

(a) Interesting (b) Uninteresting

Fig. 8. Two basic patterns in WCG ($K \geq 1$).

one. Intuitively, S represents a window consisting of *atomic* intervals that emit an aggregate for each time unit; therefore S covers all windows in $\mathcal{W}$. The computation cost of S is always $\eta \cdot R$, as it cannot be covered by any other window. This augmented graph is a directed acyclic graph (DAG) with a single "root" S. From now on, when we refer to the WCG we mean its augmented version.

Two Basic Patterns: Figure 8 presents two basic patterns in (the augmented) WCG, for an arbitrary window $W \in \mathcal{W}$. We are interested in the pattern in Figure 8(a) but not the one in Figure 8(b), as W can only affect the costs of its *downstream* windows. This eliminates windows in WCG without outgoing edges from consideration.

Analysis of Impact: As shown in Figure 9, let W_f be a factor window inserted "between" W and its downstream windows W_1, ..., W_K. We can do this for all "intermediate" vertices, i.e., windows with *both* incoming and outgoing edges, in (the augmented) WCG, thanks to the virtual "root" S. Clearly, $W_f \leq W$, and $W_j \leq W_f$ for $1 \leq j \leq K$. We now compare the overall computation costs with and without inserting W_f. The cost with the factor window W_f is $c = \sum_{j=1}^{K} \text{cost}(W_j) + \text{cost}(W_f) + \text{cost}(W)$. On the other hand, the cost without W_f is $c' = \sum_{j=1}^{K} \text{cost}'(W_j) + \text{cost}(W)$. We define the *benefit* of W_f as $\delta_f = c' - c$.

Since $\text{cost}(W_j) = n_j \cdot M(W_j, W_f)$, $\text{cost}(W_f) = n_f \cdot M(W_f, W)$, and $\text{cost}'(W_j) = n_j \cdot M(W_j, W)$, it follows that

$$\delta_f = \sum_{j=1}^{K} n_j \Big(M(W_j, W) - M(W_j, W_f) \Big) - n_f M(W_f, W).$$

By Theorem 3, $M(W_j, W_f) = 1 + (r_j - r_f)/s_f$, $M(W_j, W) = 1 + (r_j - r_W)/s_W$, and $M(W_f, W) = 1 + (r_f - r_W)/s_W$. Substituting into the above equation, we obtain

$$\delta_f = \sum_{j=1}^{K} n_j \Big(\frac{r_j - r_W}{s_W} - \frac{r_j - r_f}{s_f} \Big) - n_f \Big(1 + \frac{r_f - r_W}{s_W} \Big).$$

We now define $k_f = r_f / s_f$ and $k_W = r_W / s_W$ to simplify notation. With this notation, we have

$$\delta_f = n_f \Big(\sum_{j=1}^{K} \frac{n_j}{n_f} \Big(k_f - \frac{r_j}{s_f} + \frac{r_j}{s_W} - k_W \Big) - (1 + \frac{r_f}{s_W} - k_W) \Big). \quad (2)$$

Inserting W_f improves if and only if $\delta_f \geq 0$, i.e.,

$$\sum_{j=1}^{K} \frac{n_j}{n_f} \Big(k_f - \frac{r_j}{s_f} + \frac{r_j}{s_W} - k_W \Big) \geq 1 + \frac{r_f}{s_W} - k_W. \quad (3)$$

B. Candidate Generation and Selection

We can use Equation 3 to determine whether a factor window is beneficial. The next problem is to find candidate factor windows that are beneficial, from which we can select the best one. Algorithm 2 illustrates this candidate generation and selection procedure in detail.

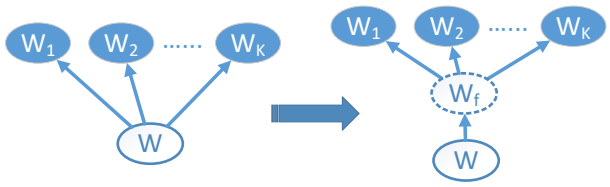


Fig. 9. Impact of factor window W_f.

Algorithm 2: Find the best factor window under "covered by" semantics.

Input: W, a window; $\{W_1, ..., W_K\}$, W's downstream windows (ref. Figure 9).
Output: The best factor window W_f w.r.t. W and $\{W_1, ..., W_K\}$.

1 // Construct the set $\mathcal{W}_f$ of candidate factor windows.
2 $\mathcal{W}_f \leftarrow \emptyset$;
3 $s_d \leftarrow \gcd\{s_1, ..., s_K\}$;
4 $\mathcal{S}_f \leftarrow \{s_f : s_d \bmod s_f = 0 \text{ and } s_f \bmod s_W = 0\}$;
5 $r_{\min} \leftarrow \min\{r_1, ..., r_K\}$;
6 **foreach** $s_f \in \mathcal{S}_f$ **do**
7 $\quad \mathcal{R}_f \leftarrow \{r_f : r_f \bmod s_f = 0 \text{ and } r_f \leq r_{\min}\}$;
8 $\quad$ **foreach** $r_f \in \mathcal{R}_f$ **do**
9 $\quad\quad$ Construct a candidate factor window $W_f\langle r_f, s_f\rangle$;
10 $\quad\quad$ **if** $W_f \leq W$ *and* $W_j \leq W_f$ *for* $1 \leq j \leq K$ **then**
11 $\quad\quad\quad \mathcal{W}_f \leftarrow \mathcal{W}_f \cup \{W_f\}$;
12 // Find the best factor window from $\mathcal{W}_f$.
13 $\delta_f^{\max} \leftarrow 0$, $W_f^{\max} \leftarrow$ `null`;
14 **foreach** $W_f \in \mathcal{W}_f$ **do**
15 $\quad$ Compute the benefit δ_f of W_f using Equation 2;
16 $\quad$ **if** $\delta_f \geq 0$ *and* $\delta_f > \delta_f^{\max}$ **then**
17 $\quad\quad \delta_f^{\max} \leftarrow \delta_f$, $W_f^{\max} \leftarrow W_f$;
18 **return** $W_f^{\max}$;

1) Candidate Generation: It looks for *eligible slides* s_f and *eligible ranges* r_f as follows (lines 1 to 11 of Algorithm 2):

- *Eligible slides*: Let $s_d = \gcd\{s_1, ..., s_K\}$. The set of eligible slides is $\mathcal{S}_f = \{s_f : s_d \bmod s_f = 0 \text{ and } s_f \bmod s_W = 0\}$. That is, s_f must be a factor of s_d and a multiple of s_W.
- *Eligible ranges*: Let $r_{\min} = \min\{r_1, ..., r_K\}$. For each $s_f \in \mathcal{S}_f$, the set of eligible ranges is $\mathcal{R}_f = \{r_f : r_f \bmod s_f = 0 \text{ and } r_f \leq r_{\min}\}$, i.e., $r_f \leq r_{\min}$ must be a multiple of s_f.

For each eligible pair (s_f, r_f), we construct a candidate factor window $W_f\langle r_f, s_f\rangle$ and further check the window coverage constraints in Figure 9, i.e., $W_f \leq W$ and $W_j \leq W_f$ for $1 \leq j \leq K$ (line 10), to only keep valid candidates.

2) Candidate Selection: Many candidate factor windows in $\mathcal{W}_f$ may be beneficial (i.e., Equation 3 holds). Only the one that leads to the maximum cost reduction (i.e., benefit) should be added. We thus compute the benefits of the candidates (by Equation 2) and select the one with the maximum benefit (lines 12 to 17 of Algorithm 2).

3) Time Complexity Analysis of Algorithm 2: Computing s_d at line 3 takes $O(s_{\max} \log s_{\max})$ time using Euclid's algorithm [24], where $s_{\max} = \max\{s_1, ..., s_K\}$. Finding all eligible slides at line 4 takes $O(\lceil \frac{s_d}{s_W} \rceil)$ time. Computing $r_{\min}$ at line 5 takes $O(K)$ time. For each $s_f \in \mathcal{S}_f$, finding its eligible ranges at line 7 takes $O(\lceil \frac{r_{\min}}{s_f} \rceil)$ time. For each $W_f\langle r_f, s_f\rangle$, it takes $O(K)$ time to check all related window coverage relationships at line 10. Hence, the candidate generation stage (lines 1 to 11) takes $O(s_{\max} \log s_{\max} + \lceil \frac{s_d}{s_W} \rceil + K + |\mathcal{S}_f| \cdot |\mathcal{R}_f| \cdot K)$ time. To simplify our analysis, we assume it is dominated by $O(|\mathcal{S}_f| \cdot |\mathcal{R}_f| \cdot K)$. Now consider the candidate selection stage (lines 12 to 17). Since we check Equation 2 once for each W_f, it takes $O(|\mathcal{S}_f| \cdot |\mathcal{R}_f| \cdot K)$ time in total. Since $|\mathcal{S}_f| = O(\lceil \frac{s_d}{s_W} \rceil)$ and $|\mathcal{R}_f| = O(\lceil \frac{r_{\min}}{s_W} \rceil)$, it follows that the time complexity of Algorithm 2 is $O(\lceil \frac{s_d}{s_W} \rceil \cdot \lceil \frac{r_{\min}}{s_W} \rceil \cdot K)$.

Algorithm 3: Find the min-cost WCG when factor windows are allowed.

Input: $\mathcal{W} = \{W_i\}_{i=1}^n$, a window set; f, aggregate function.
Output: $\mathcal{G}_{\min}$, the min-cost WCG w.r.t. $\mathcal{W}$ and f, where factor windows are allowed.

1 Construct the WCG $\mathcal{G} = (\mathcal{W}, \mathcal{E})$ w.r.t. "covered by" or "partitioned by" determined by f;
2 **foreach** $W \in \mathcal{W}$ **do**
3 $\quad W_f \leftarrow$ *FindBestFactorWindow*(W, W's downstream windows $\{W_1, ..., W_K\}$) using Algorithm 2;
4 $\quad$ Expand $\mathcal{G}$ by adding W_f and the corresponding edges (as shown in Figure 9);
5 $\mathcal{G}_{\min} \leftarrow$ Run lines 2-7 of Algorithm 1 over the expanded $\mathcal{G}$;
6 **return** *the result graph* $\mathcal{G}_{\min}$;

C. Putting Things Together

Algorithm 3 is the revised version of Algorithm 1 that returns the min-cost WCG when factor windows are allowed. It first extends the original WCG by adding the best factor windows, found by Algorithm 2, for existing windows (lines 2 to 4). It then simply invokes Algorithm 1 on the extended WCG (rather than the original one) to find the new min-cost WCG that contains factor windows (line 5).

Unlike Algorithm 1, Algorithm 3 is no longer optimal. In fact, the cost minimization problem when factor windows are allowed is an instance of the Steiner tree problem [34], which is NP-hard. Various approximate algorithms have been proposed for Steiner trees (e.g., [16], [42]), but we choose to stay with Algorithm 3 because it is simple and easy to implement. It would be interesting future work to characterize the gap between the factor windows found by Algorithm 3 and the ones that could be found by an optimal solution.[3] However, even though the factor windows found by Algorithm 3 may not be the optimal ones, the min-cost WCG with factor windows improves over the min-cost WCG without factor windows (returned by Algorithm 1), since Algorithm 3 only inserts a factor window if it is beneficial (lines 2 to 4).

Time Complexity Analysis of Algorithm 3: Construction of the WCG requires $O(|\mathcal{W}|^2)$ time as it needs to check each pair of windows to test their coverage relationship. For a given window $W \in \mathcal{W}$ and its downstream windows W_1, ..., W_K, it takes $O(\lceil \frac{s_d}{s_W} \rceil \cdot \lceil \frac{r_{\min}}{s_W} \rceil \cdot K)$ time to find its best factor window W_f using Algorithm 2. Meanwhile,

[3]Note that we restricted ourselves to consider only a subset of all possible factor windows. For example, for the WCG in Figure 7(a), our approach would not consider the factor window $W(15, 15)$, as $\gcd\{20, 30, 40\} = 10 < 15$ (ref. line 3 of Algorithm 5). An ideal, optimal solution would have also considered such candidates. In fact, it needs to generate all valid candidate factor windows, instead of finding a "locally optimal" factor window for each input window (as Algorithm 3 does), insert them into the WCG, and then solve the Steiner tree problem. Since the problem is NP-hard, the time complexity in the worst case would be exponential w.r.t. the size of the WCG.

adding W_f and the corresponding edges requires $O(K)$ time. Furthermore, running lines 2 to 7 of Algorithm 1 over the expanded graph takes $O((2 \cdot |\mathcal{W}|)^2)$ time. Thus, the time complexity of Algorithm 3 is $O(5|\mathcal{W}|^2 + |\mathcal{W}| \cdot M_{\mathcal{W}})$, where $M_{\mathcal{W}} = \max_{W \in \mathcal{W}}\{\lceil \frac{s_d}{s_W} \rceil \cdot \lceil \frac{r_{\min}}{s_W} \rceil \cdot K\}$.

D. The Case of "Partitioned By"

We can improve the procedure **FindBestFactorWindow** in Algorithm 2 if we restrict the window coverage relationships to "partitioned by" semantics, which works for more types of aggregate functions. In this special case, the candidate factor windows are restricted to *tumbling windows* (by Theorem 4).

Algorithm 4: Determine whether a factor window would be beneficial under "partitioned by" semantics.

Input: W_f, a factor window; W, a target window with downstream windows W_1, ..., W_K; λ, by Equation 4.
Output: Return `true` if adding W_f improves the overall cost, `false` otherwise.

1 **if** $K \geq 2$ **then**
2 **return** *true*;
3 // We have $K = 1$ hereafter.
4 **if** $k_1 = 1$ **then**
5 **return** *false*;
6 **else**
7 // We have $k_1 > 1$ hereafter.
8 **if** $k_1 \geq 3$ *and* $m_1 \geq 3$ **then**
9 **return** *true*;
10 **else**
11 Compute $\frac{r_f}{r_W}$ and $\frac{\lambda}{\lambda - 1} = 1 + \frac{m_1}{(m_1-1)(k_1-1)}$;
12 **return** *true if* $\frac{r_f}{r_W} \geq \frac{\lambda}{\lambda-1}$, *false otherwise*;

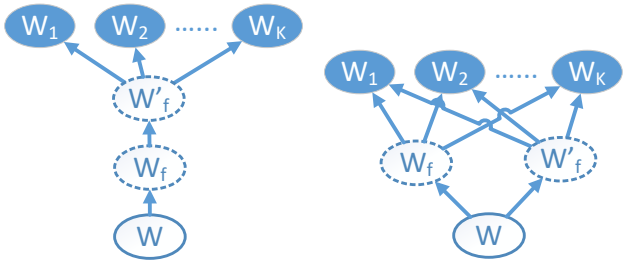
1) Revisit of Impact of Factor Windows: We first revisit the problem of determining whether a factor window is beneficial, under "partitioned by" semantics. Algorithm 4 summarizes the procedure that determines whether a factor window W_f would help in the case of "partitioned by." Here, λ is defined as

$$\lambda = \sum_{j=1}^{K} \frac{n_j}{m_j}. \tag{4}$$

The procedure in Algorithm 4 looks complicated. We offer some intuition below to help understand it:

(Case 1) If W_f has two or more downstream windows (i.e., when $K \geq 2$), then it improves the overall cost (lines 1 to 2), since now at least one downstream window would benefit from reading sub-aggregates from W_f (rather than from W). We provide more explanation using a special case (referring to Figure 9) when $K = 2$ and all windows are tumbling. We can simplify Equation 2 by noticing $k_f = k_W = 1$, $r_f = s_f$, and $r_W = s_W$, since both W_f and W are tumbling windows: $\delta_f = \sum_{j=1}^{2} n_j \cdot \left(\frac{r_j}{r_W} - \frac{r_j}{r_f}\right) - n_f \cdot \frac{r_f}{r_W}$. Moreover, since all windows are tumbling, $n_j = m_j = R/r_j$ for $j \in \{1, 2\}$, and $n_f = m_f = R/r_f$. As a result, $\delta_f = R \cdot \left(\frac{1}{r_W} - \frac{2}{r_f}\right) \geq 0$, since $r_f \geq 2r_W$ by Theorem 4.

(Case 2) If W_f only has one downstream window W_1 that is tumbling (i.e., the case when $K = 1$ and $k_1 = 1$), then it cannot reduce the overall cost (lines 4 to 5) because one now needs to use all sub-aggregates from W to compute W_f itself. Without W_f one can use the same sub-aggregates to compute W_1 directly. The case when W_f has one unique downstream window W_1 that is not tumbling (i.e., when $K = 1$ and $k_1 > 1$) can be understood in a similar way as "**Case 1**" above, since sub-aggregates from W_f can reduce cost for *intervals* in W_1 that overlap (lines 6 to 12).



(a) Dependent (b) Independent

Fig. 10. Dependent and independent factor windows with multiple candidates.

We can formally prove the correctness of Algorithm 4, using Equations 2 and 4 [51]:

Theorem 8. *Algorithm 4 correctly determines whether W_f would help when both W_f and W are tumbling windows.*

2) Revisit of Candidate Generation and Selection: We now revisit the problems of candidate generation and selection under "partitioned by" semantics.

(Candidate Generation) By restricting to tumbling windows under "partitioned by" semantics, we can significantly reduce the search space for potential candidates. By Theorem 4, the range r_f of a factor window W_f must be a *common factor* of the ranges r_1, ..., r_K of all downstream windows W_1, ..., W_K for a given target window W (ref. Figure 9). Moreover, r_f must also be a multiple of the range r_W of the target window W. As a result, one can enumerate all candidates by starting from the *greatest common divisor* r_d of r_1, ..., r_K and look for all factors r_f of r_d that are also multiples of r_W.

(Candidate Selection) To find the best factor window, we compare the benefits of two candidates W_f and W'_f. There are two cases as shown in Figure 10:

- W_f and W'_f are *dependent*, meaning either $W_f \leq W'_f$ or $W'_f \leq W_f$ – see Figure 10(a);
- W_f and W'_f are *independent* – see Figure 10(b).

Dependent Candidates: Let W_f and W'_f be two eligible factor windows such that $W'_f \leq W_f$. Then W_f can be omitted as adding it cannot reduce the overall cost. This can be understood by running Algorithm 4 against W_f, by viewing W'_f as W_f's only (tumbling) downstream window. Algorithm 4 would return `false` as this is the case when $K = 1$ and $k_1 = 1$ (line 5).

Independent Candidates: For the independent case, we have to compare the costs in more detail. Specifically, let $c_f = \sum_{j=1}^{K} \text{cost}(W_j) + \text{cost}(W_f) + \text{cost}(W) = \sum_{j=1}^{K} n_j \cdot M(W_j, W_f) + n_f \cdot M(W_f, W) + \text{cost}(W)$, and

Algorithm 5: Find the best factor window under "partitioned by" semantics.

Input: W, a window; $\{W_1, ..., W_K\}$, W's downstream windows (ref. Figure 9).
Output: The best tumbling factor window W_f that led to the minimum overall cost.

1 Compute λ using Equation 4;
2 // Generate candidate tumbling factor windows.
3 $r_d \leftarrow \gcd(\{r_1, ..., r_K\})$;
4 **if** $r_d = r_W$ **then**
5 | **return** W;
6 $\mathcal{F} \leftarrow \{r_f : r_d \bmod r_f = 0 \text{ and } r_f \bmod r_W = 0\}$;
7 $\mathcal{W}_f \leftarrow \emptyset$;
8 **foreach** $r_f \in \mathcal{F}$ **do**
9 | Create a tumbling window $W_f\langle r_f, r_f\rangle$;
10 | $b \leftarrow$ *Check*(W_f, W, $\{W_1, ..., W_K\}, \lambda$) by Algorithm 4;
11 | **if** $b =$ `true` **then**
12 | | $\mathcal{W}_f \leftarrow \mathcal{W}_f \cup \{W_f\}$;
13 // Remove candidates that are not independent.
14 **foreach** $W_f \in \mathcal{W}_f$ **do**
15 | **if** *there exists* W'_f *s.t.* $W'_f \leq W_f$ **then**
16 | | $\mathcal{W}_f \leftarrow \mathcal{W}_f - \{W_f\}$;
17 **return** *the best* $W_f \in \mathcal{W}_f$ *by applying Theorem 9*;

$c'_f = \sum_{j=1}^{K} \text{cost}(W_j) + \text{cost}(W'_f) + \text{cost}(W) = \sum_{j=1}^{K} n_j \cdot M(W_j, W'_f) + n'_f \cdot M(W'_f, W) + \text{cost}(W)$.

Theorem 9. *Let W_f and W'_f be two* independent *eligible factor windows under "partitioned by" semantics. $c_f \leq c'_f$ iff*

$$\frac{r_f}{r'_f} \geq \frac{\lambda - \frac{r_f}{r_W}}{\lambda - \frac{r'_f}{r_W}}. \tag{5}$$

Here λ has been defined in Equation 4.

Algorithm 5 presents the details of picking the best factor window for a target window W and its downstream windows W_1, ..., W_K, under "partitioned by" semantics. It starts by enumerating all candidates for W_f based on the constraint that r_f must be a common factor of $\{r_1, ..., r_K\}$ *and* a multiple of r_W (lines 3 to 6). It simply returns W if no candidate can be found (line 5). It then looks for candidates of W_f that are beneficial, using Algorithm 4 (lines 7 to 12). It further prunes *dependent* candidates that are dominated by others (lines 14 to 16). Finally, it finds the best W_f by applying Theorem 9 to compare the remaining candidates.

Example 8. *Continuing with Example 7, Algorithm 5 would generate three candidate factor windows $W(10, 10)$, $W(5, 5)$, and $W(2, 2)$, since all of them are beneficial according to Algorithm 4 ($K = 2$ indeed). However, since both $W(5, 5)$ and $W(2, 2)$ cover $W(10, 10)$, these two candidates are removed and $W(10, 10)$ is the remaining, best candidate.*

Time Complexity Analysis of Algorithm 5: Computing λ at line 1 takes $O(K)$ time. Computing r_d at line 3 takes $O(r_{\max} \log r_{\max})$ time using Euclid's algorithm [24], where $r_{\max} = \max\{r_1, ..., r_K\}$. Computing $\mathcal{F}$ at line 6 takes $O(\lceil \frac{r_d}{r_W} \rceil)$ time. Generating candidate tumbling factor windows (lines 7 to 12) takes $O(|\mathcal{F}|)$ time, as each run of Algorithm 4 takes constant time. Pruning dependent candidates (lines 14 to 16) takes $O(|\mathcal{F}|^2)$ time due to pairwise comparison. Finally, finding the best candidate by applying Theorem 9 takes $O(|\mathcal{F}|)$ time. Therefore, the time complexity of Algorithm 5 is $O(K + r_{\max} \log r_{\max} + \lceil \frac{r_d}{r_W} \rceil + |\mathcal{F}|^2 + 2 \cdot |\mathcal{F}|)$. To simplify our analysis, we assume it is dominated by $O(|\mathcal{F}|^2)$. Since $O(|\mathcal{F}|) = O(\lceil \frac{r_d}{r_W} \rceil)$, it follows that the time complexity of Algorithm 5 is $O(\lceil \frac{r_d}{r_W} \rceil^2)$. This is in contrast to the $O(\lceil \frac{s_d}{s_W} \rceil \cdot \lceil \frac{r_{\min}}{s_W} \rceil \cdot K)$ time complexity of Algorithm 2, which finds the best factor window following "covered by" semantics. In a real-world setting, we would expect $\lceil \frac{r_{\min}}{s_W} \rceil > \lceil \frac{r_d}{r_W} \rceil \approx \lceil \frac{s_d}{s_W} \rceil$, in which case Algorithm 5 improves over Algorithm 2 significantly. On the other hand, Algorithm 5 may lose some optimization opportunities due to its reduced search space for candidate factor windows. We only use Algorithm 5 when "covered by" semantics cannot be used to optimize the input aggregate function.

V. Evaluation

We report experimental evaluation results in this section. We observe that (1) the optimized query plan, even without factor windows, can significantly outperform the original query plan in terms of *throughput* [33] (up to 2.5×); (2) with factor windows, the throughput of the optimized query plan can be much higher (up to 16.8×). Moreover, our optimized plans can yield similar, and sometimes much higher, throughput compared to Scotty [48], one state-of-the-art window slicing technique. Meanwhile, our approach has negligible overhead and can scale up smoothly when increasing window-set size.

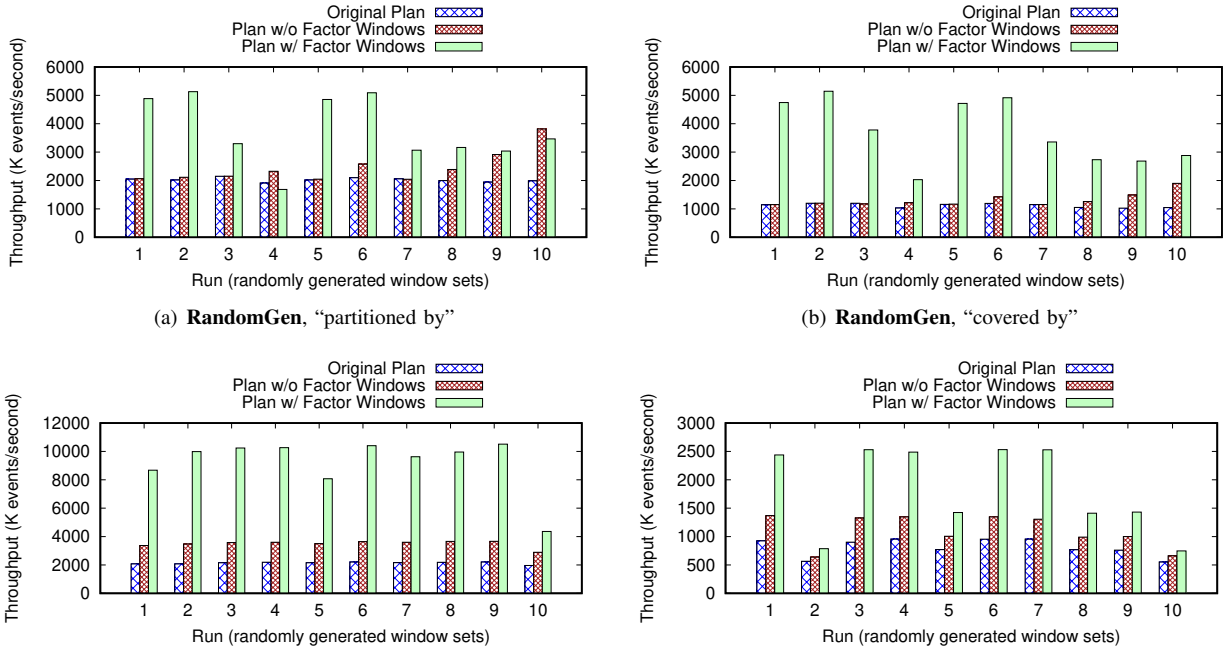
A. Experiment Settings

1) Setup: We implemented our cost-based query optimizer in C#. Given an input window-set aggregate query with its original query plan, it can produce the best query plans, with and without factor windows. All query plans are represented as Trill expressions. For each query plan, we measure its *throughput*, which is defined as the number of events processed per unit time [33]. We perform all experiments on a workstation equipped with 2.2 GHz Intel CPUs and 128 GB main memory. All results are based on single-core executions.

2) Data Sets: We used both synthetic and real data. For synthetic data, we generated data streams with 1 million and 10 million events, denoted as **Synthetic-1M** and **Synthetic-10M** respectively, where the events arrive at a constant pace. For real data, we used the same dataset as used in [17], which was derived from the DEBS 2012 Grand Challenge [32] dataset that consists of monitoring data from manufacturing equipment. Specifically, we pair the given timestamps with the values of the column `mf01`, i.e., the "electrical power main-phase 1" sensor reading. This dataset contains roughly 32 million events and is denoted as **Real-32M**. We used "`MIN`" as the aggregate function, which can be supported by both "covered by" and "partitioned by" semantics.

3) Generation of Window Sets: We generated window sets using the following approaches.

- **(RandomGen)** We generate each window $W\langle r, s\rangle$ randomly. Specifically, to generate a tumbling window where $s = r$, we first pick a "seed" range r_0 uniformly randomly



(a) **RandomGen**, "partitioned by"

(b) **RandomGen**, "covered by"

(c) **SequentialGen**, "partitioned by"

(d) **SequentialGen**, "covered by"

Fig. 11. Throughput on window sets when processing 10 million input events from **Synthetic-10M** with $|\mathcal{W}| = 5$.

Algorithm 6: The **RandomGen** window-set generator.

Input: S, the "seed" slides; R, the "seed" ranges; k_s, k_r: the multipliers; N, the size of the window set; *tumbling*: whether each window is tumbling or not.
Output: $\mathcal{W}$, the window set generated.

```
W ← ∅;
for 1 ≤ i ≤ N do
    if tumbling then
        r_0 ←Random(R);
        r ←Random({2r_0, ..., k_r · r_0});
        W ← W ∪ {W⟨r, r⟩};
    else
        s_0 ←Random(S);
        s ←Random({2s_0, ..., k_s · s_0});
        W ← W ∪ {W⟨2s, s⟩};
return the window set W;
```

from a given list and then choose r uniformly randomly from $\{2r_0, ..., k_r \cdot r_0\}$. We purposely avoid choosing $r = r_0$ to test the effectiveness of our cost-based optimizer when exploring factor windows, as $W\langle r_0, r_0\rangle$ is a valid factor window in this case that should be considered by the optimizer. To generate a hopping window, we operate in a similar manner by first picking a "seed" slide s_0 uniformly randomly from a given list and then choosing s uniformly randomly among $\{2s_0, ..., k_s \cdot s_0\}$; we finally set $r = 2s$ and return $W\langle r, s\rangle$. Algorithm 6 summarizes this procedure.

- **(SequentialGen)** In practice, the windows contained by a window set may be more correlated than those generated by **RandomGen**. Here, we focus on a common case that we observed in the real world, where the windows follow a "sequential" pattern in terms of either the range or the slide size. We presented such an example in Figure 1. This motivates us to implement the **SequentialGen** window-set generator that aims for capturing this sequential pattern. Specifically, unlike in **RandomGen** where r is randomly selected from $\{2r_0, ..., k_r \cdot r_0\}$ when generating tumbling windows , we simply pick r sequentially following the order $2r_0$, ..., $k_r \cdot r_0$. Similarly, we pick s sequentially following the order $2s_0$, ..., $k_s \cdot s_0$ when generating hopping windows.

B. Results on Synthetic Data

For the parameters in **RandomGen** and **SequentialGen**, we set the window-set size $N \in \{5, 10\}$, the "seed" slides $S = \{5, 10, 20\}$ (only for generating hopping windows, where ranges are fixed as twice the slides), the "seed" ranges $R = \{2, 5, 10\}$ (only for generating tumbling windows), and $k_s = k_r = 50$. For each window-set size N, we generated 10 window sets for both tumbling and hopping windows. We also set $\eta = 1$ in our cost model.

1) Throughput: Figure 11 reports the throughput results observed on **Synthetic-10M** for window sets of size 5 generated by both **RandomGen** and **SequentialGen**. The results on **Synthetic-10M** with window sets of size 10, as well as the results on **Synthetic-1M**, are similar and are included in [51].

*Observations on window sets by **RandomGen**:* (1) For the window sets containing *tumbling* windows, the "partitioned by" semantics were leveraged when constructing the window coverage graph (WCG) and exploring factor windows. As illustrated in Figure 11(a), compared to the original plan, the rewritten plan *without* factor windows can boost throughput by up to 1.9×, whereas the plan *with* factor windows can boost the throughput by up to 2.5×. (2) For the window sets containing *hopping* windows, the general "covered by" semantics were used to create WCG's and factor windows. Figure 11(b) presents the results. We observe similar patterns as we observed on tumbling windows, where factor windows yield significantly larger throughput (by up to 4.3×). (3) In a

Setup	w/o FW (Mean)	w/o FW (Max)	w/ FW (Mean)	w/ FW (Max)
R-5-tumbling	1.21×	1.92×	1.85×	2.54×
R-10-tumbling	1.34×	1.77×	1.88×	3.38×
R-5-hopping	1.18×	1.82×	3.26×	4.29×
R-10-hopping	1.34×	1.71×	3.20×	6.15×
S-5-tumbling	1.63×	1.67×	4.28×	4.81×
S-10-tumbling	1.98×	2.05×	7.91×	**9.38**×
S-5-hopping	1.34×	1.48×	2.17×	2.81×
S-10-hopping	1.58×	1.73×	2.92×	3.79×

TABLE I
SUMMARY OF THROUGHPUT BOOSTS ON **SYNTHETIC-10M**, WHERE 'R' STANDS FOR WINDOW SETS GENERATED BY **RANDOMGEN**, 'S' STANDS FOR WINDOW SETS GENERATED BY **SEQUENTIALGEN**, AND '5' AND '10' ARE THE SIZES OF THE WINDOW SETS GENERATED.

Setup	w/o FW (Mean)	w/o FW (Max)	w/ FW (Mean)	w/ FW (Max)
R-5-tumbling	1.19×	1.78×	1.43×	1.91×
R-10-tumbling	1.30×	1.71×	1.53×	2.86×
R-5-hopping	1.09×	1.39×	1.54×	2.63×
R-10-hopping	1.18×	1.39×	1.46×	3.53×
S-5-tumbling	1.63×	1.67×	4.12×	4.85×
S-10-tumbling	1.90×	1.97×	7.53×	**9.14**×
S-5-hopping	1.12×	1.30×	1.22×	1.77×
S-10-hopping	1.22×	1.51×	1.45×	2.31×

TABLE II
SUMMARY OF THROUGHPUT BOOSTS ON **REAL-32M**, WITH THE SAME NOTATION AS IN TABLE I.

couple of cases, the optimized plans are slightly worse than the original plans. This is possible, since our cost model does not use throughput as the cost metric. However, such cases are rare based on our evaluation, and in [51] we show that our cost metric is highly correlated with throughput.

*Observations on window sets by **SequentialGen**:* The observations are similar to those on window sets generated by **RandomGen**. Again, using factor windows significantly boosts the throughput (by up to 4.8× and 2.8× for "partitioned by" and "covered by" semantics, respectively). We further notice that the rewritten query plans *without* factor windows are more effective than they were in the case of **RandomGen**. This is not surprising, though, as the improved correlation between windows generated by **SequentialGen** leads to more overlaps and thus more sharing opportunities.

Summary: In Table I, we summarize the mean and max throughput boosts of the rewritten query plans (without and with factor windows) over the original query plans, observed when processing **Synthetic-10M** under different experimental setups for window-set generation. With factor windows, we can achieve up to 9.4× throughput boost on **Synthetic-10M**.

C. Results on Real Data

We further tested the throughput of window sets over the real dataset **Real-32M**. Table II summarizes the results on throughput boosts of the rewritten query plans, without and with factor windows, over the original plans, and the details are included in [51]. Overall, using factor windows can achieve throughput boost up to 9.1× over **Real-32M**.

D. Scalability Tests

To understand the scalability of our cost-based optimization approach, we increased the window-set size $|\mathcal{W}|$ to 15 and 20. Table III summarizes the throughput results on **Synthetic-10M**; the details are in [51]. Overall, the query plans generated by our approach scale up smoothly when increasing the window-set size, with throughput boost up to 16.8×.

Setup	w/o FW (Mean)	w/o FW (Max)	w/ FW (Mean)	w/ FW (Max)
R-15-tumbling	1.55×	1.96×	2.97×	4.34×
R-20-tumbling	1.49×	2.29×	2.10×	4.83×
R-15-hopping	1.55×	1.95×	4.67×	6.59×
R-20-hopping	1.68×	2.20×	4.23×	7.65×
S-15-tumbling	2.43×	2.49×	11.29×	13.83×
S-20-tumbling	2.42×	2.53×	14.28×	**16.82**×
S-15-hopping	1.85×	2.09×	3.51×	4.68×
S-20-hopping	1.91×	2.15×	4.02×	5.32×

TABLE III
SUMMARY OF RESULTS ON SCALABILITY TEST WITH $\mathcal{W} \in \{15, 20\}$, IN TERMS OF THROUGHPUT BOOSTS ON **SYNTHETIC-10M**. THE NOTATION HERE IS THE SAME AS IN TABLES I AND II.

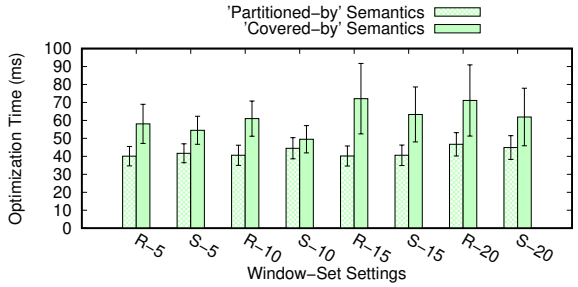


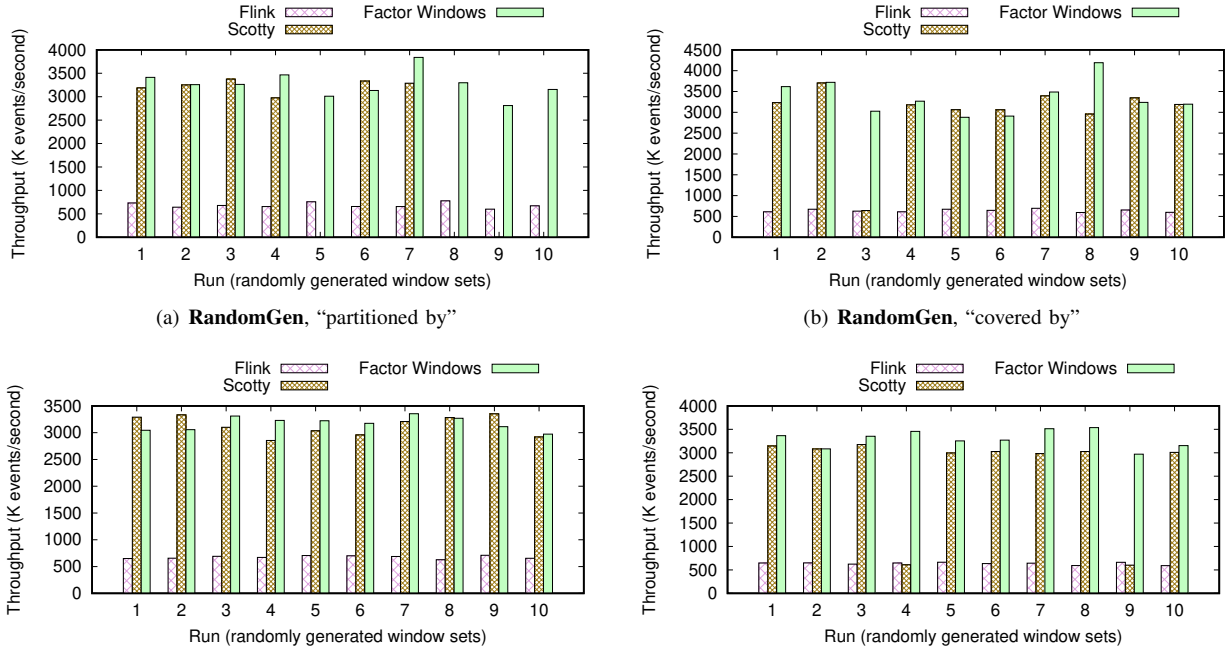
Fig. 12. Factor-window based optimization overhead (average time and standard deviation) with increasing window-set size from 5 to 20. 'R' and 'S' are shorthands for "**RandomGen**" and "**SequentialGen**".

E. Query Optimization Overhead

Figure 12 presents the average time spent on query optimization and its standard deviation (shown with error bars), when enabling factor windows and varying window-set size from 5 to 20. For each setting, the average and standard deviation were measured based on the 10 window sets generated by either **RandomGen** or **SequentialGen**. We observe that the optimization overhead is very small overall (<100 milliseconds for the settings that we tested). Moreover, the optimization overhead of "covered by" semantics is higher than that of "partitioned by" semantics. This makes sense considering the larger search space with "covered by" semantics.

F. Comparison with Window Slicing

We compare our cost-based optimization approach with Scotty [48], one state-of-the-art window slicing technology. Since Scotty does not support Trill, we translate our optimized query plans into Apache Flink queries expressed by its DataStream API [2], following a similar query rewriting procedure described in Section III-C. We compare the throughput of Flink, Scotty, and our optimized plans with factor windows, using the same data generator developed by Scotty for benchmarking its own performance [5], [48]. In our experiments we set the window-set size $|\mathcal{W}| \in \{5, 10\}$. We did not further increase $|\mathcal{W}|$ since Scotty cannot process some window sets with $|\mathcal{W}| = 10$ (see Figure 13(a)). Figure 13 summarizes the results with $|\mathcal{W}| = 10$. The results with $|\mathcal{W}| = 5$ are in [51].



(a) **RandomGen**, "partitioned by"

(b) **RandomGen**, "covered by"

(c) **SequentialGen**, "partitioned by"

(d) **SequentialGen**, "covered by"

Fig. 13. Comparison with Scotty [48] in terms of throughput on window sets with $|\mathcal{W}| = 10$.

We have two observations. First, Scotty and our factor-window based optimization significantly outperform the default Flink query execution plan, where each window aggregate is evaluated independently. Second, our approach yields similar, and sometimes much higher, throughput than Scotty. This holds for both window sets generated by **RandomGen** (Figures 13(a) and 13(b)) and **SequentialGen** (Figures 13(c) and 13(d)), where we observe up to 5.7× throughput boost (excluding cases where Scotty's throughput is unavailable).

VI. Related Work

The related work on stream query processing and optimization is overwhelming (see [31] for a survey). We focus our discussion on optimization techniques dedicated to window aggregates [19], [37]. In addition to the *window slicing* techniques discussed in the introduction (e.g. [20], [29], [30], [35], [36], [44], [47], [48]), there has been a flurry of recent work that accelerates window aggregation via better utilization of modern hardware, such as Grizzly [28] and LightSaber [45]. This line of work is orthogonal to ours.

Cost-based query optimization is the standard practice in batch processing systems [43], but is not popular in stream processing systems. There is little work on cost modeling in the streaming world [50]. One reason might be the difficulty of defining a single cost criterion, as streaming systems may need to honor various performance metrics simultaneously, such as latency, throughput, and resource utilization [22]. Although the application of static cost-based query optimization is limited [12], dynamic query optimization (a.k.a., adaptive query processing) at runtime has been extensively studied in the context of streaming (e.g., [11], [15], [25], [26], [38]–[41], [49]). Our current cost model is static and it is interesting future work to investigate how to dynamically adjust cost estimates at runtime by keeping track of the input event rates.

In recent years, a number of distributed streaming systems have been built as open-source or proprietary software (e.g., Storm [46], Spark Streaming [10], Flink [18], MillWheel [7], Dataflow [8], Quill [21], etc.). While most of these systems provide users with *imperative* programming interfaces, the adoption of *declarative*, SQL-like query interfaces [9], similar to the one that ASA exposes, has been increasingly popular. For example, both Spark Streaming and Flink now support SQL queries on top of data streams. Moving to the declarative interface raises the level of abstraction and enables compile-time query optimization. The optimization techniques proposed in this paper can be implemented in either imperative or declarative systems. We demonstrated the latter for the ASA SQL query compiler (Section III-C), but our algorithms are not tied to the ASA SQL language.

VII. Conclusion

We proposed a cost-based optimization framework to optimize the evaluation of aggregate functions over multiple correlated windows. It leverages the window coverage graph (WCG) that we introduced to capture the inherent overlapping relationships between windows. We introduced factor windows into the WCG to help reduce the overall computation overhead. Evaluation results show that the optimized query plans can significantly outperform the original plans in terms of throughput, especially when factor windows are enabled, without the need for runtime support from stream processing engines.

Acknowledgments: We would like to thank the anonymous reviewers, Badrish Chandramouli, Jonathan Goldstein, and Yinan Li for their valuable feedback.

REFERENCES

[1] Amazon kinesis. https://aws.amazon.com/kinesis/.

[2] Apache flink datastream api. https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/datastream/overview/.

[3] Azure iot central. https://azure.microsoft.com/en-us/services/iot-central/.

[4] Azure stream analytics. https://azure.microsoft.com/en-us/services/stream-analytics/.

[5] Github repository of scotty. https://github.com/TU-Berlin-DIMA/scotty-window-processor.

[6] Google cloud dataflow. https://cloud.google.com/dataflow/.

[7] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11), 2013.

[8] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[9] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2), 2006.

[10] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In *SIGMOD*, 2018.

[11] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[12] A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, 2004.

[13] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.

[14] L. Benson, P. M. Grulich, S. Zeuch, V. Markl, and T. Rabl. Disco: Efficient distributed window aggregation. In *EDBT*, 2020.

[15] P. A. Bernstein, T. Porter, R. Potharaju, A. Z. Tomsic, S. Venkataraman, and W. Wu. Serverless event-stream processing over virtual actors. In *CIDR*, 2019.

[16] J. Byrka, F. Grandoni, T. Rothvoß, and L. Sanità. An improved lp-based approximation for steiner tree. In *STOC*, pages 583–592, 2010.

[17] W. Cai, P. A. Bernstein, W. Wu, and B. Chandramouli. Optimization of threshold functions over streams. *Proc. VLDB Endow.*, 14(6), 2021.

[18] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[19] P. Carbone, A. Katsifodimos, and S. Haridi. Stream window aggregation semantics and optimization. In S. Sakr and A. Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019.

[20] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. Cutty: Aggregate sharing for user-defined windows. In *CIKM*, pages 1201–1210, 2016.

[21] B. Chandramouli, R. C. Fernandez, J. Goldstein, A. Eldawy, and A. Quamar. Quill: Efficient, transferable, and rich analytics at scale. *PVLDB*, 9(14):1623–1634, 2016.

[22] B. Chandramouli, J. Goldstein, R. S. Barga, M. Riedewald, and I. Santos. Accurate latency estimation in a distributed event processing system. In *ICDE*, pages 255–266, 2011.

[23] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[25] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.

[26] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 10(12):1825–1836, 2017.

[27] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[28] P. M. Grulich, S. Breß, S. Zeuch, J. Traub, J. von Bleichert, Z. Chen, T. Rabl, and V. Markl. Grizzly: Efficient stream processing through adaptive query compilation. In *SIGMOD*, pages 2487–2503.

[29] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *CIKM*, pages 1515–1524, 2011.

[30] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, pages 929–940, 2012.

[31] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, 2013.

[32] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The DEBS 2012 grand challenge. In *DEBS*, pages 393–398, 2012.

[33] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *ICDE*, pages 1507–1518, 2018.

[34] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

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

[36] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[37] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.

[38] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. *PVLDB*, 11(10):1303–1316, 2018.

[39] R. V. Nehme, E. A. Rundensteiner, and E. Bertino. Self-tuning query mesh for adaptive multi-route query processing. In *EDBT*, pages 803–814, 2009.

[40] R. V. Nehme, K. Works, C. Lei, E. A. Rundensteiner, and E. Bertino. Multi-route query processing and optimization. *J. Comput. Syst. Sci.*, 79(3):312–329, 2013.

[41] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, pages 353–364, 2003.

[42] G. Robins and A. Zelikovsky. Improved steiner tree approximation in graphs. In *SODA*, pages 770–779, 2000.

[43] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[44] K. Tangwongsan, M. Hirzel, S. Schneider, and K. Wu. General incremental sliding-window aggregation. *PVLDB*, 8(7):702–713, 2015.

[45] G. Theodorakis, A. Koliousis, P. R. Pietzuch, and H. Pirk. Lightsaber: Efficient window aggregation on multi-core processors. In *SIGMOD*, pages 2505–2521.

[46] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In *SIGMOD*, pages 147–156, 2014.

[47] J. Traub, P. M. Grulich, A. R. Cuellar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In *EDBT*, pages 97–108, 2019.

[48] J. Traub, P. M. Grulich, A. R. Cuellar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Scotty: General and efficient open-source window aggregation for stream processing systems. *ACM Trans. Database Syst.*, 46(1):1:1–1:46, 2021.

[49] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, pages 374–389, 2017.

[50] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.

[51] W. Wu, P. A. Bernstein, A. Raizman, and C. Pavlopoulou. Cost-based query rewriting techniques for optimizing aggregates over correlated windows. *CoRR*, abs/2008.12379, 2020.

[52] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.