Cutty: Aggregate Sharing for User-Defined Windows

Paris Carbone† Jonas Traub‡ Asterios Katsifodimos‡
Seif Haridi† Volker Markl‡

†KTH Royal Institute of Technology
{parisc,haridi}@kth.se

‡ Technische Universität Berlin & DFKI
firstname.lastname@tu-berlin.de

ABSTRACT

Aggregation queries on data streams are evaluated over evolving and often overlapping logical views called *windows*. While the aggregation of periodic windows were extensively studied in the past through the use of aggregate sharing techniques such as Panes and Pairs, little to no work has been put in optimizing the aggregation of very common, *non-periodic* windows. Typical examples of non-periodic windows are puctuations and sessions which can implement complex business logic and are often expressed as user-defined operators on platforms such as Google Dataflow or Apache Storm. The aggregation of such non-periodic or user-defined windows either falls back to expensive, best-effort aggregate sharing methods, or is not optimized at all.

In this paper we present a technique to perform efficient aggregate sharing for data stream windows, which are declared as user-defined functions (UDFs) and can contain arbitrary business logic. To this end, we first introduce the concept of User-Defined Windows (UDWs), a simple, UDF-based programming abstraction that allows users to programmatically define custom windows. We then define semantics for UDWs, based on which we design Cutty, a low-cost aggregate sharing technique. Cutty improves and outperforms the state of the art for aggregate sharing on single and multiple queries. Moreover, it enables aggregate sharing for a broad class of non-periodic UDWs. We implemented our techniques on Apache Flink, an open source stream processing system, and performed experiments demonstrating orders of magnitude of reduction in aggregation costs compared to the state of the art.

1. INTRODUCTION

Data stream analytics are becoming increasingly important for a variety of use cases in the industry and science, powering dashboards with continuous insights, approximate computations and complex stateful applications. Queries on unbounded data are typically evaluated over finite logical views of incoming streams, called *windows*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM '16 October 24 - 28 2016, Indianapolis, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN FIXME.

DOI: FIXME

Windows are a first class citizen of virtually all stream processing systems [1, 4]. Those systems typically support a set of simple, predefined primitives to construct time- and count-based windows of various forms (e.g., sliding, tumbling, hopping). This set of primitives can serve a wide variety of use cases, however, it is too restricted to support more advanced ones such as dynamic, data-driven windows. In contrast, modern UDF-heavy streaming systems do not provide any windowing primitives [2, 20], and force their users to define discretization as user-defined operators where they can encode very complex business logic. The main challenge that comes with arbitrary user-defined operators, is that they hide their semantics from the system and hinder optimization opportunities for efficient execution.

Aggregation queries over sliding windows are one of the most redundancy-prone operations in stream processing, as multiple aggregations often share underlying data (i.e., the slide) and thus computations. Seminal works in the past focused on reducing redundancy in computing aggregates on overlapping *periodic* and statically defined windows [16, 17, 18]. These techniques, however, do not apply to broad classes of windows which include punctuations [10], snapshots [11], sessions [3], etc. The main reason that the existing aggregate sharing techniques fall short in aggregating non-periodic windows is the lack of clear semantics, as well as the user-defined code in their implementation. As a result, aggregation of non-periodic windows relies on semantics-agnostic, best-effort general aggregation techniques [5, 22] at very high memory and computational costs.

The objective of this work is twofold: first we aim at giving full flexibility to users by allowing them to implement user-defined stream discretizations. Second, since user-defined discretizations hide their semantics and hinder optimization, we aim at analyzing and exposing the primitives needed to optimize and efficiently execute window aggregations.

To this end, we propose a programming model which enables expressing windows through UDFs and introduce the concept of User-Defined Windows (UDWs). We then exploit certain underlying properties of UDWs and devise a novel aggregate sharing technique that is applicable, not only to known periodic window classes, but to a broader class of windows with less space and computational costs compared to the state of the art. We implemented support for UDWs, their semantics, as well as our aggregate sharing technique on Apache Flink [8], a distributed dataflow processing system as a complete framework for streaming aggregations.

Contributions. The contributions of this paper are summarized as follows:

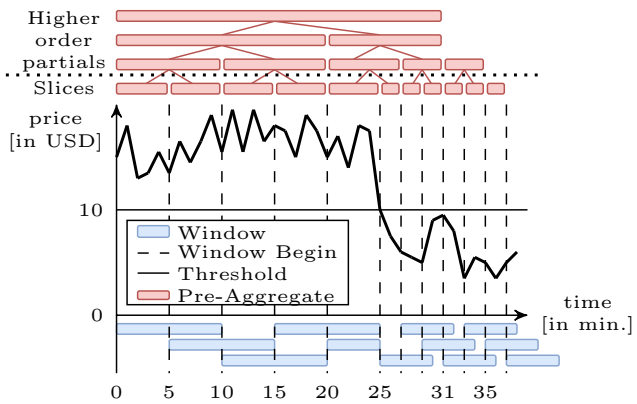


Figure 1: A dynamic window example: Reports become more frequent when the value of a stock is below 10$.

- We introduce the concept of *user-defined windows*, a programming model that allows users to define custom discretizations, allowing windows beyond simple periodic time- and count-based.
- We identify and define the semantics of a restricted but very broad family of user-defined windows which we term *deterministic*, over which window aggregates can be computed efficiently. Deterministic UDWs subsume the periodic ones.
- We introduce Cutty, a general aggregation framework for UDWs that combines discretization with aggregation to enable efficient aggregate sharing across multiple queries.
- We provide analytical and experimental results showing that our aggregate sharing technique, exhibits speedups of orders of magnitude for deterministic UDWs, compared to the state of the art.

A Motivating Example. Consider a monitoring application for stock quotes that continuously receives records representing stock trades. Each record contains a volume (how many units where traded) and a price, per traded unit. A stock trader wants to see the volume-weighted average price of a stock over the last 10 minutes, reported every 5 minutes. However, when the stock's price falls below a certain threshold (e.g., the trader can buy the stock in a low price), the trader wants to receive an update every 2 minutes, with a weighted price average of the last 5 minutes. An example of such a monitoring dashboard is depicted in Figure 1. As specified by the trader, when the price falls below \$10 on the 25^{th} minute, the slide becomes more frequent (every 2min) and the window range becomes shorter (5min). More formally, the window definition goes as follows:

$$\text{window} = \begin{cases} \texttt{SLIDE=5min; RANGE=10min} \; if \; \text{price} > \$10 \\ \texttt{SLIDE=2min; RANGE=5min} \; if \; \text{price} \leq \$10 \end{cases}$$

This is a simple example of a data-driven, user-defined window (UDW) that dynamically changes its range and slide according to the incoming stream. The semantics of such windows have not been defined or supported by systems in the past. The main idea behind the aggregate sharing technique presented in this paper is that neither discretization nor aggregation need to understand the semantics of a UDW. Instead, each UDW needs to simply specify, for each record in the stream, whether that record marks the *begin* or the *end* of a window. The rest of the UDW logic can be *arbitrarily complex*. Our aggregator uses this information to start a new partial aggregate each time a window begins (dashed vertical lines in the example of Figure 1), resulting in a *minimal* set of non-overlapping partial aggregates, called *slices*. The final aggregate for any window is computed using the slices, and possibly higher order aggregates.

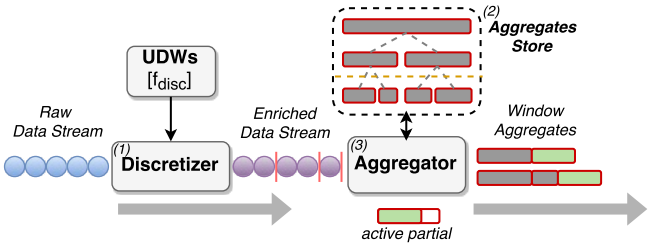


Figure 2: Architectural overview of Cutty.

Solution Overview. Our aggregator contains three main components, as depicted in Figure 2:
1) The *discretizer* enriches the incoming data stream with windowing information dictated by a set of User-Defined Window functions which share the same aggregation (e.g., SUM, AVG). The enriched stream drives the pre-aggregation decisions and abstracts necessary window bookkeeping from the aggregator. 2) The *aggregation store* which is built upon [22], maintains and pre-computes higher level partials allowing efficient aggregate look-ups for arbitrary window intervals. 3) The aggregator consumes the enriched stream and implements the core shared aggregation functionality for all types of UDWs.

The rest of this paper is organized as follows: In Section 2 we present preliminaries that the reader needs in order to follow the rest of the paper. Then, we introduce the semantics needed for efficient slicing of UDWs in Section 3. Section 4 describes an intuitive end-to-end example of our approach. We then describe the design and internals of Cutty, our aggregate sharing technique in Section 5. In Section 6 we provide an analytical comparison of Cutty contrary to the state of the art. In Section 7 we present our experimental results, and discuss the related work in Section 8. Finally, we conclude and present future work in Section 9.

2. PRELIMINARIES

Before we describe UDWs and our aggregate sharing technique, it is worthwhile to provide an overview of our data model and operators on which we base our work. For completeness of presentation, we then outline the main ideas behind window aggregate sharing and stream slicing.

2.1 Data Model and Operations

Streams and Substreams. A stream consists of records derived from a type T. In our model, a *data stream* $\overline{s}$ is a *sequence* $\overline{s} \in Seq(T)$ where $Seq(T)$ is the set of all sequences that can be derived over T. Furthermore, we denote by $s_i = \overline{s}(i)$ the element at position i in data stream $\overline{s}$.

Since a stream is conceptually infinite, we use intervals to describe finite subsequences out of $\overline{s}$. An interval $R = [a, b]$ is a set of integers from a to b, $a \leq b$. $\overline{s}(R)$ is a *subsequence* of $\overline{s}$ where $\overline{s}(R) = \{s_i | i \in R\}$. In the following, we refer to $\overline{s}(R)$ as a *substream* and use $s[a, b]$ as a shorthand notation for $\overline{s}([a, b])$. We further denote the set of all substreams of type T as $Str(T)$, where $Str(T) \subset Seq(T)$.

Operators. A `Discretize` operator transforms a stream $\overline{s} \in Seq(T)$ into a sequence $\overline{w} \in Seq(Str(T))$ of (possibly overlapping) windows, where a window is a *substream* $w_i = s[l_i, r_i]$. In the following, we refer to an output sequence $\overline{w}$ as a *discretized stream* and describe the discretization using a function f_{disc} (further analyzed in Section 3):

$$\texttt{Discretize} : f_{disc} \times Seq(T) \rightarrow Seq(Str(T))$$

Windows can overlap. However, they should maintain a FIFO order. More formally, for a pair of windows $w_i, w_j \in \overline{w}$, if $i \leq j$ then $l_i \leq l_j$ and $r_i \leq r_j$.

The `Aggregate` operator maps each window in the discretized stream to an aggregate value, given an aggregation function f_a.

$$\texttt{Aggregate} : (f_a : Str(T) \rightarrow T') \times Seq(Str(T)) \rightarrow Seq(T')$$

Examples of f_a are `SUM` and `AVG`. In the following section, we further analyze window aggregations.

2.2 Window Aggregates

Windows are finite but can be arbitrarily long. Thus, it is generally preferable to evaluate an aggregation incrementally. In this section we will briefly explain the internals of incremental and partial aggregation of overlapping windows.

Partial Aggregation. An aggregation f_a can be decomposed into partial aggregates. We denote by A the type of partial aggregates. For example, in the case of a `SUM` aggregation $A = \mathbb{R}$. A window aggregation function can be reformulated as two functions, `lift` and `lower`, and a `combine` operator $\oplus$ [23]. The function $\texttt{lift} : T \rightarrow A$ maps an element of a window to a partial aggregate. The combine operator $\oplus : A \times A \rightarrow A$ combines two partial aggregates into a new partial aggregate. Finally, $\texttt{lower} : A \rightarrow T'$ maps a partial aggregate into an element in the type T' of output values. We assume that `combine` is *associative* and that the partial aggregation values that are combined are bounded as in most other works [5, 16, 22]. By using the above functions and operator, elements in the window are lifted to partial aggregates, further combined starting from the default (identity value) 1_A for type A, and finally mapped to an output aggregate value. For example, for window $s[1,3]$ the aggregation would be unrolled as:

$$f_a(s[1,3]) = \texttt{lower}(((1_A \oplus \texttt{lift}(s_1)) \oplus \texttt{lift}(s_2)) \oplus \texttt{lift}(s_3))$$

Generally, we denote the partial aggregate of a substream $s[i,j]$ as $P(s[i,j]) = 1_A \oplus \texttt{lift}(s_i) \oplus \ldots \oplus \texttt{lift}(s_j)$.

Aggregating Overlapping Windows. A naïve execution of an `Aggregate` operator can potentially lead to redundant partial computations. To demonstrate the problem assume we have a discretized stream $\overline{w}$. For any two windows $w, w' \in \overline{w}$ there is potentially an overlap $v = w \cap w' = s([l,r] \cap [l',r'])$. Computing all partials over $\overline{w}$ would yield:

$$\begin{aligned} P(\overline{w}) &= \ldots \cup P(w) \cup P(w') \cup \ldots \\ &= \ldots \cup (P(w \setminus v) \oplus P(v)) \cup (P(v) \oplus P(w' \setminus v)) \cup \ldots \end{aligned}$$

It is clear from the formulation that if $v \neq \emptyset$, the redundant work done would be at least $P(v)$. The same redundancy also applies when executing multiple aggregate queries on a shared data stream. For example, let $\overline{w}$ and $\overline{w}'$ be two discretized streams computed over a shared data stream $\overline{s}$. For windows $w \in \overline{w}$ and $w' \in \overline{w}'$ with overlap $w \cap w' = v$ and the same aggregation function, the amount of redundant work to compute $P(w \cup w')$ would also be at least $P(v)$.

2.3 Stream Slicing

We seek to apply partial aggregation on a data stream $\overline{s}$ and derive a set of partial aggregates I, which can be used to compose full window aggregations. This way, instead of keeping all records of active[1] windows in memory, we only keep their partials. This technique has been studied in the past in the context of periodic windows and is known as *slicing* [16, 17]. Slicing guarantees that for any active window $w = s[begin, end]$ there will be a sequence of partials over contiguous intervals from *begin* to *end*. For instance, consider windows $s[1,3]$ and $s[2,7]$ for which we want to apply slicing. We can derive $f_a(s[1,3])$ and $f_a(s[2,7])$ from a set I of three shared sliced partials as follows:

$$I = \langle \overbrace{P(s_1)}^{f_a(s[1,3])=\texttt{lower}((1_A \oplus P(s_1)) \oplus P(s[2,3]))}, \underbrace{P(s[2,3]), P(s[4,7])}_{f_a(s[2,4])=\texttt{lower}((1_A \oplus P(s[2,3])) \oplus P(s[4,7]))} \rangle$$

3. USER-DEFINED WINDOW SEMANTICS

There have been various approaches to define discretization semantics on data streams [4, 7, 18]. In this work, we aim at exposing the right core primitives that can enable efficient shared aggregation without limiting window expressivity. We outline the observations that drove the design of our windowing semantics and then define and characterize semantics for user-defined windows.

3.1 Problem Definition and Intuition

The objective of this work is to apply efficient aggregate sharing for user-defined windows. Stream slicing allows sharing of partial aggregates with the least memory requirements, since slices are built from non-overlapping substreams. Ideally, a stream can be discretized into a minimal set of slices, as dictated by the window specification, in order to derive all needed window aggregations. Before we continue, we answer two fundamental questions:

– *What is the smallest set of slices of a stream which suffices to compute aggregates over a set of windows?* Intuitively, we can start pre-aggregating an active partial slice incrementally as we consume records until we reach a record s_i that marks the beginning of a window. At that point we can store a copy of our active partial and start a new one, which can be used later for the window that begins at s_i. When we reach a record that marks the end of a window we can combine the current active partial slice, together with previously stored slices to get the aggregate of the full window. Thus, at any given point we need to maintain no more slices than there are active windows.

– *Which classes of windows support slicing*? We argue that the semantics of windows do not have to be known in advance. We base the rest of this paper on the observation that minimal slicing can be applied by knowing whether a record marks the beginning of a window or not. In the following section we use this observation, in order to design semantics for UDWs.

[1]Windows are active when they have not yet been emitted.

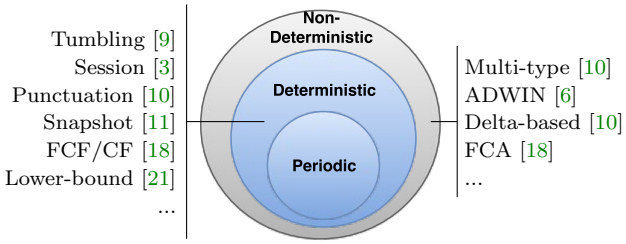


Figure 3: Overview of windowing classes.

3.2 User-Defined Discretization

We distinguish two classes of UDWs, namely *deterministic* and *non-deterministic*. In short, deterministic are all windows for which we can apply slicing efficiently as described previously, while non-deterministic are all the rest for which we cannot. Deterministic windows can be declared with a *discretization function* f_{disc}.

Intuitively a window function is deterministic if at the time that it processes a record, it can decide whether that records marks i) the beginning of a window or ii) the end of a window. For instance, a periodic count window of fixed range and slide, is deterministic, since when a record arrives, an internal counter can affirm whether a new window begins with that record. Similarly, a punctuation window is deterministic, since (by definition) a punctuation marks the beginning of a window. Formally, a deterministic window function f_{disc} is defined as follows:

$$f_{disc} : T \rightarrow \langle W_{begin} : \mathbb{N}, W_{end} : \mathbb{N} \rangle$$

where for a record $r \in T$: *i)* W_{begin} is the number of windows beginning with r and *ii)* W_{end} the number of windows ending upon processing r. For example, consider the following deterministic function for declaring periodic count windows of fixed range and slide:

$$f_{disc}(s_k) = \begin{cases} W_{begin} : 1 \; if \; \text{k } \texttt{mod slide} = 0, \text{ else } 0 \\ W_{end} : 1 \; if((k - \texttt{range}) \; \texttt{mod slide}) = 0, \text{ else } 0 \end{cases}$$

Periodic windows are supported by most existing event processing systems (e.g. CQL [4]), and are trivially subsumed by deterministic. Figure 3 depicts an overview of the window classes along with a categorization of known window types found in the literature. Tumbling windows are periodic and thus, deterministic. Session and snapshot windows by definition begin with the first record of the session that marks the window's beginning and end a timeout record which is injected in the system (e.g., a watermark). Moreover, lower-bound landmark windows begin from a given landmark record, that marks the beginning of a record and end upon a punctuation or a predefined length.

Non-Deterministic Windows. Intuitively, non-deterministic windows cannot declare immediately whether a record begins a window or not, i.e., they need to examine more records in order to take such a decision. As a result, slicing for non-deterministic windows is not possible and we have to fall back to best-effort techniques such as [5, 22]. A typical example of a window which is non-deterministic is a window which every 5 seconds outputs the last 10 records of the stream. If we assume that a record r_1, arrives during the first second of a window and the next second, another 10 records arrive, r_1 will not be part of the next window. Thus, if the rate of the stream cannot be known in advance, such a window cannot be deterministic; the window function cannot specify whether a record is going to be part of the next window at the very moment of the record's arrival. In this work, we focus on shared aggregation of deterministic windows, for which slicing is applicable.

Implementing UDWs in Practice. To make it easier for users to implement discretization functions, we implemented an API on Apache Flink inspired by IBM SPL [14], allowing users to implement custom *eviction* and *trigger* functions of arbitrary logic. Intuitively, the triggers are used by the system to mark the windows' W_{end}, and evictions are used to mark W_{begin} (used by deterministic functions) or number of evicted items e (used by non-deterministic functions). When the function which users implement cannot return a W_{begin} for all of the records which they process, the system treats them as non-deterministic.

4. DETAILED OVERVIEW OF CUTTY

In this section we describe a general framework for aggregating multiple overlapping windows. This aggregation framework exploits the properties of deterministic windows using Cutty, a novel pre-aggregation technique. Furthermore, it utilizes higher-order pre-aggregated partials which adds sharing capabilities for deterministic windows.

A Thorough Example. Figure 4 depicts a full example of the execution of Cutty for aggregating a set of deterministic UDWs. As the set of the deterministic UDWs dictate (f_{disc}'s at the top), partial aggregation is applied incrementally only within the intervals that mark the beginning of windows (dashed vertical lines). At the beginning of every interval, the active partial resets to the initial value 1_A and maintains the current pre-aggregate until it forms a complete atomic slice (in this example, $(P(s[1,2]), P(s[3,5]), P(s[6,7]), P(s[8,8]))$. Before the active partial resets, its value is stored for further reuse (central rectangle in the figure). When a window ends we have everything to compute the full aggregation from the partials. For instance, in the case of $f_A(s[3,8])$, upon getting notified at the consumption of s_9 that window $s[3,8]$ is complete we are ready to compute the full aggregation. We can derive the full window aggregation by re-using all precomputed slices $P(s[3,5]), P(s[6,7])$ and $P(s[8,8])$ which are already stored.

Sharing Higher-Order Partials. From the example in Figure 4, we observed that, simply sharing sliced partial aggregates does not eliminate redundancy when computing full window aggregations. For example, with a typical lazy evaluation slices $P(s[1,2])$ and $P(s[3,5])$ would have to be combined twice: once for computing $f_a(s[1,5])$ itself, and once for computing $f_a(s[1,6])$. Instead, if $f_a(s[1,5])$ was stored, $f_a(s[1,6])$ could be computed by $f_a(s[1,5])$ together with the current partial. In principle, the higher the number of overlapping windows, the more reduce calls are repeated for each window merging operation. To deal with this issue we can exploit an eager pre-aggregation strategy to incrementally pre-compute higher-level aggregates. Eager pre-aggregation has been used previously to support aggregate look-ups on data streams (e.g. B-Int[5], FlatFAT [22]) at the cost of additional space and update computation requirements. It has been shown, however, that these costs provide a strong trade-off when eager aggregation is applied even in a per-record granularity of a data stream [5, 22].

In our case, we apply this technique for enriching atomic sliced partials with higher order reusable partials, thus, effec-

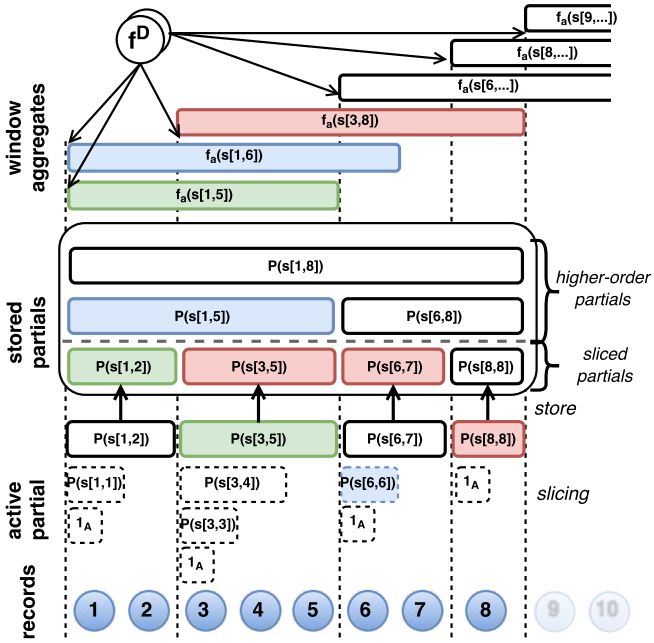


Figure 4: An example of Cutty on deterministic windows.

tively reducing aggregation cost at a low additional memory footprint. Furthermore, eager aggregation can be used as a best-effort fallback solution for non-deterministic UDWs, where slicing cannot be applied. The key idea is that we can *eagerly* evaluate higher-order aggregates and maintain them in efficient data structures in order to support arbitrary look-ups, described in more detail in the next section.

5. SHARED AGGREGATION WITH CUTTY

Incremental aggregation is the core functionality of our sharing strategy. The main component that executes the aggregation is the Cutty aggregator which consumes the enriched stream and pre-aggregates reactively. In the next section, we present the discretizer which injects windowing information into the data stream and then proceed with presenting our shared aggregator.

5.1 Shared Discretization

The shared discretizer is able to multiplex multiple UDWs, coming from multiple queries. The basic functionality of the discretizer is to consult all the UDWs for each incoming record, enrich it with the needed information and pass it downstream to the aggregator.

Deterministic UDWs. As we have seen in Section 3.2, deterministic functions allow us to know whether a record begins or ends a window, exactly at the moment of the record's arrival. This gives the discretizer the ability to give hints to the aggregator, alleviating the aggregation process from the need for window management and bookkeeping. Hence, the aggregator simply operates on a marked stream, without any knowledge about the specifics of the UDWs.

The discretizer associates a set of window begin/end identifiers with each incoming record. These identifiers are used by the aggregator to apply slicing incrementally. Consider for example the shared discretization of windows produced by two UDWs as shown in the example in Figure 5. The upper UDW has a range of 4 and slide 2 while the one in

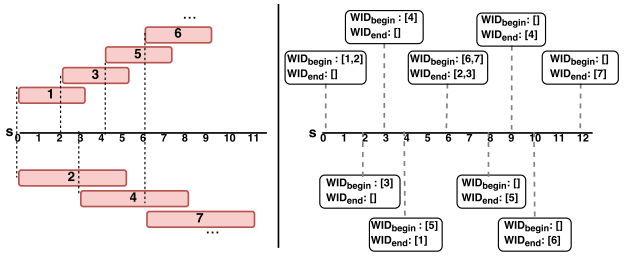


Figure 5: Mapping multiple UDWs to discretization events.

the bottom has a range of 5 and slide 3. The dashed vertical lines mark the begin of each individual window, as indicated by the UDWs. The discretizer consults the UDWs and injects "window begin" markers in the stream (as depicted on the right). For instance, on record 0, there are two window begins, namely windows 1 and 2. Similarly, the discretizer consults the UDWs and enriches the stream with "window end" markers. In our example, window 1 ends with record 3 (and indicated by the UDW upon processing record 4).

Formally, the discretizer injects window identifiers to the data stream as follows:

$$r : T \mapsto \langle r : T, \text{WID}_{begin} \subset \mathbb{N}, \text{WID}_{end} \subset \mathbb{N} \rangle$$

where WID_{begin} and WID_{end} are derived from the UDWs and represent each unique window that, respectively, begins (inclusively) or ends (exclusively) with record r.

Non-Deterministic UDWs. Non-deterministic windows cannot indicate whether the current record of the stream begins a new window. To this end, non-deterministic UDWs have to indicate possibly expired records that should be removed from the head of the current window, and notify when the window has to be emitted. Since the shared discretizer operates on multiple UDWs at the same time, it has to i) derive the records that expire across *all* UDWs i.e., the intersection of records that all UDWs have declared as expired ii) store and track the begin of each active window. Non-deterministic windows in our aggregator architecture are handled by the underlying aggregate store as described in [22]. For the lack of space, we will omit the details of how expired records are handled by our aggregator, and refer the reader to the original work.

5.2 Shared Aggregation

Efficient Aggregate Storage. The aggregator needs to maintain partials in memory and retrieve them efficiently. To this end, we designed an aggregate store that provides support for range queries over the aggregates (e.g., when multiple partials have to be combined for a window emission). The store supports three basic operations:

`append(partial_id, partial)`: Adds a partial at the end of the store where `partial_id` is an identifier for the provided partial.

`merge(from, to)`: Computes result of $P(s[from, to])$. Most of the time the aggregator is interested in looking up a full aggregation starting by `from`. In that case, we will use the shorthand call `merge(from)`.

`removeUpTo(partial_id)`: Removes all given partials from the store up to `partial_id`.

We considered two evaluation strategies for the store, a *lazy* using a circular fixed-sized array, and an *eager* which builds on FlatFAT [22] a pre-allocated memory circular heap-based

Algorithm 1 Agg $\langle partial, 1_A, \oplus, \texttt{lift}, \texttt{lower}, store, begins \rangle$

```
1: upon event ⟨ r_i, WID_begin, WID_end⟩ do
2:     if WID_begin ≠ ∅ then
3:         store.append(i, partial)
4:         partial := 1_A //reset partial
5:          for each  w ∈ WID_begin
6:             begins[w] = i //mark begin for w
7:      for each  w ∈ WID_end
8:         start := begins[w] //retrieve begin of w
9:         begins.remove(w)
10:        store.removeUpTo(min(begins)) //gc
11:        emit ⟨w | lower(store.merge(start) ⊕ partial) ⟩;
12:    partial := partial ⊕ lift(r_i) //online aggregation
13: end
```

data structure. For the rest of this section it should be assumed that the store follows an eager strategy on a binary tree unless stated otherwise. The complexity of storage and retrieval plays a very important role in the performance of our aggregation technique and is analyzed in Section 5.4.

Aggregating Enriched Streams. The functionality of the Cutty aggregator is summarized in Algorithm 1. The aggregator maintains a single active partial on which it applies incremental aggregation per record arrival. Effectively this is an execution of stream slicing, where each slice spreads between records that mark consecutive window begins. In case a new window begins (which occurs when the set WID_{begin} is not empty) the aggregator has to start a new partial pre-aggregation. In that case, the current active partial is stored in the aggregates store and the active partial resets back to its initial value (1_A). In any other case the aggregator simply applies a single `combine` operation to update its active partial. Mind that the aggregator keeps track of the first record of every active window since it has to be aware of the specific range to aggregate when a window ends. For every window that ends in W_{end}, the aggregator combines its active partial with the stored partial of the interval that starts at the beginning of the window.

Storage Costs. A very important feature of Cutty aggregation for deterministic UDWs is that it generates a *minimal* amount of sliced partials needed for any shared window computation. In the worst case, Cutty stores only as many partials as the number of active windows, compared to other slicing techniques [16] that generate twice as many partials. The main idea lies at the observation that only windows that end at a specified record r_i would need to aggregate up to i. Since no other windows would ever need to start or end at this index later, incremental aggregation can continue until it reaches a record which starts a new window.

For non-deterministic window aggregations, it is not possible to apply slicing due to the limited knowledge of the active windows. However, in that case Cutty utilizes the store, thus, bounding its performance to the current state-of-the-art approach [22]. Expired record removals and full window aggregates are executed based on the discretizer-injected information. Partial aggregation sharing is therefore achieved only via the eager strategy of the aggregate store.

5.3 Sharing Across Multiple Queries

The slicing logic of Cutty applies for both single and multiple multiplexed windows. To the best of our knowledge, Cutty is the first general slicing technique that combines deterministic windows defined on different metrics to share sliced pre-aggregate. Using Cutty, all window begins are mapped to a concrete record in the stream at runtime, which is the first record of the window. It doesn't matter which measures are used in the queries to define windows; knowing at which records windows begin, allows us to share pre-aggregates among them. Figure 6 illustrates how slices are created when two queries use different measures, namely time and distance. Imagine a vehicle which produces a stream of reports consisting of the current timestamp and mileage. We now define two periodic sliding window queries, which can share pre-aggregates: *i) Slide* = 6sec.; *Range* = 10sec. (depicted on x-axis) *ii) Slide* = 5km; *Range* = 10km (depicted on y-axis). Note that, for the sake of simplicity, the queries are periodic. However, Cutty can apply slicing and sharing on *any* combination of deterministic windows.

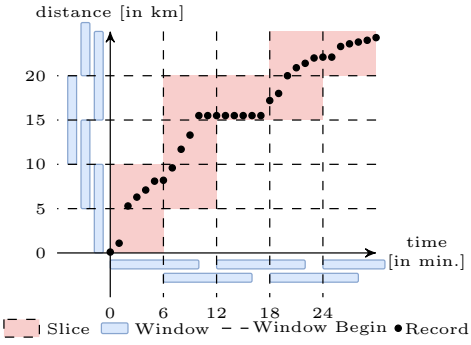


Figure 6: Incremental slicing for multiple queries which define windows over different measures.

5.4 Aggregate Store Internals

The aggregate store is an extension of FlatFAT [22], which we extended to support sharing partial aggregates among multiple queries. As in FlatFAT, partials in our store are stored in a pre-allocated heap-based data structure that is pointer-less. The data structure resembles a "*sliding binary tree*" that pre-computes high level aggregates incrementally. In its circular heap space, single-hop tree traversals (e.g. *parent*, *rightChild*) can be executed in $O(1)$ time without the need for look-ups. We briefly summarize all additions and considerations in the store internals while omitting several non-critical details that can be further studied in [22].

Multi-window Processing. *Cutty* takes care of generating shared partials, produced by slicing multiple deterministic windows accordingly, and then stores them by invoking the `append(partial_id, partial)` operation. Stored partials (as opposed to the records themselves in [22]) serve as the leafs of the tree and are uniquely addressed by their original id. To allow this we enriched the data structure with additional mappings $h(partialID) \rightarrow i$ where $i \in \mathbb{N}$ is the heap index for that partial. When required, the aggregator has to look-up for aggregates on intervals within the partial id space and retrieve them via a `merge(from, to)` operation. Both `merge` and `append` operations employ a bottom-up heap traversal for pre-computing (`merge`) and evaluating an aggregate range (`append`). The traversal yields an upper bound complexity of $O(log_2(n))$ in both cases, where n is the number of leafs. For *Non-deterministic* windows we implement an identical strategy to RA [22], by appending every individual record to the data structure and applying

Table 1: Complexities of Cutty and the state of the art over aggregating a periodic sliding window.

	Space		Update			Merge	
	Lazy	Eager	Lazy	Eager	Eager (Amortized)	Lazy	Eager
Cutty	$\lceil r/s \rceil + 1$	$\lceil 2r/s \rceil + 1$	1	$O(log(\lceil r/s \rceil))$	$^{(log(\lceil r/s \rceil)-1)}/_{s} + 1$	$\lceil r/s \rceil + 1$	$log(\lceil a \rceil)$
Pairs[16]	$\lceil 2r/s \rceil$	$\lceil 4r/s \rceil$	1	$O(log(\lceil 2r/s \rceil))$	$^{2(log(\lceil 2r/s \rceil)-1)}/_{s} + 1$	$\lceil 2r/s \rceil$	$log(\lceil 2r/s \rceil)$
Panes[17]	$\lceil r/_{gcd(r,s)} \rceil$	$\lceil 2r/_{gcd(r,s)} \rceil$	1	$O(log(\lceil r/_{gdc(r,s)} \rceil))$	$log(\lceil r/_{gdc(r,s)} \rceil) + gdc(r,s) - 1$	$\lceil r/_{gcd(r,s)} \rceil$	$log(\lceil r/_{gcd(r,s)} \rceil)$
RA [22]	r	$2r$	1	$O(log(r))$	$log(r)$	r	$log(r)$

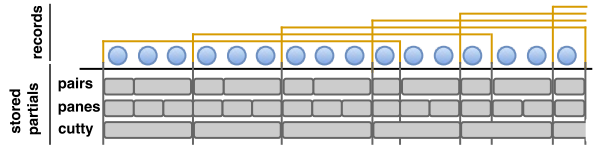


Figure 7: Slicing a count window of range 10 and slide 3.

look-ups in the granularity of records.

Memory Management. When the heap space is fully utilized upon an `append` operation, we double the array's capacity. This is a common strategy, also considered in the original implementation of FlatFAT [22]. Similarly, when a `remove` operation leaves behind an under-utilized heap down to a third of its full capacity, we half its capacity. Each storage resize invokes exactly $2n + 1$ heap operations. That means that if we need to store x partials we would pre-allocate a capacity $n = 2^k$ for $k = min\{m \in \mathbb{R} | m \geq log_2(x)\}$.

Lazy Implementation. The aggregates store also implements a *lazy* aggregation mode. This means that, when operating in this mode, no higher-order partials are precomputed. In this case, only first order partials are stored in the circular buffer (corresponding to leafs in the eager strategy). Thus, `update` and `merge` operations have $O(1)$ and $O(n)$ complexities respectively for n partials. For the sake of simplicity and comparability we preserved the same memory management strategies with the default, eager strategy that were explained above.

6. ANALYTICAL COMPARISON

So far we have introduced different window classes and aggregation techniques for each class. In order to put everything into perspective we will examine worst case and amortized spatial and computational complexities exhibited by Cutty compared to other known techniques in aggregate sharing.

Analysis Scope. We deliberately focus on a single full periodic window aggregation, with a fixed range r and slide s, since this is the single common denominator of all supported window classes. Our evaluation in Section 7 includes experimental analysis of multi-window scenarios. We cover all periodic window-centric pre-aggregation techniques, namely *panes* [17] and *pairs* [16] and the best-known general window aggregation techniques: *B-Int* [5] and *RA* [22], which are briefly described here.

Panes and Pairs. With periodic windows (constant range and slide) it is possible to pre-define all sliced partials on periodic intervals. *Panes* yields partials with a constant size, equal to the greatest common denominator of *range* and *slide*. Alternatively, *pairs* splits a slide into two partials: $p_2 = range \mod slide$ and $p_1 = slide - s_2$. Contrary to *panes*, *pairs* can also incorporate multiple periodic windows. We will examine the multi-query case further in practice in Section 7. Figure 7 depicts all slices generated by each technique for a periodic window of length 10 and slide 3.

General Techniques. General sharing techniques cover window pre-aggregation cases with no periodicity assumptions. The main idea behind RA's FlatFat [22] and B-Int [5] is to maintain a binary tree of higher order partials that "slides" together with the records of the stream. RA's FlatFAT is an adaptation of B-Int on a fixed-size circular heap with dynamic resizing support.

6.1 Complexity Analysis

Table 1 summarizes all complexities, covering worst case memory demands in terms of number of stored partials, as well as update (computation per record) and merge (computation per full window) in respect of reduce calls. We further decouple slicing methods from aggregate store strategies, i.e. eager and lazy. Evidently, general aggregation strategies conceptually slice a data stream by producing a partial per record. From the table it is clear that in all cases, the eager strategy achieves better merge performance while increasing memory and computational demands for updates. Furthermore, Cutty exhibits the most efficient execution both in terms of space and computation. This is because slices in Cutty solely correspond to the number of active windows at any given time. *Pairs*, on the other hand, require double the memory and computational resources as a side-effect of slicing twice more partials. *Panes* is the least efficient technique for periodic windows due to enforcing finer slicing granularity which corresponds to the smallest possible slice during a full window pre-aggregation. Finally, *RA*, as expected, has significantly more memory and computational demands than periodic window pre-aggregation techniques since it is agnostic of window semantics and thus, operates at the granularity of each individual record.

6.2 Amortized Costs

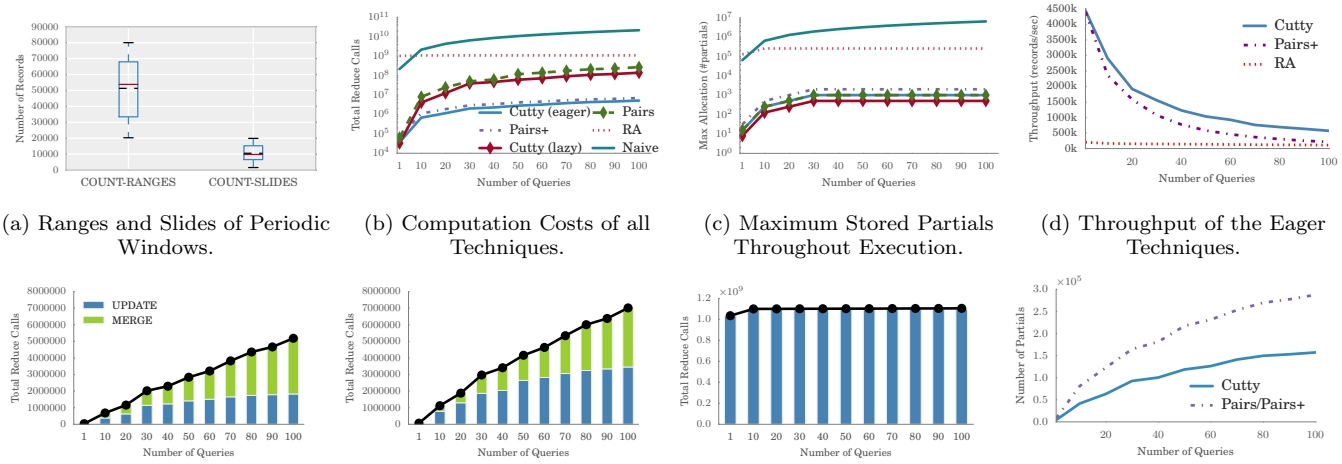
An amortized study of the operation costs of all these techniques with an eager strategy unveils further benefits for Cutty. Intuitively, the worst case update complexity in a full window aggregation applies only when a partial is stored. In the case of Cutty this occurs exactly $^{r}/_{s}$ times. In contrast, *pairs* writes to the store $2^{r}/_{s}$ times while in *panes* and *RA* this occurs exactly $^{r}/_{gcd(r,s)}$ and r times, as depicted in Table 1. Evidently, Cutty exhibits the lowest amortized cost, by invoking costly store operations half of the times compared to *Pairs*.

7. EXPERIMENTAL EVALUATION

In this section, we assess the performance of Cutty compared to *Pairs* [16] and *RA* [5, 22] for periodic and non-periodic windows.

7.1 Experiments Setup

Implementation and Competing Approaches. We implemented the API for user-defined windows and the Cutty aggregator on Apache Flink 0.9 [8]. In order to enable a



(a) Ranges and Slides of Periodic Windows.

(b) Computation Costs of all Techniques.

(c) Maximum Stored Partials Throughout Execution.

(d) Throughput of the Eager Techniques.

(e) Computation Costs Drilldown for Cutty.

(f) Computation costs drilldown for *Pairs+*.

(g) Computation Costs Drilldown for *RA*.

(h) Number of Partials Produced Throughout Execution.

Figure 8: Performance Evaluation for Periodic Queries with respect to different Workload Sizes.

fair comparison, we implemented Pairs [16] and RA [22] within the same codebase, sharing data structures and function calls. More specifically, we implemented the following techniques[2], each of which maps to specific configurations in our framework:

- **Naive:** The *Naive* aggregator does not exploit any window semantics (e.g. periodicity) nor sharing opportunities for multiple windows. It simply recomputes overlapping windows and runs a separate aggregator instance per window query. Thus, the amount of resources utilized is proportional to the number of input queries.
- **Cutty:** The main strategy that is activated in our framework for *deterministic* window functions. Unless stated otherwise, Cutty uses an *eager* aggregate store, i.e., it maintains a tree of partials as described in Section 5.4.
- **Pairs [16]:** *Pairs* is the state-of-the-art pre-aggregation sharing technique for periodic queries. Our implementation of *Pairs* uses a *lazy* aggregate store strategy, as described in the original paper.
- **Pairs+:** This is an enhanced version of the standard *Pairs* technique that is configured with an *eager* aggregate store enabling a fair comparison against Cutty.
- **RA [22]:** The Reactive Aggregator (RA) uses a general incremental aggregation strategy [22] that employs no slicing, relying solely on an *eager* aggregate store (FlatFAT).

Dataset. We used the DEBS12 grand challenge dataset [15] which contains events generated by sensors of a factory. Each record of the dataset comprises of energy metrics and sensor states sampled with 100Hz rate. In total, the dataset contains roughly 33 million events. Each event includes three energy measures and 54 binary sensor-state transitions. We decided to use the DEBS12 dataset as it serves very well to generate non-periodic session-based window queries using sensor transitions as punctuations for the experiments which include deterministic/non-periodic windows (Section 7.3).

[2]The *Panes* technique was excluded because i) it is generalized and subsumed by Pairs and ii) there is no support for multiplexing multiple windows.

Hardware. We executed all experiments on a 4.0 GHz Intel Core i7-4790K with 12GB DDR3 pre-allocated memory on a v1.8.0_91 JVM. Note that the aggregation operator in our experiments utilizes only a single core and is executed on a single task within Apache Flink's runtime.

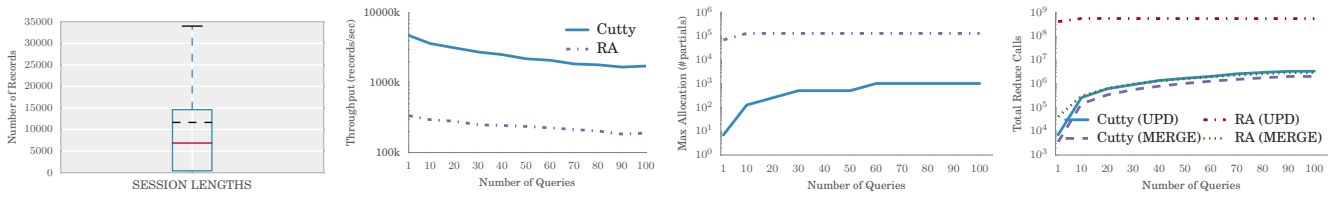
7.2 Periodic Window Aggregation

Workload. In this experiment we focused on periodic windows and generated a variable number of sliding count-based windows on top of the DEBS12 dataset. We used a rolling average of the three main energy measures as the aggregation function. We further generated queries of uniformly random window ranges and slides. The range values were selected within the range [20000;80000] and the slides within [1000;20000]. Furthermore, slides and ranges were chosen to be multiples of 10, thus, yielding 5000 possible unique range and 1900 slide values. The resulting distribution of ranges and slides is depicted in Figure 8a. In this setup, we created 11 workloads containing 1 to 100 queries.

Figure 8b depicts the total number of reduce calls which were executed by each of the techniques throughout the experiment (aggregating over $\sim$ 33M records) for different workload sizes (1-100). Respectively, Figure 8c depicts the maximum number of partials stored by each of the techniques during the experiment (i.e., maximum memory allocation). Note that the number of reduce calls correlates to the throughput and the overall performance of the techniques (we omit the graphs for the lack of space).

Benefits of Sharing. Aggregate sharing trades memory resources for less computation. A first observation in Figure 8b is that RA yields computational gains only when sharing more than 10 queries. This is because RA does not perform slicing and invokes a store operation per record in FlatFat. As mentioned earlier FlatFat operations yield additional reduce calls. Besides RA, all other techniques seem beneficial to use from a single query.

Performance Comparison. Cutty with an eager strategy clearly offers the best performance results by requiring an identical amount of memory to the *Pairs* technique (the two techniques coincide in Figure 8c) while achieving nearly



(a) Window Distribution. (b) Throughput. (c) Max Memory Utilized. (d) Reduce Calls.

Figure 9: Computational and Memory Comparisons between Cutty and *RA* for Multiplexing Non-periodic Queries

half the amount of reduce calls compared to *Pairs+* (Figure 8b). *RA* on the other hand, over-utilizes memory while also incurring heavy computational load, over three degrees of magnitude higher than Cutty (eager) and *Pairs+*.

Eager vs. Lazy Store. Throughout this paper Cutty assumed an eager aggregate store as described in Section 5. The cost trade-offs depicted in Figure 8b and Figure 8c validate this decision. We can observe that both in the case of Cutty and *Pairs+* the *eager* strategy trades off exactly and constantly twice the memory requirements to gain more than an order of magnitude less overall reduce calls. The computational gain becomes more significant when sharing large amounts of overlapping window aggregates. Since the memory overhead is very small (in the orders of hundreds), for the rest of this analysis we will only focus on the performance of the eager storage strategy for Cutty and *Pairs+* (*RA* already relies on eager pre-aggregation).

Throughput Overview. In Figure 8d we compare the throughput of Cutty, *Pairs+* and *RA* for the same workloads. As expected, throughput degrades for larger workloads in Cutty and *Pairs+* since more windows yield a larger amount of partials and final aggregations to execute. However, 100x more queries yield only a degradation of 5-6x for both, Cutty and *Pairs+*. In the case of *RA*, throughput is always at a minimum even for small workloads (1 or 10 queries). The reasons behind this are covered below.

Insights on Computational Overhead. To provide further insights on the performance results presented in this analysis we further decompose reduce calls occurred in the aggregator for Cutty (Figure 8e), *Pairs+* (Figure 8f) and *RA* (Figure 8g) for the same workloads. The calls attributed to `update` summarize the cost and frequency of the `append` operation (and associated aggregate store operations such as resizing), while the `merge` part of the calls summarizes the overall overhead caused by final window aggregation, i.e. `merge` calls. In the case of Cutty, the `merge` operations attribute to the majority of the aggregation costs, while `update` costs are kept at a minimum. On the other hand, *Pairs+* incur an almost two times higher number of pre-aggregation calls compared to Cutty with `update` operations dominating (compared to `merge` related calls) as the number of queries increases. *RA*'s operation seems to be overly dominated by pre-aggregation calls. This is because *RA* does not differentiate window specifics and eagerly stores each incoming record to its aggregation store resulting into a $log(n)$ additional cost per record, for a significantly larger n compared to Cutty and *Pairs+*. Finally, Figure 8h shows that slicing in *Pairs+* results into double the overall amount of partials compared to Cutty. That clearly explains the higher computational costs of *Pairs+*, since twice more partials yield twice more frequent updates at the store.

Experiment Summary. Cutty exhibits the highest throughput by at least 2x compared to *Pairs+* and nearly half the number of reduce calls compared to *Pairs+*. Furthermore, it has a computational cost at least three orders of magnitude lower than *RA*, the best known general pre-aggregation technique. The proportions of these results also align with our analytical comparison, described in Section 6. Concluding, Cutty is the aggregate sharing technique with the least cost for periodic queries both in theory and in practice.

7.3 Non-Periodic Window Aggregation

Workload. The DEBS12 dataset [15] consists of arbitrary binary sensor state transitions (true/false) that, in essence, can serve to pinpoint different sessions (i.e. 1 from 0 or 0 from 1 opens a new session). Since the actual dataset had only 20 active sensors (and 34 which seldom changed state) we implemented UDWs that simulated session windows, picked at random from the same distribution as the frequency of the 20 active sensor state transitions (summarized in Figure 9a). This type of dynamic windows is fundamentally non-periodic (but deterministic) and thus, we only included Cutty and *RA* in this experiment.

Performance Comparison. In Figure 9b we can see that the throughput of *RA* is already several orders of magnitude lower than Cutty, starting from a single query. Since in this scenario the windows are not sliding, the impact of additional windows is more clear. Memory utilization (Figure 9c) follows similar trends that we have seen in the previous experiment with periodic queries. Additionally, in Figure 9d we can see a drill down of the computational costs of both Cutty and *RA*. As before, the overall computational difference is around three orders of magnitude. We can further observe though, that the number of `merge` related calls in *RA* are similar to the one of Cutty's *update* and *merge* related reduce calls. Still, `merge` calls in this scenario were significantly higher than in the previous case with periodic queries since session windows could be smaller and thus, trigger full window computations more frequently.

Experiment Summary. The performance results of Cutty for non-periodic, deterministic windows highlight the core benefit of slicing, when it is applicable. To the best of our knowledge, Cutty is the first technique that can apply slicing in non-periodic, deterministic windows while doing so with a higher efficiency and better performance than state-of-the-art slicing techniques for periodic windows. However, it is fair to note here that RA can be used in more general cases (e.g., out of order eviction of window items), not only for deterministic UDWs.

8. RELATED WORK

Panes, Pairs, and RA. Pairs [16] improved upon Panes [17] by reducing the number of slices per query and by extending Panes' ideas to aggregate multiple periodic queries. However i) Pairs itself fails to scale to large numbers of queries with diverse range and slide sizes. ii) Cutty perfoms better than both Pairs and Panes theoretically and experimentally iii) Cutty applies to queries with windows which go beyond simple periodic ones. Finally, RA [5, 22] applies to non-periodic windows but with a great memory cost in order keep all individual elements of the data stream in a tree structure. In contrast, Cutty can apply slicing for the class of deterministic non-periodic windows with much less computational and memory cost.

Other Approaches. Li et al. [18, 19] did cot consider slicing techniques, but briefly classified window types by their evaluation context requirements, leaving the characterization of each class as an open research question. Our work partly fills this gap: deterministic discretization functions subsume all forward-context-free windows (no future records are required to know when a window starts), while non-deterministic discretization functions are forward-context-aware. Several heuristic-based plan optimisers have been proposed (*Weave Shared* [12], *TriWeave*[13]) to overcome limitations in the context of periodic time queries dynamically using runtime metrics (i.e. input rate and shared aggregate rate). We consider this work complimentary to ours, since our technique focuses on improving further single operator aggregation sharing. Thus, such optimisations can also be used to group multiple queries in a selection aggregators that execute Cutty instead.

9. CONCLUSIONS AND FUTURE WORK

In this paper we considered the efficient sharing of partial aggregates across a very broad range of windows, specified as user-defined functions (UDFs). We based our aggregate sharing technique on the observation that slicing can be performed simply by knowing the begins of windows, alleviating the need for complete knowledge of the semantics of the windows being aggregated. We defined a class of UDWs named deterministic, for which aggregate sharing can be performed efficiently. Moreover, multiple queries of different window semantics can share aggregates. We implemented our techniques on Apache Flink, and performed experiments demonstrating orders of magnitude of reduction in aggregation costs compared to the state of the art.

Future Work. The first version of UDWs presented in this paper has been contributed to Apache Flink v0.9 and has been in wide use for almost a year. We plan to add support for out-of-order window aggregations and comply with the bucket-per-window model [18] that Google Dataflow [3] and Apache Flink have recently adopted among others.

Acknowledgements. The authors would like to thank Lars Kroll, Gyula Fóra, and Niklas Ekström for their input. This work was supported by the End-to-End Clouds project funded by Stiftelsen för Strategisk Forskning (RIT10-0043), the BIDAF project funded by KK-stiffen (20140221), and the European Union's Horizon 2020 projects Streamline (688191), and Proteus (687691), the German Science Foundation (DFG) as FOR 1306 Stratosphere, by the German Ministry for Education and Research as Berlin Big Data Center (01IS14013A) and Software Campus (01IS12056).

10. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 2003.

[2] 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. In *VLDB*, 2013.

[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *VLDB*, 2015.

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal*, 2006.

[5] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 2004.

[6] A. Bifet and R. Gavalda. Learning from time-changing data with adaptive windowing. In *SDM*. SIAM, 2007.

[7] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. Secret: A model for analysis of the execution semantics of stream processing systems. In *VLDB*, 2010.

[8] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 2015.

[9] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *ACM SIGMOD*, 2003.

[10] B. Gedik. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience*, 2014.

[11] T. Grabs, R. Schindlauer, R. Krishnan, J. Goldstein, and R. Fernández. Introducing microsoft streaminsight. Technical report, Technical report, 2009.

[12] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Optimized processing of multiple aggregate continuous queries. In *ACM CIKM*, 2011.

[13] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *IEEE ICDE*, 2012.

[14] M. Hirzel, H. Andrade, B. Gedik, et al. IBM streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 2013.

[15] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The DEBS 2012 grand challenge. In *ACM DEBS*, 2012.

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

[17] 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. *ACM SIGMOD Record*, 2005.

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

[19] J. Li, K. Tufte, D. Maier, and V. Papadimos. Adaptwid: An adaptive, memory-efficient window aggregation implementation. *IEEE Internet Computing*, 2008.

[20] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *ACM SOSP*, 2013.

[21] K. Patroumpas and T. Sellis. Window specification over data streams. In *EDBT*. 2006.

[22] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. In *VLDB*, 2015.

[23] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *ACM SIGOPS*, 2009.