Window-Aware Load Shedding for Aggregation Queries over Data Streams [*]

Nesime Tatbul
Brown University
tatbul@cs.brown.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

Data stream management systems may be subject to higher input rates than their resources can handle. When overloaded, the system must shed load in order to maintain low-latency query results. In this paper, we describe a load shedding technique for queries consisting of one or more aggregate operators with sliding windows. We introduce a new type of drop operator, called a “Window Drop”. This operator is aware of the window properties (i.e., window size and window slide) of its downstream aggregate operators in the query plan. Accordingly, it logically divides the input stream into windows and probabilistically decides which windows to drop. This decision is further encoded into tuples by marking the ones that are disallowed from starting new windows. Unlike earlier approaches, our approach preserves integrity of windows throughout a query plan, and always delivers subsets of original query answers with minimal degradation in result quality.

1. INTRODUCTION

Stream processing engines (SPEs) [8, 10, 21] have been shown to be useful for many modern applications that have very high input rates and that need low-latency response to a set of continuous queries. Applications that have this characteristic include network traffic monitoring, industrial process control, and sensor networks.

Providing meaningful service even under system overload is one of the key challenges for stream processing engines. We assume that overloads occur as temporary bursts. If an overload is sustained, then the system is not provisioned properly, and an SPE will likely not be able to provide acceptable guaranteed service. Under such bursty conditions, queues will build up, thereby seriously increasing the latency of results. Thus, if we are to operate within the given latency bounds, there may be no alternative but to shed load by dropping some tuples. Dropping tuples will produce an approximation to the correct result. The goal then becomes to develop load shedding algorithms that remove an overload and at the same time minimize the degradation of the result.

The load shedding problem has been studied earlier. Two alternative approximation models have been commonly used. One model produces approximate answers by omitting tuples from the correct answer [6, 11, 18, 23, 26]. In this model, all delivered tuples are guaranteed to be a *subset* of the exact answer. This property is important since it allows the application to rely on the tuples that it receives to be correct. The challenge here is to provide the largest possible subset. An alternate approach to degrading the result is to emit nearly the same number of tuples, each of which might be inaccurate [7]. The challenge in this case is to ensure that the errors are bounded by some amount. It is an application-level decision as to whether it is better to have all values, some of which may be inaccurate, or fewer values, all of which are accurate. The work described in this paper is based on the subset result model.

Our subset-based approximation assumption is motivated by several real-life applications. For example, consider the case of a distributed, multi-player game. An SPE could be used as the publish/subscribe engine that distributes events to interested client machines [3]. In many cases like this, the output stream is a sequence of updates. Each new tuple updates the previous one. It is important for the play of the game that positions of players and weapons be reported accurately. An error in a position could cause the client program to simulate a “hit” when one has not occurred. Thus, in this example, it would be better to slow down the play (produce “jerky” images) than to do the wrong thing.

Aggregates play a key role in many data stream applications. Often we would like to produce a computation on ranges of consecutive input tuples (a window) as an arbitrary user-defined function. Most recent stream processing systems provide full support for user-defined aggregates (e.g., [2, 19]). With such a capability, it is highly likely to use them at arbitrary places in a query plan. Thus, *nested user-defined aggregates* have proven to be essential in various applications, ranging from habitat monitoring with sensors to online auctions [24], and highway traffic monitoring [5].

As a concrete example, consider the query plan in Figure 1, that computes the number of times in an hour that IBM’s high price and low price in a 5-minute window differ by more than 5. Box 2 is a user-defined aggregate that collects all prices for a symbol in a 5-minute window, and then emits a tuple that contains the difference between the high and the low price. This stream is then filtered to retain differences that are larger than a given threshold, in this case, 5. A downstream aggregate then counts these extreme price differences. This kind of behavior can nest to an arbitrary depth.

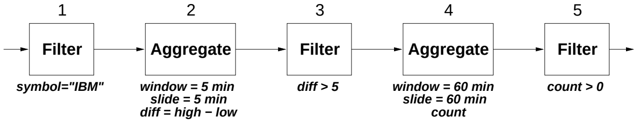


Figure 1: An example nested aggregation query

[*] This work has been supported by the NSF, under the grants IIS-0086057 and IIS-0325838.

VLDB ‘06, September 12-15, 2006, Seoul, Korea.

Load shedding techniques devised so far have applied drops in units of individual tuples. Drops are implemented as a specific operator, and this operator is pushed toward the inputs to avoid wasted work. A major limitation of this approach is that windowed operators such as aggregates, either block the motion of such drops (e.g., drop must be placed between box 4 and box 5 in Figure 1), or result in non-subset answers if drops are pushed across them. Furthermore, if inexact answers are produced through load shedding in the middle of the query plan, it is difficult to understand how this error will propagate through subsequent downstream operators. This further limits the query topologies across which such drops can be placed (e.g., there can be at most one aggregate operator in the query, at the leaf of the query tree [7]).

In this paper, we introduce a new approach which applies drops in units of windows. This approach never produces wrong values and does not suffer from any of the problems mentioned above. It further enables the placement of drops at early points in a query plan (e.g., before box 1 in Figure 1), maximizing the amount of processing saved while keeping the error under control. More specifically, in this paper, we study the problem of load shedding for aggregation queries over data streams. The main contributions of our work can be listed as follows:

- We propose a novel load shedding approach for windowed aggregation queries which guarantees to deliver subset results.
- Our technique is general enough to handle arbitrary (user-defined) aggregate functions, multiple levels of aggregate nesting, and shared query plans.
- Regardless of where the aggregates appear in a query plan, our approach enables pushing drops across them, to early points in the plan, maximizing the amount of processing saved.
- We mathematically analyze the correctness and performance of our approach.
- We experimentally evaluate the performance of our approach on a stream processing system prototype.

The rest of this paper is organized as follows: Section 2 presents an overview of models and assumptions underlying the stream processing environment that we consider. Our subset-based, window-aware load shedding approach is presented in detail in Section 3. We present an experimental evaluation of this approach in Section 4. Section 5 summarizes the related work in this area. Finally, we conclude in Section 6, outlining potential avenues for future research.

2. BACKGROUND

The work presented in this paper is part of the Aurora/Borealis [1] stream processing system [1, 2]. We first give a brief overview of our stream processing system with emphasis on its load shedder component, followed by a detailed discussion of other models and assumptions that are particularly relevant to the work described in this paper.

2.1 System Overview

We model a data stream as an append-only sequence of tuples with a uniform schema. Embedded in each tuple, is a header that carries system-assigned annotations such as tuple's arrival timestamp. Continuous queries are defined through a boxes-and-arrows-based dataflow diagram, which we call a *query network*. Each box in this diagram represents a query operator and each arc represents a data flow or a queue between the operators. Common subexpressions can be shared across multiple queries by allowing multiple arcs to emanate from a single box. Details of our data and query model are further described in earlier work [2].

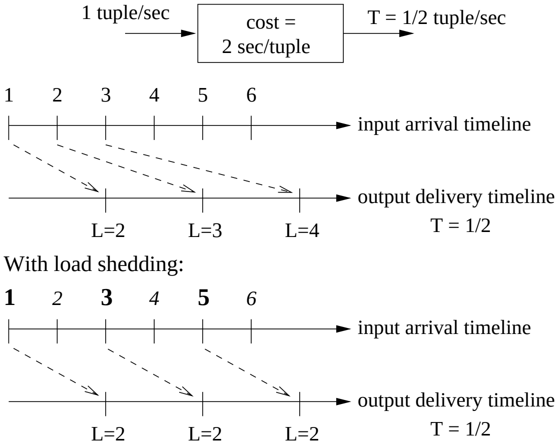


Figure 2: A simple overload scenario

The load shedder component, which is the focus of this paper, continuously monitors CPU load of a running query network. If an overload is detected, drop operators are inserted into the query network. These operators discard a certain fraction of their input tuples to remove excess load from the system with minimal degradation in quality of the delivered result.

Let us illustrate the basic idea of load shedding on a simple overload scenario. Consider the query in Figure 2 with a CPU cost of 2 seconds per input tuple. Also assume that the input arrives at the rate of 1 tuples per second. The input arrival timeline shows the points where a new tuple arrives with the indicated timestamp, and the output delivery timeline shows the points where a result with that timestamp is delivered. Since the system can only process 1 tuple every 2 seconds, a queue starts to build up and latency of output tuples (L) starts increasing due to the queue waiting time. The query throughput (T) is 1/2 tuples delivered per second. Now assume that the load shedder inserts a drop at the query input which drops 50% of its input. In Figure 2, tuples with bold timestamps are the ones that are kept, while the italic ones are dropped. In this case, the input rate will decrease to 1/2 tuples per second, and the query will be able to keep up with the reduced input rate. As a result, there will be no queueing and tuple latencies will be limited by the processing cost which is 2 seconds. Furthermore, the query throughput will still be 1/2 tuples delivered per second. Hence, by shedding load, both latency can be reduced and also the output application can be updated with the highest possible throughput frequency. Therefore, results do not get stale.

We studied this general notion of load shedding for data stream management systems in our earlier work [26]. In this paper, we build on our previous framework to enable load shedding on queries with sliding window aggregates.

2.2 Aggregation Queries

An *aggregation query* is composed of one or more aggregate operators along with other operators. Aggregate operators act on windows of tuples. Before we define the aggregate operator, we describe how we model its two important building blocks: windows and aggregate functions.

The Window Model. Data streams are continuous sequences of

[1] Borealis is a distributed stream processing system. Each Borealis node runs Aurora as its underlying query processing engine.

data records that may have no end. Traditional set operations like join or aggregate may block or may require unbounded memory if data arrival is unbounded. Most applications, however, require processing on finite portions of a stream rather than the whole. Each such excerpt is called a *window*. Windows can be modeled in various ways [16]. In our system, there are two ways to physically build windows: (i) attribute-based windows, and (ii) count-based windows. In the first case, an attribute is designated as the windowing attribute (usually time), and consecutive tuples for which this attribute is within a certain interval constitute a window (e.g., stock reports over the last 10 minutes). Here, tuples are assumed to arrive in increasing order of their windowing attributes. In the second case, a certain number of consecutive tuples constitute a window (e.g., the last 10 readings from the temperature sensor). Our system also uses a sliding window model in which a window's endpoints move by a given amount to produce the next window.

The Aggregate Function. An aggregate function $\mathcal{F}$ takes in a window of values and performs a computation on them. $\mathcal{F}$ can be a standard SQL-style aggregate function (sum, count, average, min, max) or a user-defined function. Aggregate functions in our system have the form `F(init, incr, final)`, such that the `init` function is called to initialize a state when a window is opened; `incr` is called to update that state whenever a tuple that belongs to that window arrives; and `final` is called to convert the state to a final result when the window closes. Note that, as will soon become apparent, our approach is in fact independent of the particular aggregate functions used in a query.

The Aggregate Operator. An aggregate operator $Aggregate(\mathcal{S}, \mathcal{T}, \mathcal{G}, \mathcal{F}, \omega, \delta)$ has the following semantics. It takes an input stream $\mathcal{S}$, which is ordered in increasing order on one of its attributes denoted by $\mathcal{T}$, which we call the *windowing attribute*. If $\mathcal{T}$ is not specified, $Aggregate$ requires no order on its input stream. In practice, $\mathcal{T}$ usually corresponds to tuple timestamps which can either be embedded in the tuple during its generation at the source (e.g., temperature readings from a sensor, recorded with the time they were measured), or can be assigned by the stream processing system at arrival time. From here on, we will use the terms "timestamp" and "windowing attribute" interchangeably.

$\mathcal{S}$ is divided into substreams based on optional group-by attribute(s) $\mathcal{G}$, if specified. Each substream is further divided into a sequence of windows on which the aggregate function $\mathcal{F}$ is applied. $Aggregate$'s window properties are defined by two important parameters: *window size* ω and *window slide* δ. These parameters can be defined in two alternative ways: (i) in units of the windowing attribute $\mathcal{T}$ (e.g., time-based window), (ii) in terms of number of tuples (i.e., count-based window). According to the time-based windowing scheme, a window W consists of tuples whose timestamp values are less than ω apart. When $Aggregate$ receives a tuple whose timestamp is equal to or greater than the smallest timestamp in $W + \omega$, W has to be closed. While ω denotes how large a window is and thus when it should be closed, δ denotes when new windows should be opened. Every δ time units, $Aggregate$ has to open a new window. We assume that $0 < \delta \leq \omega$. When $\delta = \omega$, we say that we have a *tumbling window*. Otherwise, we say that we have a *sliding window*. Tumbling windows constitute an interesting case because they partition a stream into non-overlapping consecutive windows.

$Aggregate$ outputs a stream of tuples of the form (t, g, v), one for each window W processed. t is the smallest timestamp of the tuples in W, g is the value of the group-by attribute(s) (skipped when $\mathcal{G}$ is not specified), and v is the final aggregate value returned by the `final` function of $\mathcal{F}$. In this paper, we will assume that $\mathcal{S}$ consists of a single group and windows are time-based. Extensions to more general forms of aggregates are provided in the technical report version of the paper [27].

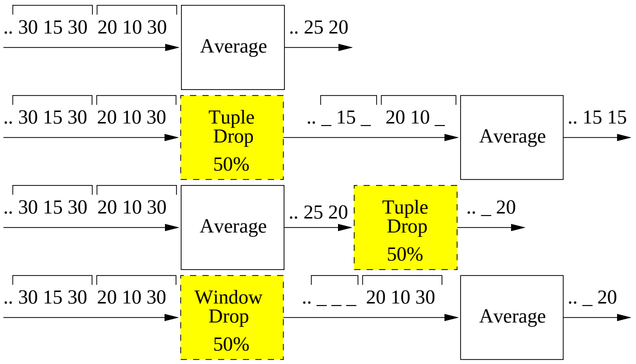


Figure 3: Drop alternatives for an aggregate

2.3 The Subset-based Approximation Model

Approximate answers result from dropping tuples. We will shed load such that the delivered answer is a subset of the original answer, and the size of this subset is the largest possible. In case of multiple queries, the goal is to maximize the amount of total percent tuple delivery.

Additionally, we assume that each output application also specifies a threshold for its tolerance to *gaps*. Gap represents a succession of tuples missing from the result. For example, we have worked on a sensor network application [25], in which a person's physiological measurements must be delivered at least once per minute, i.e., losing or choosing not to deliver results observed more frequently is acceptable. We call the maximum output gap to which an application can tolerate the *batch size*. The system must guarantee that the amount of consecutive output tuples missed due to load shedding never exceeds this value. Note that batch size can be defined in terms of tuple counts or time units. In this paper, we assume the former.

Note that batch size puts a lower bound on loss. Given a batch size $\mathcal{B}$, the query must at least deliver 1 tuple out of every $\mathcal{B} + 1$ tuples. Therefore, the fraction of tuples delivered can never be below $1/(\mathcal{B} + 1)$. Under heavy workload, it may not be possible to remove excess load while still meeting all applications' bounds on $\mathcal{B}$. In this case, we apply "admission control" on queries, where the most costly queries whose bounds can not be met have to be completely shut down (by inserting drops at their inputs with drop probability $p = 1$).

2.4 Load Shedding on Aggregates

In our subset-based load shedding framework, queries deliver values all of which would also occur in the exact answer; no new values are generated. As such, this framework has to address an important challenge when windowed aggregates are involved: dropping individual tuples from streams does not guarantee subset results when such streams are to be processed by windowed aggregates.

Let us illustrate our point with a simple example. Consider the aggregate operator in Figure 3, which computes 3-minute averages on its input in a tumbling window fashion. If we place a tuple-based random drop before the Average which cuts the load down by 50%, then we obtain a non-subset result of nearly the same size as the original. In this case, the load between the Tuple Drop and the Average is reduced by a factor of 50%, but the load is the same downstream from the Average. Alternatively, we can place the Tuple Drop after the Average, which drops tuples after the average has been computed. In this case, we produce a subset result of smaller size. However, load reduction has been achieved too late in the query plan, and we do not save from the computation of the aggregate. As a result, there is a tradeoff between achieving subset results and reducing load early in a query plan. We need a mechanism which would drop load before the Average, but would still produce a subset result. *Windowed aggregates deliver subset results if and only if they operate on original windows as indivisible units*. This observation led us to invent a new type of drop operator, called a *Window Drop*. As shown in Figure 3, the Window Drop can be placed before the Average, and applies drops in units of windows. As a result, it can achieve early load reduction without sacrificing the subset guarantee. Furthermore, our approach is general enough to allow user-defined and nested aggregates to appear anywhere in the query plan, even when there is sharing. We next describe this approach in detail.

3. WINDOW-AWARE LOAD SHEDDING

In this section, we present our subset-based, window-aware load shedding approach for aggregation queries over data streams. We show how various types of aggregation queries are handled through the use our new Window Drop operator.

3.1 The Window Drop Operator

A window drop operator $WinDrop(\mathcal{S}, \mathcal{T}, \mathcal{G}, \omega, \delta, p, \mathcal{B})$ takes six parameters in addition to an input stream $\mathcal{S}$. $\mathcal{T}$ denotes the windowing attribute, $\mathcal{G}$ denotes the group-by attribute(s), ω denotes the window size, δ denotes the window slide, p denotes the drop probability, and $\mathcal{B}$ denotes the drop batch size. The $\mathcal{T}, \mathcal{G}, \omega, \delta$ parameters of $WinDrop$ are derived from the properties of the downstream aggregate operators. p is determined by the load shedder according to the amount of load to be shed. Finally, $\mathcal{B}$ is derived based on the requirements of the output applications.

The basic functionality of $WinDrop$ is to encode window keep/drop decisions into stream tuples to be later decoded by downstream aggregate operators. $WinDrop$ logically divides its input stream $\mathcal{S}$ into time windows of size ω, noting the start of a new window every δ time units. For every group of $\mathcal{B}$ consecutive windows, $WinDrop$ makes a probabilistic keep/drop decision. Each decision is an independent Bernoulli trial with drop probability p. The drop decision for a window is encoded into the tuple which is supposed to be the window's first (or starter) element, by annotating this tuple with a *window specification* value.

Each tuple has a window specification attribute as part of its system-assigned tuple header, with a default value of -1. To allow a downstream aggregate to open a window upon seeing a tuple t, $WinDrop$ sets the window specification attribute of t to a positive value for the windowing attribute $\mathcal{T}$. This value not only indicates that a window can start at this tuple, but also indicates until which $\mathcal{T}$ value the succeeding tuples should be retained in the stream to ensure the integrity of the opened window. To disallow a downstream aggregate from opening a window upon seeing a tuple t, $WinDrop$ sets the window specification attribute of t to 0. Table 1 summarizes the semantics for the window specification attribute.

Consider an aggregate operator $Aggregate(\mathcal{F}, \omega, \delta)$ [2] and assume that we would like to place a window drop before $Aggregate$. In order to drop p fraction from the output of $Aggregate$, we insert $WinDrop(\omega, \delta, p, \mathcal{B})$ at $Aggregate$'s input. Note that the first two parameters of $WinDrop$ are directly inherited from $Aggregate$ so that $WinDrop$ can divide the input stream into windows in exactly the same way as $Aggregate$ would. Then it decides which of those windows should be dropped and marks their starter elements. Finally, when a tuple t is received by $Aggregate$, $Aggregate$ examines t's window specification attribute and skips windows that are disallowed. As a result of this, we save system resources at multiple levels. First, when a window is skipped, $Aggregate$ need not open and maintain state for that window. In other words, $Aggregate$ does less work upon seeing tuples that arrive immediately after t because there is one fewer open window that those tuples can contribute to. Second, when $Aggregate$ skips a window, it produces no output for that window, thereby reducing data rate and saving from processing in the downstream subnetwork. Third, $WinDrop$ not only encodes window specifications into tuples, but it is also capable of actually dropping tuples under certain conditions, which we call an *early drop*. More specifically, tuples that are marked with a negative window specification value and that are beyond the $\mathcal{T}$ range imposed by the most recently seen positive window specification value can be dropped right away, without waiting to be seen by a downstream aggregate. Early drops are discussed in detail in Section 3.4. It should be emphasized here that the ability to move a drop upstream from an aggregate enables us to continue pushing it toward the inputs. This is important as it can save computation for the complete downstream subquery from where it ends up.

Window Specification	Description
-1	don't care
0	window disallowed
τ	window allowed; must preserve tuples with $\mathcal{T} < \tau$

Table 1: Window specification attribute

3.2 Handling Multiple Aggregates

There are two basic arrangements of aggregates in a query network: (1) a pipeline arrangement, (2) a fan-out arrangement. Table 2 summarizes the rules for setting window drop parameters for these two arrangements. Any query network can be handled using a composition of these two rules. We now discuss these rules and how we derived them in detail.

Pipeline Arrangement of Aggregates. A query arrangement with a sequence of operators where each operator's output is fed into another one is called a *pipeline arrangement*.

Assume that we have k aggregates, $A_i(\mathcal{F}_i, \omega_i, \delta_i)$, $0 < i \leq k$, pipelined in ascending order of i, as shown in the first row of Table 2. We would like to drop p fraction from the output of A_k by placing $WinDrop(\omega, \delta, p, \mathcal{B})$ before the leftmost operator in the pipeline (A_1). $WinDrop$ must have a slide δ that is equal to the slide of the last aggregate A_k in the pipeline. The reason for this is that A_k is the last operator that divides the stream into windows and produces one output every δ_k time units. Dropping p fraction from A_k's output requires that we encode a drop decision once every δ_k time units. Furthermore, $WinDrop$ must have a window size which will guarantee the preservation of all tuples of a window W when W is kept. If we only had A_k, the window size would simply be ω_k. However, there are $k-1$ aggregates preceding A_k, each with its own corresponding window of tuples to be preserved. To be on the safe side, we consider the following worst case scenario: To produce an output tuple t_m with time m, A_k needs outputs of A_{k-1} in the range $[t_m, t_{m+\omega_k})$; A_{k-1} in turn needs outputs of A_{k-2} in the range $[t_m, t_{m+\omega_k+\omega_{k-1}-1})$; and so on. Finally, A_2 needs out-

[2] We do not show $\mathcal{S}$ when the input stream is clear from the context. From here on, we also drop the $\mathcal{T}$ and $\mathcal{G}$ parameters from both aggregate and window drop, simply assuming that windows are commonly defined on time, and $\mathcal{S}$ consists of a single group.

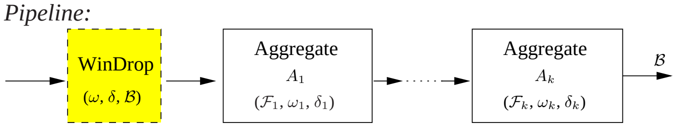
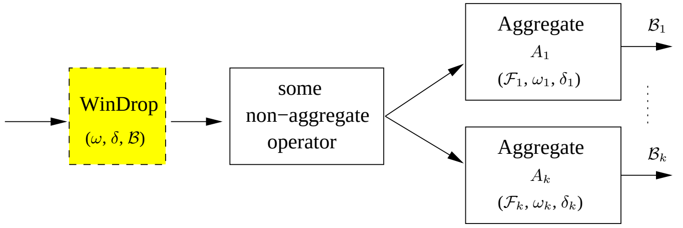
Aggregate Arrangement	Parameters for WinDrop
Pipeline:  WinDrop $(\omega, \delta, \mathcal{B})$ Aggregate A_1 $(\mathcal{F}_1, \omega_1, \delta_1)$ Aggregate A_k $(\mathcal{F}_k, \omega_k, \delta_k)$ $\mathcal{B}$	$\omega = \sum_{i=1}^{k} \omega_i - (k-1)$ $\delta = \delta_k$ $\mathcal{B}$
Fan-out:  WinDrop $(\omega, \delta, \mathcal{B})$ some non−aggregate operator Aggregate A_1 $(\mathcal{F}_1, \omega_1, \delta_1)$ $\mathcal{B}_1$ Aggregate A_k $(\mathcal{F}_k, \omega_k, \delta_k)$ $\mathcal{B}_k$	$\omega = \text{lcm}(\delta_1, \ldots, \delta_k)$ $+ \max_{i=1}^{k}\{extent(A_i)\}$ where $extent(A_i) = \omega_i - \delta_i$ $\delta = \text{lcm}(\delta_1, \ldots, \delta_k)$ $\mathcal{B} = \min_{i=1}^{k}\{\frac{\mathcal{B}_i}{\text{lcm}(\delta_1, \ldots, \delta_k)/\delta_i}\}$

Table 2: Rules for setting window drop parameters

puts of A_1 in the range $[t_m, t_{m+\omega_k+\ldots+\omega_2-(k-2)})$ and A_1 needs stream inputs $[t_m, t_{m+\omega_k+\ldots+\omega_1-(k-1)})$ in order to guarantee the desired range. Therefore, $WinDrop$ has to preserve a window of size $\omega_1 + \ldots + \omega_k - (k-1)$ whenever it decides to retain a window, which forms its effective window size. Note that this is a conservative formulation, based on the worst case scenario when each aggregate's window slide is such that the last time value in a window opens up a new window. As such, it is an upper bound on the required window size for $WinDrop$. Finally, the batch size parameter $\mathcal{B}$ of $WinDrop$ is assigned as specified by the output application at the end of the pipeline.

The simple example in Figure 4 illustrates the pipeline arrangement rule. We show a query that consists of two aggregates. A_1 has a window size and slide of 3 and 2 respectively, followed by A_2 with window size and slide of 3 each. We first show how an input stream with the indicated time values is divided into windows by these aggregates consecutively. Then we show the corresponding $WinDrop$ to be placed before this arrangement. According to our pipeline arrangement rule, $WinDrop$ must have a window size and slide of 5 and 3 respectively. Hence, it divides the input stream as shown, marking the tuples that correspond to window starts. Notice how $WinDrop$ considers input tuples with time values in the range $[1, 6)$ as an indivisible window unit to produce a result tuple with time value of 1. The original query uses exactly the same time range to produce its result with time value of 1.

Fan-out Arrangement of Aggregates. A query arrangement with an operator whose output is shared by multiple downstream branches is called a *fan-out arrangement*.

When there are aggregates at child branches of a fan-out, we need a $WinDrop$ which makes window keep/drop decisions that are common to all of these aggregates. Assume k sibling aggregates, $A_1(\mathcal{F}_1, \omega_1, \delta_1), \ldots, A_k(\mathcal{F}_k, \omega_k, \delta_k)$, as in the second row of Table 2. A common $WinDrop$ for all aggregates would have a drop probability of p, a window slide of lcm$(\delta_1, \ldots, \delta_k)$ and a window size of lcm$(\delta_1, \ldots, \delta_k)$+max$(extent(A_1), \ldots, extent(A_k)$), where $extent(A_i) = \omega_i - \delta_i$. δ = lcm$(\delta_1, \ldots, \delta_k)$ represents the lowest common multiple of slides of all sibling aggregates, i.e., every δ time units, all aggregates start a new window at the same time point. Assume T to be such a time point where all aggregates meet to start a new window. $extent(A_i)$ represents the number of time units that A_i needs beyond T in order to cleanly close its most recently opened window. A_i must have opened a window at $T - \delta_i$, because its next window will be starting at T. Therefore, its extent beyond T is $\omega_i - \delta_i$. We take the maximum of all the aggregates' extents so that all aggregates can cleanly close their open windows. As a result, the logical window that encloses all aggregate siblings must have a window size of ω = lcm$(\delta_1, \ldots, \delta_k)$ + max$(extent(A_1), \ldots, extent(A_k))$. In other words, window slide δ is formulated such that each time $WinDrop$ slides, it positions itself to where all of the aggregates, A_1 through A_k, would attempt to start new windows. Window size ω is formulated such that when a keep decision is made, enough of the range is kept to preserve integrity of all of the aggregates' windows. Finally, the batch size of $WinDrop$ is the minimum allowed by all sibling aggregates. Note that we need to scale each aggregate's batch size $\mathcal{B}_i$ before computing the minimum. This scaling is required because, when $WinDrop$ slides once, A_i slides lcm$(\delta_1, \ldots, \delta_k)/\delta_i$ times. Hence, lcm$(\delta_1, \ldots, \delta_k)/\delta_i$ consecutive windows for A_i correspond to 1 window for $WinDrop$.

The example in Figure 5 illustrates the fan-out arrangement rule. We show a query that consists of two sibling aggregates. Window sizes and slides of these aggregates are the same as in the pipeline example of Figure 4. Both aggregates receive a copy of the stream emanating from their parent, but they divide it in different ways based on their window parameters. We first show how this is done together with the extents and common window start positions for the aggregates. Both aggregates start new windows at time values 1 and 7. A_1 has an extent of 1 (i.e., its last window before a new window starts at 7 extends until 8). A_2 has an extent of 0 (i.e., its last window completely closes before a new window opens at 7). Based on these, we show the corresponding $WinDrop$ that must be placed before this aggregate arrangement. $WinDrop$ must have a window size and slide of 7 and 6 respectively. This way, it makes window keep/drop decisions at time values where both A_1 and A_2 expect to open new windows. Furthermore, in case of a keep decision, $WinDrop$ retains all tuples required to cleanly close open windows of both of the aggregates.

Composite Arrangements. We will now briefly illustrate the composition of the rules in Table 2. Assume that A_0 in Figure 5 is an aggregate with $\omega_0 = 4$ and $\delta_0 = 1$ (i.e., $A_0(4, 1)$). Thus, we have a combined arrangement with two pipelines and a fan-out. There are two alternative ways to construct $WinDrop$ for this arrangement:

1. We first apply the fan-out rule on A_1 and A_2, which gives us $WinDrop(7, 6)$ as illustrated in Figure 5. Then we apply the pipeline rule on $A_0(4, 1)$ and $WinDrop(7, 6)$, which

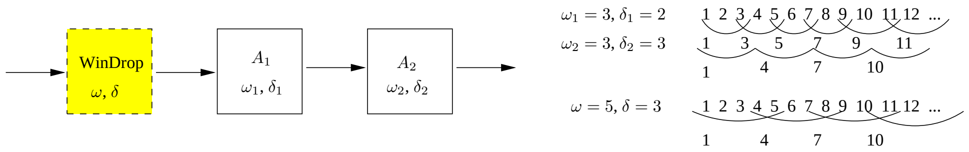


Figure 4: Pipeline example

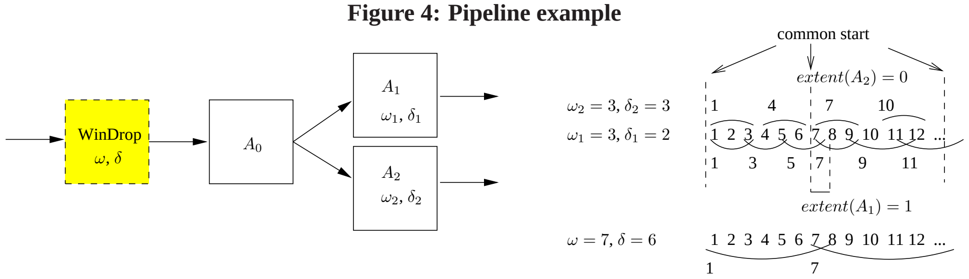


Figure 5: Fan-out example

gives us $WinDrop(10, 6)$.

2. We first apply the pipeline rule on paths $[A_0(4, 1), A_1(3, 2)]$ and $[A_0(4, 1), A_2(3, 3)]$, which gives us $WinDrop(6, 2)$ and $WinDrop(6, 3)$, respectively. We then apply the fan-out rule on these, which gives us $WinDrop(10, 6)$.

3.3 Decoding Window Specifications

As mentioned earlier in Section 3.1, window drop attaches window specifications to tuples that are potential window starters. These specifications further indicate the fate of those windows and need to be decoded by downstream aggregates in order for them to take the right action. In this section, we describe how this decoding mechanism works.

Table 3 summarizes how an aggregate $Aggregate$ with window size ω decodes the window specifications coded by a preceding $WinDrop$. First assume that $Aggregate$ receives a tuple with time value t and according to the slide parameter of $Aggregate$, a new window has to start at t (upper half of Table 3). If the tuple has a positive window specification τ, then $Aggregate$ opens a new window with a window specification attribute of $\tau - (\omega - 1)$ (i.e., when this window closes and produces an output tuple, the window specification of this output tuple will be $\tau - (\omega - 1)$). $Aggregate$ also has to make sure that all successive tuples with time values up to $\tau - (\omega - 1)$ are retained in the stream (i.e., $Aggregate$ sets its keep_until variable to $\tau-(\omega-1)$). If the tuple has a non-positive (0 or -1) window specification, then $Aggregate$ checks if t is within the time range that it must retain (i.e., if $t <$ keep_until). If so, a new window is opened with the given window specification and the keep range is set to max(keep_until, $t+\omega$). If not, $Aggregate$ skips this window.

Now assume that $Aggregate$ receives a tuple with time value t where $Aggregate$ does not expect to open a new window (lower half of Table 3). $Aggregate$ will not open any new window. However, it has to still maintain the window specification attribute in the tuple for other downstream aggregates' disposal (if any). The two important specifications are τ and 0, the former indicating the opening of a window and the latter indicating the skipping of a window. If the specification is -1, $Aggregate$ does not need to do anything. If the tuple has a positive window specification τ, $Aggregate$ updates its time range as well as the window specification of the tuple. In both of the non-negative cases, $Aggregate$ marks this tuple as a *fake tuple*. A fake tuple is one which has no real content but only carries a window specification value that may be significant to some downstream aggregates. Such tuples should not participate in query computations and should be solely used for decoding purposes.

win_start?	win_spec	keep_until	relevant action

yes	τ	within	open window
			keep_until = $\tau - (\omega - 1)$
			win_spec = $\tau - (\omega - 1)$

yes	τ	beyond	open window
			keep_until = $\tau - (\omega - 1)$
			win_spec = $\tau - (\omega - 1)$

yes	0	within	open window keep_until = $t + \omega$, (if >)
yes	0	beyond	skip window
yes	-1	within	open window keep_until = $t + \omega$, (if >)
yes	-1	beyond	skip window

no	τ	within	keep_until = $\tau - (\omega - 1)$
			win_spec = $\tau - (\omega - 1)$
			mark as fake tuple

no	τ	beyond	keep_until = $\tau - (\omega - 1)$
			win_spec = $\tau - (\omega - 1)$
			mark as fake tuple

no	0	within	mark as fake tuple
no	0	beyond	mark as fake tuple
no	-1	within	ignore
no	-1	beyond	ignore

Table 3: Decoding window specifications

We must point out here that fake tuples have one other important use. A query network may have other types of operators lying between a window drop and the downstream aggregates which are supposed to decode window specifications generated by the window drop. We must make sure that window specifications correctly survive through such operators. For example, assume that the filter between the two aggregates in Figure 1 (box 3) decides to drop a tuple t from the stream since this tuple does not satisfy its predicate. If t is carrying a non-negative window specification, then we can not simply discard it. Instead, we must mark t as a fake tuple and let it pass through the filter. This is because t is carrying a message for the downstream aggregate (box 4) about whether to open or to skip a window at a particular time point.

Note that it can be argued that fake tuples introduce additional tuples into the query pipeline. However, since these are not real tuples, operators except aggregates will just pass them along without doing any processing on them, whereas aggregates will check the flag to see if they should open a window. Hence, it is unlikely that fake tuples will drive the system into overload.

3.4 Early Drops

Window drop not only marks tuples, but it can also drop some of them. In this section, we discuss how this early drop mechanism works. We start with a useful definition.

DEFINITION 1 (WINDOW COUNT FUNCTION (WCF)). *Consider a stream $\mathcal{S}$ with tuples partially ordered in increasing order of their time values. Assume that the very first tuple in $\mathcal{S}$ has a time value of θ. Consider an aggregate $Aggregate(\mathcal{S}, \omega, \delta)$, where $\omega = m * \delta + \phi$, $m \geq 1$, $0 \leq \phi < \delta$. We define a Window Count Function $WCF : \mathbb{Z}^* \rightarrow \mathbb{N}$, that maps time value t to the number of consecutive windows to which tuples with t belong as:*

$$WCF(t) = \begin{cases} i+1, & \text{if } t \in [\theta + i * \delta, \theta + (i+1) * \delta - 1] \\ & \text{where } 0 \leq i < m \\ m+1, & \text{if } t \in [\theta + i * \delta, \\ & \quad \theta + (i-m) * \delta + \omega - 1] \\ & \text{where } i \geq m \\ m, & \text{if } t \in [\theta + (i-m) * \delta + \omega, \\ & \quad \theta + (i+1) * \delta - 1] \\ & \text{where } i \geq m \end{cases}$$

Note that the first case only occurs once at the start of the stream. Thereafter, the second and the third cases occur repeatedly one after the other. If the aggregate window is tumbling (i.e., $\omega = \delta$), then the second case has a time range length of 0, i.e., it is skipped. Also, the first case is equivalent to the third case since $m = 1$. As a result, for tumbling window aggregates, $WCF(t) = m = 1$ for all tuples (i.e., each tuple belongs to only 1 window).

RULE 1 (EARLY DROP RULE). *If a tuple with time value t belongs to k windows (i.e., $WCF(t) = k$), then this tuple can be early-dropped if and only if the window drop operator decides to drop all of these k windows.*

If a window drop operator $WinDrop$ flips a coin every time it observes a potential window start and decides to drop that window with probability p, then for an early drop, $WinDrop$ has to flip the coin for k consecutive times, which has probability p^k. Unless p is a big number or k is a small number (e.g., in the case of a tumbling window), then the probability of an early drop is very small. Instead, to take advantage of early drops, we use the following (more deterministic) drop mechanism: We mentioned in Section 2.3 that, to indicate its tolerance to gaps in the answer, each query specifies a constant $\mathcal{B}$ for the maximum number of consecutive windows that can be shed. Given a drop probability p, $WinDrop$ flips the coin once for every batch of $\mathcal{B}$ windows and drops them *all* with probability p. Based on the window count function WCF, dropping $\mathcal{B}$ consecutive windows corresponds to a certain number of early drops. Note that to satisfy $\mathcal{B}$, at least one window has to be opened after each dropped batch. If the coin yields two consecutive drops, then we allow the first window of the second batch to open and compensate for it later by skipping a window when in fact the coin yields a keep. This ensures that we satisfy both $\mathcal{B}$ and p.

3.5 Window Drop Placement

In general, load reduction should be performed at the earliest point in a query plan to avoid wasted work. However, there may be certain situations where placing drops at inner arcs of the query plan might be more preferable than placing them at the input arcs. We will briefly discuss these situations.

Unless $\mathcal{B}$ is large enough to allow early drops (i.e., $\mathcal{B} \geq \lfloor \frac{\omega}{\delta} \rfloor$), there is no benefit in placing a window drop operator $WinDrop$ further upstream than the leftmost aggregate in the pipeline. For the pipeline arrangement, as we place $WinDrop$ further upstream, the difference between ω and δ widens (i.e., $m = \lfloor \frac{\omega}{\delta} \rfloor$ in Definition 1 grows). Similarly, for the fan-out arrangement, both ω and δ may get larger across a split while $\mathcal{B}$ may get smaller. $WinDrop$ must be placed at the earliest point in the query where it saves processing while also not violating the constraints on $\mathcal{B}$.

Although not so common, a query plan may have multiple aggregates with different sliding window properties over the same data stream (e.g., a pipeline arrangement with a mix of count-based and time-based windows, and/or with different group-by attributes). In this case, the window drop must be placed at a point where such properties are homogeneous downstream. It requires further investigation to extend our framework to handle the heterogeneous case.

3.6 Analysis

Next we mathematically analyze our approach for correctness and performance.

DEFINITION 2 (CORRECTNESS). *A drop insertion plan is said to be correct if it produces subset results at query outputs.*

THEOREM 1. *Window drop inserted aggregation queries preserve correctness.*

PROOF. *The proof for this theorem has two parts, one for each aggregate arrangement. We can prove each by induction. Consider a pipeline $\mathcal{P}$ of N aggregates $A_i(\omega_i, \delta_i)$. Given a finite input stream $\mathcal{S}$, assume that the result of $\mathcal{P}(\mathcal{S})$ is the set $\mathcal{A}$, and the result for the window drop inserted version, $\mathcal{P}'(\mathcal{S})$, is the set $\mathcal{A}'$. For $N = 1$, $WinDrop(\omega, \delta)$ is inserted before A_1 such that $\omega = \omega_1$, $\delta = \delta_1$. Every δ_1 time units, $WinDrop$ marks a tuple t as either keep (τ, where $\tau = t.time + \omega$), or drop (0). When A_1 receives t with specification of τ, it opens a new window at $t.time$ and retains all tuples in time range $[t.time, \tau)$. In this case, A_1 delivers an output tuple $o \in \mathcal{A}$. When A_1 receives t with a 0 or -1 specification, it does not open a window. In this case, A_1 adds no output tuple to the result. Therefore, $\mathcal{A}' \subseteq \mathcal{A}$. Next, assume that the theorem holds for $N = n$. We will show that it must also hold for $N = n + 1$. We are given that a $WinDrop(\omega, \delta)$, with $\omega = \sum_{i=1}^{n} \omega_i - (n-1)$ and $\delta = \delta_n$, inserted before A_2 preserves correctness. Consider a window W at A_1 with a time range of $[T, T + \omega_1 - 1]$, when processed produces an aggregate output with time value T. Any aggregate downstream from A_1, that includes a tuple with time T in its window effectively incorporates S values with time up to $T + \omega_1 - 1$. Therefore, if $WinDrop'$ is placed before A_1, its effective window size must include this range to preserve window integrity. As a result, $WinDrop'$ must have a window size of $\omega' = \omega + \omega_1 - 1 = \sum_{i=1}^{n} \omega_i - (n-1) + \omega_1 - 1 = \sum_{i=1}^{n+1} \omega_i - (n+1-1)$. This proves our window size formulation for a pipeline of $n + 1$ aggregates. Finally, in order to produce subset results, the $WinDrop'$ must produce results either δ_{n+1} apart or in multiples of this quantity. Therefore, $WinDrop'$ must have a window slide of $\delta' = \delta_{n+1}$. This concludes the first part of our proof. The part for the fan-out case follows a similar inductive reasoning, therefore we do not discuss it here.* □

We now analyze the effect of window drop on CPU performance. We also compare it against the random drop alternative [26]. Consider a query network as in Figure 6, where an aggregate $A(\omega, \delta)$ is present between two subnetworks of other non-aggregate operators, whose total costs and selectivities are as shown. The CPU cycles needed to process one input tuple across this query network can be

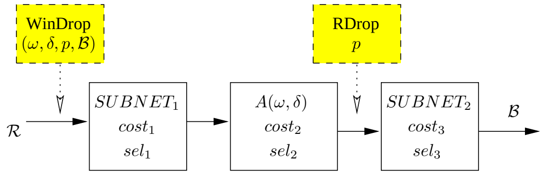


Figure 6: Inserting drops into an aggregation query

estimated as $cost_1+sel_1*(cost_2+sel_2*cost_3)$. If the input stream has a rate of $\mathcal{R}$ tuples per time unit, then the CPU load as processing cycles per time unit is $\mathcal{R}*(cost_1+sel_1*(cost_2+sel_2*cost_3))$.

If a random drop were inserted downstream from the aggregate operator, the CPU load would become:

$$L_{RDrop} = \mathcal{R}*(cost_1+sel_1*(cost_2+sel_2*(cost_{RDrop}+ (1-p)*cost_3)))$$

The CPU cycles saved as a result of this would be:

$$S_{RDrop} = \mathcal{R}*(sel_1*sel_2*p*cost_3 - sel_1*sel_2*cost_{RDrop})$$

Instead, if a window drop were inserted at the query input, the CPU load would become:

$$L_{WinDrop} = \mathcal{R}*(cost_{WinDrop}+cost'_1+cost'_2 +sel_1*sel_2*(1-p)*cost_3)$$
$$cost'_1 = cost_{fcheck}+sel_{w1}*cost_1+sel_{w2}*cost_{copy}$$
$$cost'_2 = \frac{sel_{w1}}{\delta}*cost_{wcheck}+sel_{w2}*cost_{wcheck} +sel_{w1}*sel_1*cost_2$$

$SUBNET_1$ first checks if a tuple is fake or not ($cost_{fcheck}$). Assume sel_{w1} of tuples from $WinDrop$ are normal and sel_{w2} of them are fake. Then, the former are processed normally ($sel_{w1}*cost_1$), and the latter are just copied across to A ($sel_{w2}*cost_{copy}$). A checks window specification attributes for ones to be opened ($\frac{sel_{w1}}{\delta}*cost_{wcheck}$) and for ones to be skipped ($sel_{w2}*cost_{wcheck}$). Then, tuples in the former group go through normal aggregate processing ($sel_{w1}*sel_1*cost_2$). The CPU cycles saved would be:

$$S_{WinDrop} = \mathcal{R}*(sel_1*sel_2*p*cost_3+(1-sel_{w1})*cost_1 +(1-sel_{w1})*sel_1*cost_2-cost_{WinDrop} -cost_{fcheck}-sel_{w2}*cost_{copy} -(\frac{sel_{w1}}{\delta}+sel_{w2})*cost_{wcheck})$$

If we compare S_{RDrop} with $S_{WinDrop}$, we see that $S_{WinDrop}$ has two additional savings terms: it saves from the aggregate operator's cost ($cost_2$) as well as from the first subnetwork's cost ($cost_1$) with an amount determined by sel_{w1} (as a result of any potential early drops). On the other hand, there are three additional cost terms for handling the flags introduced by $WinDrop$. We expect these costs to be much smaller than the savings of $WinDrop$. We experimentally show the processing overhead of window drop in Section 4.

Let us now briefly show how sel_{w1} and sel_{w2} can be estimated. For simplicity, we will assume a stream with one tuple per time value. We drop windows in batches of size $\mathcal{B}$. By definition, each drop-batch must be preceded and followed by at least one keep-window. The total number of tuples in a batch is $(\mathcal{B}-1)*\delta+\omega$ (see Figure 8). Given a drop-batch, $2*(\omega-\delta)$ of its tuples overlap with the preceding and the following keep-windows, therefore the number of tuples that belong only to the drop-batch is $(\mathcal{B}+1)*\delta-\omega$. These are the tuples that can be early-dropped (assuming that $\mathcal{B} \geq \lfloor\frac{\omega}{\delta}\rfloor$). This many tuples out of a total of $(\mathcal{B}-1)*\delta+\omega$ can be early-dropped and this would occur with probability p. Therefore, we end up with $sel_{w1} = 1-p*\frac{(\mathcal{B}+1)*\delta-\omega}{(\mathcal{B}-1)*\delta+\omega}$ of $WinDrop$'s output tuples being kept as normal tuples. Furthermore, one tuple out of every δ tuples may have to be retained as a fake tuple since it carries a 0 window specification. Thus, $\lfloor\frac{(\mathcal{B}+1)*\delta-\omega}{\delta}\rfloor = (\mathcal{B}+1)-\lfloor\frac{\omega}{\delta}\rfloor$ out of $(\mathcal{B}+1)*\delta-\omega$ will be additionally kept with probability p. Therefore, we end up with $sel_{w2} = p*\frac{(\mathcal{B}+1)-\lfloor\frac{\omega}{\delta}\rfloor}{(\mathcal{B}+1)*\delta-\omega}$.

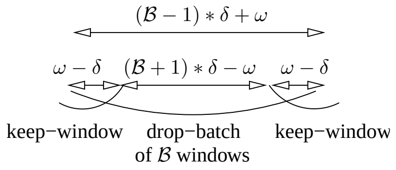


Figure 8: Drop-batch (when $\mathcal{B} \geq \lfloor\frac{\omega}{\delta}\rfloor$)

4. EXPERIMENTS

In this section, we experimentally evaluate our window drop approach. In Section 4.1, we first compare our approach against the random drop alternative which can only provide the subset guarantee by applying tuple-based drops when placed downstream from all the aggregates in a query plan. As part of this initial set of experiments, we also show the advantage of shedding load early in a query plan, by placing the window drop operator at various locations in a given plan and comparing the results. Then in Section 4.2, we examine the effect of window parameters, by varying window size and slide, and measuring the result degradation for various query plans. We also compare these experimental results against the analytical estimates of Section 3.6 to confirm their validity. Finally, in Section 4.3, we evaluate the processing overhead of our technique.

We implemented the window drop operator as part of the load shedder component of the Aurora/Borealis stream processing prototype system. We conducted our experiments on a single-node Borealis server, running on a Linux PC with an Athlon 64 2GHz processor. We created a basic set of benchmark queries as will be described in the following subsections. We used synthetic data to represent readings from a temperature sensor as (time, value) pairs. For our experiments, the data arrival rates and the query workload were more important than the actual values of the data workload. Thus, for our purposes, using synthetic data was sufficient.

4.1 Basic Performance

First we will show the basic performance of window drop for both the pipeline and the fan-out (i.e., shared) query arrangements. **Nested Aggregates.** For this experiment, we used the nested aggregation query shown in Figure 7, which is similar to the stock count example of Figure 1. There are two aggregate operators, each with tumbling windows of size 10 and 100 respectively, and both with count functions. We used a batch size of 10. We added delay operators before and after each aggregate to model other operators that may exist upstream and downstream from the aggregates. A delay operator simply withholds its input tuple for a specific amount of time (busy-waiting the CPU) before releasing it to its successor operator. A delay operator is essentially a convenient way to represent a query subplan with a certain CPU cost; its delay parameter provides a knob to easily adjust the query cost. In Figure 7, we used appropriate delay values to make different parts of the query equally costly.

The goal of this experiment is twofold. First, we show how much window drop degrades the result for handling a given level of excess load. Second, we compare it against two alternatives: one is a variation of our window drop approach, where a window drop is inserted in the middle of the query network; the other is random drop that is placed downstream from both of the aggregates. Figure 7 illustrates these three alternative drop insertion plans.

Figure 10 presents our result. The excess rate on the x-axis represents the percentage of the input rate that is over full capacity. The y-axis shows the drop rate (i.e. the fraction of the answer

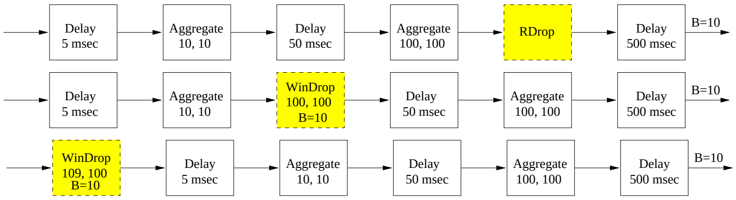
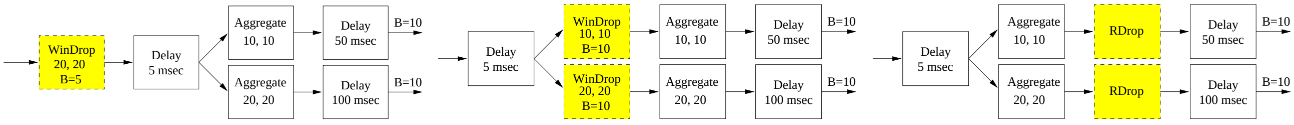


Figure 7: Drop insertion plans for the pipeline arrangement (RDrop, Nested1, and Nested2)



(a) Shared_WinDrop (b) Split_WinDrop (c) Split_RDrop

Figure 9: Drop insertion plans for the fan-out arrangement

that is missing from the result). **RDrop** represents random drop, **Nested1** corresponds to window drop inserted in the middle, and **Nested2** is for window drop placed at the input. At relatively low input rates, RDrop shows comparable performance to window drop approaches. However, as the rate gets higher, both Nested1 and Nested2 scale far better than the RDrop approach. In fact, RDrop stops delivering any results once the excess load gets beyond 65%. Nested2 either results in equal or smaller degradation in the answer compared to Nested1 at all load levels.

As a result, window drop is effective in handling system overload. It scales well with increasing input rate and outperforms the random drop alternative. Note that the RDrop case is all that would have been allowed by our previous work [26], since one could not move drops past aggregates. Placing the window drop further upstream in a nested aggregation query significantly improves the result quality, as more load can be saved earlier in the query, which reduces the total percentage of the data that needs to be shed.

Shared Query Plans. We repeated the previous experiment on a shared query plan. We used a fan-out arrangement with two aggregate queries as shown in Figure 9. The figure plots the three alternative load shedding plans that we compared. **Shared_WinDrop** is when window drop is placed at the earliest point in the query plan, **Split_WinDrop** is when each query has a separate window drop placed after the split point, and **Split_RDrop** is when we apply tuple-based random load shedding downstream from the aggregates. The parameters of the window drops are appropriately assigned based on the rules in Table 2.

Figure 11 presents our result. The y-axis shows the total drop rate for both of the queries, when the system experiences a certain level of excess load. Similar to our earlier result, shedding load at the earliest possible point in the query plan provides the smallest drop rate, and hence, the highest result quality. Again, the window drop operator enables pushing drops beyond aggregates and split points in a query plan, reducing quality degradation without sacrificing the subset guarantee.

4.2 Effect of Window Parameters

Next we investigate the effect of window parameters on window drop performance. We used a query with one aggregate operator with a count function as shown in Figure 12. We again added delay operators of 5 milliseconds each, before and after the aggregate, and set the batch size to 10.

The bar chart in Figure 12 shows the effect of window size on drops. An input rate that is 25% faster than the rate the system can handle at full capacity is fed into the query. For each window size, we measure the fraction of tuples that must be dropped to bring the system load below the capacity. We take these measurements for a window that slides by 1 (slowly sliding window) and for a window that slides by the window size (tumbling window). In most cases, the drop rates came out to be lower for the tumbling window case.

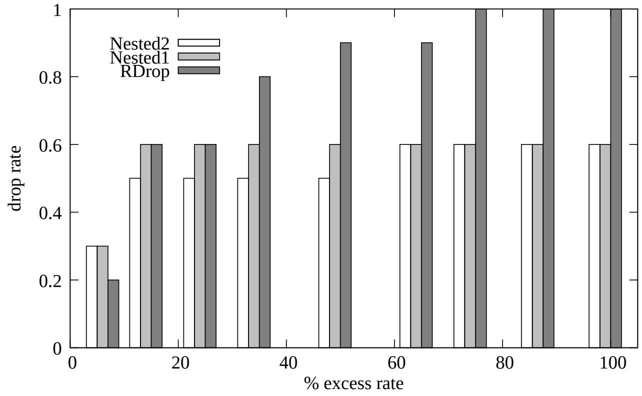


Figure 10: Comparing alternatives (pipeline)

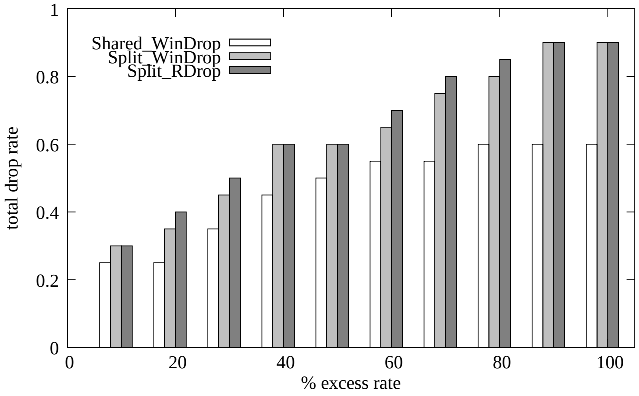
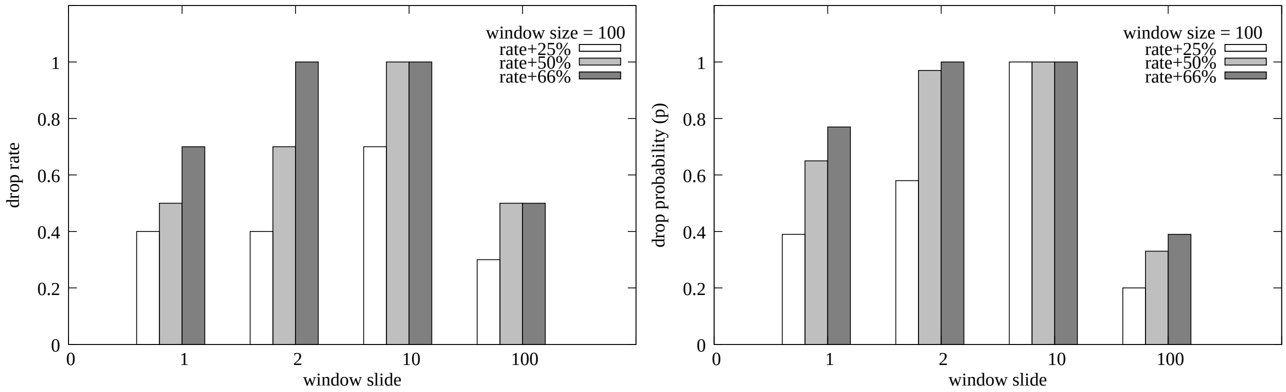


Figure 11: Comparing alternatives (fan-out)



(a) Experimental result

(b) Analytical estimate (from Section 3.6)

Figure 13: Effect of window slide

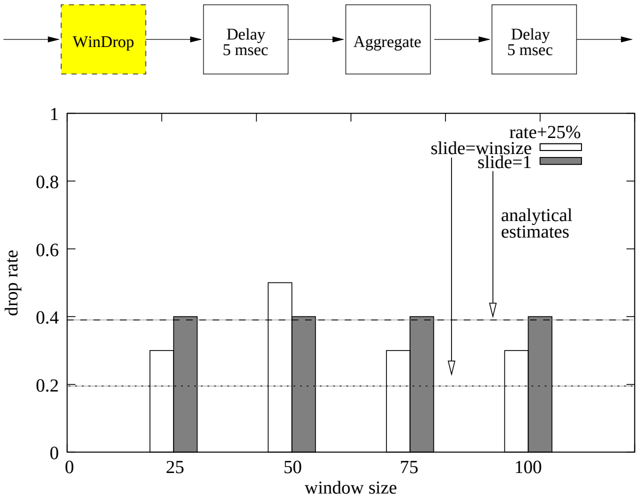


Figure 12: Effect of window size

This is because window drop can achieve early drops in this case. A second observation is that drop rates stay almost fixed as the window size increases. Interestingly, the formulas presented earlier in Section 3.6 also suggest that load should be independent of the aggregate window size when $\delta = 1$ and $\delta = \omega$ (see sel_{w1} and sel_{w2}). We also measured average operator costs and plugged them into our formulas. The formulas estimate drop rates to be 0.2 and 0.39 for the tumbling and the sliding window case, respectively (shown with dotted lines). The latter case is experimentally confirmed in Figure 12. However, our formulas underestimate the drop rate for the tumbling window case. In this case, the aggregate operator has a very small selectivity. It closes a window and produces an output once every ω tuples, at which point the downstream delay operator is scheduled. Our analysis models the average case behavior and fails to capture cases where internal load variations may occur due to changes in operator scheduling frequency.

Figure 13 details the effect of window slide on window drop performance: Figure 13(a) shows our experimental result and Figure 13(b) plots the analytical estimates of Section 3.6. A window size of 100 with four different slide values is used. A slide value of 1 corresponds to a large number of simultaneously open windows, therefore, a high degree of window overlap, and high aggregate selectivity. A slide value of 100 corresponds to one open window at a time, zero window overlap, and low aggregate selectivity, providing more opportunity for early drops. As we increase the window slide, the number of saved CPU cycles upstream from the aggregate increases (due to early drops) while the number to be saved downstream from the aggregate decreases (due to low aggregate selectivity). The required drop amount first increases, but then starts decreasing due to additional savings from early drops. Note that this decrease is observed when slide gets above 10 (i.e., when $\lfloor \frac{\omega}{\delta} \rfloor \leq \mathcal{B}$). Window drop shows the best advantage as the degree of window overlap decreases. We continue to observe this effect as excess load increases. Our analysis, plotted in Figure 13(b), captures the general behavior very well, but as window slide grows, it shows a departure from the measured results, for the same reason as explained in the previous paragraph.

As a brief note, we also compared our window drop with a random drop inserted after the aggregate. For slide=1, the performance is similar (no early drops). For slide=100, random drop fails to remove the overload, even at rate+25%. Thus, in the worst case where there is a very high degree of window overlap and zero opportunity for early drops, window drop behaves similar to the random drop. As slide grows, window drop achieves a clear advantage.

4.3 Processing Overhead

Next we evaluate the overhead of adding window drop into query plans. This overhead has several potential sources: (1) an additional operator to be scheduled in the query plan, (2) other operators interpreting window specifications, (3) fake tuples.

In this experiment, we used the query layout shown in Figure 14. We varied the predicate of the filter to obtain various selectivity values. We used a tumbling window whose window size is also varied. Table 4 shows the ratio of throughput values for a query that contains a window drop that does not drop anything ($p = 0$) and for the case when no window drop is present. We ran each query for a minute, at a rate that is 50% higher than the system capacity. Since no tuples are dropped in either case, the reduction (if any) in the number of tuples produced with window drop must be due to the additional processing overhead. First, Table 4 shows that in

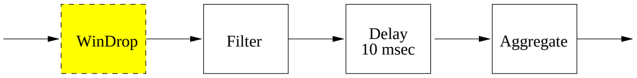


Figure 14: Filtered aggregation query

window size	selectivity=1.0	selectivity=0.5
25	0.99	0.96
50	0.99	0.98
75	1.0	0.98
100	1.0	1.0

Table 4: Throughput ratio (WinDrop($p = 0$)/NoDrop)

general, the overhead is low. Second, as the window size increases, the overhead decreases. This is due to the fact that the window drop marks tuples less frequently. Third, for lower selectivity, the overhead seems to be higher. This result accounts for the effect of handling fake tuples. As we mentioned earlier in Section 3.3, filter generates fake tuples when its predicate evaluates to false but the tuple has to be retained if it is carrying a non-negative window specification. The chance of generating fake tuples increases as the filter selectivity decreases. This may further lead to an increased overhead of processing fake tuples in the downstream query network. As shown in Table 4, we see only a slight increase in overhead when the filter selectivity is lowered to 0.5.

5. RELATED WORK

There has been a great deal of recent work in the area of data stream processing [16]. Several research prototypes have been built [2, 10, 21]. Efficient resource management, adaptivity, and approximation have been the main points of emphases.

Load shedding for aggregation queries over data streams has been the subject of recent work by Babcock et al [7]. This work inserts random drops into query trees and tries to minimize the maximum relative error at outputs. This is achieved using statistical bounds, based on mean and standard deviation statistics on windows of tuples received by aggregates. Our approach tries to produce maximum subset results, hence our approximation model is quite different. Moreover, our approach targets a general class of aggregation query topologies (i.e., aggregates can be nested and can appear anywhere in a query plan, possibly with sharing), while the proposed work assumes query trees, with a single aggregate operator at the leaf level. Lastly, in that approach, the statistical bounds apply to only a limited set of aggregate functions, whereas our approach is independent of the actual aggregate functions and can easily support user-defined functions.

Load shedding for sliding window joins in memory-limited environments have also been studied [6, 11, 18, 23]. Our approach mainly considers CPU as the limited resource. The cited works either produce maximum subset results or sampled subsets.

Punctuations are special annotations embedded into data streams to specify end of a subset of data in the stream [29, 20]. They are devised to overcome the blocking and unbounded memory problem in stateful stream operators. As such, punctuations constitute an alternative to windowed processing. Our work is relevant to punctuations in the way we attach window indicators into tuples. Although in both cases streams are annotated with information that is important in terms of optimizing query execution, the goals are quite different. In punctuations case, annotations indicate some property that naturally exists in the stream, whereas in our case, window specifications are artificially injected to cope with overload. Also, the previous work used similar windowing concepts to ours, but their focus was on optimizing query evaluation and handling disorder in data streams [20]; whereas we inject window-awareness into tuples to preserve result correctness when doing load shedding.

Lastly, approximate query processing techniques have long been studied for both traditional static data sets and continuous data streams [13]. Former techniques mostly rely on precomputed data synopsis whereas latter approaches construct one-pass summaries as streams arrive. Online aggregation [17] lies somewhere in-between by interleaving sampling with query evaluation, on stored data. More recently, the data triage approach has proposed to shed load on streams by summarizing excess data into synopsis data structures instead of dropping it [22]. In addition to samples, synopsis can take the form of histograms [14], sketches [12], or wavelets [9, 15]. In majority of the existing work, aggregate approximations are in the form of non-subset answers. To our knowledge, no previous work has studied window behaviors in depth to perform subset approximations on sliding window aggregates.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown a window-aware load shedding technique that deals with sliding window aggregate operators. Moreover, we have done this in a way that preserves the subset result guarantee. Our techniques also support load shedding in query networks in which aggregates can be arbitrarily nested. We believe that this is very important since, in our experience with the Aurora/Borealis system, user-defined aggregates have been used extensively in practice for many tasks that involve operating on a subsequence of tuples. Thus, they occur quite frequently in the interior of query networks. Our contribution is the ability to handle aggregates in a very general way that is consistent with a subset-based error model.

We have shown that, as is expected, with the added ability to push drops past aggregates, we can recover more load early; thereby, regaining the required CPU cycles while minimizing the total utility loss. By focusing on dropping windows, we can better control the propagation of error through the downstream network.

Some of the complexity in our solution is a result of the simple flat data model. For example, not being able to denote windows as sets of tuples results in a tuple marking scheme. However, a simple model simplifies implementation and allows for faster execution in the general case.

We plan to extend this work in the following directions:

Prediction-based Load Shedding. Subset-based load shedding approaches lead to gaps in query results. One way for the output application to interpret these gaps is to predict what might be missing from the result based on what is delivered (e.g., based on linear interpolation). Using a prediction-based interpretation of the subset result also gives us an opportunity to compare our approach against the relative-error based approaches (e.g., [7]). We have conducted some preliminary experiments in this direction based on linear interpolation. Our results on real data traces show that for small gaps (i.e, low batch size), our approach produces results with lower average error. Additionally, we have also observed that larger slide values result in higher error for both approaches, but our approach seems to scale better with increasing slide. We need to conduct a more comprehensive experimental study to provide a detailed quantitative comparison.

Window-awareness on Joins. Joins also operate on windows of tuples. However, the semantics is quite different. A join window involves two input streams, A and B. It defines which tuples from input B are in the range of a given tuple from input A so that the join predicate can be applied on them. Unlike aggregates, where window behavior is crucial in producing subset results, this is not the main issue for joins. Dropping inputs necessarily produces a subset and load shedding on joins is mostly about controlling the size of that subset. Consider a query with an Aggregate followed by a Join. Window Drop placed before the Aggregate causes Aggregate to produce a somewhat random subset per aggregate group.

This further affects the overall query result from the Join in different ways, depending on the form of the join predicate as well as tuple values.

Memory-constrained Environments. Query networks with large number of stateful operators like aggregates may also require load shedding due to insufficient memory. An aggregate, with window size ω and window slide δ has at most $\lceil \frac{\omega}{\delta} \rceil$ open window states per group. However, depending on the form of the aggregate function (i.e., distributive/algebraic vs. holistic), a window state may become unbounded [4]. We are planning to adapt our window drop approach to such memory-constrained environments.

Distributed Load Shedding. We are also extending this work into the distributed setting. Borealis [1, 3], the successor to Aurora, maps query networks to a large number of machines of varying computational capability (sensors, servers, and everything in between). In this case, load shedding requires that nodes cooperate in choosing where and how much load to shed. A downstream node will often request that its upstream nodes shed a certain amount of load on its behalf. That request can be further subdivided and passed (in part) upstream. We are currently investigating heuristic approaches to the distributed load shedding problem [28].

7. REFERENCES

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR Conference*, Asilomar, CA, January 2005.

[2] D. 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*, 12(2), August 2003.

[3] Y. Ahmad, B. Berg, U. Çetintemel, M. Humphrey, J. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik. Distributed Operation in the Borealis Stream Processing Engine (demo). In *ACM SIGMOD Conference*, Baltimore, MD, June 2005.

[4] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. *ACM TODS*, 29(1), March 2004.

[5] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *VLDB Conference*, Toronto, Canada, September 2004.

[6] A. Ayad and J. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *ACM SIGMOD Conference*, Paris, France, June 2004.

[7] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE ICDE Conference*, Boston, MA, March 2004.

[8] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *VLDB Conference*, Hong Kong, China, August 2002.

[9] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing Using Wavelets. In *VLDB Conference*, Cairo, Egypt, September 2000.

[10] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, Asilomar, CA, January 2003.

[11] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing Over Data Streams. In *ACM SIGMOD Conference*, San Diego, CA, June 2003.

[12] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing Complex Aggregate Queries over Data Streams. In *ACM SIGMOD Conference*, Madison, WI, June 2002.

[13] M. Garofalakis and P. Gibbons. Approximate Query Processing: Taming the Megabytes. In *VLDB Conference*, Rome, Italy, September 2001.

[14] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams Databases. In *ACM SIGMOD Conference*, Santa Barbara, CA, May 2001.

[15] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-pass Summaries for Approximate Aggregate Queries. In *VLDB Conference*, Rome, Italy, September 2001.

[16] L. Golab and T. Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2), June 2003.

[17] J. Hellerstein, P. Haas, and H. Wang. Online Aggregation. In *ACM SIGMOD Conference*, Tucson, AZ, May 1997.

[18] J. Kang, J. Naughton, and S. Viglas. Evaluating Window Joins over Unbounded Streams. In *IEEE ICDE Conference*, Bangalore, India, March 2003.

[19] Y. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *VLDB Conference*, Toronto, Canada, September 2004.

[20] 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 Conference*, Baltimore, MD, June 2005.

[21] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR Conference*, Asilomar, CA, January 2003.

[22] F. Reiss and J. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *IEEE ICDE Conference*, Tokyo, Japan, April 2005.

[23] U. Srivastava and J. Widom. Memory Limited Execution of Windowed Stream Joins. In *VLDB Conference*, Toronto, Canada, September 2004.

[24] Stream Query Repository. http://www-db.stanford.edu/stream/sqr/.

[25] N. Tatbul, M. Buller, R. Hoyt, S. Mullen, and S. Zdonik. Confidence-based Data Management for Personal Area Sensor Networks. In *DMSN Workshop*, Toronto, Canada, September 2004.

[26] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[27] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Data Streams. Technical Report CS-04-13, Brown University, Computer Science, November 2004.

[28] N. Tatbul and S. Zdonik. Dealing with Overload in Distributed Stream Processing Systems. In *IEEE International Workshop on Networking Meets Databases (NetDB'06)*, Atlanta, GA, April 2006.

[29] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE TKDE*, 15(3), May 2003.