No False Positives: Window-aware Load Shedding for Data Streams

Nesime Tatbul and Stan Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-05-06
April 2005

No False Positives: Window-aware Load Shedding for Data Streams

Nesime Tatbul Stan Zdonik

Brown University, Department of Computer Science, Providence, RI USA
{tatbul, sbz}@cs.brown.edu

Abstract

Data stream management systems may be subject to higher input rates than their resources can handle. In this case, results get delayed and Quality of Service (QoS) at system outputs may fall below acceptable levels. Load shedding addresses this problem by allowing data loss in exchange for reduced latency. In this paper, we describe a load shedding technique for queries consisting of one or more aggregate operators with sliding windows. We introduce a sophisticated drop operator, called a "Window Drop". This operator is aware of window properties (i.e., window size and window slide) of downstream aggregate operators in the query plan. Accordingly, it logically divides the 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 overall QoS utility.

1 Introduction

Stream processing engines (SPEs) like Aurora [7, 10, 20] 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 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 used. One model produces approximate answers by omitting tuples from the correct answer [5, 11, 18, 21, 24]. In this model, all delivered tuples are guaranteed to be a subset of the "exact" answer. In other words, we never produce *false positives*. This property is important since it allows the application to rely on the tuples that it receives to be correct. Our work is based on this "no false positives" model. An alternate approach to degrading the result is to emit nearly the same number of values, each of which might be inaccurate. The trick is to ensure that the errors are bounded by some amount. This is the approach that was taken in recent work [6]. This model is reasonable as well. 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. Thus, the mentioned approach [6] is complementary to the approach taken here.

Let's consider some examples for which no false positives makes sense. In many cases, the output stream is a sequence of updates. Each new tuple updates the previous one. For example, for a query network that produces 1-second temperature averages, a missing value will be corrected when the next one is received one second later. Losing a tuple increases the staleness of the previous report; however, for slowly changing variables, this might not be so bad. As a second example, consider the case of a distributed, multiplayer game. A stream processing engine could be used as the publish/subscribe engine that distributes events to interested client machines. 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.

On the other hand, *aggregates* play an important role in many data stream applications. Often we would like to make a computation on ranges of consecutive input tuples (a window). This computation could be in the form of a traditional SQL-style function such as sum, count, average, min, max as well as an arbitrary user-defined function. As a matter of fact, UDAs have been supported by recent stream processing systems [2, 19]. Moreover, *nested* aggregates as well as *user-defined* ones have proven to be essential in various applications from habitat monitoring with sensors to online auctions, as exemplified in the Stream Query Repository [22]. Lastly, the Linear Road stream data management benchmark also includes such aggregation queries [4].

As a concrete example, consider the query network 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.

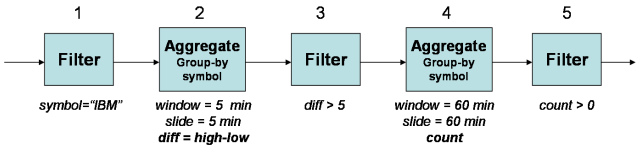


Figure 1: An example aggregation query

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 4 and 5 in Figure 1), or result in false positives if drops are pushed across them (e.g., [6]). Furthermore, if false positives (wrong answers) are produced through load shedding in the middle of the query network, 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 downstream aggregate and it must be at the leaf of the query tree).

In this paper, we introduce a new approach which applies drops in units of windows (hence the term "window-aware"). This approach never produces false positives 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. The main contributions of our work can be summarized as follows:

- We propose a novel load shedding approach for aggregation queries which delivers subset results. This technique is general enough to handle arbitrary aggregate functions and multiple levels of aggregate nesting.
- We mathematically analyze the correctness and performance of the proposed approach.
- We experimentally show the performance of our approach on a stream processing system prototype.

The rest of this paper proceeds as follows: Section 2 presents an overview of models and assumptions underlying the stream processing environment that we consider. Our 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 Models and Assumptions

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

2.1 System Overview

Aurora is a data stream management system for processing continuous queries over data streams [2]. Queries are defined through a boxes-and-arrows-based dataflow diagram. Each box represents an Aurora 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 box.

A stream is an append-only sequence of tuples with a uniform schema. Queries are composed using the operators defined by the Aurora Stream QUery ALgebra (SQuAl) [2], including Filter, Map, Union, Aggregate, Join, BSort, and Resample. Briefly, Filter applies a predicate to tuples and retains those for which the predicate is true. Map applies a function to each stream element. Union merges two input streams into one. Aggregate applies a function on a window of tuples. Join correlates tuples from two streams that are within the same time band. BSort is an approximate sort operator. Finally, Resample is used to align pairs of streams.

Aurora has a number of run-time components including a Scheduler, a Storage Manager, and a Load Shedder. The scheduler makes decisions about operator execution order [8]. The Storage Manager is responsible for efficient storage and retrieval of tuple queues on arcs between operators. Finally, the Load Shedder is responsible for managing overload due to high input rates [24].

Most of Aurora optimizations are driven by a Quality of Service (QoS) model. Query results are sent to external applications for which a number of QoS functions are defined. For each application, these functions relate a characteristic of the output to its utility. For example, a delay-based QoS function maps tuple latencies to utility values such that as tuples get delayed, their utility degrades.

The Load Shedder component, which is the focus of this paper, continuously monitors CPU load of a running Aurora query network. If an overload is detected, drop operators are inserted into the query network. These operators discard tuples based on a criteria linked to QoS specifications at outputs. The goal is to remove excess load from the system with minimal degradation in QoS.

We introduced this general notion of load shedding for data stream management systems in our earlier work [24]. In this paper, we build on our previous framework to enable load shedding on queries with sliding window aggregates. Our load shedding framework uses QoS specifications defined at query outputs, various system statistics like relative input rates, operator costs and selectivities to generate a static sequence of drop insertion plans. Each plan corresponds to an overload level and gets activated whenever the system reaches that level. As load levels change, the load shedder adaptively switches from one plan to another. These plans are generated to guarantee minimal utility loss; i.e., for any overload level, drops are inserted such that utility loss for gained cycles in return is minimized [24].

One important assumption of this load shedding framework is that it always delivers subsets of query answers. In other words, even though tuples get dropped, queries deliver values all of which would also occur in the exact answer; no new values are generated. As such, this framework has a major shortcoming when windowed operators are involved: dropping individual tuples from streams does not guarantee subset results when such streams are to be processed by windowed operators. Windowed operators 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*, which is the main focus of this paper.

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 how an aggregate operator works, we describe how we model its two important building blocks: windows and aggregate functions.

2.2.1 The Window Model

Data streams are continuous sequences of 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 Aurora, 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). Additionally, Aurora uses a *sliding window* model in which a window's endpoints move by a given amount to produce the next window.

2.2.2 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, AVERAGE, COUNT, MIN, MAX) or a Postgres-style user-defined function (UDF). As in Postgres, aggregate functions in Aurora 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.

2.2.3 The Aggregate Operator

An aggregate operator $Aggregate(\mathcal{S}, \mathcal{T}, \omega, \delta, \mathcal{F}, \mathcal{G})$ has the following semantics. $Aggregate$ 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 the arrival time of the tuple. 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. $Aggregate$ outputs a stream of tuples of the form (t, v, g), one for each window W processed. t is the smallest timestamp of the tuples in W, v is the final aggregate value returned by the `final` function of $\mathcal{F}$, and g is the value of the group-by attribute(s) (skipped when $\mathcal{G}$ is not specified).

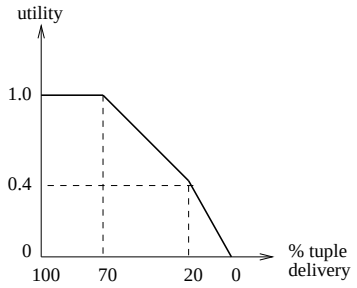


Figure 2: Loss-tolerance QoS function

$Aggregate$'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 + ω, 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 timestamp 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. 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 [25].

2.3 The Approximation Model

Approximate answers result from dropping tuples. In Aurora, we shed load such that the total utility loss measured at the output is minimized. To be more specific, a *loss-tolerance QoS function* maps the percent tuple delivery to utility values (see Figure 2). The larger the percentage of output tuples delivered, the higher its utility to the receiving application. In the case of a single query, minimizing utility loss corresponds to providing the largest possible subset of the actual query result. In the case of multiple queries, the load to shed from each is based on their (possibly different) tolerance to loss.

In addition to a loss-tolerance QoS function, we assume that each output application also specifies a threshold for its tolerance to gaps or "lack of responsiveness". We call the maximum gap that an application can tolerate the *batch size*. The system must guarantee that the number of consecutive output tuples missed due to load shedding never exceeds this value. As an example, we are working on a sensor network problem with the army [23], in which soldiers' physiological measurements must be delivered at least once per minute,

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

i.e., losing or choosing not to deliver results observed more frequently is acceptable.

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 $p = 1$).

3 Window-aware Load Shedding

Window-based computation is an essential component of stream processing systems. Often times query results are obtained by grouping tuples into windows in various ways, as described in the previous sections. Therefore, if the goal is to deliver subset results as approximate answers, the windowing mechanisms of a query must be taken into account when discarding tuples. Thus, load shedding on windowed queries must be *window-aware*.

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

3.1 The Window Drop Operator

A window drop operator $WinDrop(\mathcal{S}, \mathcal{T}, \omega, \delta, p, \mathcal{B})$ takes five parameters in addition to an input stream $\mathcal{S}$. $\mathcal{T}$ denotes the windowing attribute, ω denotes the window size, δ denotes the window slide, p denotes the drop probability, and $\mathcal{B}$ denotes the drop batch size. The $\mathcal{T}, \omega, \delta$ 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 from the QoS specifications at the query outputs. As will be described in Section 3.4, this parameter allows us to exploit an application's tolerance to gaps in making early drop decisions. We describe how values of these parameters are assigned in Section 3.2.

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*, with a default value of -1. This is a hidden attribute which is used internally by the system for QoS management purposes. 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 $\mathcal{T}$ value. 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 $A(\mathcal{T}, \omega, \delta, \mathcal{F})$ [1] and assume that we would like to place a $WinDrop$ before A. In order to drop p fraction from the output of A, we insert $WinDrop(\mathcal{T}, \omega, \delta, p, \mathcal{B})$ at A's input. Note that the first three parameters of $WinDrop$ are directly inherited from A so that $WinDrop$ can divide the input stream into windows in exactly the same way as A would. Then it decides which of those windows should be dropped and marks their starter elements. Finally, when a tuple t is received by A, A 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 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.

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 $WinDrop$ 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.

3.2.1 Pipeline Arrangement of Aggregates

When there is a sequence of operators such that each operator's output is fed into another one, we call such an arrangement a *pipeline arrangement*.

Assume that we have k aggregates, $A_i(\omega_i, \delta_i, \mathcal{F}_i)$ [2], $0 < i <= k$, pipelined in ascending order of i. We would like to drop p fraction from the output of A_k by placing

[1] We do not show $\mathcal{S}$ when the input stream is clear from the context.

[2] From here on, we drop the $\mathcal{T}$ parameter from both aggregate and window drop, simply assuming that windows are commonly defined on time.

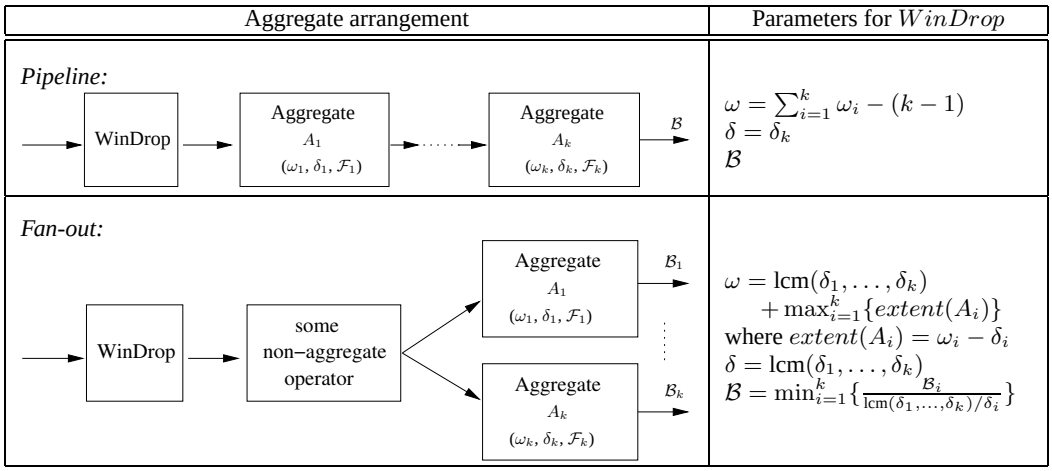


Table 2: Rules for setting $WinDrop$ parameters

$WinDrop(\omega, \delta, p, \mathcal{B})$ before the leftmost operator in the pipeline denoted by A_1 in the first row of Table 2. $WinDrop$ 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 outputs 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 3 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 1.

The pipeline arrangement rule can also be applied on the example that was presented in Figure 1. For this example, we would need a $WinDrop$ with a window size and slide of 64 and 60 minutes, respectively. Consequently, we would be missing the counts for some of the 1-hour windows when load is shed.

3.2.2 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, $A_1(\omega_1, \delta_1, \mathcal{F}_1), \ldots, A_k(\omega_k, \delta_k, \mathcal{F}_k)$, 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 $\text{lcm}(\delta_1, \ldots, \delta_k)$ and a window size of $\text{lcm}(\delta_1, \ldots, \delta_k) + \max(extent(A_1), \ldots, extent(A_k))$, where $extent(A_i) = \omega_i - \delta_i$. $\delta = \text{lcm}(\delta_1, \ldots, \delta_k)$ 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 encapsulates all aggregate siblings must have a window size of $\omega = \text{lcm}(\delta_1, \ldots, \delta_k) + \max(extent(A_1), \ldots, extent(A_k))$. 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

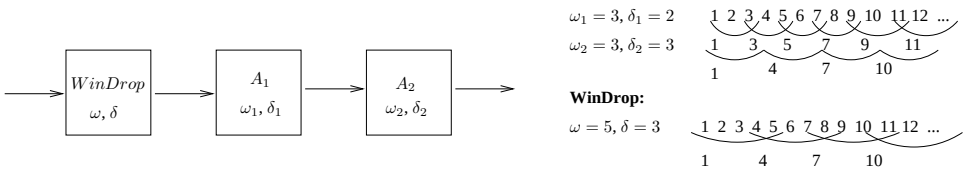


Figure 3: Pipeline example

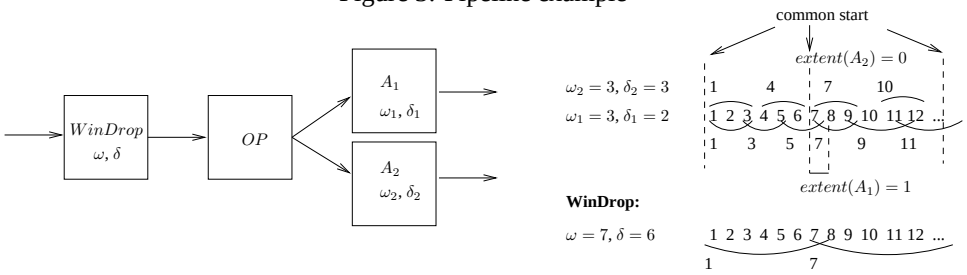


Figure 4: Fan-out example

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 $\text{lcm}(\delta_i)/\delta_i$ times. Hence, $\text{lcm}(\delta_i)/\delta_i$ consecutive windows for A_i translate to 1 window for $WinDrop$.

The example in Figure 4 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. 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.

3.2.3 Rule Composition

We will now briefly illustrate the composition of the rules in Table 2. Assume that OP in Figure 4 is an aggregate with $\omega = 4$ and $\delta = 1$ (i.e., $A_0(4,1)$). There are two ways to construct $WinDrop$:

1. We first apply the fan-out rule on A_1 and A_2, which gives us $WinDrop(7,6)$ as illustrated in Figure 4. Then we apply the pipeline rule on $A_0(4,1)$ and $WinDrop(7,6)$, which 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, $WinDrop$ 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 A with window size ω decodes the window specifications coded by a preceding $WinDrop$. First assume that A receives a tuple with time value t and according to the slide parameter of A, a new window has to start at t (upper half of Table 3). If the tuple has a positive window specification τ, then A 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 $\tau - \omega - 1$). A also makes sure that all successive tuples with time values up to $\tau - \omega - 1$ are retained in the stream. If the tuple has a non-positive (i.e., 0 or -1) window specification, then A checks if t is within the time range that it must retain. If so, a new window is opened with the given window specification and the keep range is set to $t + \omega$. If not, A skips this window.

Now assume that A receives a tuple with time value t where A does not expect to open a new window (lower half of Table 3). A 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, A does not need to do anything. If the tuple has a positive window specification τ, A updates its time range as well as the window specification of the tuple. In both of the non-negative cases, A 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.

We must point out here that fake tuples have one other important use. A query network may have other types of

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 = $t + \omega$
yes	0	beyond	skip window
yes	-1	within	open window keep_until = $t + \omega$
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

operators lying between a $WinDrop$ and the downstream aggregates which are supposed to decode window specifications generated by the $WinDrop$ (see the example in Figure 1). 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 skip a window at a particular time point.

3.4 Early Drops

$WinDrop$ is not just an operator that marks tuples; it can also drop tuples. We call this functionality an *early drop*. In this section, we discuss how a $WinDrop$ accomplishes early drops. We start with a useful definition.

Definition (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 $A(\mathcal{S}, \omega, \delta)$, where $\omega = m * \delta + \phi$, $m \geq 1$, $0 \leq \phi < \delta$. We define a Window Count Function $WCF : Z^* \rightarrow N$, which maps a given time value t to the number of consecutive windows to which tuples that carry t belong:*

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

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 (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 $WinDrop$ operator decides to drop all of these k windows.*

If $WinDrop$ 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 that, as part of its loss-tolerance QoS function, 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 translates 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.

For the pipeline arrangement, as we place $WinDrop$ further upstream, the gap between ω and δ widens and hence m grows. In this case, more consecutive windows must be dropped to achieve early drops. This would only be possible to the extent allowed by $\mathcal{B}$. Therefore, if an application defines its $\mathcal{B}$ small, then there is no point in placing $WinDrop$ further upstream than the earliest aggregate in the pipeline.

For the fan-out arrangement, the window slide δ gets larger across a split. This means that when a window is dropped upstream from a split, in fact, more than one consecutive window may get dropped downstream from the split. We need to make sure that this does not exceed $\mathcal{B}$ defined at the outputs. Table 2 shows how this is achieved by appropriately scaling $\mathcal{B}_i$. As in the pipeline arrangement case, $WinDrop$ must be placed at the earliest point in the query network, not violating the constraints on $\mathcal{B}$.

3.5 Analysis

Next we mathematically analyze our approach for correctness and performance.

3.5.1 Correctness

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

Theorem. *$WinDrop$ 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.

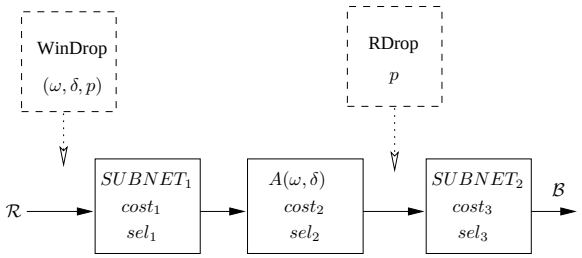


Figure 5: Inserting drops into an aggregation query

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 $WinDrop$ 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 as given in Table 2. 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. □

3.5.2 Performance

In this section, we analyze the CPU cycles saved by a $WinDrop$. We also compare our approach against an alternative one which uses a random drop [24]. Consider a query network as in Figure 5, 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 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 $WinDrop$ were inserted at the query input, the CPU load would become:

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

$SUBNET_1$ 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, former are processed normally ($sel_{w1} * cost_1$), and 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:

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

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 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 $WinDrop$ 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 6). 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. 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 $sel_{w2} = p * \frac{(\mathcal{B}+1) + \lfloor \frac{\omega}{\delta} \rfloor}{(\mathcal{B}-1)*\delta + \omega}$.

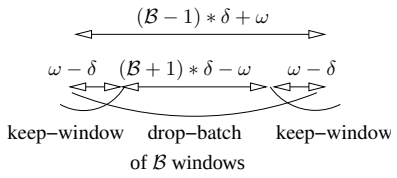


Figure 6: Drop-batch

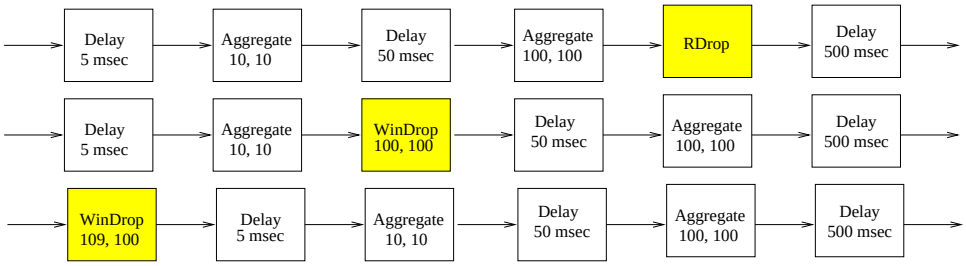


Figure 7: Drop insertion plans (RDrop, Nested1, and Nested2)

4 Experiments

In this section, we experimentally evaluate our window drop approach. We first compare it against a simple random drop approach which provides subset approximation by shedding individual tuples downstream from the aggregates. Note that we do not compare our approach against the relative error-based approaches (e.g., [6]) since there is no fair way to do this when error models are different. We vary both window parameters and input rates, and measure the result degradation for various query plans. We also compare our experimental results against the analytical estimates of Section 3.5.2 to confirm their validity. Finally, we evaluate the processing overhead of our technique.

We implemented the WinDrop operator as part of the load shedder component of Aurora, which itself serves as the query processor component of the Borealis distributed stream processing prototype system [1]. We conducted our experiments on a single-node Borealis server, running on a Linux PC with an Athlon 64 2GHz Processor.

We constructed 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.

4.1 Basic Performance

First we will show the basic performance of window drop. 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 boxes before and after each aggregate to model other operators that may exist upstream and downstream from the aggregates. A delay box simply withholds its input tuple for a specific amount of time before releasing it to its successor box. 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 WinDrop 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 WinDrop is inserted in the middle of the query network; the other is random drop [24] that is placed downstream from both of the aggregates. Figure 7 illustrates these three alternative drop insertion plans.

Figure 8 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 fraction of the answer that is missing from the result. RDrop represents random drop,

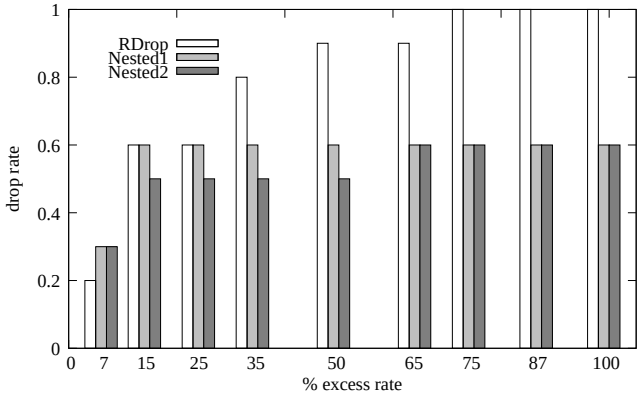


Figure 8: Comparing alternative approaches

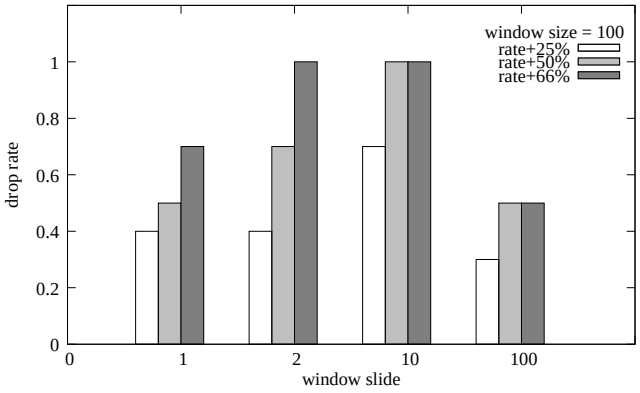
Nested1 corresponds to WinDrop inserted in the middle, and Nested2 is for WinDrop placed at the input. At relatively low input rates, RDrop shows comparable performance to window drop approaches. However, as the rate gets higher, both Nested1 and Nested2 scale far better than the RDrop approach. In fact, 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 [24] since we could not move drops past aggregates. Placing WinDrop further upstream in a nested aggregation query significantly improves the result quality.

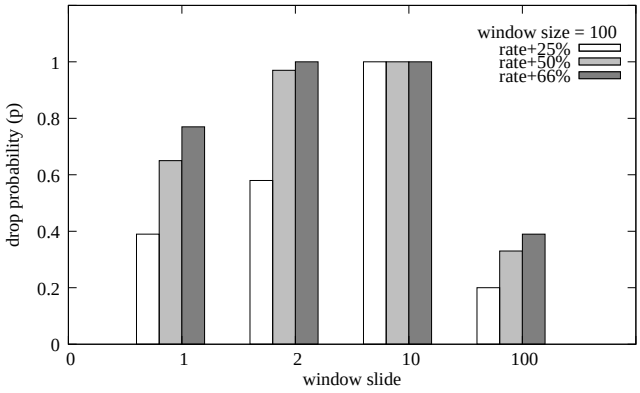
4.2 Effect of Window Parameters

Next we investigate the effect of window parameters on WinDrop performance. We used a simple query with one aggregate operator with a count function as shown in Figure 9. We again added delay boxes of 5 milliseconds each, before and after the aggregate, and set batch size to 10.

The bar chart in Figure 9 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. This is because WinDrop can achieve early drops in this case. A second observation is that drop rates stay almost



(a) Experimental result



(b) Analytical estimate (from Section 3.5.2)

Figure 10: Effect of window slide

fixed as the window size increases. Interestingly, the formulas presented earlier in Section 3.5.2 also suggest that load should be independent of 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. The latter case is experimentally confirmed in Figure 9. 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 box is scheduled. Our analysis models the average case behavior and fails to capture such cases where internal load variations may occur due to changes in box scheduling frequency.

Figure 10 details the effect of window slide on WinDrop performance: Figure 10(a) shows our experimental result and Figure 10(b) plots the analytical estimates of Section 3.5.2. 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 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 $\lfloor \frac{\omega}{\delta} \rfloor \leq \mathcal{B}$). WinDrop shows the best advantage as the degree of window overlap decreases. We continue to observe this effect as excess load increases. Our analysis captures the general behavior very well, but as window slide grows, it shows a diversion from the measured results, for the same reason as above.

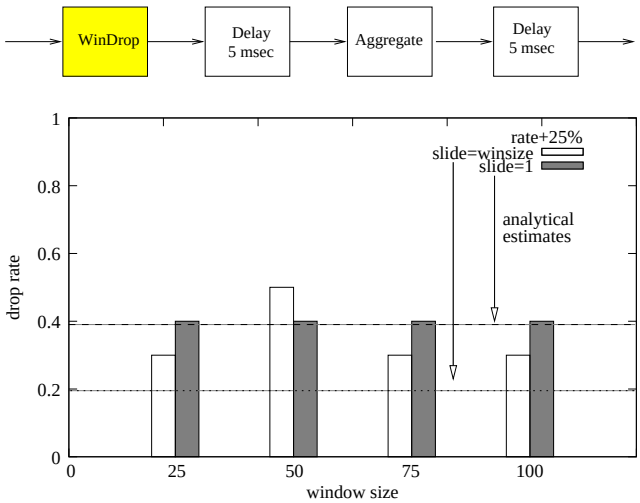


Figure 9: Effect of window size

As a brief note, we also compared WinDrop with random drop inserted after the aggregate. For slide=1, random drop performs almost the same as WinDrop (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, WinDrop converges to the simple random drop. As slide grows, WinDrop achieves a clear advantage.

4.3 Processing Overhead

Next we evaluate the overhead of adding WinDrop 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 11. 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 when query contains a WinDrop that does not drop anything ($p = 0$) and for when no WinDrop is present. The rate is 50% higher than the capacity level, and we ran each query for a minute. D denotes the amount of delay in the Delay box, representing the cost of the subnetwork between the filter and the aggregate. Since no tuples are dropped in either case, the reduction (if any) in the number of tuples produced with WinDrop must be due to the additional processing overhead. First, Table 4 shows that in general, the overhead is low. Second, as window size increases, the overhead decreases. This is due to the fact that WinDrop marks tuples less frequently. Third, for lower selectivity, overhead seems to be higher. This result accounts

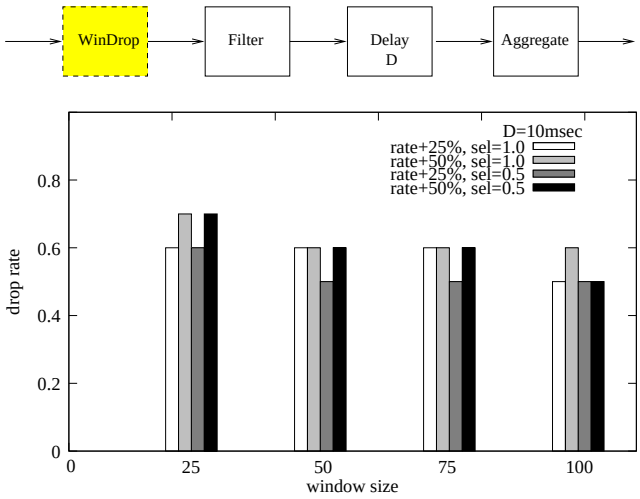


Figure 11: Filter selectivity

	D=10msec		D=20msec	
win_size	sel=1.0	sel=0.5	sel=1.0	sel=0.5
25	0.99	0.96	1.0	0.96
50	0.99	0.98	0.98	0.97
75	1.0	0.98	1.0	0.97
100	1.0	1.0	1.0	1.0

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

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 our results indicate. Lastly, we see only a slight increase in overhead when the cost of processing between filter and aggregate is doubled.

Figure 11 shows the degradation in result quality as window size, filter selectivity and input rate changes. We do not observe any significant difference in drop amounts due to processing overhead, confirming the results in Table 4.

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, including Aurora [2], STREAM [20], and Telegraph [10]. 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 [6]. 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 produces subset results and tries to minimize QoS utility loss, hence our approximation model is quite different. Moreover, our approach can handle a more general class of aggregation queries by preserving window integrity, while the proposed work applies to 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.

Load shedding for sliding window joins in memory-limited environments have also been studied [5, 11, 18, 21]. Our approach mainly considers CPU as the limited resource although it can also be used to solve the memory-limited case. The cited works also produce subset results. The goal in these works is to either produce maximum subsets or sampled subsets, whereas we determine relative subset sizes based on QoS specifications.

Punctuations are special annotations embedded into data streams to specify end of a subset of data in the stream [26]. 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 both approaches annotate streams with information that is important in terms of optimizing query execution, their goals are quite different. Furthermore, in punctuations case, annotations indicate some property that naturally exists in the stream, whereas in our case window specifications are artificially generated to cope with overload. It would be interesting to investigate how our window-aware load shedding idea could be used under a windowing model based on punctuations.

Lastly, approximate query processing techniques have long been studied both for 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 between by interleaving sampling with query evaluation, on stored data. 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 no false positives 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 Aurora, 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 networks. Thus, 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 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:

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. WinDrop 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 [3]. We are planning to adapt our window drop approach to such memory-constrained environments.

Distributed Load Shedding. We also plan to extend this work into the distributed setting. Borealis [1], 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.

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] 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.

[4] 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.

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

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

[7] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal*, 13(4), December 2004.

[8] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, Sep 2003.

[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, 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] 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.

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

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

[23] 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.

[24] 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.

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

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