Window-aware Load Shedding for Data Streams

Nesime Tatbul and Stan Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-04-13
September 2004

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. Drop operators are placed at carefully chosen points in a query plan, in order to relieve overload with minimal loss in answer quality. 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 "Windowed 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 partitions 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 [5, 8, 18] 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 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.

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

We have studied this problem earlier in a limited context [20]. In that work, load shedding is accomplished by inserting load reducing operations (i.e., random drops or semantic filters) into the query network. The drops are initially inserted at the outputs and then moved closer to the inputs to avoid dropping tuples in which we have already invested significant work. Our earlier work, however, was not able to move a drop across an aggregate operator. This is a serious limitation since aggregates are potentially expensive and play an important role in many streaming applications. This paper attempts to remedy that.

In our previous work [20], we produced an approximate answer that only omitted tuples from the correct answer. Thus, all tuples that were delivered are a part of the real answer. In other words, we never produced *false positives*. This property is important since it allows the application to rely on the tuples that it receives to be correct. In many cases, the answer shows some periodicity. For example, for a query network that produces 5-second temperature averages, a missing value will be corrected when the next one is received. Moreover, if false positives are produced through load shedding in the middle of the query network, it is unclear how this error will propagate through subsequent downstream operators. The approach taken in this paper preserves this "no false positives" principle.

Nested aggregates occur naturally in a system like Aurora. Built-in aggregates can be nested as in the 1-hour minimum of 5-minute averages. Aurora also supports user-defined aggregates as a way to produce answers that depend on a range of input values (a window). This capability has been used extensively in real applications. For example, in one financial service application, the quality of a feed can often be determined by the delay between ticks [5]. We use a user-defined aggregate to identify these delays by forming windows as pairs of tuples. For each window, the aggregate emits a tuple that contains the time difference between the two tuples. This stream is then filtered to retain differences that are larger than a given threshold. A downstream aggregate operator then counts these extreme delays. This kind of behavior can nest to an arbitrary depth. Thus, this work deals well with nested aggregates.

The rest of this paper is outlined 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. Section 4 briefly discusses how our basic approach can be extended in various ways. Theoretical and experimental evaluation of our approach is presented in Section 5. Section 6

summarizes the related work in this area. Finally, we conclude in Section 7 outlining potential avenues for further research.

2 Models and Assumptions

The work presented in this paper is part of the Aurora stream processing system [1]. We first give a brief overview of Aurora 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 Aurora and its Load Shedder

Aurora is a data stream management system for processing continuous queries over data streams [1]. 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 single 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) [1], 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. We omit further details about these operators and refer the interested reader to [1]. For the purposes of this paper, we assume that query networks do not contain any Join, BSort or Resample operators.

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 [6]. 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 [20].

Most of Aurora optimizations are driven by its 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 [20]. 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 has a corresponding 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. To generate these plans, we first identify the candidate drop locations in a query network. Next, for each location, we compute a loss/gain ratio, indicating the unit QoS utility loss per CPU cycles gained if a drop were inserted at that location. These locations are sorted in increasing order of their ratios. Finally, drop insertion plans are generated by processing each location in order. This algorithm guarantees minimal QoS utility loss; i.e., for any overload level, drops are inserted such that utility loss for gained cycles in return is minimized.

One important assumption of our 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. This is implied by our QoS model which provides us with two alternative functions:(1) a loss-tolerance QoS function, which maps percent tuple delivery to utility values, (2) a value-based QoS function, which maps tuple values to utility values. The load shedder uses (1) for random load shedding, where tuples are dropped based a drop probability, and (2) for semantic load shedding, where tuples are dropped based on a drop predicate.

The above 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 *Windowed 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.

The Window Model. Data streams are continuous data records that may be infinitely large in number. Traditional set operations like join or aggregate may block or may require unbounded memory if there is no end to data arrival. On the other hand, most applications require processing on excerpts of a stream rather than the whole. Each such excerpt is called a *window*. Windows can be modeled in various ways [14]. In Aurora, there are two ways to physically build windows: (i) time-based windows, and (ii) count-based windows. In the former, an attribute is designated as the windowing attribute (usually time) and consecutive tuples whose such attribute is within a certain interval constitute a window (e.g., stock reports over the last 10 minutes); whereas in the latter, a certain number of

consecutive tuples constitute a window (e.g., the last 10 readings from the temperature sensor) [1]. Additionally, Aurora uses a *sliding window* model where window endpoints on the stream move by a certain amount to produce new windows.

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 it 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 $Aggregate(\mathcal{S}, [\mathcal{T}], \omega, \delta, \mathcal{F}, [\mathcal{G}])$ 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* [2]. In practice, this attribute 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 the group-by attribute $\mathcal{G}$ [3]. Each substream constitutes a sequence of windows on which the aggregate function $\mathcal{F}$ is to be 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 (skipped when $\mathcal{G}$ is not specified). Until Section 4.1, we will assume that $\mathcal{S}$ is a single group.

$Aggregate$'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 $\mathcal{T}$ (i.e., time-based window), (ii) in terms of number of tuples (i.e., count-based window). Unless otherwise stated, we assume the time-based windowing scheme. Section 4.2 discusses extensions to the count-based model. 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 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*; and a *sliding window* otherwise. Tumbling windows constitute an interesting case because they partition a stream into non-overlapping consecutive windows.

[1] Note that in (i), tuples are assumed to arrive to the aggregate in increasing order of their windowing attributes.

[2] If $\mathcal{T}$ is not specified, $Aggregate$ requires no order on its input stream.

[3] If $\mathcal{G}$ is not specified, then $\mathcal{S}$ is treated as a single group.

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 1). 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.

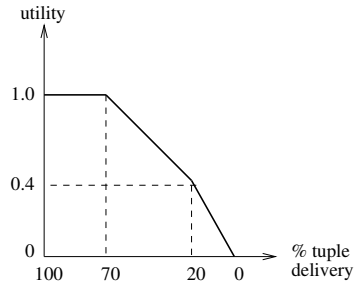


Figure 1. Loss-tolerance QoS function

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 the *batch size*. The system must guarantee that the number of consecutive output tuples missed due to load shedding never exceeds this value.

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 percentage of tuples delivered can never be below $1/(\mathcal{B} + 1)$. If overload requires more drops, then some applications may have to be completely shut down.

3 Window-aware Load Shedding

In this section, we present our 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 *Windowed Drop* operator.

3.1 The Windowed Drop Operator

A windowed 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 batch size. The $\mathcal{T}, \omega, \delta$ parameters of $WinDrop$ are derived based on the properties of the downstream aggregate operators. p is determined by the load shedder according to the amount of load to be shed along the subquery downstream from $WinDrop$. Finally, $\mathcal{B}$ is derived from the QoS specifications at

Window spec	Description
-1	don't care
0	window disallowed
τ	window allowed; must preserve tuples with $ts < \tau$

Table 1. Window specification attribute

query outputs. 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 partitions 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 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, as explained below.

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 certain tuple t, $WinDrop$ sets the window specification attribute of t to a positive timestamp value. This value not only indicates that a window can start at this tuple, but also indicates until which timestamp 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.

When a tuple t is received by an aggregate, aggregate examines the window specification attribute and skips windows that are disallowed. As a result of this, we save system resources at multiple levels:

- When a window is skipped, aggregate need not open and maintain state for that window.
- Aggregate does less work upon seeing tuples that arrive immediately after t because there is one less open window that those tuples can contribute to.
- Less state means less memory usage.
- When aggregate skips a window, it produces no output for that window, thereby reducing the data rate and saving from processing in the downstream subnetwork.

The last effect could also be achieved by inserting a *random drop* [20] right after aggregate. As such, it is not purely a contribution of the windowed drop. However, it is important to note that in the presence of multiple aggregates in a query, a simple random drop could only be placed after the very last aggregate in the query in order to guarantee subset results. Windowed drop removes this limitation and can be placed upstream from any of the aggregates in the query, thereby saving from processing early in the query network. *Our goal is to perform drops as early as possible while retaining the same effect of placing a random drop following final aggregate in a path.*

An alternative approach to windowed drop could be to have the aggregates themselves decide which windows to skip. There are several reasons why using a separate windowed drop operator is more desirable: First, in the presence of a sequence of aggregate operators, the first aggregate in the sequence can not decide which windows to drop without considering the windowing properties of the downstream aggregates. It has to not only preserve its own window integrity but also window integrity of all downstream aggregates. Second, windowed drop not only encodes window specifications into tuples, but can also drop tuples under certain circumstances, which we call an *early drop* (discussed in Section 3.4). More specifically, tuples that are marked with a non-positive window specification attribute and that are beyond the timestamp range imposed by the most recently seen positive window specification attribute can be dropped right away, without waiting to be seen by a downstream aggregate. We would like such an operator to be placed as early in the query as possible to save the most processing. By having a separate operator, we can move it further upstream from the aggregate. If the window drop logic was embedded inside each individual aggregate, we would not be able to detach and push it upstream. Lastly, we would like to avoid changing the original aggregate implementation to the extent possible for easy deployment of our technique. With windowed drop, the only change needed to the aggregate code is the addition of the decoding logic to interpret window specifications annotated into tuples (see Section 3.3).

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 windowed drop parameters for these two arrangements. Any query network can be handled using a composition of these two rules. We now discuss how we derived these rules in detail.

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*. First consider the simple case where there is a single aggregate operator A. We would like to place a windowed drop D in front of A. Given an aggregate operator $A(\mathcal{T}, \omega, \delta, \mathcal{F})$, in order to drop p fraction from the output of A, we insert a windowed drop $D(\mathcal{T}, \omega, \delta, p, \mathcal{B})$ at A's input [4]. Note that the first three parameters of D are directly inherited from A so that D can partition the input stream into windows in exactly the same way as A would. Then it decides which of those windows should be dropped.

To generalize, assume that we have k aggregates, $A_i(\omega_i, \delta_i, \mathcal{F}_i)$, $0 < i <= k$, pipelined in the order of i[5]. We

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

[5] From here on, we drop the $\mathcal{T}$ parameter from both aggregate and windowed drop; simply assuming that windows are commonly defi ned on time.

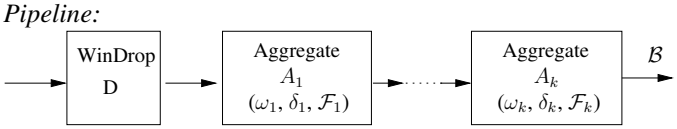
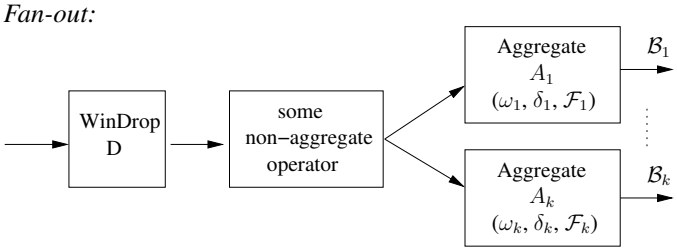
Aggregate arrangement	Parameters for D
Pipeline: 	$\omega = \sum_{i=1}^{k} \omega_i - (k-1)$ $\delta = \delta_k$ $\mathcal{B}$
Fan-out: 	$\omega = \text{lcm}(\delta_1, \ldots, \delta_k)$ $+ \max_{i=1}^{k}\{extent(A_i)\}$ where $extent(A_i) = \omega_i - \delta_i$ $\delta = \text{lcm}(\delta_1, \ldots, \delta_k)$ $\mathcal{B} = \min_{i=1}^{k}\{\frac{\mathcal{B}_i}{\text{lcm}(\delta_1,\ldots,\delta_k)/\delta_i}\}$

Table 2. Rules for setting Windowed Drop parameters

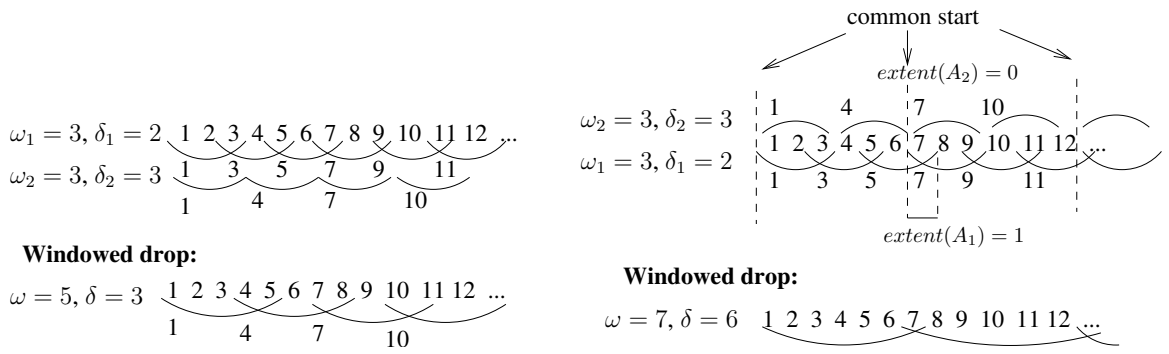
would like to drop p fraction from the output of A_k by placing a windowed drop $D(\omega, \delta, p, \mathcal{B})$ in front of the leftmost operator in the pipeline denoted by A_1 in Table 2. D must have a slide δ that is equal to the slide of the last aggregate A_k in the pipeline. The reason for this is that A_k is the last operator that partitions the stream into windows and produces one output every δ_k time units. Dropping p fraction from A_k's output requires that we make a drop decision once every δ_k time units. Furthermore, D 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 timestamp 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, D 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 when each aggregate's window slide is such that the last timestamp in a window opens up a new window. As such, it is an upper bound on the required window size for D. Finally, the batch size parameter $\mathcal{B}$ of D is assigned as specified by the output application connected to the pipeline. The example in Figure 2(a) illustrates this rule.

Fan-out Arrangement of Aggregates. 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 windowed drop D which makes window keep/drop decisions that are common to all the 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 Table 2. A common D 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 it's 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 D 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. The example in Figure 2(b) illustrates this rule.

Finally, the batch size for D, i.e., the maximum number of consecutive windows that D can drop 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 D slides once, A_i slides $\text{lcm}(\delta_i)/\delta_i$ times. Hence, $\text{lcm}(\delta_i)/\delta_i$ consecutive windows for A_i translates to 1 window for D.

3.3 Decoding Window Specifications

Table 3 summarizes how an aggregate A with window size ω decodes the window specifications coded by a preceding windowed drop. First assume that A receives a tuple with timestamp t and according to slide 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



(a) Pipeline example

(b) Fan-out example

Figure 2. Illustration of aggregate arrangement rules (Numbers indicate tuple timestamps.)

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

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 timestamps 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 timestamp range that it must retain. If so, a new window is opened with the same 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 timestamp 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 to use it. The two important specifications are τ and 0, former indicating opening of a window and latter 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 the timestamp range it has to maintain and also updates the window specification of the tuple. In both of the non-negative cases, A marks this tuple as a *fake tuple*, i.e., a tuple which has no real content but only carries window specification value that may be significant to some downstream aggregates. Such tuples should not participate in aggregate computation and should be solely used for decoding purposes. As a related note, a query network may have other types of operators between a windowed drop D and the downstream aggregates which will decode window specifications of D. We must make sure that window specifications survive through such operators. For example, assume that a filter 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 must mark t as a fake tuple and let it through the filter because t is carrying a message for the downstream aggregates.

3.4 Early Drops

In this section, we discuss opportunities for early drops by windowed drop operators and how the batch size parameter $\mathcal{B}$ is used to support early drops.

Definition 1 (Window Count Function (WCF)). *Consider a stream $\mathcal{S}$, which has possibly an infinite number of tuples which are partially ordered in increasing order of their timestamps* [6]*. Assume that the very first tuple in $\mathcal{S}$ has a timestamp of θ. Consider a windowed aggregate A (through which the stream $\mathcal{S}$ flows), with window size ω and window slide δ, where $\omega \geq \delta$, $\omega = m * \delta + \phi$, $m \geq 1$, $\phi < \delta$. We define a Window Count Function $WCF : Z^* \rightarrow N$, which maps a given tuple timestamp to the number of consecutive windows to which tuples with the given timestamp would belong:* [7]

[6] Timestamps are represented as non-negative integers.

[7] Note that each tuple belongs to at least one window when $\delta \leq \omega$.

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

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. When the aggregate window is tumbling (i.e., $\omega = \delta$), then the second case has a timestamp 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 t with timestamp t_m belongs to k windows, i.e., $WCF(t_m) = k$, then t can be early-dropped if and only if the windowed drop operator D decides to drop all of the k windows that the tuple belongs to.*

If D 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, D 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, then the probability of an early-drop is very small. Instead, to take advantage of early drops, we suggest the following (more deterministic) drop mechanism: 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, D flips a coin and drops $\mathcal{B}$ consecutive windows with probability p. Based on the window count function WCF, dropping $\mathcal{B}$ consecutive windows translates to a certain range of early drops.

For the pipeline arrangement, as we place D 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. The overall drop percentage is still under our control (due to p), but we have to sacrifice more contiguous drops, which may be undesirable for some applications. So, if an application defines its $\mathcal{B}$ small, there is no point in pushing D further than the earliest aggregate in the pipeline as aggregates can still benefit from window drops regardless of if an early-drop is possible or not.

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, D must be placed at the earliest point in the query network, not violating the constraints on $\mathcal{B}$.

4 Extensions

4.1 Multiple Groups

Windowed drop operator takes an additional parameter $\mathcal{G}$ to denote the group-by attribute. Thus, it recognizes substreams for each group and makes window keep/drop decisions on each separately. Windowed drop uses the same ω, δ, $\mathcal{T}$ parameters on all groups. However, the p and $\mathcal{B}$ parameters may have to be assigned differently for each group.

First of all, some groups may be more important than others and therefore they may have more strict QoS requirements. For example, given an aggregate which computes the average price for the last 1 hour's stock prices grouped by company, the application may have defined loss-tolerance to Company A tuples to be smaller compared to Company B tuples. Besides, the application may not want to miss more than 5 output tuples in a row for A, whereas it may tolerate to miss upto 10 consecutive tuples for B. Hence, we need to extend windowed drop to have a separate drop probability p and batch size $\mathcal{B}$ for each group-by value. Secondly, even if the groups have the same QoS, group tuples into an aggregate may be arriving at different rates, bearing different selectivities and contributing to CPU load at different amounts. Therefore, shedding different amounts of load from each group may be more optimal. As a result, our basic approach can be easily extended to handle multiple groups by:

- allowing an output application to define separate loss-tolerance QoS function and batch size for each group,
- allowing a windowed drop to accept separate p and $\mathcal{B}$ parameters for each group,
- computing separate loss/gain ratios for each group and treating them like independent drop locations when generating drop insertion plans,
- keeping track of group statistics along with operators'.

The rest of the algorithms can be applied in the same way within each group.

4.2 Count-based Windows

We have so far assumed time-based windows for aggregates. We now discuss the extensions to handle the count-based ones.

When windows are specified in terms of the number of arriving tuples, the windowing attribute becomes redundant. However, for uniform treatment of both windowing schemes and to simplify the tuple counting process, we could replace the timestamp attribute with a tuple-id attribute. In this case, each operator must renumber tuples that it outputs. Also, positive window specification attribute τ in Table 3 now refers to the number of tuples to be preserved (or in case of tuple-id's, an id range), rather than a time range.

Although the fan-out arrangement rule in Table 2 directly applies to count-based windows, a modification is required to the pipeline arrangement rule. The window size parameter for D across a pipeline of aggregates must be modified as $\omega = \omega_k * \prod_{i=1}^{k-1} \delta_i + \sum_{i=1}^{k-1} (extent(A_i) * \prod_{j=1}^{i-1} \delta_j)$, and the window slide parameter for D must be modified as $\delta = \prod_{i=1}^{k} \delta_i$.

There is an additional problem with count-based windows when filters and unions are involved. A filter may remove tuples from the stream when its predicate returns false on them. This would cause its downstream aggregates to count tuples in a different way than a windowed drop operator upstream from the filter. Similarly, a union merges two streams into one. Therefore, a windowed drop counting tuples upstream from the union would do it in a different way than an aggregate that is downstream from the union. In both cases, window integrity would be lost. As a result, under the count-based windowing model, windowed drops must never be placed across filters and unions.

4.3 Memory-constrained Environments

We have primarily considered CPU-limited environments so far. However, high data rates or other characteristics of input streams may congest other system resources, too. For example, windowed operators in general have high demand on memory.

Memory requirements for queries over data streams have been characterized in recent work, where queries are classified based on whether they can be evaluated using bounded memory or not [2]. Bounded memory computable queries require bounded memory independent of the input characteristics. This work shows that an aggregate query can be computed in bounded memory *only when* there is no aggregate expression with holistic function on an unbounded attribute and every grouping attribute is bounded.

In this section, we assume a system with enough CPU to handle input rates, but a fixed amount of memory M which may be insufficient due to a large number of aggregate operators, each with a large number of simultaneously open window states. We briefly discuss how our load shedding approach could be utilized to handle memory overload in such a system.

Since we assume that there is no CPU overload, i.e., tuples are consumed as fast as they arrive, queue build-up at operators is not a concern in terms of memory. An aggregate, with window size ω and window slide δ has at most $\lceil \frac{\omega}{\delta} \rceil$ open window states per group at any given time (see WCF defined in Section 3.4). If aggregate function is *distributive* (e.g. SUM), a window state is occupied by a single scalar value, which is the aggregate value based on the tuples seen so far (e.g., current sum). If aggregate function is *algebraic* (e.g. AVERAGE), a window state consists of a synopsis of the tuples seen so far (e.g., current sum and count) from which the final result can be computed. In both cases, a window state consumes a fixed amount of memory. However, if the function is *holistic* (e.g. MEDIAN), then a window state may be unbounded under the time-based windowing model as the number of tuples arriving during a window period may be arbitrarily large. Assume that a window state takes up μ amount of memory. If the aggregate has seen g groups so far, then the maximum total memory consumed by that aggregate is $g * \lceil \frac{\omega}{\delta} \rceil * \mu$. Dropping p fraction of the windows from a group would reduce memory requirements for that group by a factor of p. However, as in the CPU-constrained case, this would lead to QoS utility loss for that group. As before, we need to compute loss/gain ratios for each group at each drop location (as discussed in Section 4.1) and drop sufficient amounts from these groups in increasing order of their ratios until expected memory consumption drops below M. Here, loss refers to QoS utility loss at outputs whereas gain refers to total memory gain. As a result, our approach can be easily adapted to memory-constrained environments by a simple change in load formulation. An important point is that each aggregate and each group within an aggregate may have different contribution to the overall memory consumed to due to its window, function, and input characteristics (e.g., holistic function on a time-based window with unbounded number of groups requires more memory). Therefore, memory gain per window dropped from each group may frequently change and this has to be reflected on the drop insertion plans for optimal load shedding. It is a future work item to investigate load shedding in environments with simultaneous overload on multiple resources (i.e., CPU and memory overload occurring at the same time).

5 Evaluation

This section presents a theoretical and experimental evaluation of our approach.

5.1 Load Analysis

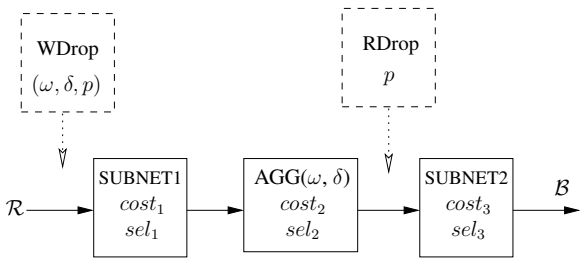


Figure 3. Inserting drops into an aggregation query

In this section, we analyze how much CPU cycles are saved by a Windowed Drop versus a Random Drop. Consider a query network as in Figure 3, where an aggregate with window size w and slide δ 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:

$$L = \mathcal{R} * (cost_1 + sel_1 * (cost_2 + sel_2 * cost_3))$$

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 Windowed Drop were inserted at the query input, the CPU load would become:

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

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

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

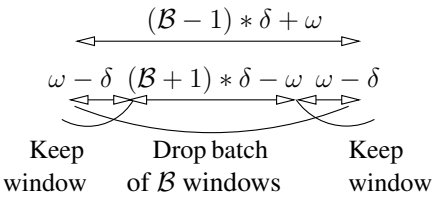


Figure 4. Drop batch

$$+(1-sel_{WDrop})*cost_1$$
$$+sel_1*p*cost_2)$$

If we compare S_{RDrop} with S_{WDrop}, we see that S_{WDrop} has two additional savings terms. It is guaranteed to save from the aggregate operator's cost ($cost_2$) with an amount proportional to p. Secondly, it may save an additional amount from the first subnetwork's cost ($cost_1$) when $sel_{WDrop} < 1$, as a result of early drops.

Let us now briefly show how Wdrop's tuple selectivity sel_{WDrop} can be estimated. For simplicity, we will assume a stream with one tuple per timestamp. 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 4). 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. However, one tuple out of every δ tuples may have to be retained as a fake tuple since it carries a 0 window specification. Therefore, the number of tuples that can be early-dropped is estimated as: $(\mathcal{B}+1)*\delta-\omega-\lfloor\frac{(\mathcal{B}+1)*\delta-\omega}{\delta}\rfloor=(\mathcal{B}+1)*\delta-\omega-(\mathcal{B}+1)+\lfloor\frac{\omega}{\delta}\rfloor$. This many tuples out of a total of $(\mathcal{B}-1)*\delta+\omega$ can be early-dropped and this would occur with probability p. Thus, we can estimate sel_{WDrop} as:
$sel_{WDrop}=1-\frac{p}{(\mathcal{B}-1)*\delta+\omega}*((\mathcal{B}+1)*\delta-\omega-(\mathcal{B}+1)+\lfloor\frac{\omega}{\delta}\rfloor)$

Generalization of this analysis to query networks with more complex aggregate arrangements is left out from this paper due to space considerations. The basic idea is similar.

5.2 Correctness Analysis

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

Theorem 1. *Windowed Drop inserted aggregation queries preserve correctness.*

Proof. The proof for this theorem has two parts, one for each aggregate arrangement. We 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 windowed drop inserted version of $\mathcal{P}$ is the set $\mathcal{A}'$. For $N=1$, $D(\omega,\delta)$ is inserted in front of A_1 such that $\omega=\omega_1$, $\delta=\delta_1$, and $\omega=m*\delta+\phi$. Every δ_1 time units, D marks a tuple t as either keep (τ, where $\tau=t.timestamp+\omega$), or drop (0). When A_1 receives t with specification of τ, it opens a new window at $t.timestamp$ and retains all tuples in time range $[t.timestamp,\tau)$. 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 windowed drop $D(\omega,\delta)$, with $\omega=\sum_{i=1}^{n}\omega_i-(n-1)$ and $\delta=\delta_n$, inserted in front of 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 timestamp T. Any aggregate downstream from A_1, that includes a tuple with timestamp T in its window effectively incorporates S values with timestamp up to $T+\omega_1-1$. Therefore, if D' is placed in front of A_1, its effective window size must include this range to preserve window integrity. As a result, D' 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 windowed drop must produce results either δ_{n+1} apart or in multiples of this quantity. Therefore, D' 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. We omit this second part due to lack of space. □

5.3 Experiments

In this section, we first verify that the Windowed Drop (**WinDrop**) approach achieves less QoS utility loss compared to the Random Drop (**RDrop**) alternative. This is a result that directly follows from the load analysis presented in Section 5.1. Secondly, we show how WinDrop's performance changes with changes in some of the query parameters. Finally, we compare our window-aware approach to an alternative window-ignorant approach (**TupleDrop**) using a well-known metric from information retrieval. We show that our approach always achieves perfect *precision*, while sacrificing from *recall*. However, on a measure that weighs precision and recall equally, WinDrop outperforms TupleDrop in most cases. Furthermore, our approach scales well as windows get larger, which the alternative approach fails to handle.

In what follows, we first briefly describe our experimental environment; then we present our results.

Experimental Setup. To experiment on our load shedding algorithms, we have built an Aurora simulator on CSIM [17]. This program uses CSIM facilities for process and event scheduling. The rest of the data structures and algorithms are implemented in almost the same way as in Aurora. CSIM enables us to practically play with performance factors like operator costs, selectivities and input stream rates for our experiments.

We generated an input stream of 10,000 tuples as $(timestamp,value)$ pairs, whose values were chosen from an integer range of $[0,100]$ in a random fashion. Timestamps are ordered integers, assigned at the source, with one tuple per timestamp. Various levels of CPU load are obtained by changing tuple inter-arrival times.

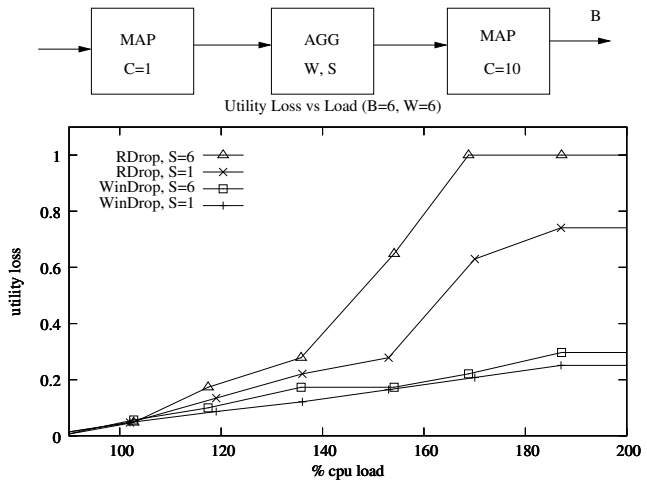


Figure 5. Window Slide

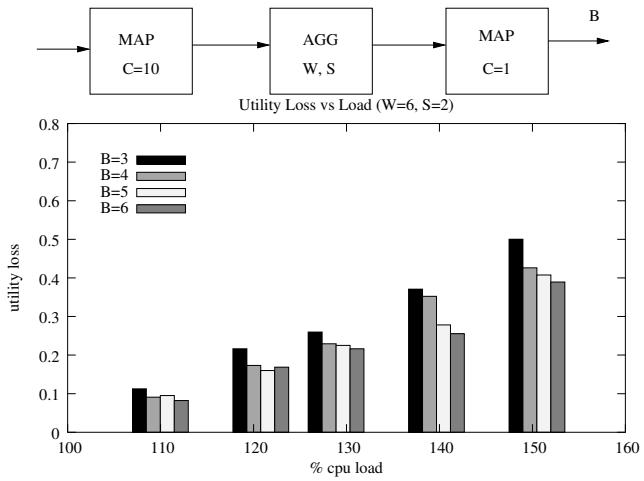


Figure 6. Batch Size

For each query output, we generated loss-tolerance QoS, as piece-wise linear functions with two pieces. This is achieved by randomly picking a utility value u in the range $[0.3, 1.0]$ and then a percent delivery value p in the range $[30, u * 100]$. Hence, (p, u) formed the turning point between two linear function pieces. Generating QoS this way also guaranteed a concave shape. This is required so that slope of the function for the percent delivery range $[p, 100]$ is not larger than that for $[0, p]$.

We used different query layouts for experimenting on different aspects of our algorithms. In addition to aggregates, we used map operator in these queries. We essentially used map to represent a subnetwork rather than a single operator. We assigned it different cost values to represent relative cost of that part of the query network. Map always has a selectivity of 1.0 as it produces one output tuple per input. Below we show our results together with these different query layouts.

Experimental Results. For our first experiment, we used a query layout as shown at top of Figure 5. An aggregate operator lies between a subnetwork of unit cost and another one which is 10 times more expensive. We fixed aggregate's window size as 6, and varied its slide. We used AVERAGE as the aggregate function. The output batch size $\mathcal{B}$ is set to 6, which enables early drops for any value of slide and provides an allowance to shed load down to about 15% of the output. We used a loss-tolerance QoS function whose turning point is $(40, 0.75)$, i.e., utility drops linearly from 1.0 to 0.75 when % tuple delivery drops from 100 to 40 (and then again drops more steeply from 0.75 to 0 when % tuple delivery drops from 40 to 0). Our goal is twofold: (1) to show how WinDrop and RDrop perform in terms of utility loss as input rates (and hence CPU load) increase, (2) to show how this performance is affected when aggregate's window slide changes.

When CPU load is over 100%, WinDrop inserts a Windowed Drop at the query input. Under the same condition, RDrop inserts a Random Drop between the aggregate and the costly map. We purposely used a costly subnetwork downstream from the aggregate in order to give the RDrop algorithm an advantage, because it can only save CPU cycles at this portion of the query. The graph in Figure 5 shows two curves for each algorithm, one for window slide of 1 and another for window slide of 6. WinDrop achieves obviously less utility loss compared to RDrop in both cases. It also degrades more gracefully than RDrop. RDrop essentially drops to zero tuple delivery at an overload level of about %70, whereas WinDrop drops to about 50% tuple delivery at the same overload level.

An interesting result is due to the effect of window slide. Given an aggregate with a fixed window size, if window slide increases, then both the per-tuple cost of the aggregate and its selectivity decrease. The reason for lower cost is that as slide increases, aggregate has less number of simultaneously open windows, hence `incr` of the aggregate function needs to be called fewer times upon new tuple arrival. The selectivity also decreases because it is proportional to $1/\delta$, as aggregate opens one new window every δ time units and produces one output for each such window. Increasing slide not only results in a query with reduced CPU requirements, but also the subnetwork downstream from the aggregate constitutes less fraction of the total query cost. Consequently, as slide increases, RDrop has to shed more load to be able to save the required CPU cycles. This is clear from our result in Figure 5, where with slide of 1, RDrop performs much better than the case with slide of 6. On the other hand, WinDrop performance is not affected from the change in window slide.

Our second experiment is to show the performance benefits achieved due to early drops enabled by dropping batches of consecutive windows. For this experiment, we used the query layout shown at top of Figure 6. It is the symmetric version of the earlier query layout. We purposely used a costly map upstream from the aggregate because that portion of the query would benefit if tuples were early-dropped by a Windowed Drop placed at the query input. We set aggregate's window size to 6 and window slide to 2, i.e., early drops are possible if $\mathcal{B} \geq 3$, as $WCF(t) = 3$ for any given t.

As we increase the batch size, Windowed Drop gets more opportunities to early-drop tuples. As a result, we expect it to

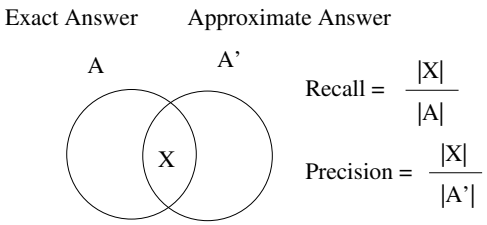


Figure 7. Recall and Precision

achieve less utility loss. The result presented in Figure 6 verifies this expectation. As $\mathcal{B}$ is increased from 3 to 6, excess load is shed with losing less QoS utility. The effect becomes more clear at higher overload levels.

As a last set of experiments, we compared WinDrop against a TupleDrop approach. This approach, similar in spirit to the approach in a recent related work [3], drops individual tuples rather than complete windows and performs this upstream from an aggregate. Therefore, the approximate result it produces does not guarantee a subset of the real answer. Rather, false positives are highly likely. Such approaches use relative error as their evaluation metric which is different from our evaluation metric based on the result set size. To be able to compare our approach to such an approach on a common ground, we used an evaluation metric often used in information retrieval, called *F-Measure* [4]. F-Measure is the harmonic mean of recall and precision. Translated into our context, recall corresponds to the fraction of the correct result tuples returned in reference to the exact query answer; precision is the fraction of the correct result tuples returned in reference to the approximate query answer. Figure 7 graphically illustrates what we mean by recall and precision. F-Measure equally weighs recall and precision as:

$$F = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}}$$

High F-Measure value indicates that both recall and precision are high.

For this experiment, we used the query layout as shown at top of Figure 8. We ran WinDrop and TupleDrop for increasing load levels and for different window sizes for the aggregate. We used the same slide as the window size (tumbling window). We show two graphs in Figure 8 for the cases $\omega = 2$ and $\omega = 4$. WinDrop always achieves full precision since there are no false positives. As load increases its recall drops. TupleDrop loses both from precision and recall as load increases. When window size is increased, TupleDrop is less likely to produce correct results. Because, to avoid an incorrect result, it must either keep all tuples in a window or drop all of them. Since drop decisions are made per tuple, the chances for this is highly unlikely as windows get larger. This can clearly be seen in our results. WinDrop delivers higher score results for most of the cases and its performance does not degrade with increasing window size.

6 Related Work

There has been a great deal of recent work in the area of data stream processing [14]. Several research prototypes have been

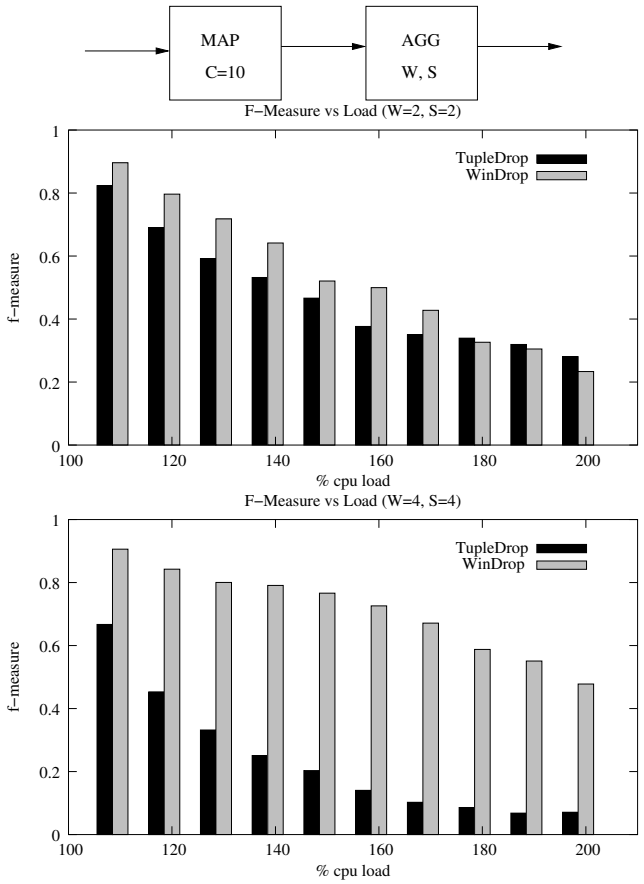


Figure 8. F-Measure

built, including Aurora [1], STREAM [18], and Telegraph [8]. Efficient resource management, adaptivity, and approximation have been the points of emphases in these works.

Load shedding for aggregation queries over data streams has been the subject of recent work by Babcock et al [3]. When time to process tuples that arrive in unit time exceeds 1, random drops are inserted into query trees so as to minimize maximum relative error at outputs. This is achieved using statistical bounds, based on mean and standard deviation statistics on sliding window of tuples received by aggregates. Our approach always produces subset results to minimize QoS utility loss, hence our approximation model is quite different. Moreover, since we preserve window integrity, our approach can handle a more general class of aggregation queries. The proposed work only applies to query trees, with a single aggregate operator at the leaf level. Also, the statistical bounds apply to only a limited set of aggregate functions whereas our approach is independent of the actual aggregate functions. Lastly, this approach heavily relies on window statistics. Sliding window statistics are subject to frequent change. This would directly affect the statistical bounds, often invalidating the load shedding policies due to these bounds. Therefore, approaches that rely heavily on window statistics might incur high overhead in adapting to changes.

Load shedding for sliding window joins in memory-limited

environments have also been studied [9, 16, 19]. Our approach mainly considers CPU as the limited resource although it can easily be extended to the memory-limited case as described in Section 4.3. Similar to our approximation model, the cited works 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 [21]. 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 stream tuples. Although both approaches annotate streams with additional 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 [11]. Former techniques mostly rely on precomputed data synopsis whereas latter approaches construct one-pass summaries as streams arrive. Online aggregation [15] lies somewhere between by interleaving sampling with query evaluation, on stored data. In addition to samples, synopsis can take the form of histograms [12], sketches [10], or wavelets [7, 13]. 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.

7 Conclusions

In this paper, we have shown how our previous load shedding work can be extended to deal with *sliding window aggregate* operators. Moreover, we have done this in a way that preserves the *no false positives* guarantee of our previous work. Our techniques also support load shedding in 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 our previous error model.

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

Next, we intend to extend this work into the distributed setting. Borealis, the successor to Aurora, attempts to map 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. Thus, we are beginning to investigate heuristic approaches to the *distributed load shedding* problem.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), 2003.

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

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

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[5] 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 Special Issue on Data Stream Processing*, 2004 (to appear).

[6] 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, September 2003.

[7] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing Using Wavelets. In *VLDB Conference*, 2000.

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

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

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

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

[12] 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.

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

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

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

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

[17] Mesquite Software, Inc. CSIM18 Simulation Engine. `http://www.mesquite.com/`.

[18] 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*, January 2003.

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

[20] 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.

[21] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3), May 2003.