Frames: Data-driven Windows

Michael Grossniklaus[1,2] David Maier[1] James Miller[1] Sharmadha Moorthy[3] Kristin Tufte[1]

[1] Computer Science Department, Portland State University, Portland, OR 92701, USA
{maier, jgm2, tufte}@cs.pdx.edu
[2] Department of Computer and Information Science, University of Konstanz, 78457 Konstanz, Germany
michael.grossniklaus@uni-konstanz.de
[3] Microsoft, One Microsoft Way Redmond, WA 98052, USA
sharmm@microsoft.com

ABSTRACT

Traditional Data Stream Management Systems (DSMS) segment data streams using windows that are defined either by a time interval or a number of tuples. Such windows are fixed—the definition unvarying over the course of a stream—and are defined based on external properties unrelated to the data content of the stream. However, streams and their content do vary over time—the rate of a data stream may vary or the data distribution of the content may vary. The mismatch between a fixed stream segmentation and a variable stream motivates the need for a more flexible, expressive and physically independent stream segmentation. We introduce a new stream segmentation technique, called *frames*. Frames segment streams based on data content. We present a theory and implementation of frames and show the utility of frames for a variety of applications.

CCS Concepts

•**Information systems** → **Stream management;**

Keywords

data streams, stream processing, stream segmentation

1. INTRODUCTION

Data stream management systems (DSMS) process potentially unbounded sequences of tuples or data items; due to their continuous nature, data streams are often segmented before they are processed. Traditionally, data streams are segmented using windows defined by a time interval or a number of tuples. Such windows are fixed in size and are defined on physical properties unrelated to the content of the stream. It seems intuitive that a fixed, unvarying, externally-based segmentation may not function well for a long-running query over a data stream that is subject to bursts and changes in data distribution.

DEBS '16, June 20 - 24, 2016, Irvine, CA, USA

ISBN 978-1-4503-4021-2/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2933267.2933304

Consider that rolling averages are commonly used to reduce noise and smooth signals from jittery sensors. A rolling average can be represented by a sliding window that averages the values in the window. With traditional windows, a fixed length must be selected for the window for the duration of the stream. For a bursty stream, one may want shorter windows during a burst to capture detailed characteristics of the burst, while longer (or no) windows may be desired during non-burst periods. Similarly with changes in data distribution, fixed-length window are often not ideal. Finally, in our experience, window-length selection is often done based on application domain knowledge, but with the goal of capturing a higher-level idea such as "smoothing out noise from a sensor." Again, variable window lengths that capture application semantics may be most appropriate.

In this paper, we propose frames, or data-dependent windows. In contrast with traditional windows, frames are more flexible, expressive and robust. In contrast with more sophisticated pattern matching systems, frames support a wide range of applications that do not require the full complexity of such systems. Frames segment data based on stream content and as such can adapt to changing stream characteristics and can capture higher-level concepts. Frames are expressive so that they can better capture user query needs and produce high accuracy with fewer results. In contrast with predicate windows, frames are simply subsequences of a stream (technically the starts and ends of those subsequences), while a predicate window [11] captures state (in other words, a view) that may be updated as the stream progresses.

The goal of our work is to improve the flexibility, expressiveness and robustness of data stream processing, while maintaining simplicity and performance. In general, we believe frames are useful for improving stream segmentation when the user does not have a good sense of specific window parameters for the segment and when segmentation needs to vary over time. In addition, in many cases, frames may better capture application semantics.

In the following section, we describe frames through examples. The rest of the paper describes theory, implementation, quality-improvement studies using new task-based metrics and performance feasibility studies.

2. FRAME TYPES AND USAGE

As a prelude to the more formal frame specification in the next section and to the description of frame implementation and performance experiments in later sections, we

present an example-based description of frames in this section. We argue that the four types of frames defined in this paper and presented below support a wide variety of application needs and—further—that frames support that variety of application needs with relatively low implementation complexity thereby avoiding the performance and code maintenance headaches that come with complex implementations. Frames are designed to be a compromise that is expressive enough for many applications while maintaining a desired level of implementation simplicity.

This paper defines four types of frames: *threshold frames*, *delta frames*, *boundary frames* and *aggregate frames*. We introduce each below.

2.1 Threshold Frames

Consider data on dye concentrations gathered by oceanographers interested in the movement and mixing of coastal waters.[1] In this research, oceanographers first discharge fluorescent dye at the ocean surface; then a research vessel towing a probe containing a fluorimeter cruises back and forth through the *dye field* recording dye concentrations at various depths and locations. Due to the collection process, the data contains long sequences of readings where dye levels are near zero, which are uninteresting to scientists. *Threshold frames* directly detect time periods (episodes) where a specified attribute (dye concentration in this example) is over (or under) a specified threshold. Threshold frames may include a minimum duration. The output of threshold frames is not a set of tuples, but rather the start time and end time of the episode (time period) wherein the tuples are above (below) the threshold.

We observe that this pattern of desiring to select "interesting regions" of data for additional processing is present in many applications. Network traffic analysts are interested in episodes of spikes in network traffic, transportation analysts are interested episodes where traffic speeds fall below a threshold, indicating traffic congestion.

Two advantages of threshold frames over traditional windows for this application are: 1) windows are emitted continuously, even during uninteresting periods, wasting resources; and 2) in order to capture episode boundaries accurately using windows, we may need to use a fine window granularity, resulting in more windows requiring more resources.

In contrast to sampling, frames produce data regions (start points and end points), while sampling produces a sub-sample of the data itself. Sampling the original data without a filter would include samples of the "uninteresting" regions, which have little value to the researchers. One could use frames to select "interesting" regions and then sample over those regions. Finally, when trying to detect a spike in activity (i.e. dye concentration, network traffic or vehicle traffic), frames can locate the starts and ends of the spike; in contrast, a periodic sample (i.e. every 100th reading) along with a notification when a high reading occurs can tell the user when a spike occurs, but will not identify the starts and ends of the "interesting" region. In other words, frames determine periods of interest while sampling finds points of interest.

2.2 Delta Frames

In studying the dye data, oceanographers also want to detect periods of rapid change in the data. *Delta frames*, in which a new frame starts whenever the value of a particular attribute changes by more than an amount x, can be used in this case. Delta frames adapt to the monitored attribute, with shorter frames during periods of rapid change allowing frames to capture the full range of values in the "spikes" in the data, whereas with fixed-size windows, many windows are "wasted" on relatively constant parts of the data.

Similar to threshold frames, delta frames produce the starts and ends of the regions of rapid change. Sampling does not produce those boundaries and, depending on the sampling method, may miss the region of rapid change—if, for example, the sampling interval is longer than the region of rapid change. We note that high-velocity streams may be particularly sensitive to the complexity of segmentation methods due to the need to process data at very high rates.

2.3 Boundary Frames

In the ACM DEBS 2013 GrandChallenge[2], one of the tasks is to build a series of heat maps showing how much time a soccer player has spent in different parts of the field. This task specifies a gridding of the field into cells to be used in the heat map. *Boundary frames* end a frame whenever an attribute value crosses on of a prescribed set of breakpoints and are useful for this example. A boundary-frame scheme on the x and y components of player position with the gridlines as breakpoints accounts for player time accurately.

2.4 Aggregate Frames

When monitoring vehicle traffic data, a common calculation is average vehicle traffic speed over the past five minutes. However, traffic analysts may want a shorter time window during high-traffic period, with finer-grained time windows to better capture condition changes, while the five-minute window may be shorter than ideal in overnight periods with low traffic. In contrast, an approach rooted in application needs would be to calculate the average speed for every ten (or X) vehicles. Such a segmentation would naturally use longer windows overnight when there is low traffic and fine-grained windows during congested time periods. As traffic data is most often reported as (speed, volume) pairs for a specified time interval, this calculation would require segmenting the stream based on a cumulative aggregate over traffic volume. *Aggregate frames*, which end a frame when an aggregate of the values of a specified attribute within the frame exceeds a threshold work for this case.

2.5 Discussion

From an end-user perspective, we believe that specifying the frame properties (i.e. average speed over ten vehicles, in this example) is often more intuitive than "guessing" a window length that will approximate a desired traffic property. Frames and their parameters are designed to match common application needs and as such frame parameters are designed to be intuitive to domain experts. Experiments also show that frame parameters are less sensitive than window parameters and thus need less tuning.

Frames support *intrinsic* segmentation of streams and take application characteristics into account. Thus, with frames, system resources and stream capacity can be better targeted at the key aspects of the stream and task. Traditional windows use *extrinsic* segmentation based on fixed time periods

[1] http://damp.coas.oregonstate.edu/latmix/

[2] http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails

or event counts, thus devoting equal resources to "more interesting" and "less interesting" portions of a stream. Furthermore, frames differ from windows in that frames are intervals that are specified by a start and an end, while windows are sets of tuples, sometimes seen as views. Note that frames are a generalization of windows in the sense that frames can be used to emulate windows.

However, not all applications need frames. Applications for which a traditional window specification using a time or tuple-based window size and slide is intuitive may not want to use frames. Additionally, applications that need a regular reporting schedule will not work well with frames. Other considerations are data skew and whether all periods of a stream are important. Finally, applications that look for patterns whose detection has arbitrary computational complexity or requires complex state to be kept cannot be done with frames. As stated above, frames are designed to be a compromise—expressive enough for many applications while maintaining a desired level of implementation simplicity.

3. FRAME SPECIFICATION

To give a formal specification of frames, we first define a framing of a data stream and then specify the functions that can be used to define a framing. We then introduce examples of commonly used framings and describe how they can be expressed based on the specification we give.

Definition 1. A *data stream* S is defined as an infinite sequence of tuples $S = [t_1, t_2, t_3, \ldots]$. All tuples of a data stream have the same schema. A distinguished *progressing* attribute A defines the logical order of the tuples: for any $n \in dom(A)$, there is an i s.t. $t_i.A > n$. While the progressing attribute implies a logical ordering, it does not require the stream tuples to be physically ordered.

3.1 Framing of a Data Stream

For this paper, we define the framing of a data stream incrementally by introducing the set of possible frames that is then restricted through local and global conditions to obtain the set of candidate and final frames, respectively.

Definition 2. The *possible frames* $F_p(S)$ of a data stream S are given by the infinite set of intervals $F_p(S) = \{[s_1, e_1], [s_2, e_2], [s_3, e_3], \ldots\}$, such that $\forall [s, e] \in F_p(S) : s, e \in dom(A)$, where A is the progressing attribute, $\forall i : s_i < e_i$.[3] For an interval $[s, e] \in F_p(S)$, we define an $extent([s, e])$ to be the set of tuples $\{t | t \in S \wedge s \leq t.A < e\}$. A *framing* of S is any subset of $F_p(S)$ including $F_p(S)$ itself.

We further constrain framings with *local* and *global conditions* that restrict the intervals in a framing. A local condition p_l is a (conjunction of) predicates that can be individually checked for each interval $[s, e] \in F_p(S)$. We distinguish data-dependent and data-independent predicates. For *data-dependent predicates*, expressions of the form $lhs\ \theta\ rhs$ can be used, where θ is a comparison operator, rhs is a constant, and lhs is a sub-expression built from arithmetic operators, aggregates, and universal quantification. The predicate $t.X > c$, for instance, restricts the framing to intervals where the value of attribute X of each tuple $t \in extent([s, e])$ is larger than a constant c.

[3]The names s_i and e_i stand for start and end point, respectively.

A *data-independent predicate* specifies the minimum (or maximum) duration of the interval. Duration can be expressed either in terms of the progressing attribute A or the number of tuples contained in the $extent([s, e])$. In the former case, the predicate is given by $e - s \geq n$ (or $e - s \leq n$), while in the latter case, it is given by $|extent([s, e])| \geq n$ (or $|extent([s, e])| \leq n$), where $|\cdot|$ denotes set cardinality.

Definition 3. The *candidate framing* $F_c(S)$ of a data stream S are those possible frames for which the local condition p_l is true, i.e., $F_c(S) = \{[s, e] | [s, e] \in F_p(S) \wedge p_l([s, e])\}$.

Example. Suppose we want to specify threshold frames as introduced in Section 2 to detect periods where the dye concentration (measured by its fluorescence $flSP$) is higher than 0.05 units for at least 10 measurements. In this case, the local condition p_l would be given by $t.flSP > 0.05 \wedge e - s \geq 10$, assuming that the domain of progressing attribute *scan* is the sequence number of a measurement. All intervals meeting this local condition would be in the candidate framing for this example.

To further restrict a candidate framing towards a final framing $F_f(S)$, global conditions p_g are applied to all intervals in the set of candidate frames $F_c(S)$, rather than to individual intervals. As a global condition, we can require that all final frames are either *minimal* or *maximal* among candidate frames. A final frame is minimal (maximal) if there is no candidate frame that is a proper sub-interval (super-interval). Another global condition that guides the selection of candidate frames is whether a set of intervals is *saturated* or *drained*. A set of frames is saturated if it satisfies all conditions and there is no candidate frame that can be added to it without violating a condition (set maximality). A final framing is drained if it satisfies all conditions and there is no frame that can be removed from it without violating a condition (set minimality).

Definition 4. A *final framing* $F_f(S)$ of a data stream S are the candidate frames for which all global conditions $p_g^1, \ldots, p_g^n$ are true. The set of local and global conditions that define a final framing is referred to as a *framing scheme*.

Example. Continuing the running example of specifying threshold frames over dye data, we use global conditions to require that all final frames are maximal and that the set of final frames is drained.

In general the final framing defined by a framing scheme is not unique. For example, requiring a final framing using delta frames to be maximal (or minimal) and drained is not sufficient to obtain a unique framing. However, if a framing scheme obeys certain additional characteristics, the resulting framing can be proven to be to unique.

Definition 5. A framing scheme has *union closure* if the union of overlapping candidate frames is also a candidate frame. A framing scheme has *super-interval closure*, if any possible frame that contains a candidate frame is also a candidate frame. Vice-versa, it has *sub-interval closure*, if any possible frame that is contained in a candidate frame is also a candidate frame. Finally, a set of intervals $[s_1, e_1], [s_2, e_2], \ldots, [s_n, e_n]$ covers an interval [a, b] if $extent([a, b]) \subset \bigcup_{i=1}^{n} extent([s_i, e_i])$. A framing $F(S)$ is said to cover a framing $G(S)$ if its set of intervals covers every interval in G.

Lemma 1. If a framing scheme has union closure and the set of final frames is maximal and covers the candidate frames, then the final framing is unique.

Proof (by contradiction). If the framing scheme is not unique then there exist two framings F and G and an interval $[a, b]$ such that $[a, b] \in F$ and $[a, b] \notin G$. Since G covers the candidate frames, there exists a subset of intervals E in G that covers $[a, b]$. Let $[s, e]$ be a frame in E. Since there is union coverage, $[s, e] \cup [a, b] = [c, d]$ is also a candidate frame and strictly larger than $[s, e]$ and $[a, b]$. Therefore, neither $[s, e]$ nor $[a, b]$ are maximal, contradicting the hypotheses. □

Proposition 1. A framing scheme that obeys Lemma 1 is also drained. Since it contains only maximal frames, no frame can be removed without violating coverage.

Proposition 2. In a framing scheme that has union closure, maximal frames do not overlap.

Lemma 2. Assuming there is a minimum value of the progressing attribute A, a final framing is unique if the framing scheme has super-interval (sub-interval) closure and the set of final frames is drained, minimal (maximal), and covers the candidate frames.

Proof. Given that the framing has super-interval (sub-interval) closure, a minimal (maximal) frame with starting point T_1 cannot properly contain a minimal (maximal) frame with starting point T_2, where $T_2 > T_1$. Since the framing has coverage of the candidate frames and is drained, it is therefore unique as the first candidate frame needs to be final, which then defines the starting point of the next frame, etc. □

3.2 Specific Framing Schemes

Based on this frame specification, we introduce specific framing schemes that we defined to address the requirements of the use cases outlined in Section 2.

Threshold Frames: This framing scheme reports periods of the stream where the value of a user-defined attribute a is greater (smaller) than a given threshold value x. Therefore, the local condition is given by the data-dependent predicate $a > x$ ($a < x$), whereas the global conditions require maximal frames that cover the candidate frames. Since this framing scheme has union closure, it is unique (Lemma 1), drained (Proposition 1) and non-overlapping (Proposition 2). In fact, this framing scheme is *separated* in the sense that there is a gap of at least one tuple between two consecutive frames.

Example. A concrete example of threshold frames is described in Section 3.1 to illustrate the definitions.

Boundary Frames: This framing scheme segments the stream whenever the value of a user-specified attribute a crosses a (multiple of) a given boundary x. Its local condition is therefore given by $\exists n : \forall t \in extent([s, e]) : (n-1)x < t.a \leq nx$, whereas the global conditions require maximal frames that cover the candidate frames. Since this framing scheme has union closure, it is unique (Lemma 1), drained (Proposition 1) and non-overlapping (Proposition 2). Since every tuple of the stream is contained in exactly one frame, this framing scheme is said to *partition* the stream.

Example. To use (two-dimensional) boundary frames to compute heat maps for soccer players, the following local condition can be used: $\exists n, m : \forall t \in extent([s, e]) : (n-1)x < t.x \leq nx \wedge (m-1)y < t.y \leq my$, where $0 < n \leq 16$, $0 < m \leq 25$, $x = 4.25$, and $y = 4.2$ for a 16x25 grid on a standard 68x105 metres (international) soccer pitch.

Delta Frames: In this framing scheme, a frame is emitted whenever the delta between the minimum and maximum value of a user-specified attribute a becomes greater (or smaller) than a predefined value x. The local condition is therefore given by $\exists t_1, t_2 \in extent([s, e]) : |t_1.a - t_2.a| \ \theta \ x$, where θ is a comparison operator. We distinguish two cases.

1. $\theta \in \{<, \leq\}$: Global conditions require maximal frames that cover the candidate frames.
2. $\theta \in \{>, \geq\}$: Global conditions require minimal frames that cover the candidate frames.

In both cases the final framing must be drained. Since the framing scheme in the first case has sub-interval closure and the framing scheme in the second case has super-interval closure, both of these framing schemes are unique, assuming there exists a minimal value for the progressing attribute (Lemma 2). Since both framing schemes are drained and maximal, they are non-overlapping and partition the stream.

Example. Delta frames can be used to bin the dye track data for a dye mass histogram by monitoring shifts in the water density (ρ) with the following local condition: $\exists t_1, t_2 \in extent([s, e]) : |\rho(t_1.t090C, t_1.sal00, t_1.prSM) - \rho(t_2.t090C, t_2.sal00, t_2.prSM)| > x$, where the density is derived from water temperature, salinity, and pressure. The value of x depends on the desired width of the density bins in the histogram.

Aggregate Frames: This framing scheme monitors a predicate over an aggregation of an attribute a and reports a new frame if the aggregate value becomes greater (or smaller) than a given constant x. The local condition is therefore given as $\forall t \in extent([s, e]) : f_\Sigma(t.a) \ \theta \ x$, where f_Σ is an aggregation function and θ is a comparison operator. In terms of global conditions, the same observations as for delta frames apply to aggregate frames.

Example. In order to use aggregate frames to monitor average vehicle traffic as described in Section 2, the following local can be used: $\Sigma_{t \in extent([s,e])} t.volume > 25$, which causes the stream to be segmented after each passing of 25 cars.

Figure 1 illustrates these four framing schemes by plotting candidate frames along the x-axis, which represents the logical arrival time of tuples. In Figure 1(a), all intervals in the set of possible frames that do not overlap a boundary are candidate frames for the boundary framing scheme. Figure 1(b) shows the case of a threshold framing, in which candidate frames are all intervals that fall in a contiguous region where the data value is above (or below) the given threshold. In both cases, a frame can start with any arriving tuple and can end with any tuple in the same contiguous region, bounded by the crossing of a boundary or a threshold value. In particular, each tuple within this region is a possible candidate frame on its own. Finally, Figures 1(c) and (d) illustrate delta and aggregate frames for predicates

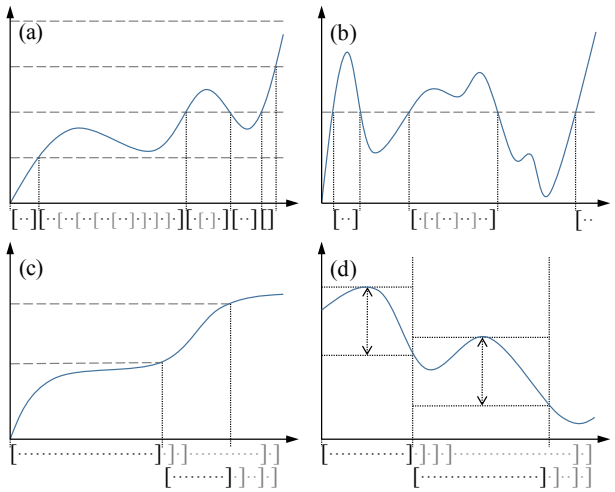

Figure 1: Boundary (a), threshold (b), aggregate (c), and delta (d) frames.

in which the delta or aggregate is specified as larger than a given value. A frame can start with any arriving tuple, but frames can only end at tuples that delimit a region for which the predicate is satisfied. Once that tuple is encountered, any super-interval is also a candidate frame.

The framing schemes presented in this section give examples of overlapping and non-overlapping schemes as well as schemes that cover, partition or separate the stream. In this context, we note that in our approach, these properties follow from the local and global conditions used in its definition. The fact that these properties are guaranteed at a high level is a major difference with existing solutions, where they have to be explicitly enforced in the program or query.

4. QUALITY-IMPROVEMENT STUDY

In this section, we present a quality-improvement study to analyze the task-based performance of frames. By *task-based performance*, we mean the quality of a data product judged by an application-specific metric. In Section 6, we report on a feasibility study that shows frames to have equal or better run-time performance compared to windows. Thus, we argue that frames improve task-specific performance without reducing run-time performance. The results support our claims that (a) frames have greater expressive power than windows, (b) frames can capture more complex criterion than windows, and (c) frames naturally adapt to changes in the data stream.

4.1 Experiment #1: Scatter Plots

The first experiment uses data from dye-tracking cruise W0908B, conducted by Oregon State University (OSU) in the Pacific Ocean off the coast near Newport, OR from Aug. 26 to Sept. 2, 2009 (*cf.* Section 2). Dye is measured at a sub-second frequency and used to understand how different bodies of water intermix. We report results obtained with data sets tow05 (325,000 tuples), tow08 (509,955 tuples), tow13 (407,992 tuples), and tow15 (375,000 tuples). The task of this experiment is to summarize the data set for visual representation as a scatter plot. The quality of this summarization corresponds to the visual similarity of

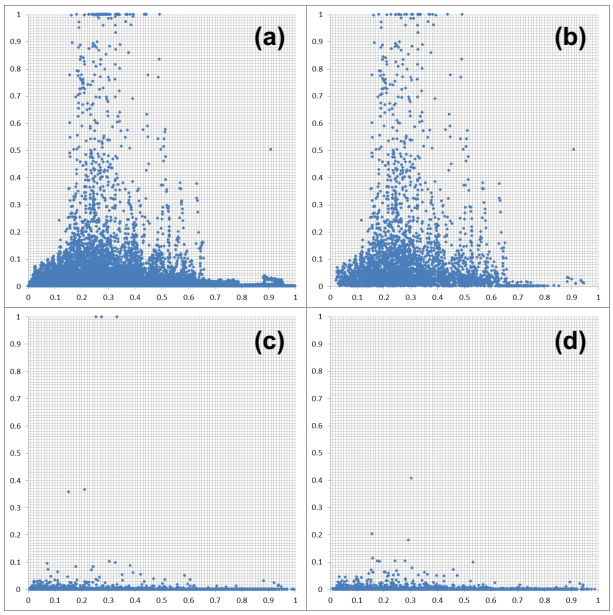

Figure 2: Florescence (y-axes) vs. Depth (x-axes) scatterplots, on normalized scales: (a) original data, ∼510K points; (b) ∼3600 frames; (c) regular sample of ∼3600 points; (d) ∼3600 equal-sized windows

the plot of the summarized data set to the plot of the original data set. Apart from visual similarity that we apply as a measure to assess task-based performance in this case, we additionally experimented with a second measure that is based on rendering the scatter plots as rasterized bitmaps. We normalize both axes of the scatter plot and then choose a grid resolution that yields approximately the same number of rendered points in both bitmaps. Finally, we compare these bitmaps by treating them as binary sets and applying Jaccard distance $1 - (A \cap B)/(A \cup B)$.

We use *delta frames* to segment the data stream. Depending on the range of measured fluorescence values, we use a delta value of 0.05 or 0.005 intensity units. For each frame, its average depth and its average fluorescence value are reported. Fixed-size windows are used to segment the stream into a similar number of intervals as the frame-based approach by setting the number of tuples contained in each window accordingly. Average depth and average fluorescence are reported for each window. Finally, the data stream is sampled at fixed intervals to generate a similar number of samplings as frames in the frame-based approach. Depth and fluorescence values are reported for each sample.

Figure 2 shows four scatter plots based on the tow08 data set, with Plot 2(a) showing the original data. Plot 2(b) shows average values for depth and florescence for roughly 3600 frames, which was created using a delta of 0.05 intensity units. Plot 2(c) shows 3600 uniformly-sampled points from the original data set. Plot 2(d) shows the average values over 3600 fixed-interval windows. Visually, the representation created using frames (b) is more similar to the full data set than the representation created using samples (c) or fixed-interval windows (d). About the only divergence between Plot 2(b) and the original is near the maximum depth, where the values are sparse.

Table 1: Deltas used in frame-based rendering and data set statistics after processing.

Data Set	Delta	Frames	Windows	Samples
tow05	0.005	8488	8335	8334
tow08	0.05	3614	3593	3592
tow13	0.05	4218	4208	4207
tow15	0.005	3045	3026	3025

Table 2: Jaccard distances for frame, window and sampling-based rendering of scatter plot bitmaps.

Data Set	Grid	Frames	Windows	Samples
tow05	25	0.073643	0.5	0.437984
	50	0.171501	0.591139	0.539241
	100	0.233249	0.684119	0.682747
tow08	25	0.024793	0.772727	0.789256
	50	0.035772	0.813008	0.819512
	100	0.107168	0.83032	0.845993
tow13	25	0.027491	0.636986	0.635739
	50	0.077361	0.754247	0.746303
	100	0.253538	0.833925	0.840855
tow15	25	0.057143	0.703571	0.689286
	50	0.144175	0.773148	0.791667
	100	0.418952	0.851351	0.835071

We also report results for Jaccard distance on rendered bitmaps of all four tow data sets. Table 1 describes the processing of these data sets based on the delta values provided by domain experts for each data set; Table 2 shows the Jaccard distances for frames, windows, and sampling. For each data set, we show three grid sizes: the one where the number of points rendered in both bit maps is approximately the same (50x50), one with half (25x25), and one with double (100x100) that resolution. The frame-based rendering of the bitmap consistently outperforms the window and sampling-based approaches in terms of Jaccard distance.

4.2 Experiment #2: Histograms

The second experiment aims to demonstrate that frames compute specific data products more precisely than windows. Again, the dye data set is used. The task is to approximate a dye mass versus depth histogram used by oceanographers at OSU, which we call the *Oceanographer's Histogram*. While the Oceanographer's Histogram yields an accurate representation of the dye distribution with respect to density bins, computing it is expensive as it involves a large number of depth slices that are created regardless of the dye concentration in a given water region. We compare the baseline Oceanographer's Histogram to window-based and frame-based approximations. To evaluate results, we use two measure. The first (qualitative) measure is again a visual comparison of the resulting histograms. As a second quantitative measure, we use *Earth-Mover Distance* (EMD). EMD measures how many data units have been assigned to a wrong bin in the histogram in terms of how "far" they would have to be moved to be in the correct bin.

The baseline Oceanographer's Histogram segments the dye data set into predefined depth slices of a fixed size. In order to approximate this baseline histogram using windows, we segment the dye data set based on the scan number attribute into windows of a fixed size. The frame-based approximation uses *delta frames* with a combination of two predicates. The first predicate is used to start a new frame whenever there is a shift in density (of the water), whereas the second predicate starts a new frame based on a shift in dye mass. We have fixed the ratio between the two deltas to 1:2. For each of the histograms, to reduce noise, the average dye mass of each segment is calculated and multiplied by the Δdepth of the segment. Finally, this value is summed to the corresponding density bin of the histogram. We expect frames to perform better than windows as they segment the stream based on physical properties such as water density (which relates to temperature and salinity) or the measured dye mass, rather than scan numbers.

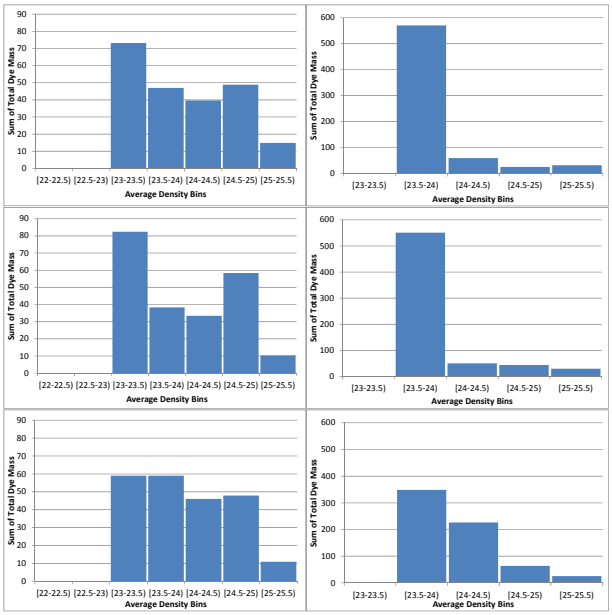

Figure 3: Baseline histogram, frame-based and window-based approximation for tow06 (left) and tow13 (right).

Figure 3 shows the baseline histogram (top) together with the frame-based (middle) and window-based (bottom) approximation. Results from tow06 are shown on the left, while results from tow13 are shown on the right. We have chosen these two tows as the relative task-based error of frames with respect to windows is worst on tow06 and best on tow13. The tow06 data set is segmented into 51,756 depth slices for the baseline histogram, capturing a total of 225.6 dye units. The frame-based approximation is computed based on just 897 frames and has earth-mover distance 19.3 with respect to the baseline histogram. The window-based approximation is computed using 897 windows but has EMD 23.7. The tow13 data set requires 38,172 depth slices for the baseline histogram capturing a total dye mass of 687.7 units. The frame-based approximation uses 4,217 frames and yields an error of EMD 60.5. Similarly, the window-based approximation needs 4,207 windows, but has EMD 294.6, i.e., almost 45% of the total dye mass. Finally, Figure 4 plots the relative error of windows over frames for all six tow data streams of the dye data set, calculated as $1 - EMD_{frame}/EMD_{window}$. As can be seen, frames consistently outperform windows in the task of approximating the

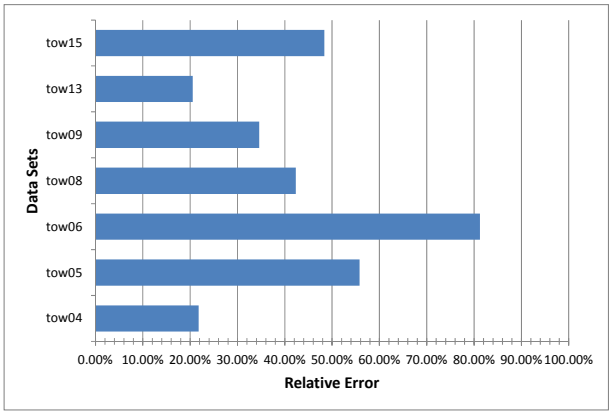

Figure 4: Relative error of windows over frames

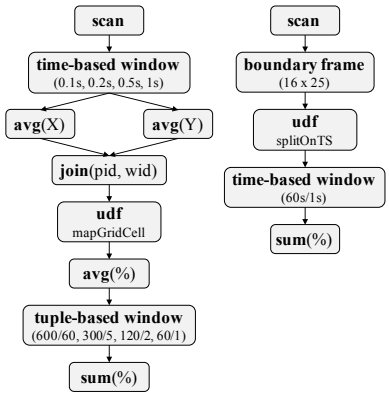

Figure 5: NiagaraST query plans for Query #3.

Oceanographer's Histogram based on this task-based measure.

4.3 Experiment #3: Heat Maps

For the last experiment, we use the sample data set from the DEBS 2013 Grand Challenge, which contains sensor data from a real-time locating system that is used to monitor a soccer game. The sensors report position, velocity, and acceleration. One sensor is in the ball and there are two sensors per player, one in each shoe. The goal keeper has two additional sensors in his gloves. The data set was recorded during a one-hour game of two halves, with two eight-player teams. The sensors in the players' shoes and gloves report with 200 Hz frequency, whereas the ball's sensor reports with a frequency of 2,000 Hz. The task is to calculate the heat map from Query #3. A heat map graphically visualizes how much time each player spent in which region of the pitch. For this purpose, the field is overlaid with a 16x25 grid (400 cells), which is one of the options required in the challenge. For each half, the heat map is reported after 1, 5, and 10 minutes as well as for the full half.

We compare the results of window-based queries with those of a query that uses *two-dimensional boundary frames* (or *grid frames*). The corresponding query plans are shown in Figure 5, with the window query on the left and the frame query on the right. These query plans are based on our implementation of frames in NiagaraST, which we will discuss in the next section. In the window query, the sensor data stream is first segmented based on the timestamp of the tuples. Then, the X and Y-average of each player's position during the window is computed and mapped to a grid cell. Note that the split and join are required because NiagaraST can only compute one aggregation value at a time. After computing the average percentage spent in each cell, the result tuples are mapped to tuple-based windows to conform to the reporting schedule required by the challenge. Finally, percentages are summed up and reported. As can be seen from the initial time-based window, we experimented with different window granularities, as finer-granular windows are able to improve the error in the final heat map. The frame query uses a boundary frame that corresponds to the layout of the grid. In order to conform to the required reporting schedule, the resulting frames have to be split based on the timestamps of the tuples they contain and summarized into time-based windows. In the last step, the total percentages are again summed up and reported.

In order to evaluate the quality of the computed heat map, we computed a ground truth offline by sorting all measured player positions and mapping them to the corresponding cells. Based on this ground truth, we compute the total root-mean-square error (RMS) for a pair of heat maps as the sum of all cell-wise errors. Figure 6 plots the relative error of the window-based and the frame-based heat maps, computed as $1 - RMS_{frame}/RMS_{window}$, for four different reporting schemes described in the challenge (1 and 5 minute windows that are updated every 1 or 10 seconds). As can be seen, the frame-based heat map consistently has a lower error than the window-based heat map. The relative error of windows increases for reporting schemes with a larger window size and/or slide. Better results can be achieved by dividing these large windows into smaller windows (10%, 20%, and 50% of the original window size) and recombining those to match the reporting scheme. However, as we will show in Section 6, this improvement comes at the price of increased run time.

4.4 Summary, Discussion and Critique

This section presented three use cases using real-life data sets. In each task examined, frames have consistently outperformed windows in terms of the quality of the computed data product. We conclude that frames do indeed improve the task-based performance of data-stream applications. Furthermore, we argue that these experiments demonstrate that frames are better suited to specify the complex queries required in these use cases as they are defined based on specifications that are given directly by application. This claim

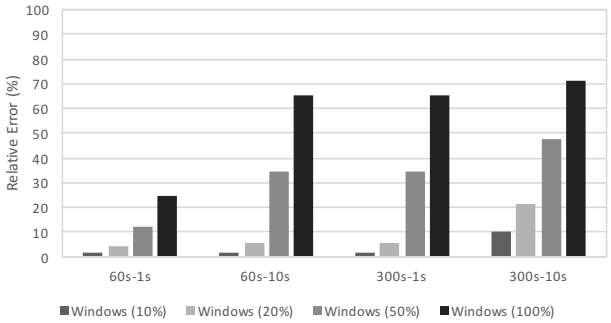

Figure 6: Relative error of windows over frames.

is corroborated by the heat map query where the window query plan had to be modified to use smaller windows than specified in the DEBS 2013 Grand Challenge in order to obtain a result quality that is comparable to the frames query.

There are caveats to our study. For example, there are other possible measures than the ones presented here that could be used to evaluate task-based performance of frames versus windows. In the case of heat maps, for example, a two-dimensional variant of earth-mover distance could be used. However, this metric is not easily computed. Furthermore, in the scatter-plot experiment, it would be possible to preprocess the data set to remove zero and near-zero values. Naturally, this would improve the task-based performance of windows and sampling. However, we argue that it is precisely the point of frames that such cumbersome preprocessing steps are not required.

Another possibility to obtain better task-based performance than windows would be to use a technique different from frames. The example given at the end of Section 3.2 for aggregate frames can also be expressed in a pattern-based approach. In the pattern matching syntax implemented by Oracle [26, 32], the query could be expressed as follows.

```
SELECT start, end FROM Traffic MATCH_RECOGNIZE (
         ORDER BY ts
         MEASURES MIN(A.ts) AS start, MAX(A.ts) AS end
         AFTER MATCH SKIP PAST LAST ROW
         MINIMAL MATCH
         PATTERN (A+) DEFINE A AS SUM(A.volume) >= 25)
```

Doing so will compute the same result as the proposed aggregate frame and therefore yield the same task-based performance. Nevertheless, there are some important points to note here. First, the semantics of pattern-based approaches is different from frames as they are not intended as a concept to just segment but rather to fully process data streams. Second, frames cover a large number of use cases as we demonstrate in this paper. We argue that the complexity of pattern-based approaches is not required for these use cases and instead may introduce adverse effects such as poor code maintainability. Finally, frames are supported by a concise formal foundation that could be applied to reason over frames or to optimize queries involving frames.

5. IMPLEMENTATION

In this section, we first present a generic (logical) frame operator to compute such framings. Based on this generic operator, we then discuss concrete (physical) frame operator implementations that correspond to the framing used in the motivating examples in Section 2. Our implementation computes unique framings (*cf.* Section 3) incrementally with bounded look-ahead and space requirements.

5.1 Logical Frame Operator

As our frame implementation computes framings incrementally, the frame operator template processes one tuple at a time. At any point, a frame operator maintains a, possibly empty, list of start and end points for open frames. For every arriving tuple, the operator must decide whether to (a) close an open frame, (b) update an existing frame, or (c) open a new frame. For (a), the frame is either reported or discarded. The latter can occur if a frame does not meet certain requirements as for example a minimum duration. The frame predicates (p_{close}, p_{update}, p_{open}) that trigger these actions depend on the framing scheme, which also determines if an action is applied once or multiple times.

Denoting the set of candidate frames as F_c, the set of final frames as F_f, and the state of the frame operator as C, the logical frame operator is defined as follows.

```
ProcessTuple()
1  if p_close(t, C)
2     then C ← Close(ts(t), C, F_c, F_f)
3  if p_update(t, C)
4     then C ← Update(ts(t), C, F_c)
5  if p_open(t, C)
6     then C ← Open(ts(t), C, F_c)
```

The operator begins by checking if the arrival of tuple t necessitates the closing of one or more frames (line 1). If so, it invokes the Close function and passes the timestamp of the current tuple $ts(t)$, the state of the frame operator (C), and the candidate and final frames sets (line 2). The Close function may remove closed frames from F_c. If it adds a corresponding frame to F_f, it is reported and discarded otherwise. The next check is whether any candidate frames need to be updated (line 3). To do so, the Update function is called with the timestamp of the current tuple, the operator state, and the set F_c (line 4). Typically, updating candidate frames corresponds to extending one or more frames to include the current tuple. The last check is if one or more new frames need to be opened (line 5). The opening of frames is handled by the Open function that is passed the timestamp of the current tuple, the frame operator state, and the set (line 6). A new frame is created by inserting the interval $[ts(t), ts(t)]$ into F_c. Finally, all three functions may update the state C of the frame operator.

5.2 Physical Frame Operators

We present different implementations of this logical frame operator for threshold, boundary, delta, and aggregate frames. Our implementation uses the NiagaraST [21–24] stream-processing system, which is written in Java and developed at Portland State University. NiagaraST is based on the Niagara [28] system from the University of Wisconsin-Madison.

All streams have a timestamp attribute, but tuples need not arrive in order; and the input stream may or may not have a known reporting schedule. Frame detection depends on how and when data arrives. If the data arrives on a schedule, the frame operator uses it to output frames as they are detected. If not, frames are processed over the input stream only when *punctuation* is received to deal with disordered data. On receipt of punctuation, the list of tuples is sorted on the timestamp attribute and frames are processed over tuples with a timestamp less than or equal to the punctuation timestamp. As described above, we maintain state (C), which consists of the start time ($C.ts_{start}$), end time ($C.ts_{end}$) and the size ($C.count$), if any, of the current frame. On receipt of each punctuation, a list of new frames is output. If the input punctuation timestamp is greater than the end time of the last frame, the punctuation is passed on.

To achieve a general and modular implementation, framing functionality uses three physical operators (*cf.* Figure 7). The first (existing) operator, Apply, processes the frame predicates (p_{close}, p_{update}, and p_{open}) for each tuple. More specifically, Apply applies the predicates to each input tuple

Table 3: Open, update, close functions for threshold, boundary, delta, and aggregate frame operators.

Frame Type	Threshold	Boundary	Delta	Aggregate
$p_{open}(t, C)$	$t.A > c \wedge C.count = 0$	$t.A > b_i \wedge C.b = b_{i-1}$	$C.ts_{start} = \perp$	$C.ts_{start} = \perp$
$p_{update}(t, C)$	$t.A > c \wedge C.count > 0$	$t.A > b_i \wedge t.A < b_{i+1}$	$\lvert C.v - t.A\rvert < \Delta$	$C.v < c$
$p_{close}(t, C)$	$t.A < c \wedge C.count > 0$	$t.A \geq b_{i+1}$	$\lvert C.v - t.A\rvert \geq \Delta$	$C.v \geq c$
$\textsc{Open}(ts, C, F_c)$	$F_c \leftarrow \{[ts, ts]\}$ $C.ts_{start} \leftarrow ts$ $C.count \leftarrow 1$	$F_c \leftarrow \{[ts, ts]\}$ $C.ts_{start} \leftarrow ts$ $C.b \leftarrow b_i$	$F_c \leftarrow \{[ts, ts]\}$ $C.ts_{start} \leftarrow ts$ $C.v \leftarrow t.A$	$F_c \leftarrow \{[ts, ts]\}$ $C.ts_{start} \leftarrow ts$ $C.v \leftarrow agg(t.A)$
$\textsc{Update}(ts, C, F_c)$	$F_c \leftarrow \{[C.ts_{start}, ts]\}$ $C.count \leftarrow C.count + 1$	$F_c \leftarrow \{[C.ts_{start}, ts]\}$	$F_c \leftarrow \{[C.ts_{start}, ts]\}$	$F_c \leftarrow \{[C.ts_{start}, ts]\}$ $C.v \leftarrow agg(C.v, t.A)$
$\textsc{Close}(ts, C, F_c, F_f)$	**if** $C.count > min$ **then** $F_f \leftarrow F_f \cup F_c$ $F_c \leftarrow \emptyset$ $C.count \leftarrow 0$ $C.ts_{start} \leftarrow \perp$	$F_f \leftarrow F_f \cup F_c$ $F_c \leftarrow \emptyset$ $C.ts_{start} \leftarrow \perp$	$F_f \leftarrow F_f \cup F_c$ $F_c \leftarrow \emptyset$ $C.ts_{start} \leftarrow \perp$	$F_f \leftarrow F_f \cup F_c$ $F_c \leftarrow \emptyset$ $C.ts_{start} \leftarrow \perp$ $C.v \leftarrow \perp$

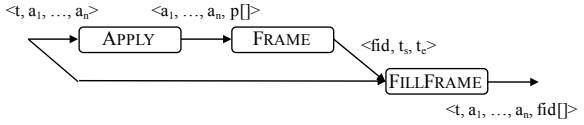

Figure 7: Physical frame operators.

and appends corresponding *true* or *false* values to it. These tuples then pass to the physical FRAME operator that interprets sequences of predicate results. If it detects a frame, it emits a tuple with frame id, start and end time. Finally, the FILLFRAME operator processes these metadata tuples and uses them to tag data tuples that fall between the start and end time with the corresponding frame id.

5.2.1 Threshold Frame Operator

We illustrate the case where the threshold operator frames periods in which the signal is above a threshold (c). In this setting, the p_{close} predicate is defined as shown in Table 3. If the value of the progressing attribute (A) of the current tuple (t) is below the threshold, a frame is reported if at least one immediately preceding tuple was above the threshold. As can be seen from the definition of the CLOSE function, a minimum duration (min) can optionally be set, which suppresses short frames. As there is ever at most one candidate frame, we report frames by unioning the set of final frames with the set of candidate frames. The UPDATE function is invoked if the predicate p_{update} is true, i.e., if the current tuple's value of the progressing attribute is above the threshold and it is not the first such tuple. Finally, $p_{open}(t, C)$ checks if the value of the progressing attribute is above the threshold and whether this is the first tuple (in a sequence) meeting this condition. If so, the OPEN function is invoked.

5.2.2 Boundary Frame Operator

The boundary frame operator is similar to the threshold frame operator in that it also monitors a particular attribute. However, boundary frames partition a stream rather than identifying periods of interest. Hence, the internal state and the three functions are implemented differently. The internal state tracks the last boundary that was crossed plus the value of the progressing attribute of the tuple that marks the beginning of a frame. The p_{open} predicate checks if the current tuple lies over the next boundary. If so, the OPEN function records the progressing attribute of the current tuple and the new boundary ($C.b$) in the the operator state. The UPDATE function simply extends the interval of the current frame. Finally, p_{close} checks whether the current tuple lies on over the next boundary. If so, the CLOSE function emits a frame for the currently recorded.

The definitions above assume that the monitored attribute is increasing monotonically. However, the definitions are easily extended to the general case. While we have presented our implementation of boundary frames in terms of boundaries for one attribute, our operator supports multi-dimensional boundary frames.

5.2.3 Delta Frame Operator

The delta frame operator maintains an internal state with the values of the delta attribute ($C.v$) and of the progressing attribute of the first tuple in a frame ($C.ts_{start}$). The p_{open} predicate checks if there is currently an open frame. If not, it initializes the internal state of the operator. The UPDATE function simply extends the interval of the current frame. The p_{close} predicate checks if the difference between the value of the delta attribute of the current and the first tuple exceeds the configured threshold. If so, the CLOSE function reports a frame for the currently recorded interval.

5.2.4 Aggregate Frame Operator

The aggregate frame operator is similar to the threshold frame operator. However, the internal state is slightly different. In contrast to the threshold frame operator, it maintains a running aggregate over all tuples of a frame ($C.v$). As shown in Table 3, the definition of the three predicates as well as the OPEN, UPDATE, and CLOSE function are analogous to those of the delta frame operator.

6. FEASIBILITY STUDY

The quality-improvement study in Section 4 shows that frames outperform window in terms of task-based performance. However, we have also observed that by using smaller windows, the error of the window-based approach can often be reduced. In order to understand the trade-offs involved, we present a feasibility study that compares the run-time performance of windows and frames. As the main benefit of frames lies in better task-based performance, we do not expect frames to outperform windows in terms of run-time performance. The research question we address in this study

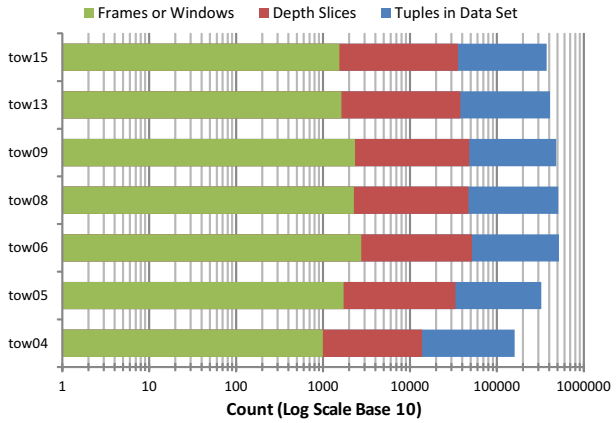

Figure 8: Data points in computation.

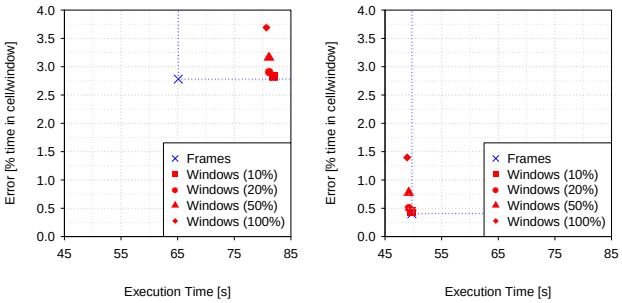

Figure 9: Run-time vs. task-based performance for 60/1s (left) and 300/10s (right) reporting scheme.

is therefore: *Is the performance of frames comparable to the performance of windows?* All experiments presented in this section were run in NiagaraST, using its existing window implementation and the frame implementation presented in the previous section. The figures reported for the first experiment were measured on a Dell OptiPlex 780 with a 3 GHz CPU and 4 GB of main memory, whereas the second experiment was conducted on an iMac with a 3.4 GHz Intel Core i7 CPU and 32 GB of main memory.

6.1 Experiment #1: Points in Computation

The first experiment seeks to quantify the reduction of computational overhead in window and frame queries. Both types of queries segment the infinite data stream and then process these segments. Apart from dealing with the infinite nature of data streams, this technique also serves to reduce the points in the computation by summarizing each stream segment to a single new data point with an aggregation function. As window or frame operators typically occur at the beginning of a query plan, the number of points directly affects the run-time performance of all succeeding operators and therefore *potentially* leads to better overall performance.

In order to quantify this potential run-time performance benefit, we revisit Experiment #3 in Section 4, which computed an approximation of the Oceanographer's Histogram using both windows and frames. In Figure 8, we compare the number of data points in the original data sets, to the number of depth slices, and to the number of frames or windows. Since we configured the window-based approach to produce (roughly) the same number of stream segments as the frame-based approach, both approaches reduce the number of points in the computation by the same factor. In comparison to the baseline approach that uses depth slices, this reduction is by an order of magnitude (and two orders relative to the original dataset).

This experiment demonstrates the reduction of computational overhead in window and frame queries. The next experiment addresses how much of this reduction potential can be realized by using either windows or frames.

6.2 Experiment #2: Run-Time Performance

To study this run-time performance of frames versus windows, we return to Query #3 of the DEBS 2013 Grand Challenge, which computes heat maps for soccer players. The query plans used to measure task-based performance are shown in Figure 5. In order to compare run-time performance fairly, we modified the windows query plan. Specifically, we removed the split and join operator introduced to aggregate a player's X and Y position in NiagaraST. This join is very costly and not necessary in other DSMSs.

Figure 9 shows the run-time and task-based performance results for the 60s/1s and 300s/10s reporting schemes. We selected these specific results because in the 60s/1s scheme the difference between frames and windows was the largest, whereas in the 300s/10s scenario it was smallest.

The overall error is generally smaller for reporting schemes that cover a longer duration of the game, as player position can be calculated more accurately. As with task-based performance, we measured the run-time performance of four variants of the window query. In the base variant, the original stream is split into tumbling windows with a length that corresponds to the slide of the reporting scheme, i.e., 1s and 10s. These tumbling windows are processed and combined into sliding windows before reporting results. In addition, we considered window queries that use initial tumbling windows of sizes 10%, 20%, or 50% of the the base variant. For the 60s/1s reporting scheme, the frame query consistently dominates all four window variants both with respect to run-time and task-based performance. Setting the reporting scheme to 300s/10s significantly reduces the number of windows required to process the query and the execution times of the four window queries decrease accordingly. The number of frames is unaffected by the change of the reporting scheme since they are created whenever a player crosses a boundary of the overlaid grid. Nevertheless, the execution time of frames also improves slightly, an effect which can be attributed to the window operator used at the end of the frames query to align its results with the reporting scheme. We note that the approach based on 10%-windows, which most closely matches frames in terms of error, still has almost the same execution time as the frame-based approach.

6.3 Pattern Matching Approaches

A limitation of our feasibility study is that the run-time performance of frames is not compared to that of pattern-based approaches, which share the same task-based performance. In order to fairly compare these two techniques a pattern-match operator would have to be implemented in NiagaraST, as very few systems that have such an operator are freely accessible, which is an expensive undertaking. We believe that a fully optimized pattern-match operator could

match the performance of frames, if it could avoid buffering tuples and compute pattern matching incrementally. However, this requirement is not met in many existing implementations. For example, Fischer *et al.* [10] indicate that their implementation of the `all` keyword is inefficient as it buffers all tuples and reevaluates predicates for newly-arriving tuples. In summary, it should be noted that the state managed by frames (*cf.* Table 3) is always bounded by the number of open frames, whereas in pattern-based approaches it can be bound by both the number of concurrently calculated matches and the number of tuples in every match.

6.4 Summary

This section first quantified the potential gain in run-time performance achieved by using frames for the dye data set, but without sacrificing task-based performance. A second experiment related run-time to task-based performance of frames and windows for a more complex query. While there are scenarios where frames are able to outperform windows on both dimensions, the opposite is not true.

In order to choose windows, one must select a window size and slide, which is often done based on experience or experimentation, not based on application criteria. Our experiments in this paper and in prior work [25] show that for many combinations of window size and slide, either the run-time or task-based performance is borderline unacceptable.

Finally, we note that there are significant opportunities to optimize our frames implementation. We assert that the small performance time penalty paid for the frame execution is well worth the improved accuracy of using frames.

7. RELATED WORK

There are two broad classes of related work: proposals to extend windows beyond traditional forms and techniques for optimal segmentation or summarization of data sets. We are not the first to propose enhancement of window functions, but we believe our theoretical formulation for frames can ultimately unify many of those approaches, and provide them with less-operational semantics. We examine some prior approaches, grouping them by the main mechanism used.

Predicate-based approaches: These approaches involve a predicate over tuples, and resemble frames in that the window extent can depend on stream attributes other than timestamp or sequence number. Predicate windows [11], define different types of stream subsets than frames. With predicate windows, a stream tuple is considered an update to the previous tuple with the same key. An tuple is in the window if it satisfies the predicate and is the most recent tuple for its key. Thus, the contents of a predicate window may be a non-contiguous subset of the stream, while frames are contiguous subsets. In contrast, a frame can span multiple tuples with the same key. IBM's InfoSphere Streams Punctor operator [16] can only look back a fixed distance in deciding where to end a window, hence is unable to express frames that involve an arbitrary look-back, such as delta frames. A proposal for window functions in XQuery relies in part on explicit `START` and `END` predicates [5]. This approach starts with the collection of all possible subsequences of a stream and restricts that collection; we feel the candidate-frame approach is a better basis for determining global properties.

More recently, Google's Dataflow Model [3] and Apache Flink [15] aimed at supporting broader classes of windows by supporting separate conditions for different aspects, such as the window range, when (and how often) a window result is produced, and how the window contents are adjusted. Such packages might provide alternative implementation mechanisms for frames. We also note that both approaches aim to support *session* windows, which could viewed as a variant of threshold frames where the constraint is a minimum duration *between* rather than *within* intervals.

Pattern-based approaches: Another group of window extensions involve pattern matching over stream elements. Matching a pattern is an alternative means for specifying candidate frames, hence it is compatible with our general framework. Simple pattern matching, such as unadorned regular expressions, cannot express frames requiring a delta between min and max values, or a time-based minimum duration. Complex pattern matching can express such frames; however, it is much more computationally expensive than frames. One of our goals was to maximize expressiveness while minimizing loss of performance.

Pattern-based approaches, such as SASE+ [2], Cayuga [6], and AFAs [7] go beyond finite-state automata and support manipulation of auxiliary state. Pattern-matching work not specific to streams includes SQL-TS, which supports searching for complex patterns in database systems [29] and S-OLAP [8], a flavor of OLAP system that supports grouping and aggregation of data based on patterns. Zemke *et al.* [32] and McReynolds [26] propose a `MATCH_RECOGNIZE` clause for SQL to support pattern matching. Golab *et al.* [12] propose SQL extensions for pattern detection in stream warehouses. Such proposals could be useful starting points for a query-language syntax for frames. Fischer *et al.* [10] propose a pattern matching facility for XQuery that supports predicates over the entire window using the `all` keyword.

Various proposals target composite events in complex-event processing (CEP) systems. Artikis *et al.* [4] present a model using fluents and Hirzel [14] also considers composite events as patterns over streams of simpler patterns, embodied in the MatchRegx operator of System S. Many proposals for composite-event detection seem more concerned with discrete events, whereas our focus has been numeric streams.

Contexts: Etzion and collaborators [1, 9] consider extending windows to more general "contexts" that can group events temporally, spatially, and based on system or event state. Whiteneck *et al.* [31] made initial forays into episodes with both temporal and spatial clustering. Semantic Windows [19] also consider defining segments of a multi-dimensional space based on content, though for static versus streaming data. In previous work [25], we demonstrated the use of frames to detect so-called episodes. A detailed performance study showed that there is in fact no "sweet spot" for windows, i.e., a configuration that outperforms frames both in terms of run-time and task-based performance.

Interval-based stream models: Stream systems that use an interval-based model for stream elements could incorporate frames. The *snapshot window* capability [27] of StreamInsight is a simple form of content-based segmentation. Our *FillFrame* operator is essentially the StreamInsight *Join* between an interval stream and a point stream.

Piecewise representation: Some frames types—delta frames in particular—resemble piecewise representation (PR) approaches. In PR, a sequence of values is broken into segments, and each segment is summarized by a simpler function, with linear functions being common (called piecewise linear approximation). PRs are used for purposes such

as preprocessing for data mining [20], approximate pattern matching [30], and detecting context changes [13]. In nearly all cases we are aware of, the "framing" and the "filling" attributes for PR are the same, whereas our approach accommodates "cross filling." In addition, PR techniques are typically off-line and on-line approaches seem to focus on one-pass algorithms on stored data sets.

Histograms: Histograms segment data sets and summarize each segment [17]. As with PR methods, most histogram construction is aimed at the stored-data case. There are different techniques to optimize against different error measures. However, the choice of error measure does not generally get connected to task-based performance. There is one interesting example of task-based evaluation that we encountered. Ioannidis and Poosla [18] use histograms for approximate query answering, and they offer a figure in the style of our Figure 2 that draws a similar conclusion.

8. CONCLUSION

We presented a new technique, called *frames*, that segments data streams based on content, providing a more flexible, expressive and physically independent stream segmentation than traditional approaches. We developed and described a formal specification of frames that we believe can unify existing stream segmentation approaches. Based on an implementation of our new technique, we provided two classes of evaluation—a quality-improvement study, based on task-based quality, and a feasibility study based on run-time performance. The combination of these studies demonstrates how frames improve task-specific performance without reducing run-time performance, our original goal.

Acknowledgements

This work is supported in part by National Science Foundation grant IIS-0917349. Michael Grossniklaus' participation is partially funded by the Swiss National Science Foundation (SNSF) grant number PA00P2_131452.

9. REFERENCES

[1] A. Adi and O. Etzion. Amit – The Situation Manager. *VLDB J.*, 13(2):177–203, 2004.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient Pattern Matching over Event Streams. In *SIGMOD*, pages 147–160, 2008.

[3] T. Akidau et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[4] A. Artikis, M. J. Sergot, and G. Paliouras. Run-time Composite Event Recognition. In *DEBS*, 2012.

[5] I. Botan, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending XQuery with Window Functions. In *VLDB*, pages 75–86, 2007.

[6] L. Brenna et al. Cayuga: A High-Performance Event Processing Engine. In *SIGMOD*, 2007.

[7] B. Chandramouli, J. Goldstein, and D. Maier. High-Performance Dynamic Pattern Matching over Disordered Streams. *PVLDB*, 3(1):220–231, 2010.

[8] C. K. Chui, B. Kao, E. Lo, and D. W. Cheung. S-OLAP: An OLAP System for Analyzing Sequence Data. In *SIGMOD*, pages 1131–1134, 2010.

[9] O. Etzion, Y. Magid, E. Rabinovich, I. Skarbovsky, and N. Zolotorevsky. Context-Based Event Processing Systems. In S. Helmer, A. Poulovassilis, and F. Xhafa, editors, *Reasoning in Event-Based Distributed Systems*, Studies in Computational Intelligence, pages 257–278. Springer, 2011.

[10] P. M. Fischer, A. Garg, and K. S. Esmaili. Extending XQuery with a Pattern Matching Facility. In *Proc. Intl. XML Database Symposium (XSym)*, 2010.

[11] T. M. Ghanem, A. K. Elmagarmid, P. Larson, and W. G. Aref. Supporting Views in Data Stream Management Systems. *ACM Trans. Database Syst.*, 35(1), 2010.

[12] L. Golab, T. Johnson, S. Sen, and J. Yates. A Sequence-Oriented Stream Warehouse Paradigm for Network Monitoring Applications. In *Proc. Intl. Conf. on Passive and Active Measurement (PAM)*, 2012.

[13] J. Himberg, K. Korpiaho, H. Mannila, J. Tikanmäki, and H. Toivonen. Time Series Segmentation for Context Recognition in Mobile Devices. In *ICDM*, pages 203–210, 2001.

[14] M. Hirzel. Partition and Compose: Parallel Complex Event Processing. In *DEBS*, pages 191–200, 2012.

[15] F. Hueske. Introducing Stream Windows in Apache Flink. https://flink.apache.org/news/2015/12/04/Introducing-windows.html, 2015.

[16] IBM Streams Processing Language Standard Toolkit Reference, IBM Infosphere Streams V 2.0.0.4, 2012.

[17] Y. E. Ioannidis. The History of Histograms (abridged). In *VLDB*, pages 19–30, 2003.

[18] Y. E. Ioannidis and V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *SIGMOD*, pages 233–244, 1995.

[19] A. Kalinin, U. Cetintemel, and S. Zdonik. Interactive Data Exploration Using Semantic Windows. In *SIGMOD*, pages 505–516, 2014.

[20] E. J. Keogh, S. Chu, D. M. Hart, and M. J. Pazzani. An Online Algorithm for Segmenting Time Series. In *ICDM*, pages 289–296, 2001.

[21] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *SIGMOD Record*, 34(1):39–44, 2005.

[22] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *SIGMOD*, pages 311–322, 2005.

[23] J. Li, K. Tufte, D. Maier, and V. Papadimos. AdaptWID: An Adaptive, Memory-Efficient Window Aggregation Implementation. *IEEE Internet Computing*, 12(6):22–29, 2008.

[24] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. *PVLDB*, 1(1):274–288, 2008.

[25] D. Maier, M. Grossniklaus, S. Moorthy, and K. Tufte. Capturing Episodes: May the Frame Be with You. In *DEBS*, pages 1–11, 2012.

[26] S. McReynolds. Complex Event Processing in the Real World. Oracle White Paper, Sept. 2007.

[27] MSDN Library. Snapshot Windows. In Developers Guide (StreamInsight): Writing Query Templates in LINQ.

[28] J. F. Naughton, D. J. DeWitt, D. Maier, et al. The Niagara Internet Query System. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[29] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Expressing and Optimizing Sequence Queries in Database Systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.

[30] H. Shatkay and S. B. Zdonik. Approximate Queries and Representations for Large Data Sequences. In *ICDE*, pages 536–545, 1996.

[31] J. Whiteneck, K. Tufte, A. Bhat, D. Maier, and R. J. Fernández-Moctezuma. Framing the Question: Detecting and Filling Spatial-temporal Windows. In *Proc. Intl. Workshop on GeoStreaming (IWGS)*, 2010.

[32] F. Zemke, A. Witkowski, M. Cherniak, and L. Colby. Pattern Matching in Sequences of Rows. Draft SQL Change Proposal, Mar. 2007.