Frames: Data-driven Windows for Task-Based Performance

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

[1] Computer Science Department
Portland State University
Portland, OR 92701, USA

{maier, tufte}@cs.pdx.edu

[2] Department of Computer and Information Science
University of Konstanz
78457 Konstanz, Germany

michael.grossniklaus@uni-konstanz.de

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 defined based on external properties unrelated to the data content of the stream. In contrast, streams themselves are highly variable. This mismatch motivates the need for more a flexible, expressive and physically independent stream segmentation. We present a new stream segmentation technique, called *frames*. Frames segment streams based on data content. We present a theory and implementation of frames and show that frames improve task-based performance for a variety of applications.

1. INTRODUCTION

Data streams, like their natural counterparts, are capricious; subject to bursts of data volume as well as sudden changes in data distribution. It seems intuitive that a fixed, unvarying, externally based segmentation would not fit the naturally variable nature of data streams. Traditionally, data streams have been segmented using windows defined either by a time interval or a number of tuples. Such traditional windows are fixed in size and are defined on physical properties unrelated to the content of the stream. In contrast, we propose frames, or data-dependent windows, which segment data streams based on stream content. In contrast with traditional windows, frames are more flexible, expressive and robust resulting in improved performance on task-based metrics.

In streaming applications, windows are used for purpose such as reducing stream volume for high-speed, high-volume streams and smoothing fluctuations in streams. For example, a one-minute rolling average may be used to reduce stream volume or smooth data from a jittery sensor. Windows can capture a one-minute rolling average using a one- minute sliding average. However, the selection of the one-minute length is not necessarily obvious.

We make two observations. First, different lengths may be appropriate for different stream conditions. Second, the selection of the one-minute length is often an attempt to capture a higher-level, more complex idea such as "smooth out noise from a sensor." In our experience, window lengths are often selected by users based on experience and intuition with the goal of capturing such higher-level ideas.

Consider Figure 1, which shows a set of scatter plots of a stream of data from an ocean sensor showing intensity of florescence of dye being tracked versus depth. The upper-left hand plot (a) is the original data with more than a half-million data points, the other three plots (b), (c), (d) are representations of that data using a small number of frames, samples and fixed-interval windows (3600 frames, 3600 samples and 3600 windows, respectively). 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).

This figure illustrates how frames can perform better on task-based performance metrics. Clearly, frames provide a better visual representation of the scatter plot in Figure 1(a).

In general, we believe frames are useful for improving stream segmentation in three cases:

- When the user does not have a good sense of specific parameters for the segment
- When the segmentation needs to vary over time.
- When windows are not expressive enough

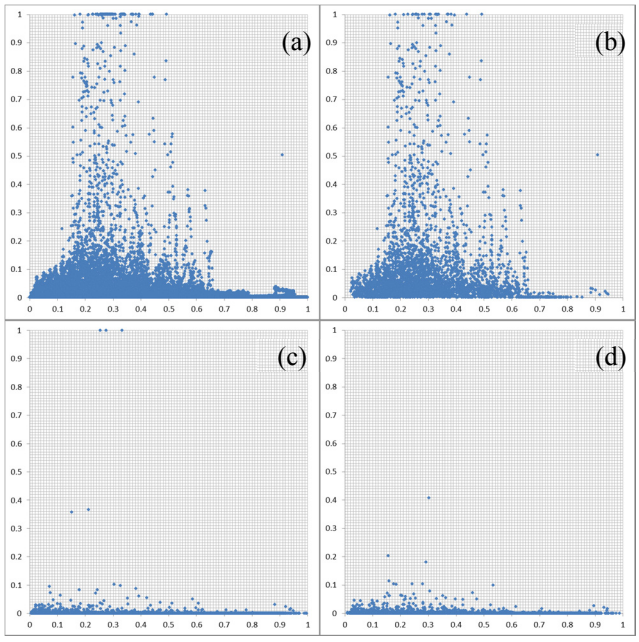

Figure 1. Florescence (y-axes) versus Depth (x-axes) scatterplots, on normalized scales: (a) original data, ~510K points; (b) ~3600 frames; (c) regular sample of ~3600 points; (d) ~3600 equal-sized windows.

We propose frames as an alternative to windows. Frames are defined on stream data and as such can adapt to changing stream characteristics and can capture higher-level concepts. Frames allow dynamic, adaptable stream segmentation so that framing schemes can adapt to changing data distributions as is inevitable in a long-running data stream query. Frames are expressive so that they can better capture user query needs and produce more accuracy with, in many cases, significantly fewer results.

The goal of our work is to improve the flexibility, expressiveness and robustness of data stream processing. The paper describes theory, implementation, quality-improvement studies using new task-based metrics and performance feasibility studies.

2. FRAME BENEFITS AND USE CASES

Frames support *intrinsic* segmentation of streams, which allows us to take application characteristics into account, both in terms of which information in a stream is most important and optimizing the results of application tasks. Thus, 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 or event counts, thus devoting equal resources to "more interesting" and "less interesting" portions of a stream, and only giving coarse control over task performance.

The dye data set provides several examples of how frames can target the most important aspects of a data stream. This data set was gathered by oceanographers interested in the movement and mixing of coastal waters.[1] A quantity of fluorescent dye is discharged at the ocean surface from a research vessel. The vessel then tracks the "dye field" by towing a probe rack equipped with a fluorimeter (as well as sensors for temperature, salinity, pressure, *etc.*). The vessel attempts to traverse the dye field, raising and lowering the probe to obtain readings at different depths.

Figure 3 in Section 4 shows a portion of fluorescence readings for one particular "tow". We see that the probe is in the dye field intermittently and that there are long sequences of uninteresting readings where dye levels are near 0. Consider the simple task of detecting episodes where significant dye is present. These periods can be used to find the corresponding segments in streams containing depth and lat-long position. That information can indicate the current extent of the dye field, in order to direct subsequent tows. *Threshold frames* are well matched to this task. A threshold frame is specified by a threshold value x for an attribute a (and an optional duration d). It returns a frame—consisting of start and end positions in the stream—for each stream segment where the value of a is above x (and which is longer than d). In Figure 3, threshold frames are specified on the fluorescence (flsp) attribute being greater than 0.2 and lasting at least 20 readings. Note the advantages here of threshold frames over traditional windows:

- Windows are emitted continuously, even during uninteresting periods, wasting resources and output capacity. Uninteresting windows could be filtered out later, but resources have already been expended to compute them.
- In order to capture episode boundaries accurately (so as to get a good estimate of the dye-field extent), we may need to use a fine window granularity, resulting in more windows requiring more resources.
- Depending on how we summarize fluorescence in each window (min, max, avg), we might overestimate or underestimate the total duration of episodes.
- Since an episode might be longer than a window range, there will be an additional step to stitch together sequences of windows to determine the episode period.

Other applications of the dye data can benefit from an alternative scheme, called *delta frames*, in which a new frame starts whenever the value of a particular attribute changes by more than an amount x. Delta frames adapt to the monitored attribute, with shorter frames during periods of rapid change. The scatterplot application in Figure 1 uses delta frames. They make sense for this task, as the scientists are interested in the extent of variation of dye intensity at different depths. For a comparable number of segments, delta frames do better, as they capture the full range of values in the "spikes" in the data, whereas many windows are "wasted" on relatively constant parts. Delta frames can track multiple attributes, ending a frame when any reaches its threshold. The histogram application in Section 4 uses this variation.

Sometimes, aspects of an application indicate a natural way to segment a stream. *Boundary frames* end a frame whenever an attribute value crosses on of a prescribed set of breakpoints. An example arises in the ACM DEBS 2013 Grand Challenge,[2] where one of the tasks is to build a series of heat maps showing how much time a given 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. Using a boundary-frame scheme on the x and y components of player position with the gridlines as breakpoints accounts for player time more accurately than with fixed-duration windows that can span multiple cells on the field.

In some applications, it may make sense to have frames track a cumulative property of the data. *Aggregate frames* support such cases, ending a frame when an aggregate of all values of an attribute within the frame exceeds a threshold. An example use for aggregate frames is in monitoring network traffic, where we might want to segment a stream based on total packet volume, rather than on a simple count of packets or fixed time period.

We remark here on another degree of flexibility with frames. A particular framing scheme dictates only the start and end of each frame. It does not require that the "filler" for frames come from the same stream, or that there be any filler at all. In the case of episode frames, knowing the time interval of each might be sufficient for some tasks. In other cases, we might fill frames from another stream. For the example of threshold frames above, we might fill each frame with the min and max latitude and longitude from the vessel's GPS stream.

3. FRAME SPECIFICATION

To give a formal specification of frames, we first define the framing of a data stream and then specify the functions that can be used to define a framing. We also describe how a framing can be used to fill frames with tuples from a stream. Finally, we introduce example 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 $S = [t_1, t_2, t_3, \dots]$. 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.

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

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

$t_i.A > n$. 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 the purpose of 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 $F_p(S) = \{[s_1, e_1], [s_2, e_2], [s_3, e_3], \dots\}$, such that $\forall [s,e] \in F_p(S) : s, e \in dom(A)$, where A is the progressing attribute, $\forall\, i\colon s_i < e_i$, and $\forall\, i < j\colon s_i = s_j \Rightarrow e_i \neq e_j$.[3] For an interval $[s,e] \in F_p(S)$, we define an $extent([s,e])$ to be the set of tuples $\{t | t \in S \wedge s \leq t.A < e\}$, where A is the progressing attribute. A *framing* is any subset of $F_p(S)$ including $F_p(S)$ itself.

In practice, however, it is useful to further constrain framings. We distinguish *local* and *global conditions* that can be applied to restrict which intervals are contained 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.

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 are interested to detect periods in a stream of traffic data, where the speed is consistently below 40 mph for at least five minutes. In this case, the local condition p_l would be given by $t.speed < 40 \wedge e - s \geq 300$, assuming that the domain of progressing attribute A is time in seconds.

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 an individual interval. As a global condition, we can require that all final frames are either *minimal* or *maximal* among candidate frames. A final frame is minimal (or maximal) if there is no candidate frame that is a sub-interval (or 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).

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

Definition 4: A *final framing* $F_f(S)$ of a data stream S are the candidate frames for which all global conditions $p_g^1, \dots, p_g^n$ are true, i.e., $F_f(S) = \{[s,e] | [s,e] \in F_c(S) \Leftrightarrow \bigwedge_{1 \leq i \leq n} p_g([s,e])\}$. The set of local and global conditions that define a final framing is referred to as a *framing scheme*.

Notice that 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 fully contains a candidate frame is also a candidate frame. Vice-versa, it has *sub-interval closure*, if any possible frame that is fully contained in a candidate frame is also a candidate frame. Finally, a set of intervals $[s_1, e_1], [s_2, e_2], \dots, [s_n, e_n]$ *covers* an interval $[a,b]$ if $t \in extent([a,b]) \Rightarrow t \in \bigcup_{i=1}^{b} extent([s_i, e_i])$. 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 sequence of intervals $[s_i, e_i][s_{i+1}, e_{i+1}]$ in G that covers $[a,b]$, such that $s_i \leq a \leq e_i \wedge s_{i+1} \leq b \leq e_{i+1}$. Since there is union coverage $[s_i, e_i] \cup [a,b] = [s_i, b]$ is also a candidate frame. Therefore, neither $[s_i, e_i]$ nor $[a,b]$ are maximal, which is in contradiction to the lemma. ■

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 closure (or sub-interval closure), a minimal (or maximal) frame with starting point T_1 cannot properly contain a minimal (or 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 possible frame needs to be picked 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 1.

Threshold Frames: This framing scheme reports periods of the stream where the value of a user-defined attribute a is greater (or smaller) than a given threshold value x. Therefore, the local condition is given by the data-dependent predicate $a > x$ (or $a < x$), whereas the global conditions require maximal frames that cover

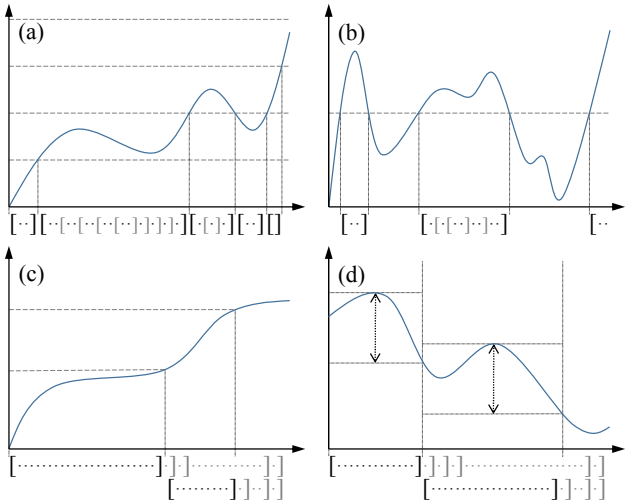

Figure 2. Boundary (a), threshold (b), aggregate (c), and delta (d) frames.

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 subsequent frames.

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.

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

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

In both cases the final framing is required to be drained. Since the framing scheme in case (a) has sub-interval closure and the framing scheme in case (b) has super-interval closure, both of these framing scheme 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.

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. 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.

Figure 2 illustrates these four framing schemes by plotting candidate frames along the x-axis, which represents the logical arrival time of tuples. In Figure 2(a), all intervals in the set of possible frames that do not overlap a boundary are candidate frames for the boundary framing scheme. Figure 2(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 that is 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, Figure 2(c) and (d) summarize the case of delta and aggregate frames for predicates in which the delta or aggregate is specified to be 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, as shown in the figure.

While the four framing schemes defined in this section are non-overlapping, that property is not a requirement of our frame specification. To illustrate this point, we define the hypothetical framing scheme of *double-boundary frames* that can cross exactly one boundary. The local condition of this framing scheme is given as $\exists n: \forall\, t \in extent([s,e]): (n-1)x < t.a \leq (n+1)x$, while the global conditions require this framing to cover the candidate frames and to be maximal and saturated. Although this framing scheme is unique, every $t \in S$ is in two overlapping frames.

Summarizing the framing schemes presented in this section, we have seen examples of overlapping and non-overlapping framing schemes as well as framing schemes that cover, partition or separate the stream. In this context, we point out that in our approach to stream segmentation, these properties of a framing follow from the local and global conditions used in its definition. Our approach of guaranteeing properties at a high level is a major difference to existing solutions, where these properties have to be manually enforced in the program or query.

4. QUALITY-IMPROVEMENT STUDY

This section presents the first of two studies designed to compare frames and windows. We undertake a quality-improvement study to see if frames have improved task-based performance versus windows on a variety of tasks and over several data sets. 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 research question of our quality improvement study is: *Do frames improve task-based performance?* This study will 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. In the following, we compare the quality of frame-based data products to window-based and manually created alternatives.

4.1 Experiment #1: Episode Detection

The first experiment uses data from dye track cruise W0908B, conducted by Oregon State University (OSU) in the Pacific Ocean off the coast near Newport, OR from August 26 to September 2, 2009 (*cf.* Section 2). Dye is measured at a sub-second frequency and used to understand how different bodies of water intermix. Therefore, a common task is to identify and select "regions of interest", so-called episodes, in the data stream. Figure 3 shows

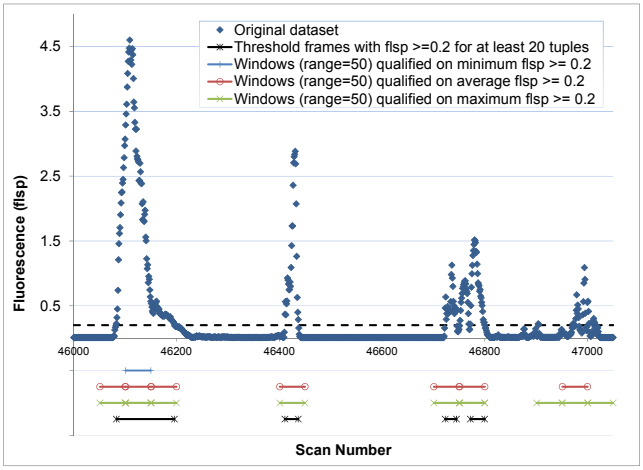

Figure 3. Comparison of the three heuristics versus frames.

"dye intensity" data from a scientific cruise. This data has intermittent sharp peaks, representing locations of high dye density, in between long periods of stability. The "stable" data indicates no dye and is of little interest to scientists. This pattern of intermittent (interesting) peaks in otherwise stable (uninteresting) data occurs in other data sources such as network and vehicle traffic.

As discussed, using windows to detect the relatively sparse episodes is intuitively problematic due to fixed window sizes. Small windows will effectively detect episodes, but will produce a large number of result windows, most containing only the information that the data is in a stable state. If window size is increased, the number of windows produced will reduce; however, episodes may be lost (undetected) due to averaging effects. In this setting, three types of errors can occur. A *duration error* occurs if window boundaries do not align with episode boundaries. If episodes are mistakenly detected or missed due to the fixed window size, an *existence error* in terms of false positives and negatives can occur.

In the first experiment, the task is to detect ranges of data where the fluorescence values exceed a threshold of 0.2 for at least 20 consecutive measurements. We compare *threshold frames* to three window variants. For all window variants a size of 50 tuples is used. The first variant selects a window if the minimum fluorescence value is larger or equal to 0.2. In the second variant the average fluorescence value is used to qualify windows, whereas the third variant uses the maximum value. Consecutive qualified windows are merged to form the detected episodes.

The bottom of Figure 3 shows the results of the experiment. The episodes detected by the threshold frames are exact results, since they directly applied the episode definition. Comparing windows to frames, it can be seen that all window variants suffer from duration errors. Additionally, existence errors can be observed. The window variant using the minimum fluorescence value suffers from false negatives for all but the first episode. The other two window variants report a false positive for the last episode. A full discussion of these results can be found in Maier *et al.* [30].

4.2 Experiment #2: Scatter Plots

The second experiment uses the same dye data set as before. In particular, 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 summarize the data set for visual representation as a scatter plot. The quality of this summarization corresponds to the visual similarity of the plot of the summarized data set to the plot of the original data set. The first measure we apply to assess task-based performance in this case is visual similarity. Additionally, we 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 1 in Section 1 shows four scatter plots based on the tow08 data set, with Plot 1(a) showing the original data. Plot 1(b) shows average values for depth and florescence for roughly 3600 *frames*, which was created using a delta of 0.05 intensity units. Plot 1(c) shows 3600 uniformly-sampled points from the original data set. Plot 1(d) shows the average values over 3600 fixed-interval windows. The scatter plot of the frame representation is clearly the most visually similar to the plot of the original data. About the only divergence between Plot 1(b) and the original is near the maximum depth, where the values are sparse.

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

Data Set	Grid Size	Frames	Windows	Sampling
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

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

We also report results based on applying Jaccard distance to rendered bitmaps of all four tow data sets. Table 1 gives information about the processing of these data sets, whereas Table 2 shows the measure 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 approach in terms of Jaccard distance.

4.3 Experiment #3: Histograms

The third 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 vs. 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 a large number of depth slices have to be processed that are created regardless of the dye concentration in a given water region. We compare the baseline Oceanographer's Histogram to a window-based and a frame-based approximation. 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 the so-called *earth-mover distance* (EMD). The earth-mover distance 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 whenever there is a shift in density (of the water), whereas the second predicate starts a new frame based on a shift in dye

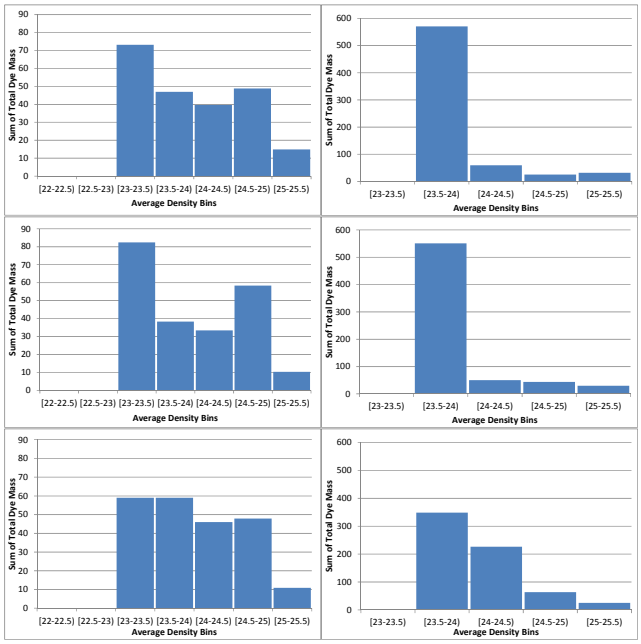

Figure 4. Baseline histogram, frame-based and window-based approximation for tow06 (left) and tow13 (right).

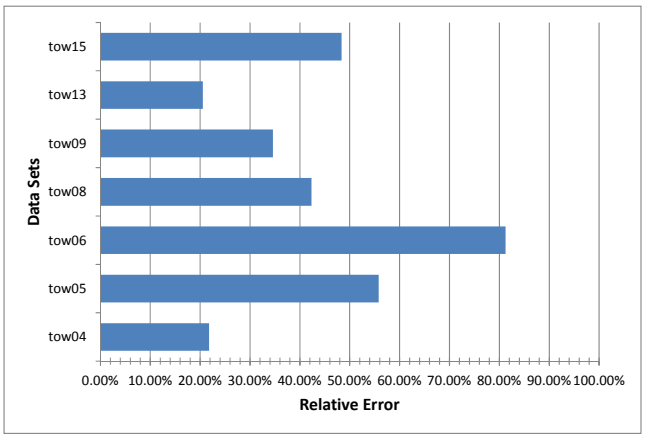

Figure 5. Relative error of windows over frames.

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.

In Figure 4 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 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 earth-mover distance 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 earth-mover distance 60.5. Similarly, the window-based approximation needs 4,207 windows, but has earth-mover distance 294.6, i.e., almost 45% of the total dye mass. Finally, Figure 5 plots the relative error of windows over frames for all six tow data streams of the dye data set. The relative error is calculated as $1 - EMD_{frame}/EMD_{window}$. As can be seen, frames consistently outperform windows in the task of approximating the Oceanographer's Histogram based on this task-based measure.

4.4 Experiment #4: Heat Maps

For the last experiment, we use the sample data set provided by 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 in Nuremberg, Germany during a one-hour game (two 30-minutes halftime) 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 of the game each player spent in which region of the pitch. For this

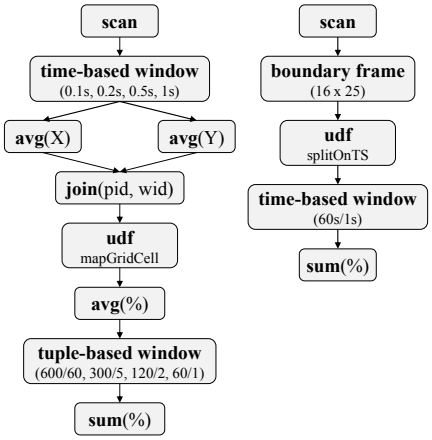

Figure 6. NiagaraST query plans for Query #3.

purpose, the field is overlaid with a 16x25 grid (400 cells), which is one of the options required in the challenge. For each halftime, the heat map is reported after 1, 5, and 10 minutes as well as after the full halftime.

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 6, with the window query on the left and the frame query on the right. 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 windows granularity. The reason for this decision is that 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.

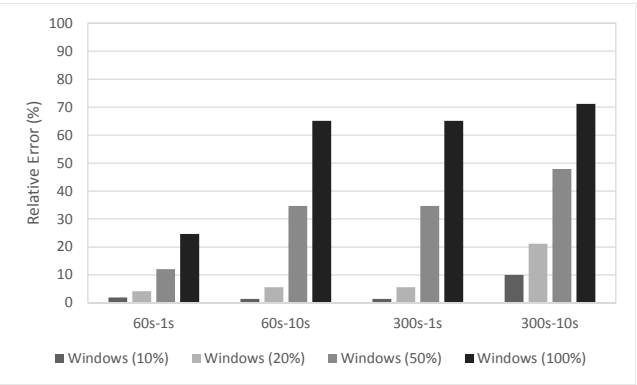

Figure 7. Relative error of windows over frames.

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 7 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. However, scheme 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. However, as we will show in Section 6, this improvement comes at the price of reduced run-time performance.

4.5 Summary, Discussion and Critique

In this section, we have presented four use cases based on real-life data sets. In each task that we examined, frames have consistently outperformed windows in terms of the quality of the computed data product. With respect to the research question of this quality improvement study, we therefore conclude that frames do indeed improve the task-based performance of data stream applications. Moreover, 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 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 one of the frames query.

Nevertheless, there are possible threats to the validity of our study. For example, there are other possible measure than the ones presented here that could be used to evaluate task-based performance of frames versus windows. Choosing a different measure might invalidate some of the results that we have obtained. 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 (expensive and cumbersome) preprocessing steps are not required.

5. IMPLEMENTATION

Our implementation of frames supports framings that are unique (*cf.* Section 3) and that can be computed incrementally with bounded look-ahead and space requirements. 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 1.

5.1 LOGICAL FRAME OPERATOR

As our frame implementation requires that framings be computed 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

Frame Type	Threshold	Boundary	Delta	Aggregate
$p_{open}(t, C)$	$t.A > c \wedge C.count = 0$	$t.A > b_i \wedge C.b = b_{i-1}$	$C.ts_{start} = \perp$	$C.ts_{start} = \perp$
$p_{update}(t, C)$	$t.A > c \wedge C.count > 0$	$t.A > b_i \wedge t.A < b_{i+1}$	$\|C.v - t.A\| < \Delta$	$C.v < c$
$p_{close}(t, C)$	$t.A < c \wedge C.count > 0$	$t.A \geq b_{i+1}$	$\|C.v - t.A\| \geq \Delta$	$C.v \geq c$
OPEN(ts, C, F_c)	$F_c \leftarrow \{[ts, ts]\}$ $C.ts_{start} \leftarrow ts$ $C.count \leftarrow 1$	$F_c \leftarrow \{[ts, ts]\}$ $C.ts_{start} \leftarrow ts$ $C.b \leftarrow b_i$	$F_c \leftarrow \{[ts, ts]\}$ $C.ts_{start} \leftarrow ts$ $C.v \leftarrow t.A$	$F_c \leftarrow \{[ts, ts]\}$ $C.ts_{start} \leftarrow ts$ $C.v \leftarrow \text{agg}(t.A)$
UPDATE(ts, C, F_c)	$F_c \leftarrow \{[C.ts_{start}, ts]\}$ $C.count \leftarrow C.count + 1$	$F_c \leftarrow \{[C.ts_{start}, ts]\}$	$F_c \leftarrow \{[C.ts_{start}, ts]\}$	$F_c \leftarrow \{[C.ts_{start}, ts]\}$ $C.v \leftarrow \text{agg}(C.v, t.A)$
CLOSE(ts, C, F_c, F_f)	**IF** $C.count > min$ **THEN** $F_f \leftarrow F_f \cup F_c$ **END** $F_c \leftarrow \emptyset$ $C.count \leftarrow 0$ $C.ts_{start} \leftarrow \perp$	$F_f \leftarrow F_f \cup F_c$ $F_c \leftarrow \emptyset$ $C.ts_{start} \leftarrow \perp$	$F_f \leftarrow F_f \cup F_c$ $F_c \leftarrow \emptyset$ $C.ts_{start} \leftarrow \perp$	$F_f \leftarrow F_f \cup F_c$ $F_c \leftarrow \emptyset$ $C.ts_{start} \leftarrow \perp$ $C.v \leftarrow \perp$

Table 3: OPEN, UPDATE, CLOSE functions for threshold, boundary, delta, and aggregate frame operators.

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(t)
1: **IF** $p_{close}(t, C)$ **THEN**
2: $C \leftarrow$ **CLOSE**$\big(ts(t), C, F_c, F_f\big)$
3: **END**
4: **IF** $p_{update}(t, C)$ **THEN**
5: $C \leftarrow$ **UPDATE**$(ts(t), C, F_c)$
6: **END**
7: **IF** $p_{open}(t, C)$ **THEN**
8: $C \leftarrow$ **OPEN**$(ts(t), C, F_c)$
9: **END**

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 4). To do so, the UPDATE function is called with the timestamp of the current tuple, the operator state, and the set F_c (line 5). 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 7). 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 F_c (line 8). 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 now present different implementations of this logical frame operator for threshold, boundary, delta, and aggregate frames. Our implementation uses the NiagaraST [26-29] stream-processing system. NiagaraST is written in Java and developed at Portland State University. NiagaraST is based on the Niagara [33] system from the University of Wisconsin-Madison.

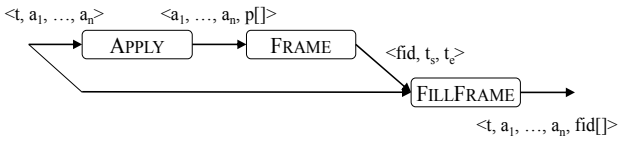

Figure 8. Physical frame operators.

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. At 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. At 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 operators at the physical level (*cf.* Figure 8). 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 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 consisting of a 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 frame operator frames periods in which the signal is *above* a certain 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 immediate-

ly 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 that consists of the value of the delta attribute ($C.v$) and the value 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 has shown 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 limited. In order to understand the trade-offs involved, we present a feasibility study that compares the run-time 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 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 two experiments were measured on a Dell OptiPlex 780 with a 3 GHz CPU and 4 GB of main memory, whereas the third experiment was conducted on an iMac with a 3.4 GHz Intel Core i7 CPU and 32 GB of main memory.

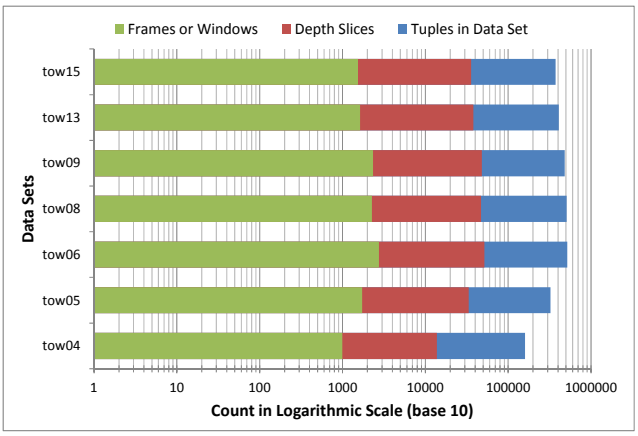

Figure 9. Data points in computation.

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 often 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 9, we compare the number of data points in the original data sets, to the number of depth slides, 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. With respect to the original data set, the reduction of points in the computation is even by two orders of magnitude for all tow data streams.

This experiment demonstrates the reduction of computational overhead in window and frame queries. Thus, it also quantifies the potential run-time improvement that can theoretically be achieved. However, this experiment does not provide any information about how much of this potential can be realize by using either windows or frames. The next two experiments address this question.

6.2 Experiment #2: Run-Time Performance

We conduct the first run-time performance experiment in the setting of Experiment #1 in Section 4, where we detected episodes in a dye data set. Since we observed in our quality improvement study that using smaller windows can contribute to limiting the

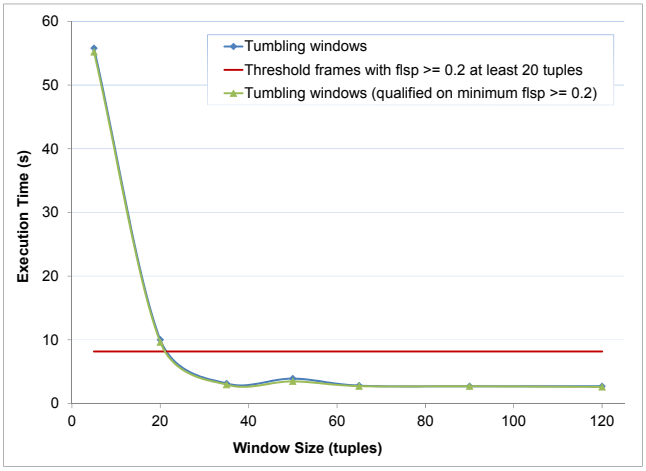

Figure 10. Query execution time on the dye data set.

error of the result, we measured execution time of frames and windows of varying sizes. The results are plotted in Figure 10, with different window sizes (number of contained tuples) on the X-axis. The execution time of the frame-based approach is shown as a horizontal red line, since it does not depend on the window size. For the window queries, we have measured two data points for each size. First, the cost of computing tumbling windows only (blue line) and then the combined cost of computing windows and qualifying them as part of an episode (green line) was measured. As can be seen, the qualification overhead is negligible and dominated by the cost of computing the windows.

We can observe that for small window sizes the execution times of the window queries are larger than the frame execution time. For larger window sizes, window execution time is somewhat lower than frames, but still comparable. Recall that the task-based performance results in Section 4 were measured using windows containing 50 tuples. While this configuration slightly outperforms frames in terms of run-time performance, it is noticeably less accurate than the frame-based approach with respect to task-based performance. Maier *et al.* [30] provide a detailed discussion of this performance study and shows 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.

6.3 Experiment #3: Run-Time Performance

Our first experiment on run-time performance used a very simple query to compare windows and frames. As episodes are detected in terms of window and frame boundaries, this query contained almost no additional computing. While this is a suitable experimental setup to quantify the net cost of windows versus frames, it does not provide any information on how these performance differences affect the overall execution time of a more complex query. In order to study this question, we return to Query #3 of the DEBS 2013 Grand Challenge, which computes heat maps for players during a soccer game.

The query plans that we used to measure task-based performance are shown in Figure 6 in Section 4. In order to measure run-time performance fairly, we slightly modified the windows query plan. Recall that it was necessary to introduce a split and join operator in order to aggregate a player's X and Y position in NiagaraST at the same time. In particular, the join is very costly. Since this limitation is not present in other DSMS, we chose to remove both the split and the join operator to measure run-time.

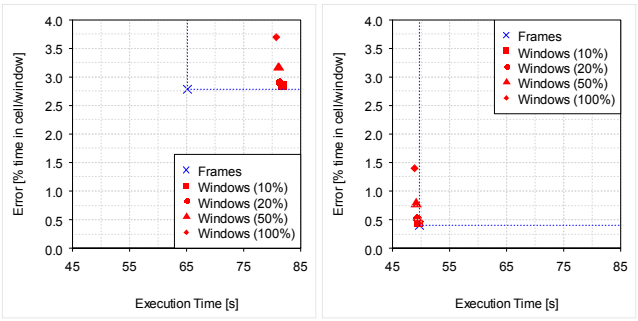

Figure 11. Run-time versus task-based performance for 60/1s (left) and 300/10s (right) reporting scheme.

Figure 11 shows the run-time and task-based performance results for the 60s/1s and 300s/10s reporting scheme. Again, we selected these specific result because in the 60s/1s scheme the difference between frames and windows was the most significant, whereas in the 300s/10s scenario it was least significant.

We note that 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 for 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 experimented with 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.

6.4 Summary, Discussion and Critique

In this section, we quantified the potential gain in run-time performance achieved by segmenting and summarizing a stream. We have already shown in Section 4 that frames support this approach without sacrificing task-based performance. A second experiment compared the net cost of computing frames to that of computing windows of different sizes. Finally, we have 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.

These experiments confirm one of our central criticisms of windows, i.e., that the window size and slide are typically not defined by the application scenario. Therefore, to choose windows to compute such a query, one must select a window size and slide based on experience or experimentation with a subset of the data stream. It is true that certain combinations of window size and slide provide similar levels of task-based and run-time perfor-

mance as frames. However, the experiments in this section show that for many combinations of window size and slide, either the run-time or task-based performance are border-line unacceptable. It is interesting to point out that frames are even able to outperform windows in an application scenario (heat maps), where a window-based reporting scheme was specified. Finally, we note that there are significant opportunities to optimize our frames implementation, which we will pursue in further work. 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 prior work: proposals to extend windows beyond traditional forms and techniques for optimal segmentation or summarization of datasets. We are not the first to propose enhancement of window functions. We believe our theoretical formulation for frames can ultimately unify many of those approaches, and provide them with less-operational semantics. That basis should permit easier analysis of properties of stream-segmentation schemes, such as maximality, disjointness and coverage. We examine some of the alternative 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. Differences from frames include use of per-tuple predicates that may not handle sum-based frames (IBM's InfoSphere Streams *Punctor* operator [22]), the application of predicates only to elements that are "live" at a given time t, (Predicate windows [12]). In addition, a proposal for window functions in XQuery relies in part on explicit `START` and `END` predicates [11]. This approaches 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.

Pattern-based approaches: Another group of window extensions involve pattern matching over stream elements. Matching a pattern is an alternative means to specify 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+ [3], 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 [31] and S-OLAP [8], a flavor of OLAP system that supports grouping and aggregation of data based on patterns. Zemke et al. [40] and McReynolds [31] propose a `MATCH_RECOGNIZE` clause for SQL to support pattern matching. Golab et al. [13] propose SQL extensions for pattern detection in stream warehouses. Such proposals could be useful starting points for a query-language syntax for frames.

Various proposals target composite events in complex-event processing (CEP) systems. Artikis et al. [5] present a model using *fluents* and Hirzel [17] also considers composite events as patterns over streams of simpler patterns, embodied in the MatchRegx operator of System S. That operator enforces certain global properties, such as left- and right-maximality of matches. Many proposals for composite-event detection seem more concerned with discrete events, whereas our focus has been numeric streams.

Scan Statistics: Scan statistics [14] identify local regions where density of elements is greater than expected by chance such as detecting distinctive episodes in temporal series [34]. We believe that episodes based on clustering statistics can fit into our model [30]. Etzion and collaborators [2, 10] consider extending windows to more general "contexts" that can group events temporally, spatially, and based on system or event state. We made initial forays into episodes with both temporal and spatial clustering [39].

Interval-based stream models: Stream systems that use an interval-based model for stream elements could support the incorporation of frames. The *snapshot window* capability [32] 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. The PIPES stream system [22] provides a facility similar to fragments for count-based windows over a stream partitioned into groups. PIPES provides a parameter to the partitioned window operator that causes a lengthy window to be broken into shorter pieces.

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*). Histogram construction (below) can be viewed as piecewise constant approximation. PRs are used for purposes such as preprocessing for data mining [24], detecting context changes [16] and approximate pattern matching [38]. 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." PR techniques are typically off-line; there are some on-line approaches to PR [24], but the focus seems to be on one-pass algorithms on stored datasets, rather than segmenting a dynamic stream. While PR techniques have addressed a variety of error measures, we have not seen systematic work that relates optimality relative to a given error measure to performance on a specific task.

Histograms: Histograms segment datasets and summarize each segment [24]. As with PR methods, most histogram construction is aimed at the stored-data case. There is a bit more consideration of task performance in this area, though some of it seems anecdotal, such as preferring equi-depth to equi-width histograms for selectivity and cardinality estimation. Specific tasks examined include join-size estimation [19], calculating selectivity of range predicates [35] and estimating cardinality of equality selections [21]. There are different techniques to optimize against different error measures. However, the choice of error measure does not generally get connected to task performance. There is one interesting example of task-based evaluation that we encountered. Ioannidis and Poosla [20] look at using histograms for approximate query answering, and they offer a figure in the style of our Figure 1 that draws a similar conclusion about what a good approximation of the data looks like.

8. CONCLUSION

We have 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 have developed and described a formal

specification of frames which we believe can unify existing stream segmentation approaches. We have implemented our new technique in the NiagaraST data stream processing system and provide two classes of evaluation – a quality improvement study, based on task-based performance, and a feasibility study based on run-time performance. The combination of these studies demonstrates how frames can improve task-specific performance without reducing run-time performance; the original goal of our work.

9. ACKNOWLEDGMENTS

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.

10. REFERENCES

[1] Abadi, D., et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal* 12(2): 120-139, August 2003.

[2] Adi, A. and Etzion, O. 2004. Amit - The Situation Manager. *The VLDB Journal* 13(2): 177-203, May 2014.

[3] Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. Efficient Pattern Matching over Event Streams. In *SIGMOD 2008*.

[4] Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal* 14(1), March 2005.

[5] Artikis, A., Sergot. M. and Paliouras. G. Run-time Composite Event Recognition. *DEBS 2012*, pages 69-80.

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

[7] Chandramouli, B., Goldstein, J., and Maier, D. High-Performance Dynamic Pattern Matching over Disordered Streams. In *VLDB 2010*.

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

[9] Cranor, C., Johnson, T., Spatashek, O. Gigascope: A Stream Database for Network Applications. In *SIGMOD 2003*.

[10] Etzion, O., Magid, Y., Rabinovich, E., Skarbovsky, I. and Zolotorevsky, N. Context-Based Event Processing Systems. *Reasoning in Event-Based Distributed Systems*. Springer Studies in Computational Intelligence 347, 2011, 257-278.

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

[12] Ghanem, T. M., Elmagarmid, A., Larson, P., and Aref, W. Supporting Views in Data Stream Management Systems. *ACM Trans. Database Syst.* 35(1): 1-47, February 2010.

[13] Golab, L., Johnson, T., Sen, S., and Yates, J. A Sequence-oriented Stream Warehouse Paradigm for Network Monitoring Applications. In *PAM 2012*, pages 53-63.

[14] Glaz, J. Naus, J., and Wallenstein, S., Ed. Scan Statistics: Methods and Applications. *Birkhauser Boston, Springer Science + Business Media*. 2009.

[15] Goldstein, J., Hong, M., Ali, M. and Barga, R. Consistency Sensitive Operators in CEDR. *Technical Report MSR-TR-2007-158, Microsoft Research*. 2007.

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

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

[18] Ioannidis, Y. The History of Histograms (abridged). In VLDB 2003.

[19] Ioannidis, Y.E. and Christodoulakis, S. Optimal Histograms for Limiting Worst-case Error Propagation in the Size of Join Results. *ACM TODS* 18(4): 709-748, Dec. 1993.

[20] Ioannidis, Y.E. and Poosala, V., Balancing Histogram Optimality and Practicality for Query Result Size Estimation, In *SIGMOD 1995.*

[21] Ioannidis, Y.E. Universality of Serial Histograms. In *VLDB 1993.*

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

[23] Jadagish, H.V.. Personal communication, 2014.

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

[25] Krämer, J. Continuous Queries over Data Streams–Semantics and Implementation. *PhD Thesis, Univ. of Marburg.* 2007.

[26] Li, J. Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T. and Maier, D. Out-of-Order Processing: a New Architecture for High-Performance Stream Systems. *VLDB Endow.* 1(1): 247-288, 2008.

[27] Li, J., Tufte, K., Maier, D., Papadimos, V. AdaptWID: An Adaptive Memory-Efficient Window Aggregation Implementation. *IEEE Internet Computing*, Nov-Dec 2008.

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

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

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

[31] McReynolds, S. Complex Event Processing in the Real World. *Oracle White Paper.* (September 2007)

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

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

[34] Preston, D. and Protopapas, P. Searching for Events in Time Series Using Scan Statistics. (Poster) *Initiative in Innovative Computing Open House, Harvard Univ.* (2008)

[35] Poosala, V., Haas, P.J, Ioannidis, Y.E. and Shekita, E.J. Improved Histograms for Selectivity Estimation of Range Predicates. In *SIGMOD 1996.*

[36] Rudensteiner, E.A., Ding, L., Sutherland, T., Zhu, Y., Pielech, B., and Mehta, N. CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. In *VLDB 2004*.

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

[38] Shatkay, H. and Zdonik, S.B.. Approximate Queries and Representations for Large Data Sequences. In *ICDE 1996*.

[39] Whiteneck, J., et al. Framing the Question: Detecting and Filling Spatio-Temporal Windows. In *IWGS 2010*.

[40] Zemke, F., Witkowski, A., Cherniak, M., and Colby, L. Pattern Matching in Sequences of Rows. *Draft SQL Change Proposal.* (March 2007)