# Capturing Episodes: May the Frame Be with You

David Maier, Michael Grossniklaus, Sharmadha Moorthy, Kristin Tufte
Computer Science Department
Portland State University
Portland, OR 92701
{maier, grossniklaus, moorthy, tufte}@cs.pdx.edu

## ABSTRACT

We are interested in detecting episodes in a data stream that are characterized by a period of time over which a condition holds, usually with a minimum duration. For example, we might want to know whenever any router has a packet-drop rate above 0.3% continuously for more than two minutes. Such episodes can be interesting in their own right for monitoring purposes, but they can also specify target regions for examination over the original or other stream. For instance, for each router-drop episode we detect, we might want to count the number of control messages the router received. We assert the key requirements are to *detect the episodes*, *detect them accurately*, and *detect them promptly*.

Current capabilities for data-stream management systems (DSMSs) include functionality, such as pattern-matching and windowed aggregates, that can help with detecting some kinds of episodes. We offer a third alternative, *frames*, which generalizes the other two. Frames are intervals that segment a data stream into regions of interest. In contrast to windows, frame boundaries can be data dependent, such as when a predicate holds for a given duration, or the maximum and minimum values of an attribute diverge more than a certain amount. We introduce frames and their theory, plus their implementation in the NiagaraST DSMS. We then demonstrate some advantages of frames versus windows, such as better characterization of episodes, on real data sets and explore an extension, *fragments*, to deal with long episodes.

## Categories and Subject Descriptors
H.2 DATABASE MANAGEMENT

## General Terms
Algorithms, Performance, Theory

## Keywords
Data streams, DSMS, episodes, windows, frames, NiagaraST

## 1. INTRODUCTION

In a data stream, there can be long periods of relatively "normal" activity, with periodic "episodes of interest." An episode of interest might be a period of high loss rate in a stream of network-router reports or a period of severe congestion, perhaps induced by a vehicle breakdown, in a stream of vehicle-traffic data. Stream applications that deal with normal operations, such as a display of current router loads or a traffic-speed map, can usually suffice with a relatively simple reduction of the data, such as 30-second or 1-minute summaries. For episodes of interest, by contrast, we want processing of the stream to depend on the episode itself.

To illustrate, let us consider in more detail a utility for monitoring the status and performance of a digital backbone network. One condition of concern is a router consistently dropping packets for an extended period. Assume the router produces a stream that reports on its activity every few seconds for monitoring purposes. A message in this stream might report a time (`rptTime`), total packets processed (`totalPkts`), aggregate size of the packets (`pktBytes`) and percentage of packets dropped (`dropPct`). Using this stream, we are interested in episodes of "sustained loss": For example, every period where `dropPct` is over 0.3% for at least two minutes. Once such a period is detected, we might want to simply flag the occurrence for a human operator to investigate. Or, we might use the episode to delimit an aggregate computation over the same stream (such as the average of `totalPkts` over the period) or a different stream (such as the number of control-plane commands during the episode).[*]

Note that in this scenario we should not overlook any sustained-loss episodes, and it is undesirable to report such an episode where none exists. The precise period of the episode is critical for proper correlation with other streams. Further, we likely do not want to wait for an extended period to find out that an episode is underway. Therefore, it is important to *detect the episodes*, *detect them accurately*, and *detect them promptly*.

Traditionally, the main mechanism for working with segments of a continuous stream has been windows [1][3][8][14][16]. A window is usually defined as a finite sub-stream, based either on a count of tuples or a fixed time period. A window-based operator performs its calculations on the contents of successive windows, rather than over the entire stream. While windows do deal in segments of streams, we find that they are not well suited for detecting and representing episodes. A key problem is that the boundaries of windows are explicitly set; either by a tuple count or fixed time duration. Thus, window boundaries will not necessarily line up well with the duration of an episode and windows may incorrectly report the existence of an episode (false positive) or totally miss one (false negative). We see at least three issues with using windows to identify episodes:

- Window sizes are fixed in terms of tuples or time. Reducing the window size to obtain better accuracy on


DEBS '12, July 16–20, 2012, Berlin, Germany.


[*] A control-plane command, such as a change to routing information, generally consumes considerably more computational resources than a regular data-plane packet.

episode boundaries leads to an increased number of windows.

- Windows are continually produced, meaning that we continue to generate windows (and other results that must be processed) during uninteresting periods.
- Windows are often incorporated into aggregate operators to produce summary values over the windows. We would like the flexibility combine episodes detected on one stream with data from another stream.

How can we improve on windows for capturing episodes in data streams? The key shortcoming of windows in such settings appears to be that windows are externally delimited, and do not actually derive from the data contents of the stream (apart from timestamps). We have been developing techniques for data-aware segmentation of streams, which we term *frames*. Broadly, a frame is an interval over a stream where a certain condition holds. So, for our sustained-packet-loss example, the condition is that the interval only contains tuples with `dropPct` > 0.3 and lasts at least 2 minutes.

Our initial experience with frames has shown their value in different kinds of stream applications. This paper focuses on the use of frames for detecting and representing episodes. In other work, we are looking at their utility in giving a better representation of a stream for particular tasks; we almost always get lower error levels as compared to the same number of windows (or the same error levels with fewer frames than windows).

We have been developing a theory of frames, which we introduce in Section 3. That theory covers both local and global conditions on frames in a manner independent of physical stream aspects, such as specific arrival order. It does provide a basis, however, for the incremental generation of frames, which is the basis for a practical implementation. Our implementation approach separates the detection of frames, in a *Frame* operator, from further processing, such as filling a frame with tuples then computing an aggregate over those tuples. There are several reasons for this separation. For one, in some applications all we need to know is the existence or interval of a frame. Second, we might want to use different tuples to fill the frame than we used to define it (such as the stream of control-plane commands in the router example). Third, we might want to post-process the frame, such as expanding its interval, before further processing of it. We have also looked at complications that arise when frames get very long, and support frame *fragments* in our implementation to help detect frames promptly.

While the general theory of frames accommodates a wide range of specification styles, our evaluation here concentrates on what we term Threshold plus Duration (T+D) frames, as many kinds of episodes can be captured with them. T+D frames are specified via a threshold on some attribute (e.g., `DropPct` > 0.3) and a minimum duration (e.g., 2 minutes). T+D frames typically cover only a subset of the time range of a stream, though we also consider a "partitioning" version where both qualifying episodes and intervening periods become frames. Our experiments with various datasets and frame specifications show that while shortening the window range can reduce duration and existence errors, such errors cannot be eliminated entirely with windows. Moreover, as the window range decreases, the processing cost, not surprisingly, goes up. We also see that frame fragments can be beneficial to downstream operators, while incurring no processing penalty.

## 2. EXAMPLES OF FRAMES

We present additional scenarios in which frames may be useful and for which traditional DSMS windows might not perform well. The first two correspond to data sources that we use in our evaluation section.

*Example 1—Dye Study of Lateral Mixing:* We have been working with data from sensors used to track the dispersion of fluorescent dye released in the ocean. Dye is measured with a photometer at a sub-second frequency, but is generally aggregated to over a longer period to reduce noise before further analysis. While windows can be used for this purpose, there are long periods in the sensor stream where no dye is present. Computing window aggregates during such periods is wasted effort. However, T+D frames can focus computation effort on episodes where dye is present in significant amounts. (We are also experimenting with *delta* frames with this data. With delta frames, we start a new frame whenever we have seen a shift in dye intensity—or other sensed variables such as salinity and temperature—over a given amount. Decomposing the stream in this manner ensures that the aggregate values for each period are in fact representative of the whole period.)

*Example 2—Vehicle-Traffic Monitoring:* In some cases, windowing can be seen as a form of approximation. Consider a stream of traffic speed and vehicle-count data; the user wants to know the "current" speed for a particular location. Cumulative speed and vehicle count are reported every 20 seconds; however the 20-second data has high variance. A window can be used to smooth the jitter in the data. However, we observe that the length of window desired during high-volume traffic (rush hour) may be different than the length of window during low-volume (overnight) time periods. In the overnight period, one may wish to use a longer window to account for the lower flow of traffic and to avoid the window speed being biased by a single particularly fast (or slow) vehicle. Instead of windowing on time, one may wish to window based on the number of vehicles over which the average speed is recorded. Thus, we may want windows defined over the stream such that sum(`VehicleCount`) > 100 for each window. In this example, a window with a fixed time period is a compromise, whereas a more desired and sophisticated window definition can be provided with frames.

*Example 3—Fine-grained Bursts:* An IPTV service was sometimes seeing packet loss when running at only 3-4% of capacity. The standard monitoring system for the service produced summary reports at 15-minute intervals, which were not revealing anything that might point to the cause of the poor performance. The problem was ultimately traced to orphaned sessions arising during short periods of frequent channel changing ("surfing" if you will). These episodes typically lasted for tens of seconds, and were not standing out in the 15-minute aggregates. It can be hard to know in advance the right granularity for monitoring; a technique such as frames, which can dynamically adapt granularity, might have been more enlightening as to the source of the problem.

## 3. THEORY OF FRAMES

For the purpose of this paper, we give a brief theory of frames that defines threshold frames with a minimum duration (T+D frames) that can be used to detect episodes in a data stream. Based on the formal specification of data streams, we first introduce the notion of a set of possible frames. Local conditions are applied to this set to obtain the set of candidate frames. The set of final frames is

then obtained by applying global conditions to the set of candidate frames.

**Definition (Data Stream):** A data stream $S$ is defined as an potentially unbounded sequence of tuples $S = [t_1, t_2, t_3, \ldots]$. All tuples of a data stream have the same schema. One attribute of the schema, the *progressing attribute*, is distinguished as it defines the logical order of the tuples. The stream progresses on this attribute. That is, if $A$ is the progressing attribute, then for any $n$, there is an $i$ such that $t_i.A > n$. Note that while the presence of a progressing attribute implies a logical ordering, it does not require that the tuples within the stream be physically arranged in this order.

## 3.1 Framing of a Data Stream

A framing of a data stream is defined incrementally via a sequence of sets of frames that are step-wise reduced by applying conditions, retaining only the relevant frames for a specific scenario. In the context of this paper, we focus on conditions that produce T+D frames, instead of presenting the theory of frame in its full generality.

**Definition (Possible Frames):** The *possible frames* $F_p(S)$ of a data stream $S$ are given by the infinite set of intervals (frames) $F_p(S) = [s, e] \mid s, e \in dom(A) \wedge s < e$, if $A$ is the progressing attribute.[†] Note that any subset of $S$ (including $S$ itself) is a *framing* of $S$.

Each interval $[s, e] \in F_p(S)$ has an *extent* that is the set of tuples $\{t \mid t \in S \wedge s \leq t.A < e\}$, where $A$ is the progressing attribute.

To obtain a framing consisting of T+D frames, further conditions need to be applied. We distinguish *local* and *global conditions* that are applied to restrict which intervals are contained in a framing. Local conditions apply to individual frames, whereas global conditions involve the entire set of frames.

In the case of T+D frames, there are two local conditions, which can be checked individually for each interval $[s, e]$ contained in the set of possible frames $F_p$. The first condition is a so-called *data-dependent predicate* requiring every tuple $t$ in *extent*$([s, e])$ to have a value above a given threshold $c$ on a specified attribute $X$, i.e., $\forall\, t \in$ *extent*$([s, e])$: $t.X > c$. (The threshold can also be defined as an upper bound: $\forall\, t \in$ *extent*$([s, e])$: $t.X < c$.) The second condition is a so-called *data-independent predicate* that guarantees the minimum 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 second condition is given by $e.A - s.A \geq n$, while in the latter case, it is given by $|\textit{extent}([s, e])| \geq n$, where $|\cdot|$ denotes set cardinality. The two conditions are used in a conjunction to form the local condition $p_l$ in the following definition.

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

For T+D frames, local condition $p_l$ ensures that all candidate frames are of minimum duration or longer and that they only contain tuples with an attribute value above or below the threshold, as required. It does not, however, guarantee that all frames have the maximal duration possible and are not over-lapping. In order to accurately detect the start and end point of episodes in a data stream, these additional requirements must also be satisfied. To obtain the set of final frames, we further constrain the set of candidate frames $F_c$ by applying a global condition $p_g$ to to the collection of all its intervals, rather than to individual intervals. To filter out frames that begin after or end before a full episode, we require that the framing is *maximal*: no frame is strictly contained in another frame. To avoid using two frames to cover a single episode, we require that the framing is *drained*, i.e., contains the minimal number of frames possible. Note that for any two overlapping or consecutive frames, the set of candidate frame also contains a frame that is the union of these two frames. Therefore, we can always replace a pair of overlapping and consecutive frames by their union to reduce the overall number of frames in a framing. In order to not lose any episodes, we require the final framing to *cover* the set of candidate frames: any candidate frame is a sub-interval of some final frame.[‡]

**Definition (Final Frames):** The *final frames* $F_f(S)$ of a data stream $S$ is the subset of candidate frames for which global condition $p_g$ holds. (While we do not prove it here, there is only one such subset.)

We note that an implementation of a frame operator will not follow the sequence of steps in our formal definition. Taking this approach would imply complete knowledge of entire data streams, which is not possible due to their unbounded nature. However, we note that T+D frames can easily be implemented online and incrementally. One implementation is to scan the data stream until a tuple is encountered that satisfies the data-dependent local conditions. The operator makes note of the start point corresponding to that tuple and continues to scan the stream as long as the encountered tuples meet the data-dependent local conditions. If a tuple is encountered that violates these conditions, the frame operator checks whether the data-independent conditions are met by the interval that spans from the first tuple to the last tuple that satisfied the data-dependent conditions. If so, the frame is reported. Otherwise it is discarded. It is not hard to demonstrate that this implementation also meets the global conditions of maximality, being drained and coverage.

## 3.2 Filling Frames

Once a framing of a stream has been defined in terms of a sequence of intervals, we can "fill" its frames, using the same stream or any other.

**Definition (Frame Filling):** For stream $S'$, we designate a filling attribute $B$. Let $\textit{fill}(S', [s, e]) = \{u \in S' \mid s \leq u.B \leq e\}$. The filling of a framing $F$ from $S'$, $\textit{fill}(S', F)$ is $\{\textit{fill}(S', [s, e]) \mid [s, e] \in F\}$.

We note that it is possible to modify the filling by applying a function $g(\cdot)$ to each frame (to possibly change its start and end points). We will generally then apply aggregation or another operation to the filled frames, to reduce each to a single tuple.

# 4. IMPLEMENTATION

This section describes an initial implementation of frames in the NiagaraST system, including frame fragments.

[†] The names *s* and *e* stand for start and end point, respectively.

[‡] A framing can also be *minimal*, *saturated*, *disjoint*, or *partitioning*. However, these global conditions cannot be applied to T+D frames, hence are not discussed in this paper.

## 4.1 NiagaraST Implementation

We have implemented frames in the NiagaraST [16] stream-processing system. NiagaraST is written in Java and is being developed at Portland State University. NiagaraST is based on the Niagara [21] system from the University of Wisconsin-Madison. The NiagaraST prototype implements T+D frames. In our implementation, the frame specification has two parts—a threshold predicate and a (time) duration: "Threshold Predicate T holds for duration at least D." T is a tuple-wise predicate over the stream; currently, D can be expressed as a number of tuples. The implementation supports frames such as those required for the dye-data, traffic-monitoring and router-monitoring examples (and can actually support more general "predicate plus duration" frames). A possible frame specification for the dye-data example is: dye fluorescence > 0.2 for at least 20 reports (20 tuples). We define a frame to be a maximal period satisfying the condition. In this example, if there were 25 consecutive reports in a row with dye fluorescence > 0.2, we would report it as a single frame instance. To implement frames in NiagaraST, we have implemented two operators: *ThresholdFrame* and *FillFrame*.

Figure 1 shows a sample query plan that uses the *ThresholdFrame* and *FillFrame* operators. The input data streams 1 and 2 are streams of continuous water-profile measurements in CSV format. In this example, data streams 1 and 2 contain the same data. The *CSVStream* operators parse the input stream into tuples with a set of attributes. One stream is used as input to *ThresholdFrame*, which produces a frame stream, say using a threshold value of 0.2 for dye-fluorescence and a minimum duration of 20 tuples. Each frame output by *ThresholdFrame* is given a unique frame-id. The frame stream (from the *ThresholdFrame* operator) and the output from the second *CSVStream* are used as input to the *FillFrame* operator, which tags each tuple from the second stream with the frame-id of the frame whose interval it matches (if any). After tuples are tagged with the frame-id, the average aggregate is applied to the depth and the dye-fluorescence attributes, with frame-id as the grouping attribute. We note that the input data stream 2 could be a different stream from input data stream 1 (as long as there is a comparable progressing attribute).

Also note that we could to modify the start and end times of a frame before filling it. Consider our example for extended packet loss. If we are using such frames to look at numbers of control-plane messages, then we could extend the start time of each frame earlier by a couple of minutes, to possibly capture control messages leading up to the episode.

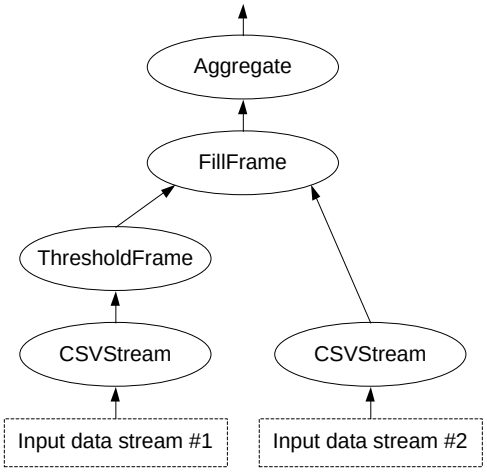


**Figure 1: Query Operator Tree Using Frames**

We implemented two new operators in NiagaraST to support frames: *ThresholdFrame* and *FillFrame*.

### *4.1.1 ThresholdFrame Operator*

The *ThresholdFrame* operator processes incoming data tuples (dye-data tuples in our example) and produces a stream of frames. Specifically, the output of *ThresholdFrame* is tuples each containing a frame-id plus frame start and end times. The start and end times are expressed in terms of the progressing attribute (usually a sequence number or timestamp) of the data tuples, so frames can correlate with the same or other streams. In many scenarios, one wishes to create frames over sub-groups of the input as opposed to creating frames over the entire input. For example, in the network-router scenario, we may wish to group by router-id so that we can detect periods of high loss at each router. We call such frames groupwise frames, and include grouping attributes in the output.

The arguments to the *ThresholdFrame* operator include the name of the progressing attribute, the minimum duration and the name of any group-by attributes for groupwise frames. The progressing attribute is used to apply the minimum duration condition and to identify frame start and end times. In the NiagaraST implementation, the *Apply* operator handles the application of the threshold condition. *Apply* evaluates the threshold condition on the input stream tuples and appends a Boolean attribute to each tuple indicating whether the threshold condition is met or not.

In the current implementation, frame processing is done only when stream punctuation is received. A *punctuation* is an in-stream marker noting that all tuples with a value of the progressing attribute before a certain point have been seen (possibly on a per-group basis). The *ThresholdFrame* operator buffers regular tuples it receives on its input. When a punctuation is received, an ordered list of buffered tuples is traversed to examine tuples logically earlier than the punctuation. When a tuple for which the threshold predicate is true (as determined by the *Apply* operator) is encountered, a tentative frame is created with a frame-start time equal to that of the tuple. Subsequent tuples are traversed and while the threshold predicate of those tuples is true, the frame end time is updated. When a false predicate value is detected, the system checks if the frame satisfies the minimum duration condition. If so, a frame tuple of the form (frame-id, group-by value, start time, end time) is generated. Whether the minimum-duration condition of the frame is met or not, the current frame state is reset and tuple processing begins anew. Thus at each punctuation, a list of frames, if any, is output. If a frame is under consideration (minimum duration condition not met yet) at the punctuation point, frame state is retained (but see the frame fragments discussion below). If the input punctuation timestamp is greater than the end time of the last frame, the same punctuation is output. Otherwise, a punctuation with the end time of the last frame is output. Processing only upon receipt of punctuation allows the system to tolerate tuple disorder. If an assumption was made about tuples arriving in order, tuples could be processed as they arrived.

### *4.1.2 FillFrame Operator*

The *FillFrame* operator combines a frame stream and a second input stream to fill the frames. *FillFrame* tags tuples from the filling stream with the appropriate frame-id, if a matching frame can be found, and outputs any such tagged tuple.

*FillFrame* maintains a hash map of (group-id, list of frames) entries and a hash map of (group-id, list of tuples) entries to buffer tuples from the frame and filling streams, respectively. When a

frame tuple *f* = (*fid*, *gid*, *fstime*, *fetime*) arrives from the frame stream, we output every matching data tuple, *t*, with progressing attribute *A* from the list of tuples corresponding to *gid*. Matching tuples are those such that *A* falls between *fstime* and *fetime*. After that, *f* is buffered in the list of frame tuples for *gid*. When a tuple *t* arrives from the filling stream with progressing attribute *A* and group-id g*id*, we search the list of frame tuples corresponding to *gid* for a matching frame (*fid*, *gid*, *fstime*, *fetime*). A matching frame is one such that *A* falls between *fstime* and *fetime*. Tuple *t* is then tagged with *fid* and output. If a matching frame is not found we buffer *t* in the list for *gid*.

When a punctuation arrives from the filling stream, we purge all frames from the list-of-frames hash table whose frame end time is less than or equal to the punctuation point. If the punctuation has values for both the group-by and progressing attributes, the processing is done just for the list of frames corresponding the group-by value. If it has a value only for the progressing attribute, it is processed for all the groups. We also pass on punctuation with the frame-id of the last frame purged from the buffer. When a punctuation arrives from the frame stream, we purge tuples in the list-of-tuples buffer whose progressing attribute is less than or equal to the punctuation point, as we do not expect any matching frames for these tuples at that point.

## 4.2 Frame Fragments

One issue with frames is that the duration of episodes, and the periods between episodes, can be arbitrarily long. This possibility can have adverse effects for both the client and for processing performance. For the client, there are delays in finding out about the onset of an episode. For example, in the router scenario, if a sustained-loss episode lasts for an hour, the client will not know about it until it ends. From a performance standpoint, our *FillFrame* operator has to hold onto an hour's worth of tuples that will be used when the frame is finally complete.

In our setting, punctuations can help with these issues somewhat. Since a frame has both a start and end time, either is a candidate for punctuation. Punctuation on end time does not actually tell a client or downstream operator whether an episode is underway—it just indicates that no new episode has completed before a certain time. (But we will describe an approach shortly where end-time punctuation is useful.) Punctuation on start time can let a client know that there are certain periods that no episode is going to span. However, if an episode is (or might be) underway, then start-time punctuation will be blocked. However, a client cannot reliably distinguish between no punctuation because a frame is under consideration versus there is a delay in stream delivery.

Long frames can also lead to performance problems. A particular issue arises in the *FillFrame* operator, where tuples from the filling stream will accumulate in operator memory waiting for the corresponding frame to arrive from the *Frame* operator. When the frame does arrive, time is required to work through this backlog of tuples.

To address these problems with long frames, we note that the *Frame* operator often knows that there will be a frame forthcoming long before it emits the frame. For example, in a T+D framing scheme, once the framing attribute has been above the threshold for the requisite duration, there definitely will be a frame emitted. Thus, once a frame is guaranteed to be forthcoming, the *Frame* operator can inform downstream operators or clients. We do so by having the *Frame* operator emit frame *fragments*. A fragment is a piece of a larger frame emitted periodically when a long frame is "underway". Our implementation emits frame fragments upon receipt of punctuation. In some sense, splitting a frame into fragments is similar to dividing a window into panes [17]. Our *FillFrame* operator can use a frame fragment to process any tuples it is holding (or that subsequently arrive) that fall in the fragment interval. While the use of fragments does mean that the *Frame* operator emits more output, we will see in the Evaluation section that the net effect appears beneficial. *FillFrame* tends to have lower memory requirements, and overall query time does not grow. Furthermore, the rate of fragment production can be controlled via the frequency of punctuation. Punctuations also reveal when a frame is complete: if the end time of a fragment is strictly before the time in a punctuation, it must be the final fragment in its frame. (If the fragment happens to end exactly on the punctuation, then the following punctuation will reveal the end of the frame.)

# 5. EVALUATION

We evaluate frames as a technology to detect episodes based on a comparison with windows. We compare frames to windows in terms of accuracy and run-time performance followed by an evaluation of the benefits of using frame fragments.

All experiments were run in NiagaraST, using its existing window implementation and the frame implementation presented in the previous section. The figures reported in this section were measured on a Dell Optiplex 780 with a 3 GHz CPU and 4 GB of main memory. We used data sets from two different application domains – dye data and traffic data (cf. Examples 1 and 2 in Section 2).

The second data set contains measurements from dye track cruise W0908B (tow4 and tow13), conducted by Oregon State University in the Pacific Ocean off the coast near Newport on August 31, 2009 (cf. Example 2 in Section 2). All performance results reported in this section are averages over ten execution runs.

## 5.1 Accuracy Metrics

By definition, frames capture episodes precisely, as they directly evaluate the condition that defines an episode. As a consequence, we examine how closely windows can approximate a given framing of a data stream. To create the base case, we frame a data set using threshold frames with a minimum duration. For comparison, we create time-based tumbling windows to segment the data set; note that by using time-based tumbling windows, we obtain a set of non-overlapping windows that partition the data set. From this initial set of windows, we use a second processing step to select a subset of those windows to be used to represent the episodes in the data set. Thus we have a set of frames and a set of "selected windows" to be used for comparison. In our experiments, we vary the window size and the window selection criteria. We quantify the accuracy of windows with respect to frames by using two definitions of error: *duration error* and *existence error*.

*Duration error* measures all regions (in terms of tuple count) that are misclassified by the window approximation. Duration error includes, but is not limited to, error due to frame-boundary alignment. Since windows start at pre-defined, data-independent points in the stream, they may capture the start or end point of an episode at too early or too late a point in the stream.

*Existence error* compares the set of windows selected to represent the episodes in the data set with the frames that represent the data set. *Existence error* consists of two cases: false negatives and false positives. False negatives are frames for which there is no

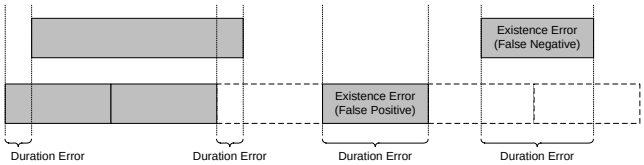


**Figure 2: Duration and Existence Error**

overlapping selected window (episodes that are totally missed by the window approximation). False positives are selected windows for which there is no overlapping frame (episodes that are incorrectly suggested to exist by the window approximation). In contrast to *duration error*, *existence error* is measured in counts of windows or frames.

Figure 2 graphically displays *duration error* and both types of *existence error*. In this figure, the frames are represented by the upper level of gray boxes. The window segmentation is shown in the lower layer with selected windows shown in gray. As described, duration error measures all misclassified regions. The figure also shows the two cases of existence error—where a selected window overlaps no frame and where a frame exists, but no selected window overlaps it.

## 5.2 Run-time Performance

We evaluate the run-time performance of frames with respect to windowing schemes with different window sizes. Since the motivation of frames in not to introduce more efficient processing, but rather to provide additional functionality, our aim is to confirm that this functionality does not come at too high a price. Specifically, we intend to demonstrate that frames can be implemented to have run-time characteristics that are comparable to windows. Our performance figures include the time to send the query results to an off-node client.

## 5.3 Traffic Data Set

The first evaluation for frames uses the traffic data set (cf. Example 1 in Section 2). This data set contains records of the number of cars that pass over a specific location and their average speed at 20-second intervals. The data set used was collected during January 2012 at a detector on I-5 Northbound at Terwilliger Blvd. in Portland.

The baseline framing of the traffic data set was created using a threshold framing scheme with a threshold of "speed less than 40 mph" with a minimum duration of 3 minutes, i.e., episodes where the average speed drops under 40mph for at least 3 minutes. We have created eight different stream segmentations using windows of increasing duration from 1 to 16 minutes (60 to 960 seconds). For each of these window-based stream segmentations, we have applied three heuristics to determine which windows to select to represent episodes. The first heuristic, which we term the "minimum heuristic," selects windows where the minimum speed value of tuples in the window is below the threshold. This heuristic is very liberal and is likely to yield many false positives. The second heuristic, the "maximum heuristic," selects windows whose maximum speed value is below the threshold. It is very conservative and likely to produce many false negatives. Finally, the third heuristic, the "average heuristic," checks whether the average value of a window is below 40mph and is thus expected to lie between the other two approaches in terms of error.

### *5.3.1 Traffic Data Set – Accuracy*

Figure 3 plots duration error (in tuples) for different window ranges (in seconds) for each of the three heuristics. A solid black line indicates the total frame duration – that is the total number of

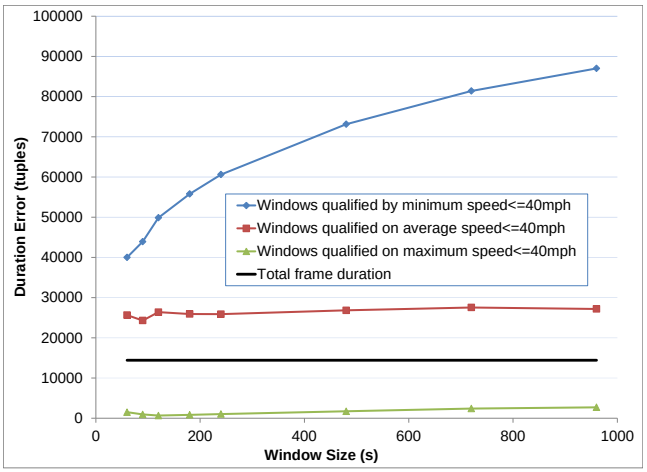


**Figure 3: Duration Error – Traffic Data Set (minimum duration 3 minutes)**

tuples in frames. The frame duration is necessary to understand the severity of the duration errors for windows.

The maximum heuristic yields a small duration error. However, the average and minimum heuristics yield duration errors greater than the frame duration. Thus, for these two heuristics, the number of tuples incorrectly selected or not selected as being part of episodes is greater than the number of tuples in actual episodes (recall frames are accurate). The magnitude of this error is significant and potentially unacceptable.

Figure 4 plots the existence error for the case of false negatives. In this figure, the y-axis contains the percentage of frames that were missed by the window approximation versus the total number of frames. Experiments for different window sizes are shown. The experiment confirms our expectations with respect to the behavior of the three heuristics. Selecting windows based on their maximum value is too conservative to cover all frames and therefore misses more episodes (frames) as the window size increases. The percent of missed frames is up to almost 70% for the maximum heuristic. The other two heuristics, in contrast, produce no more than 10% false negatives in all cases.

Consider Figures 3 and 4 together. We observe that in Figure 3, the duration error of the average and minimum heuristics is significant, bordering on unacceptable. In Figure 4, the error for

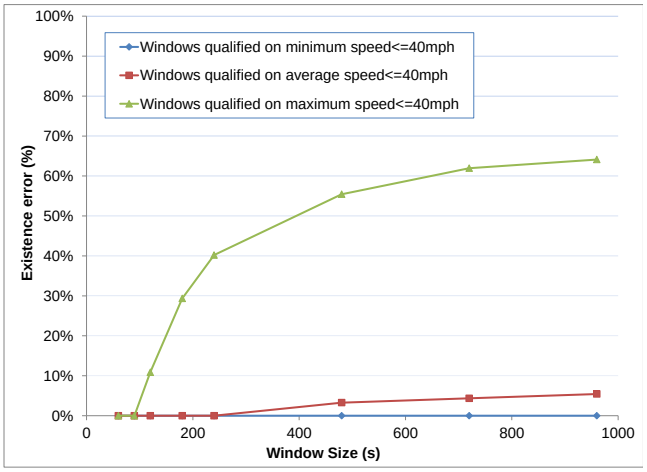


**Figure 4: Existence Error (False Negatives) – Traffic Data Set (minimum duration 3 minutes)**

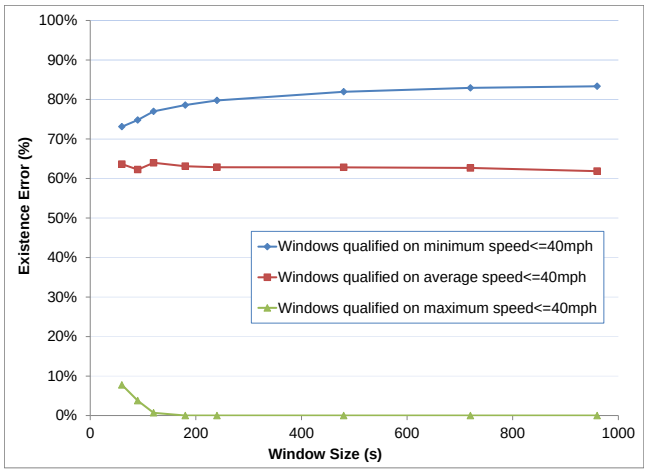


**Figure 5: Existence Error (False Positives) – Traffic Data Set (minimum duration 3 minutes)**

the maximum heuristic is borderline unacceptable. Except for very small windows, there is no configuration or heuristic capable of capturing episodes with nearly the same accuracy as frames.

In Figure 5, false positives are plotted as the percentage of falsely identified windows versus the total number of qualified windows. Experiments with different window sizes are shown. As above, the performance of the heuristics varies widely and is as expected based on the conservative or liberal nature of the heuristics.

### *5.3.2 Traffic Data Set – Performance*

The query execution time that we measured while running the experiments with the traffic data set as described in the previous sub-section is plotted in Figure 6. Execution time on the y-axis (in seconds) is related to the different window sizes on the x-axis (in seconds). The baseline framing of the data stream does not depend on window sizes and is shown as a horizontal line for comparison. We observe that the windowing schemes with very small window sizes that are required for accurate approximation of the framing have longer query-execution times than the baseline framing. The query execution time of the window-based approach falls below the one using frames only once the window size is increased.

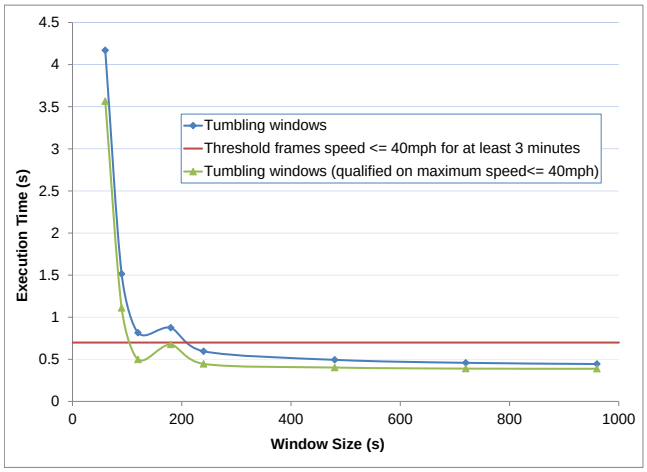


**Figure 6: Query Execution Time on the Traffic Data Set (minimum duration 3 minutes)**

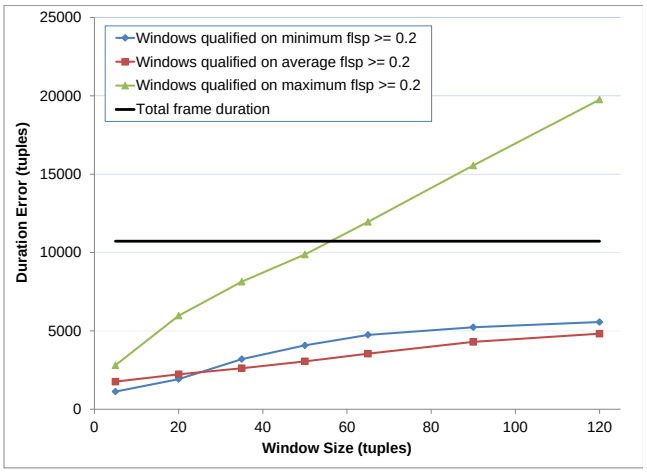


**Figure 7: Duration Error – Dye Data Set (tow 13, minimum duration 20 tuples)**

## 5.4 Dye Data Set

In the dye data set we create a baseline framing that detects episodes where the measured fluorescence is above 0.2 flsp for at least 20 tuples. Again, we create a series of window-based stream segmentations by varying the window size from 5 to 120 tuples. In the selection step, we use the same three heuristics as for the traffic data set, that is, the minimum, maximum, or average value of a window determines whether it is selected as representing an episode. In contrast to the previous example, we note that in this example an episode is detected if a measured value is *above* the threshold. As a consequence, the roles of the heuristic using the minimum value of a window and the one that looks at the maximum are now reversed, with the former being conservative and the latter being liberal.

### *5.4.1 Dye Data Set – Accuracy*

Figure 7 plots the duration error on the y-axis (number of tuples) for the three heuristics with respect to different window sizes (number of tuples) on the x-axis. In order to contextualize these results, we also include a horizontal line in the graph that indicates the total number of tuples that are included in frames. Even for very small window sizes, the liberal heuristic of selecting windows based on their maximum value selects more than twice as many tuples as the frames. For larger window sizes, this strategy degenerates rapidly and yields a massive duration error. For a window size of 120 tuples (the largest size we have measured), it mistakenly selects about 20000 tuples, which is 5% of the entire data set (400000 tuples). The conservative and average heuristics both have a lower duration error. For a window size of 10 tuples (where a total of 40000 windows need to be post-processed), the conservative heuristic introduces a duration error of about 25% relative to the total number of tuples contained in frames. The average heuristic yields a relative duration error of about 50% for this small window size already. Both heuristics quickly degrade to a relative duration error of almost 100% as the window size is increased.

The existence error for the case of false negatives is shown in Figure 8. The y-axis plots existence error as a percentage that relates the number of frames that have no corresponding windows to the total number of frames (100 for this data set). Not surprisingly, the conservative heuristic fails to identify the most frames and already yields an existence error of over 50% for small window sizes. The liberal heuristic does not introduce any false negatives as one tuple above the threshold is enough to select a

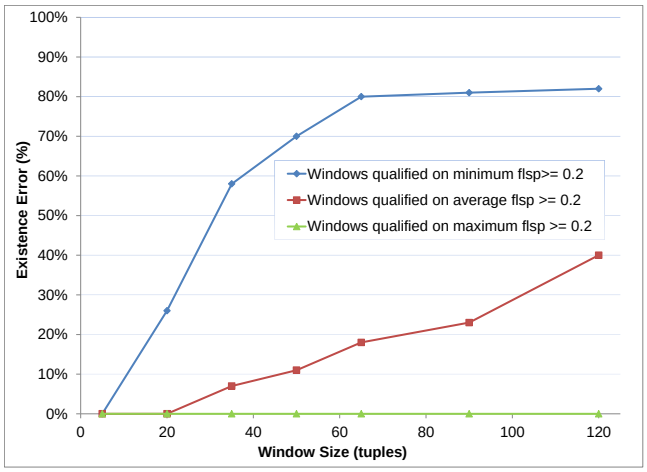


**Figure 8: Existence Error (False Negatives) – Dye Data Set (tow 13, minimum duration 20 tuples)**

window. Finally, the average heuristic lies in the middle, but quickly has a non-negligible existence error of more than 10% as the window size increases.

Figure 9 shows the case of false positives for the existence error, defined as the percentage of windows selected without a corresponding frame with respect to the total number of qualified windows. The existence error is plotted on the y-axis and related to different window sizes (in number of tuples) on the x-axis. As can be seen, the liberal strategy trades off avoiding false negatives (cf. Figure 8) by conceding an existence error of about 30% in terms of false positives. In the case of the conservative heuristic, the trade-off is reverse, i.e., false positives are avoided at the expense of false negatives. Again, the average heuristics lies between the other two, but yields a relatively high percentage of 10% of false positives for the smaller window sizes that correspond to an acceptable percentage of false negatives (cf. Figure 8).

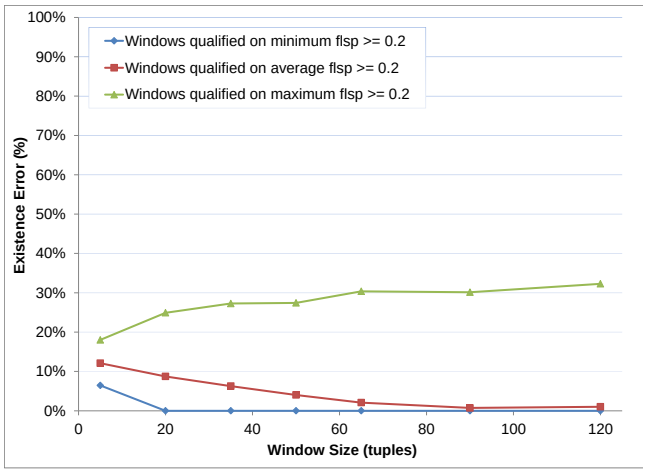


**Figure 9: Existence Error (False Positives) – Dye Data Set (tow 13, minimum duration 20 tuples)**

So far, we have only experimented with T+D frames with a minimum duration greater than one. In addition, for the dye data set, we focused on one instance of data – tow13. To conclude our evaluation of accuracy, we look at the case where there is no minimum duration: a frame only needs to contain at least one tuple and switch to a different dye data set – tow4. In Figure 10, the duration error for each of the three heuristics on the two4 data set is plotted. As a comparison, Figure 11 plots the duration error on the same data set for minimum duration 20.

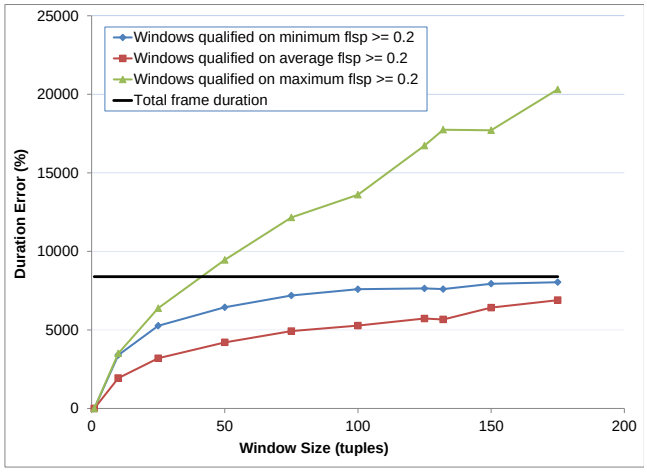


**Figure 10: Duration Error – Dye Data Set (tow4, minimum duration 1 tuple)**

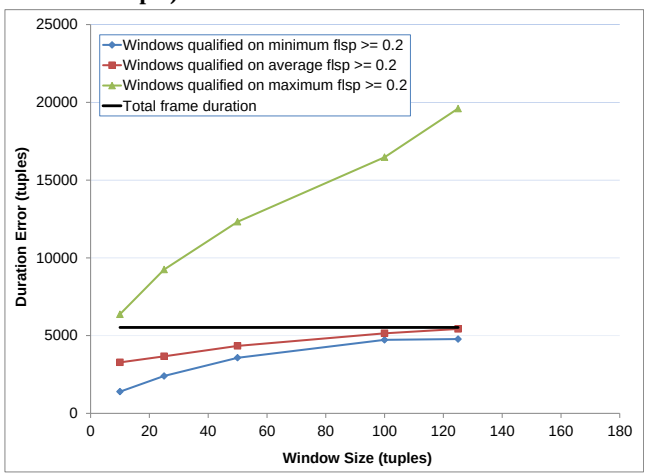


**Figure 11: Duration Error – Dye Data Set (tow4, minimum duration 20 tuples)**

Figure 12 shows how the windows selected by different heuristics compare to frames on a stretch of data. The frames defined by the threshold condition are shown in black at the bottom of the figure. Green lines represent the windows selected by the maximum heuristic, red the average heuristic, and blue the minimum

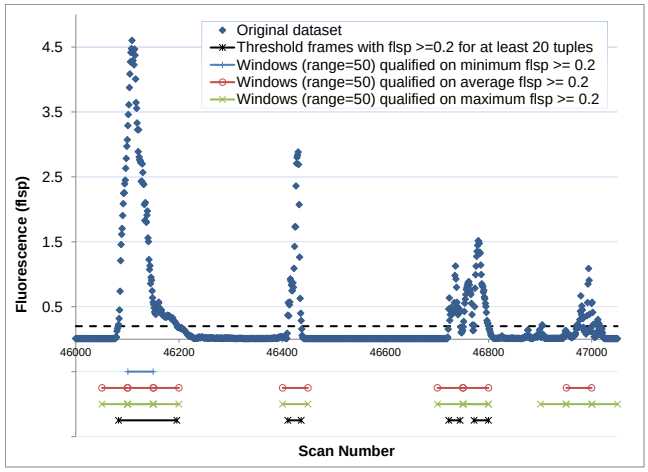


**Figure 12: Comparison of the three Heuristics versus Frames**

heuristic. Duration error can be seen near scan number 46400 and 46800. False positives can be seen for the maximum and average heuristics on the right hand side of the plot and the false negatives can be seen for the minimum heuristic.

### *5.4.2 Dye Data Set – Performance*

The query execution time measured on the dye data is shown in Figure 13. The blue lines show execution times for windows and frames on the dye data set. For small window sizes, the execution times for windows are larger than the frame execution time (shown as a horizontal red line). For larger window sizes, window execution time is somewhat lower than frames. Frames are noticeably more accurate than windows for most window sizes as shown in the previous sub-section.

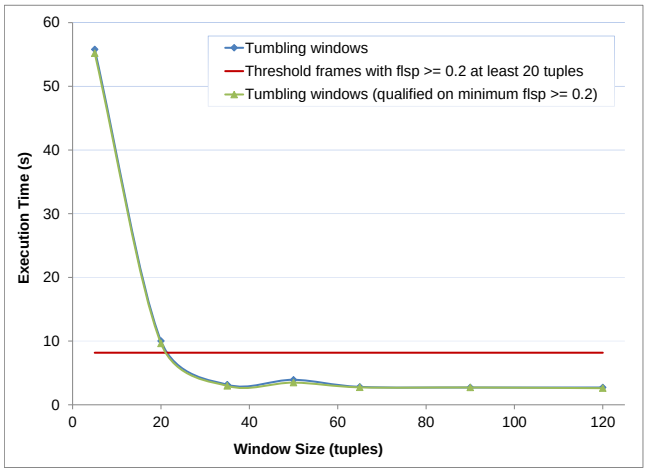


**Figure 13: Query Execution Time on the Dye Data Set**

## 5.5 Evaluation Discussion

If one were to choose windows to approximate episodes, one must choose a window size and a selection heuristic. It is true that certain combinations of window size and selection heuristic provide similar levels of accuracy and execution time performance as frames. However, the experiments in this section show that for many combinations of window size and selection heuristic, the accuracy errors (duration and existence error) are border-line unacceptable. In addition, for small window sizes, the execution time can be significantly greater than the frame execution time. We also note that we have not optimized our frames implementation and we believe there are significant optimizations to be made, 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 existence and duration accuracy of using frames.

## 5.6 Frame Fragments

To conclude this section, we study the benefits and overheads of using fragments (or partial frames) to address the issue of very long frames. In particular, we demonstrate that the use of frame fragments reduces the memory requirements of operators that consume frames. At the same time, it does not have a significant effect on query execution times. The experiments were conducted using tow13 from the dye data set. We have selected this particular tow as it produces particularly long frames, which is required to study the effect of fragments.

In Table 1, we quantify the benefits of using fragments in terms of reducing the memory requirements of frame. If long frames occur in the data stream, a large number of tuples need to be buffered within the *FillFrame* operator while the frame is growing. We therefore study memory requirements of large frames in terms of the maximum and average size of this buffer. Table 1 contains the corresponding numbers for the previously introduced experiments with threshold frames of minimum duration 1 and 20, respectively. As can be seen, fragments reduce memory requirements in all cases. In the case of the maximum size of the buffer, the reduction is quite significant.

**Table 1: Size of tuple buffer in *FillFrame* operator**

| **dye data set** (tow13) | **no fragments** | | **fragments** | |
|---|---|---|---|---|
| | **max** | **avg** | **max** | **avg** |
| T+D frames (T > 0.2, D = 1) | | | | |
| over all frames | 151026 | 687 | 459 | 209 |
| over all punctuations on frame stream | 151026 | 1972 | 459 | 215 |
| T+D frames (T > 0.2, D = 20) | | | | |
| over all frames | 960 | 324 | 459 | 215 |
| over all punctuations on frame stream | 1029 | 322 | 460 | 302 |

In Table 2, we summarize the query execution times for these two types of threshold frames, with and without fragments. As can be seen, the use of fragments has almost no effect on query execution times. In the case of threshold frames with minimum duration 1, they even improve query execution time slightly.

**Table 2: Execution times**

| **dye data set** (tow13) | **no fragments** | **fragments** |
|---|---|---|
| T+D frames (T > 0.2, D = 1) | 4.765 | 4.128 |
| T+D frames (T > 0.2, D = 20) | 7.798 | 8.169 |

# 6. RELATED WORK

Past research on data-driven windows includes work on predicate-windows by Ghanem *et al.* [9][10]. Tuples enter into and expire from a predicate-window depending on whether they satisfy the predicate associated with the window. Predicate windows have features in common with frames in that the extent of a window can depend on values of the stream elements. Some predicate windows may be expressible as framing schemes and vice versa. However, there are two key differences. First, a predicate window instance does not necessarily correspond to a contiguous range over the input stream. Second, the contents of a window is determined by applying a predicate individually to each stream element that is "live" at a particular time *t*, and generally does not depend on the arrangement of those elements. (The predicate windows approach posits a *correlation* attribute CORAttr, such that a later tuple $e_2$ with the same CORAttr value as a previous tuple $e_1$ is assumed to be an update of $e_1$. In this instance, a later tuple can affect the inclusion of an earlier tuple.)

The motivations for frames overlap with those for pattern matching over streams. Matching a pattern is a possible way to define candidate frames, and thus fits in our model. Additional conditions of maximality of matches or non-overlap might be imposed in addition. Simple pattern matching, such as unadorned regular expressions, is not expressive enough to express frames where, say, the delta between min and max values of an attribute is needed, or a time-based minimum duration is required. (If minimal duration were expressed as a tuple count, then one could implement T+D frames with by applying a predicate for the

threshold to flag tuples, followed by a finite automaton to check the duration.) Most pattern-matching approaches, such as SASE+ [2], Cayuga [5] and AFAs [6], go beyond simple finite automata and support manipulation of auxiliary state. Such enhanced matching facilities can generally be adapted to detect frames, especially if the pattern specification allows user defined functions for manipulating state. Other pattern-matching work that is not stream-based includes SQL-TS, an extension to SQL that supports searching for complex patterns in database systems [23] and S-OLAP [7], a flavor of online analytical processing system that supports grouping and aggregation of data based on patterns [6]. Zemke et al. [25] and McReynolds [18] have proposed a MATCH_RECOGNIZE clause for SQL to support pattern matching. Such proposals could be useful starting points for a query-language syntax for frames.

Scan statistics [11] identify local regions where density of elements is greater than expected by chance. These regions can be based on either fixed-size windows or variable-size windows. One use of scan statistics is to detect distinctive episodes in temporal series [22]. We believe that episodes based on clustering statistics can fit into our frame model, although scoring the significance of a local cluster would require estimating the base distribution without seeing the entire stream.

Stream systems that use an interval-based model for stream elements, such as StreamInsight and PIPES, have existing features that could support the incorporation of frames. The *snapshot window* capability [12][20] of StreamInsight is a simple form of data segmentation based on content. A new snapshot window is generated every time the set of active elements changes. It seems a short step to have window boundaries determined by change in some function of the window contents, such as the sum over a particular attribute. Furthermore, our *FillFrame* operator is essentially the StreamInsight *Join* with an interval stream on one side and a point stream on the other. The PIPES stream-processing system [13] provides a facility similar to fragments, but for slightly a different purpose. The target is count-based windows over a stream partitioned into groups. For a group with very few tuples, a count-based window can grow very large in terms of time. PIPES has interval-based elements, and depends on streams being ordered by increasing start time of elements. A "lengthy" window in a sparse group can thus come to block the output of later windows from other groups (as all partitions feed to a single output stream). PIPES provides a parameter $\Delta$ to the partitioned window operator that causes a lengthy window to be broken into pieces of no more than $\Delta$ in duration. In a sense, the fragmentation derives from issues arising from latency with both PIPES and frames, but for PIPES the problem originates with enforcing order, whereas in NiagaraST, physical streams do not need to be strictly ordered (as long as they are correctly punctuated).

In contrast to their window counterparts, frames are adaptive. Adaptive mechanisms and systems have been proposed in the past for stream management systems, including CAPE [19], StreaMon [4] and AdaptWID [15] adaptive query processing in general. Some of these techniques adapt to stream contents in terms of tuple density or data-distribution properties. Frames, by comparison, adapt based on properties of the data and its arrangement.

# 7. CONCLUSION

While still a work in progress, we think the frame approach goes a long way towards fulfilling the desiderata we listed for supporting episodes—*detect the episodes*, *detect them accurately*, and *detect them promptly*—especially as compared to windows. T+D frames can detect a large class of episodes and their boundaries accurately. Our evaluation shows that windows may do poorly at this task, even when the number of windows is much greater than the number of frames required. Execution overhead for frames is comparable to that for windows (and frames might have an advantage if processing overhead at the client were included). We have seen that the use of fragments aids the prompt detection of, and action upon, frames; fragments do not seem to impose a processing overhead, when considered from the full-query level.

Going forward, we are implementing and testing other kinds of frames, such as those based on "maximum deltas" for one or more attributes, and ones on a derived property of the frame extent, such as sum over vehicle counts. We are particularly interested in which framing schemes work best for which tasks. We are also extending our theory beyond T+D frames.

Frames as described here reflect a temporal segmentation of a stream. Many of the data sources we work with also have a spatial component: mileposts for traffic data; latitude, longitude and depth for dye data. Having frames that capture episodes with both temporal and spatial extent, such as traffic congestion on a particular stretch of highway, are a logical extension, and we have made initial forays in this direction [24].

# 8. ACKNOWLEDGMENTS

The authors thank Ted Johnson for the sustained-loss router example, which actually started us on this line of work. The IPTV example also comes from conversations with him. James Whiteneck and Rafael Fernández helped with early implementations and examination of frames. Oregon State University's "Team Florescin" collected the dye data: Murray Levine, Steve Pierce and Brandy Cervantes.

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

# 9. 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] Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. Efficient Pattern Matching over Event Streams. In *SIGMOD 2008*. (Vancouver, Canada, June 2008).

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

[4] Babu, S., and Widom, J. StreaMon: An Adaptive Engine for Stream Query Processing. In *SIGMOD 2004*. (Paris, France, June 2004).

[5] Brenna, L., *et al*. Cayuga: A High-Performance Event Processing Engine. In *SIGMOD 2007*. (Beijing, China, June 2007).

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

[7] Chui, C., Kao, B., Lo, E., and Cheung, D. S-OLAP: An OLAP System for Analyzing Sequence Data. In *SIGMOD 2010*. (Indianapolis, Indiana, USA, June 2010).

[8] Cranor, C., Johnson, T., Spatashek, O. Gigascope: A Stream Database for Network Applications. In *SIGMOD 2003*. (San Diego, California, June 2003)

[9] Ghanem, T. M., Aref, W. G., and Elmagarmid, A. K. Exploiting Predicate-Window Semantics over Data Streams. *SIGMOD Rec.* 35(1): 3-8, March 2006.

[10] 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.

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

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

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

[14] 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.

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

[16] 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*. (Baltimore, Maryland, June 2005).

[17] 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.

[18] McReynolds, S. Complex Event Processing in the Real World. *Oracle White Paper*. (September 2007) Available at http://www.docstoc.com/docs/17900898/Complex-Event-Processing-in-the-Real-World

[19] 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* (Toronto, Canada, 2004).

[20] MSDN Library. Snapshot Windows. In *Developers Guide (StreamInsight):Writing Query Templates in LINQ*. Available at: http://msdn.microsoft.com/en-us/library/ff518550.aspx

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

[22] Preston, D. and Protopapas, P. Searching for Events in Time Series Using Scan Statistics. (Poster) *Initiative in Innovative Computing Open House, Harvard Univ.* (2008) Available at http://iic.seas.harvard.edu/posters/scanstats-poster.pdf

[23] 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.

[24] Whiteneck, J., et al. Framing the Question: Detecting and Filling Spatio-Temporal Windows. In *IWGS 2010*. (San Jose, California, Nov 2010)

[25] Zemke, F., Witkowski, A., Cherniak, M., and Colby, L. Pattern Matching in Sequences of Rows. *Draft SQL Change Proposal.* (March 2007) Available at http://www.cs.ucla.edu/classes/spring12/cs240B/notes/row-pattern-recogniton-11.pdf