Frames: Intrinsic Windows over Data Streams

James Whiteneck, Kristin Tufte, David Maier, Amit Bhat, Rafael J. Fernández-Moctezuma
Department of Computer Science
Portland State University
P.O. Box 751, Portland, OR
+1 (503) 725-2406
{whitenej, tufte, maier, amitb, rfernand}@cs.pdx.edu

ABSTRACT
Data Stream Management Systems (DSMS) segment potentially unbounded data streams into windows for processing. DSMS windows typically have the following three characteristics: they are defined on a timestamp or sequence-number attribute so that the window definition is independent of other data values in the stream; the window definitions are static and do not change in the course of query execution; and, finally, windows are filled with data from the stream over which the window is defined. These three aspects limit the power of traditional DSMS windows, while many applications have a need for more advanced windowing mechanisms. In this paper, we introduce *frames*: a mechanism for defining dynamic, data-dependent windows that can be filled with data from other streams or even historical tables. We describe frame specification and implementation in the NiagaraST DSMS and show how frames can support sophisticated windows common to the needs of many applications including network monitoring and tracking vehicle traffic.

Categories and Subject Descriptors
D.3.3 [**Database Management**]: Systems – *query processing*

General Terms
Algorithms, Management, Performance, Design, Languages

Keywords
Frames, Windows, Intrinsic Windows, Data-stream management.

1. INTRODUCTION
Data Stream Management Systems (DSMS) have become commonplace in research and industry. DSMS process potentially unbounded data streams, producing results in real-time or near real-time. More recently, systems have been introduced that combine stream data with historical archives, providing the power of combining live stream data with historical archived data. The goal of DSMS is to give users "current" information about dynamic, unbounded data streams and, in the case of stream-archive systems, to put that current information in historical context.

To provide a "current" view over a stream, DSMS use windows. Windows segment the unbounded stream into finite subsets, which can be processed, aggregated over, selected on, and joined using variations on traditional DBMS operators. However, traditional DSMS windows have limitations that restrict their usefulness in certain applications. We see a need for new, more flexible and dynamic windowing mechanisms.

Consider a network-monitoring application for analyzing the causes of poor router performance. The input data stream for this application consists of `loss_rate` reports, which are issued from routers on a regular schedule. The application first detects extended periods of time during which a router has a high drop percentage (say, > 0.3%). The application then uses the selected time periods to probe auxiliary data to shed light on the reason for the high loss. For example, we may discover that a high-loss period coincides with a large number of routing-table updates (BGP messages), which can be expensive to process. Consider supporting this application with traditional DSMS window functionality. Windows are typically defined over a timestamp attribute, based on a time range (window length) and an update frequency (slide). Such windows are less than ideal for this network-monitoring application for several reasons:

- They are defined in terms of a timestamp attribute; however the interval of interest (high-loss period) in this application depends on the loss rate reported by the router – a non-timestamp attribute in the stream.
- Windows are filled with data from the stream over which the window is defined. However, to analyze the high-loss periods, we want to fill the window with data from an auxiliary stream.
- The window definitions are static and do not change in the course of query execution. While the number of tuples per window may vary, the length of windows and their rate of production are fixed. In contrast, in the desired scenario above, windows adapt to patterns in the data: a window will only be produced if the high-loss condition occurs and the window will extend to the end of the high-loss period.

In this paper, we introduce *frames*, a mechanism for dynamic, data-dependent windows. As the name suggests, a frame defines only the boundaries of a window, that is, a start time and an end time. Frames segment data streams in an intrinsic, data-dependent way, based on the values in the data stream (e.g., `loss_rate` in the example above), and can be filled with data from any stream (including the stream over which the frame is defined) or even historical data. Frames provide a sophisticated windowing mechanism that can support advanced monitoring applications and other scenarios that we describe later in the paper.

We have implemented a version of frames in NiagaraST [11], via a Frame operator. In the implementation, we define frames as a period of time, with a given minimum duration, during which a predicate is satisfied. This definition satisfies the needs of the router-monitoring scenario described above. The implementation supports filling frames from an auxiliary stream via a "Between-Join", handles disordered and missing data, and can take advantage of a regular data-delivery schedule for a stream, if such a schedule exists.

In this paper, we begin by discussing frame benefits, including differences between intrinsic and extrinsic segmentation, and then describe example scenarios that can benefit from frames. We present a general definition of frames, discuss our initial implementation of a Frame operator in the NiagaraST system – along with describing the kinds of frames it can handle – and follow with performance results.

2. Frame Benefits

As discussed in the introduction, windowing is ubiquitous in stream systems. Windowing was developed to deal with stateful and blocking operators, but current windowing mechanisms are limited in their ability to adapt to changing data streams and support segmentation that fits naturally to the particular phenomenon a user wants to examine. We discuss benefits of intrinsic segmentation, and the flexibility in filling frame in this section, followed in the next section by examples of applications that benefit from the intrinsic segmentation provided by frames.

Frames support *intrinsic* segmentation – the division of a stream based on some property of the data (thus segmentation will be different if data values are different). Furthermore, frames need not cover every element of a stream – they need only be reported for "interesting" segments. Traditional windowing mechanisms operate by *extrinsic* segmentation – the division of the stream is fixed in advance, and is independent of data values in the stream, other than a timestamp or sequence number. Typically, windows cover all elements of a stream and are emitted regularly, even during "uninteresting" periods. Of course, the output of a windowing operator can be sent through a filter to select out the notable window instances, however, the effort expended to construct the filtered-out windows is wasted. There are several reasons that the intrinsic segmentation provided by frames might be more beneficial than the extrinsic segmentation of normal windows:

- With intrinsic segmentation, the boundaries of segmentation adapt to the data, and might convey some information themselves, such as the duration of poor router performance.
- Windows often serve to reduce data volumes to more manageable levels. However, fixed-size windows may not maximize information per window compared to an intrinsic segmentation that adapts to stream characteristics. For example, we might want to divide a network stream into periods of roughly equal payload volume, rather than by equal time intervals or numbers of packets. (The distinction is reminiscent of that between equi-depth and equi-width histograms [10].)
- Not every segment of the stream might be interesting – windows tend to put every tuple in at least one window; in the network router monitoring example, only periods of high loss are interesting, low loss is not interesting.

In addition to the benefits listed, frames derive from intrinsic segmentation that adapts to data in the stream, providing flexibility in how frames are filled. By "filling" frames, we mean associating a set of tuples with the frame, which tuples can then be the target of operations, such as aggregation or duplicate elimination. Such filling will typically be accomplished with a "between join" relating the timestamps of filling tuples to the bounds of the frame.

- Frames may be filled from the same stream over which the frame was defined; this is most similar to windows. Note that the filling tuples need not be exactly the ones that contributed to frame detection. For example, once we have detected a high-loss period for a router, we might want to fill that frame with just the "very-high-loss" reports from the router (> 1.5%) and count them.
- Frames may be filled from another stream. This case corresponds to our router monitoring example, where the filler comes from the BGP message stream to the router.
- – Frames can be filled from stored data. For example, suppose we are defining frames using highway sensor data to select extended periods of slow traffic speed. We can fill such frames with stored data from the same time on previous days in order to understand whether this slow down is a normal occurrence or represents an unusual event.
- Alternatively, frames themselves can be stored, and filled from later-arriving data. For example, we can hold on to the high-loss frames from the router, and fill them with Netflow records that are periodically dumped from the router, to get a sense of the number of different sessions using the router during the frame period.
- Frames need not be filled at all, but might serve simply as alerts to trigger other actions, such as running network or router diagnostics.

Finally, we note that a single frame can be replicated and filled from several different sources, and that the filling tuples can be offset relative to the frame boundaries. For example, in filling a high-loss-period frame with BGP messages, we might include messages starting 30 seconds before the actual frame-start time.

3. Frame Examples

We present three examples scenarios in which frames may be useful and for which traditional DSMS windows will not suffice.

Example 1—Vehicle-Traffic Monitoring: In some cases, windowing can be seen as a form of approximation. Consider a stream of traffic speed and 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 jitteriness 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(vehicle_count) > 100 for each window. In this example, a window with a fixed time period is a compromise and while a more desired and sophisticated window definition can be provided with frames.

Example 2—Freezer Temperature: A commercial walk-in freezer contains a sensor that reports freezer temperature. One may wish to know periods of time greater than one minute during which the freezer temperature was too warm (> 32 degrees Fahrenheit). We could then correlate these periods with a stream of door open and close events to see if the door openings and closings are correlated with the high freezer temperatures.

Example 3—Fine-grained Bursts: A network-monitoring device produces reports of network usage every fifteen minutes. During certain 15-minute intervals, the reports show usage levels of only 3-4% of total capacity but unusually high packet loss. Further inspection shows the presence of packet loss occurring during short-lived (< 30 second) bursts that saturate the network. We observe that it may be hard to know in advance the right granularity for monitoring; a technique like frames, which can dynamically adapt granularity, may be better for this application.

4. Frame Definition

A frame is a dynamic, data-dependent mechanism for computing window boundaries based on data values in a stream. As its name suggests, a frame represents only the beginning and end of a window extent and not does specify the data used to "fill" the window. Typically a frame will define the start timestamp and end timestamp of a window.

At an abstract level, a *framing scheme* F is a mapping from a sequence of elements of type T to a set of frames, each of which bounds a portion of the sequence:

F: Seq(T) → {(fs: Time, fe: Time)}

Here 'fs' is for "frameStart" and 'fe' for "frameEnd". We assume each frame element has an attribute drawn from a totally ordered domain, such as timestamps or sequence numbers. For the moment, we do not constrain F. Thus it can depend in any manner on the values of elements and their arrangement.

Consider a sequence S with elements of type `<Time, Temperature>`:

`<1, 30>, <2, 31>, <3, 33>, <4,34>, <5, 30>, <6, 34>, <7, 33>, <8, 34>, <9, 35>, <10, 32>`

Some example framing schemes, along with their results on S:

F1. (locally) maximal periods where `Temperature` > 32. F1(S) = {(fs: 3, fe: 4), (fs: 6, fe: 9)}. Note that (fs: 7, fe: 9) is not in F1(S), because it is not maximal – (fs: 6, fe; 9) contains it.

F2. (globally) maximum period where `Temperature` > 32. F2(S) = {(fs: 6, fe: 9)}. Note that (fs: 3, fe: 4) is not in F2(S), since it does not have maximum duration.

F3. maximal periods where `Temperature` is increasing over at least three time units. F3(S) = {(fs: 1, fe: 4), (fs: 7, fe: 9)}.

In these particular examples, the result set will not contain overlapping frames, but in general overlap is allowed.

Not every framing scheme will be realizable in a stream setting where the sequence of elements is presented over time and we want to detect and report frames incrementally. Framing schemes F1 and F3 above can be computed incrementally — the existence of a frame and its bounds can be determined once the next element after the frame end has been seen. For framing scheme F2 however, the existence of a frame is not definite until all the input has been seen, which could be arbitrarily in the future. Hence, we could only report a result for F2 over a stream if the stream ends. Thus, we are interested in classes of framing schemes that can be evaluated incrementally over streams.

Our own prototyping efforts are based on framing schemes that arise naturally in the monitoring of network or vehicular traffic, two application areas on which we have focused. Our framing schemes are specified as a maximal interval over which some predicate holds, possibly with a minimum duration. Thus, F1 above is expressible in our specification, as would be F1' that also requires that the period must be at least 3 time units long. Because our frame specifications mandate maximality, frame instances will not overlap. For example, suppose our specification requires `loss_rate` > 0.3% for at least 5 minutes. Then we could not have two frame instances (fs: 10:13, fe: 10:20) and (fs: 10:18, fe: 10:23), because (fs: 10:13, fe: 10:23) must be a period where the `loss_rate` condition holds continuously, and it subsumes either of the other frames. However, for other styles of frame specification, such as based on event density, maximality might not exclude overlap.

5. Implementation Considerations

Many issues arise in moving from the abstract notion of framing schemes to an implementation in a DSMS.

Long Frames: Consider again the router-monitoring example, high-loss periods could extend for minutes or tens of minutes depending on the cause of the problem. It does not seem appropriate to wait until the end of the frame to notify the user; instead, the user should be notified as soon as the start and end of the frame is detected. It may make sense to periodically report the extension of the frame between those two points, but probably not every time it is extended.

Disorder: Another pragmatic issue is stream disorder. We assume that a framing scheme is relative to a logical order induced by some timestamp-like attribute. If physical delivery order is different from this logical order, we may need additional information, such as punctuation [13] or an expected delivery schedule, to know that a frame has definitely been detected.

Scheduled vs. Non-scheduled Data: Data in some streams arrives on a regular schedule. For example, router `loss_rate` reports are sent at specified intervals and the highway-sensor reports arrive every 20 seconds. In contrast, a freezer sensor may report temperature only on temperature change and trades in a stock market ticker have no schedule. If the stream has a reporting schedule, frames can be detected based on the presence of a set of expected tuples; if there is no reporting schedule, one must wait for punctuation to determine frame existence.

Missing Data: Missing data can be detected if there is a reporting schedule. (If not, missing data might not be detected or may require a sophisticated algorithm for detection.) Missing data must be addressed in the frame specification. For example, in the router example, should a missing data value be considered a high-loss or low-loss record? It turns out that the answer may depend on context – a missing data item in the middle of a period of high loss is likely to be a high loss record that got dropped due to network saturation; however, a missing record during a time of low loss should not be assumed to be a high-loss record. In

general, if the stream has a reporting schedule, consideration must be given to how missing data is interpreted as it will directly affect the extents of the frames. In Figure 1, tuples A and B are missing. If A is true and B is false, there are 3 and 4 true tuples in a row. If both are true there are 8 true tuples in a row.

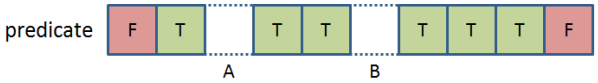

Figure 1. Tuples A and B are missing.

Groupwise Frames: So far we have discussed framing schemes being applied over the entirety of a stream. However, frames can also be defined "groupwise"—that is, the framing scheme is applied per substream defined by a group-by attribute, and frame instances are tagged with the value of that attribute. A consequence of groupwise treatment is that frames for different groups will have different start and end times; in contrast with time-based windows where the window start and end times are consistent across groups.

Data Semantics: Data arrives as tuples with timestamps, however the tuple data can be interpreted in different ways. Given a tuple *t*, representing freezer temperature with data value *d* and timestamp *ts*, how do you interpret that tuple? Does *t* represent a point-in-time event at time *ts*? Or does *t* represent an event beginning at time *ts* and continuing until the next event? Different interpretations lead to different frame boundaries and may affect the implementation. Addition of start and end timestamps to each tuple as is done in CEDR [9] may help.

Uniqueness of Time: We require that timestamps on tuples be unique. Multiple readings at one timestamp may occur for reasons including multiple sensors contributing to one logical reading or an error. In either case, preprocessing can be done—such as to aggregate together multiple sensors or to clean the data—to resolve the issue. This cleaning or aggregation should happen before the data is sent to the frame-processing operators, though the specification of those operators must be mindful of preprocessing procedures, as they can affect frame existence and duration.

Frame Extent: There is some subtlety in how the framing scheme picks the starts and ends of frames. A liberal framing scheme might apply to the freezer example, wherein we would like regions when the freezer temperature was possibly > 32 degrees, and hence the frame extends from the last sub-freezing report all the way to the next report at or below 32. In contrast, a more conservative scheme might apply to the router example, wherein we might want only regions where we know for sure that the loss rate is high.

6. Related Work

Past research on data-driven windows includes work on predicate-windows by Ghanem *et al.* [6]. 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 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 e2 with the same CORAttr value as a previous tuple e1 is assumed to be an update of e1. In this instance, a later tuple can affect the inclusion of an earlier tuple.)

Various pattern-matching methodologies have been proposed previously for stream-processing systems, including SASE+ [1], Cayuga [3] and AFAs [4]. 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 [14] and S-OLAP [5], a flavor of online analytical processing system that supports grouping and aggregation of data based on patterns [4]. Pattern-matching techniques for streams or relations could be the basis for a framing scheme, with a frame corresponding to a span of events that match the pattern. Additional conditions of maximality of matches or non-overlap might be imposed in addition. There would need to be extensions to the techniques to deal with out-of order and missing data, and to exploit reporting schedules.

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 [14], StreaMon [2] and AdaptWID [12] 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 clustering.

7. NiagaraST Implementation

As a test framework, we have implemented one version of frames in the NiagaraST [10] stream processing system. NiagaraST is a data stream processing system written in Java and developed at Portland State University. NiagaraST is based on the Niagara [14] system from the University of Wisconsin-Madison In this prototype, the frame specification consists of two parts—a predicate and a (time) duration: "Predicate P holds for duration at least D". P is a tuple-wise predicate over the stream. D can be expressed as a number of tuples or minimum time interval. The implementation supports frames such as those required for the router-monitoring example. A possible frame specification for the router example is: "`tuple.loss_rate` > 0.3% for at least 3 reports". We take the maximal period satisfying the condition. In this example, if there were 5 consecutive high-loss reports in a row, we would report it as a single frame instance (rather than three instances of length 3).

We have implemented two operators in NiagaraST to support frame processing: an Apply operator to process the predicate P and a Frame operator to interpret the predicate sequences, looking for ones that meet the duration condition D. More specifically, Apply applies P to each input tuple and appends a true or false value to it. These tuples are fed to the Frame operator, which uses the duration D to form frames.

The frame operator takes parameters to specify the minimum frame duration, a missing data predicate, and existence of reporting schedule. Frame output consists of a unique frame id, frame start and end times, and a flag to indicate if the frame is partial or final. A partial frame is an early indication that D has been satisfied and is still true. A final frame indicates that D is no

longer true and the frame start and end times are final. All data is assumed to have a timestamp attribute, but tuples are not assumed to arrive in order; and the input stream may or may not have a known reporting schedule. The methods for detecting a frame depend on how and when data arrives. If the data arrives on a schedule, the Frame operator takes advantage of that schedule and uses it to output frames as they are detected. If there is no reporting schedule, the operator waits for punctuation to finalize frame detection. Punctuation is also used to deal with disorder.

Missing data is handled with two conditions. First, we require that any frame begin with a known value that satisfies the predicate P; that is, a frame cannot begin with a missing value. Second, the frame specification specifies whether missing data values should be considered as satisfying the predicate P or not. We plan to extend this simple specification to allow slightly more complex patterns. For example, in the router example, missing values should be assumed to satisfy the predicate (that is, be high-loss records) if they are in the midst of other high-loss records and should be assumed to not satisfy the predicate in other cases.

To handle the "Long Frame" problem mentioned in Section 5, we support breaking a frame into a sequence of *fragments*. We emit an initial fragment when were a certain a new frame instance will occur: we have seen a sequence of tuples that satisfy P of at least duration D, and we know (via a reporting schedule or punctuation) that this sequence cannot merge with an earlier frame nor be divided by a non-satisfying tuple. As additional tuples arrive that extend the sequence, the Frame operator will periodically output incremental fragments, with a final fragment once the end time of the frame is known. The different fragment types are indicated by a flag in the output tuples of Frame.

We expect that fragmenting frames may have benefits for latency and memory use when filling frames. A frame may be filled via a join with the filling stream and the resulting tuples will likely be further aggregated. For example, high-loss frames in the router example can be joined on time with a BGP message stream, and then passed to an aggregate to perform a count of the different message types. For a long frame without fragmentation, the join process would not start until the end of the frame is detected; resulting in latency and the need to buffer the BGP stream at the join. In contrast, if the frame is delivered as fragments, each fragment can be joined with the BGP stream as received, allowing the matching tuples to be purged and the result tuples to be forwarded to the aggregate operator for processing.

8. Evaluation

We evaluated the performance of our frame implementation in the NiagaraST system over various sizes of input data and with various frame specifications selected to vary the density and length of frames. We also compared our frame implementation with a frame-like window query that produces similar results and evaluated the latency impacts of various frame fragment sizes.

All experiments were run on a 3.00 GHz Intel Core2 Duo 3.00 GHz Processor with 4GB of memory, running Windows 7 Enterprise. We used Java Runtime Environment Version 6 (1.6.0_21), with the –server option, and with 1.5 GB memory allocated to the JVM. For all queries, we use a data set of traffic records. The data is from approximately 600 sensors on the Portland-Vancouver metropolitan area freeways that report traffic speed and count every 20 seconds.

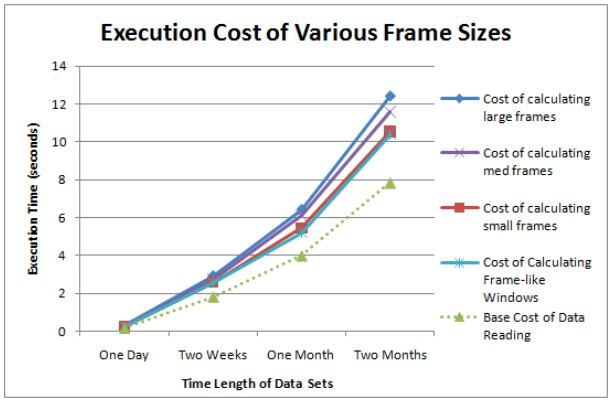

Figure 2. Experiment 1: Frame performance over various data sets and frame specifications.

For the first experiment, shown in Figure 2 above, we ran three frame specifications over four data sets – ranging from 1 day of traffic data (4320 reports) to two months of data (267840 reports). We measured performance of frame length, short, medium and long in order to test different frame densities. By density of frames, we refer to the percent of tuples in a frame, not the density of the tuples within the frames themselves. Figure 3 lists the frame specifications as well as the average frame length and percentage of tuples in frames for the 2 month data set. All specifications have minimum duration of 5 minutes. Figure 2 shows the performance of the framing queries for these specifications over the various data sets. The base cost of scanning the data file is indicated as a dotted line. As the size of the dataset increases the cost of computing frames also increases. These results are based on our initial implementation and we believe the cost can be decreased by optimizing the algorithm.

Frame Specifications	# of Frames	Avg Length	% of Tuples
Short (Speed <= 30)	302	38	0.042844
Medium (Speed >= 50)	3071	39	0.447169
Long (Speed >= 30)	866	278	0.89885

Figure 3. Frame specifications used in experiments 1 & 2.

We can create a result similar to a frame result using traditional NiagaraST windows. The basic idea is logically illustrated in three steps: 1) create sliding windows of length equal to the minimum frame duration and slide by each time step; 2) aggregate over those windows with a predicate derived from the frame predicate; and 3) finally select windows that match the frame predicate. This process creates output similar to frame; however further processing would be required to coalesce windows to create the maximal window. The cost of calculating a frame-like window is indicated in Figure 2.

For our third experiment we used a modified join query to test performance of detecting and filling frames. To fill frames, we joined the frame operator output with a traffic stream to fill in the framed time periods and aggregated vehicle speed. We used a special Between-Join that allowed punctuation to free blocked tuples to the aggregating operator and process frame fragments individually. We also tested the impact of frame fragment sizes on the latency of the results, as indicated in Figure 4 above. When

frame lengths are short and/or punctuations are frequent, the fragment size makes little difference. However, when frame fragments are long, having short punctuation can provide a substantial benefit as seen in Figure 4.

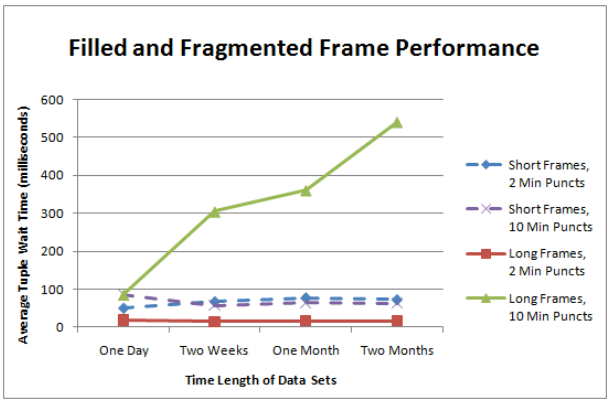

Figure 4. Frame fragment size and latency performance.

9. Future Work

We believe frames are a valuable addition to the windowing capabilities of DSMS. Their computational requirements are similar to those for fixed-period windows, but they can express data in segments and over predicates in ways not expressible by windows.

One of our main interests for future work is other styles of frame specification. In particular, frames may be defined on density criteria. For example, we might be interested in periods where vehicle count at a highway sensor exceeds 30 vehicles per minute. One issue with density-defined frames is that maximality of duration might not be the most useful choice. Consider a 2-minute period with 90 vehicles followed by a 2-minute period with 30 vehicles. Do we want to report the full 4-minute period, or would reporting just the initial 2-minute period be more informative? Scan statistics [8] may provide guidance here. Scan statistics consider the local density of events in a sequence or region, and can help calculate the likelihood that a particular density occurs by chance given an a priori global distribution. Thus, in our traffic example, it may be possible to determine whether 90 vehicles in 2 minutes is more unusual than 120 vehicles in 4 minutes and select a frame on that basis. Scan statistics might also be used directly to specify frames. We may want to report a frame of duration *d* spanning *k* tuples if the probability of such a cluster occurring by chance (assuming some distribution) is .05 or less.

A second area we have been investigating is frames with both spatial and temporal extent [16], such as stretches of highway at least 2 miles long that have reduced speeds for at least 30 minutes. One interesting issue here is that maximality does not guarantee non-overlap. There might be a 3-mile stretch with slow traffic for 30 minutes that overlaps a 2-mile stretch with slow traffic for 45 minutes. While we do not intend to prohibit overlapping frames outright, we may want some prioritization mechanism so that we do not over-report the same underlying phenomenon.

10. ACKNOWLEDGMENTS

The authors thank Ted Johnson for his router packet-loss example and early feedback on this paper and Johannes Gehrke for pointing us at scan statistics. This work is supported in part by the National Science Foundation (grant IIS-0917349) and CONACyT México (Fellowship 178258).

11. REFERENCES

[1] Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. Efficient pattern matching over event streams. In *SIGMOD 2008.* (Vancouver, Canada, June 2008).

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

[3] Brenna, L., *et al.* Cayuga: a high-performance event processing engine. In *SIGMOD 2007*. (Beijing, China, June 2007).

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

[5] 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).

[6] Ghanem, T. M., Aref, W. G., and Elmagarmid, A. K. *Exploiting predicate-window semantics over data streams.* SIGMOD Rec. 35(1). (Mar. 2006), pp. 3–8.

[7] Ghanem, T. M., Elmagarmid, A., Larson, P., and Aref, W. 2010. Supporting views in data stream management systems. *ACM Trans. Database Syst.* 35, 1 (Feb. 2010), 1-47.

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

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

[10] Ioannidis, Y.E. and Poosala, V., Balancing Histogram Optimality and Practicality for Query Result Size Estimation, *Proc. of the 1995 ACM SIGMOD Conference* , San Jose, CA, May 1995.

[11] 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*. Proc. VLDB Endow. 1(1). (2008) pp. 274–288.

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

[13] 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).

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

[15] Sadri, R., Zaniolo, C., Zarkesh, A., and Adibi, J. *Expressing and optimizing sequence queries in database systems*. ACM Trans. Database Sys. 29(2). (June 2004), pp. 282–318.

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

Frames: Intrinsic Windows over Data Streams Demo Proposal

James Whiteneck, Kristin Tufte, David Maier, Amit Bhat, Rafael J. Fernández-Moctezuma
Department of Computer Science
Portland State University
P.O. Box 751, Portland, OR
+1 (503) 725-2406

{whitenej, tufte, maier, amitb, rfernand}@cs.pdx.edu

ABSTRACT

We demonstrate a novel extension to windows, called Frames, which has been implemented in the NiagaraST data stream processing system. Frames are dynamic, data-dependent windows that define periods of interest in a data stream; these periods can be "filled" with auxiliary data so as to support correlation of the period of interest with other data sources. In contrast to traditional DSMS windows: 1) frames are dynamic and change based on values in the data stream; 2) frames can be filled with data from any source stream; and 3) frames define periods of interest in the stream, so that processing can be focused only on the "interesting" time periods. Real-world examples show that frames are useful for many applications, including sensor-monitoring applications.

1. DEMO SCENARIO

For this demonstration, we use a data stream of vehicle traffic obtained from the Oregon Department of Transportation (ODOT), which contains vehicle speed and volume information reported for Portland-Vancouver metropolitan region. The goal of the demonstration application is to help the user obtain insight into the causes of freeway traffic slowdowns. Traffic slowdowns have many causes, including recurring bottlenecks, weather, accidents, construction and special events. To understand freeway behavior, traffic managers desire to place traffic information in context of other such information. The demo leverages frames to do just that.

In the traffic example, we define periods of interest as those with slow traffic; for example, we may be interested in locations with speeds < 30 mph for at least 5 minutes. The predicate (speed < 30 mph) and duration (5 minutes) define the frame. We will fill these frames with probe-vehicle, weather, accident and historical speed information to shed light on the cause of the slowdown.

2. DEMO INTERFACE

The demo interface consists of five primary pieces: frame specification; map interface, apply operator inspector, frame operator inspector and frame filler. All will be implemented in an Internet-accessible web site.

Conference'10, Month 1–2, 2010, City, State, Country.

Frame Specification: A window will be available to specify the frame. Selectors will exist to specify the frame predicate, duration, reporting schedule existence.

Map Interface: A map interface will be provided using Google maps, similar to that shown in Figure 1, which displays sensor locations. Sensors will be colored based on whether their current speeds meet the frame criteria or not. Sensors will be clickable to reveal the Apply Insight, Frame Insight and Frame Filler windows for that location.

Apply Inspector: The Apply Inspector window shows the operation of the Apply operator, which applies the frame predicate to tuples. Tuple data will be shown in this window colored by predicate satisfaction or not.

Frame Inspector: The Frame Inspector window demonstrates the operation of the Frame operator. The current frame length, missing data (if reporting schedule exists), and frame fragments will all be shown. A horizontally expanding line will be used to show frame size; if a reporting schedule exists, missing data and present data will be shown in a grid and frame fragments will be graphically displayed.

Frame Filler: The Frame Filler window will show the auxiliary data associated with the frame – probe vehicle data (travel time), weather data (rainy, clear), accident information (nearby accidents that may affect traffic), and finally historical information to understand if current conditions are "normal."

This demo will use the NiagaraST frame implementation to associate data with traffic speed data, providing insight into causes of current traffic slowdowns.

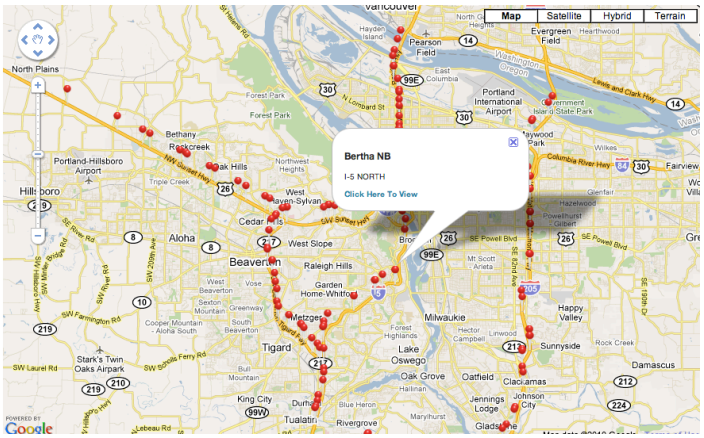

Figure 2 Traffic Locations Map - Image Courtesy PORTAL (http://portal.its.pdx.edu