Data-driven Windows: The Theory and Practice of Frames

Michael Grossniklaus, David Maier, Sharmadha Moorty, Kristin Tufte[*]
Computer Science Department
Portland State University
Portland, OR 97201, USA
{grossniklaus,maier,moorthy,tufte}@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.

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.

[*]Alphabetical order for the time being.

We have implemented a version of frames in NiagaraST [18], 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 [13].)

- 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. USE CASES

- Threshold Frames
  - All segments where data value $>$ (equiv $<$) some threshold for some length of time
  - Maximal span
  - No need for explicit start

- Boundary Frames
  - Frame on some boundary in the data–i.e., frame every one meter of depth

  - What is the predicate?
  - How do we do this?
  - Max/min explicit, start explicit

- Aggregate Frames
  - $sum(vol) > threshold$
  - Minimal span
  - Minimal frames overlap; do need some sort of start info if we want to eliminate some of those frames

- Delta Frames
  - $max(x)min(x) > value$
  - start/minimal/maximal all seem undefined by predicate
  - idea is to use frames to capture regions of change

- Michaels Capture Trend Changes Idea

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. THEORY OF FRAMES

To give a formal specification of frames, we first define the framing of a data stream and then specify the functions that can be used to define a framing. We also describe how a framing can be used to fill frames with tuples from a stream. Finally, we introduce example of commonly used framings and describe how they can be expressed based on the given specification.

Definition 1. A *data stream* S is defined as an infinite sequence of tuples $S = [t_1, t_2, t_3, \ldots]$. All tuples of a data stream have the same schema. One attribute of the schema, the *progressing attribute*, is distinguished as is defines the logical order of the tuples. The stream progresses in this attribute, i.e. if A is this attribute, then for any n, there is an i s.t. $t_i.A > n$. Note that while the presence of a progressing attribute implies a logical ordering, it does not require that the tuples within the stream are physically ordered.

4.1 Framing of a Data Stream

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

Definition 2. The *possible frames* $F_p(S)$ of a data stream S are given by the infinite set of intervals (or framing) $F_p(S) = \{[s_1, e_1], [s_2, e_2], [s_3, e_3], \ldots\}$, such that $\forall [s, e] \in F_p(S) : s, e \in \text{dom}(A)$, if A is the progressing attribute and $\forall i < j : s_i \leq s_j \wedge s_i = s_j \Rightarrow e_i \neq e_j$.[1] Each interval $[s, e] \in F_p(S)$ defines an extent as a set of tuples $\{t | t \in S \wedge s \leq t.A < e\}$, where A is the progressing attribute. Each interval $[s, e] \in F_p(S)$ has an *extent* that is the set of tuples $\{t | tinS \wedge s \leq t.A < e\}$, where A is the progressing attribute.

Theoretically, all set of intervals that satisfy the conditions given in the definition are valid framings of a stream. In practice, however, it is useful to further constrain framings. We distinguish *local* and *global conditions* that can be applied to restrict which intervals are contained in a framing.

A local condition p_l is a (conjunction of) predicates that can be checked individually for each interval $[s, e]$ contained in the set of possible frames F_p. We distinguish data-dependent and data-independent predicates. For *data-dependent predicates*, an expressions of the form *lhs*θ*rhs* has to hold for every tuple t in extent$([s, e])$, where θ is a comparison operator, *rhs* is a constant, and *lhs* is a sub-expression built from arithmetic operators, aggregates, and universal quantification. The predicate $t.X > c$ for instance restricts the framing to intervals where the value of attribute X of all contained tuples t is consistently larger than a constant c.

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

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

set cardinality. The two conditions are used in a conjunction to form the local condition p_l in the following definition.

Definition 3. 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., $F_c(S) = \{[s,e] | [s,e] \in F_p(S) \wedge p_l([s,e])\}$.

Global conditions p_g apply to all intervals in the set of candidate frames F_c, rather than to an individual interval. As a global condition, we can require that all intervals are either *minimal* or *maximal*. For example, a set of candidate frames obtained by the threshold predicate above is not guaranteed to report a unique set of intervals as for every interval that satisfies the predicate all sub-intervals down to individual tuples could be reported. In this case, requiring the framing to be maximal ensures that only non-overlapping intervals of maximal length are reported.

Another global condition that guides the selection of candidate frames is whether a set of intervals is *saturated* or *drained*. A set of frames is saturated if it satisfies all conditions and there is no candidate frame that can be added to it without violating a condition (set maximality). A final framing is drained if it satisfies all conditions and there is no frame that can be removed from it without violating a condition (set minimality).

Definition 4. The *final frames* $F_f(S)$ of a data stream S are the candidate frames for which all global conditions $p_g^1, \ldots, p_g^n$ are true, i.e., $F_f(S) = \{[s,e] | [s,e] \in F_c(S) \wedge [s,e] \in F_c(S) \Leftrightarrow \bigwedge_{1 \le i \le n} p_g^i(F_f(S))\}$.

4.2 Properties of Framings

Depending on the local and global conditions chosen for a framing, we can observe different framing schemes that describe how a stream is segmented. Figure 1 summarizes important framing schemes.

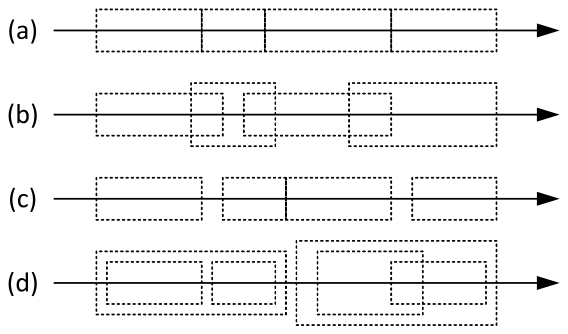


Figure 1: Different framing schemes

Partition. Each interval $[s_i, e_i]$ begins where the previous frame $[s_{i-1}, e_{i-1}]$ ended, i.e., $s_i = e_{i-1}$. Note that this segmentation of the stream is similar to tumbling windows. However, intervals still vary in length, whereas windows are of fixed length, either in terms of time or tuples.

Cover. $\forall a \in \text{dom}(A) : \exists [s_i, e_i] \in F : s_i < a \le e_i$, where A is the progressing attribute, i.e., $s_i \le e_{i-1}$. Note that this segmentation of the stream is similar to sliding windows. However, neither the slide offset nor the interval length is fixed. A special case of this segmentation is the case where intervals overlap by fixed value c, i.e., $s_i = e_{i-1} - c$. We will refer to this stream segmentation as adjacent. Another special case are advancing intervals, where we require that $b_i < b_{i+1} \wedge e_i < e_{i+1}$.

Disjoint. $\forall [s_i, e_i] \in F : \nexists [s_j, e_j] \in F : s_j \le s_i \le e_j \vee s_j \le e_i \le e_j$, i.e., $s_i \ge e_{i-1}$. Note that with this segmentation it is possible that some values $a \in \text{dom}(A)$, where A is the progressing attribute, are not contained in any intervals.

Unconstrained.

4.3 Examples

Coming back to the use cases presented in Section 3, we now describe a series of types of frames that address the requirements outlined earlier. We also discuss how each of these types can be specified within the framework introduced in this section.

Threshold Frames: This type of frame reports periods of the stream where the value of a user-defined attribute a is greater (or smaller) than a given threshold value x. As a consequence, this type of frame does not partition or cover the stream. It is defined by the frame predicate $a > x$ (or $a < x$). The start point is data dependent, i.e., a frame starts as soon as the predicate is true. The span is maximal in the sense that the frame keeps growing as long as the predicate holds true. An example of use for this type of frame is to report high-loss periods.

Boundary Frames: Boundary frames segment the stream whenever the value of a user-specified attribute crosses a (multiple of) a given boundary x. This type of frames partitions the stream. The frame predicate for the n^{th} frame is given by $a < nx$, whereas the data-dependent starting point is defined by $a > (n-1)x$. The span is minimal. Note that if attribute a is a progressing attribute, this equivalent to windows.

Delta Frames: This type of frame monitors a user-specified attribute a of the tuples in the data stream. A frame is emitted whenever the delta between the minimum and maximum value of this attribute becomes greater than a predefined value x. This type of frame partitions the stream. The frame predicate can therefore be given as $max(a) - min(a) < x$. The frame starting point is the end point of the previous frame and the span of the frames is maximal.

Aggregate Frames: This type of frames monitor a predicate over an aggregation of an attribute. Aggregate frames segment the stream. The frame predicate is given by $f_{\text{aggr}}(a)\ \theta\ x$, where f_{aggr} is an aggregation function and θ is a comparison operator. The start point of aggregate frames is the end point of the previous frame and its span can either be minimal or maximal, depending on the desired reporting scheme. An example of this type of frame is segmenting on the maximum sum of dye mass.

4.3.1 *Notes*

Threshold and Boundary Frames

- the union of overlapping candidate frames is a candidate frame

Boundary Frames

- any tuple is in some candidate frame

- full coverage comes from the fact that any single tuple frame satisfies the predicate

Delta and Aggregate Frames

Below *minimal* is used for predicates that require the delta or aggregate to be greater than a constant, whereas *maximal* is used when the delta or aggregate needs to be smaller than a constant.

- a minimal/maximal frame from a starting point T_1 cannot properly contain a minimal/maximal frame from starting point T_2, where $T_1 > T_2$
- without minimal duration, these framings have coverage using a drained framing of minimal/maximal frames (note that this framing is unique as the first frame needs to be picked which then defines the starting point of the next frame, etc.)

4.4 Framing Function

A framing as defined in the previous section is a function that returns a sequence of frame intervals for a given data stream. However, in order to process the data stream as it progresses, we require that this function can be computed incrementally, based on the previously computed frame intervals and the next tuple of the stream.

Definition 5. A *framing function* $f(F_{i-1}, C_{i-1}, t_i) \rightarrow (F_i, C_i)$ applies the i^{th} tuple to the current framing F_{i-1} and returns a new framing F_i. The framing F_i is an extension of the framing F_{i-1}, i.e., $F_{i-1} \subseteq F_i$. C_{i-1} and C_i denote the respective state before and after application of f for the i^{th} tuple.

C manages both the internal state of the framing function and the list of frames that are currently being processed, i.e., the open frames. For every arriving tuple, the framing function needs to decide whether (a) to open a new frame, (b) update an existing frame, or (c) close and emit a frame. In the latter case, the frame is removed from state C and inserted into the framing F. Depending on the framing scheme, all of these actions can be applied once or multiple times. In order to control the behavior of a framing function, we have identified the following frame properties.

We conclude this subsection by outlining an example template of a framing function that generates partitioning frames.

```
PARTITION($F_{in}, C_{in}, t, p$)
1  $f_{old} \leftarrow [s_{old}, e_{old}] | [s_{old}, e_{old}] \in C_{in}$
2  if $p(t, C_{in})$
3     then $F_{out} \leftarrow F_{in}$
4           $f_{new} \leftarrow [s_{old}, ts(t)]$
5     else $F_{out} \leftarrow F_{in} \cup \{f_{old}\}$
6           $f_{new} \leftarrow [ts(t), ts(t)]$
7  $C_{out} \leftarrow C_{in} \backslash f_{old} \cup \{f_{new}\}$
8  return $(F_{out}, C_{out})$
```

The function template begins by extracting the current frame from the state C. Note that there is only one open frame in the state as this template follows the partition scheme. On line 2, the predicate is evaluated. If it still holds, the current frame is extended (lines 3 and 4). If not, the current frame is emitted (line 5) and a new frame is opened (line 6). Finally, the state is updated to reflect the updated or new frame (line 7). The function $ts(\cdot)$ is used to return the value of the progressing attribute of a tuple.

4.5 Filling Frames

Once a framing of a stream has been defined in terms of a sequence of intervals, it can be used to fill frames. A framing that has computed based on one stream can be used to produce frames of that stream or any other stream. A frame filling $\hat{F}$ is given as a set of tuple sequences $\hat{F} = \{\langle t_1, t_2, \ldots t_i\rangle, \ldots \langle t_{n-j}, \ldots t_{n-1}, t_n\rangle\}$.

Definition 6. A *frame filling* is defined by a function $f : (S' \times g(F) \times Q) \rightarrow \hat{F}$, where S' is the data stream furnishing the tuples, F is the framing, and Q is a query.

The function $g(\cdot)$ allows intervals in the framing to be modified. For example, the function could be used to extend or reduce the lengths of intervals by a fixed amount. In particular, it can be used to produce adjacent frames. In contrast, the query Q defines a subset of the tuples that fall into the interval of the frame, or performs aggregation.

5. IMPLEMENTATION

6. EVALUATION

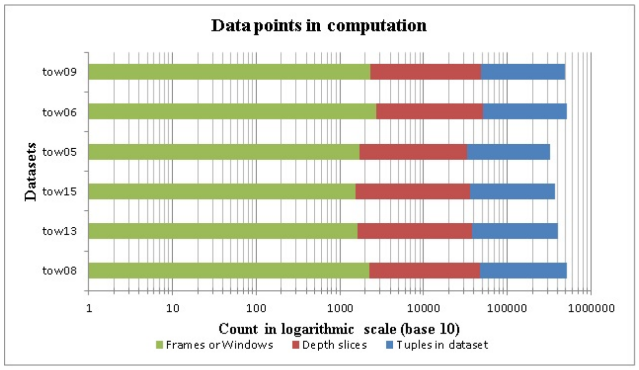


Figure 3: Data point in computation

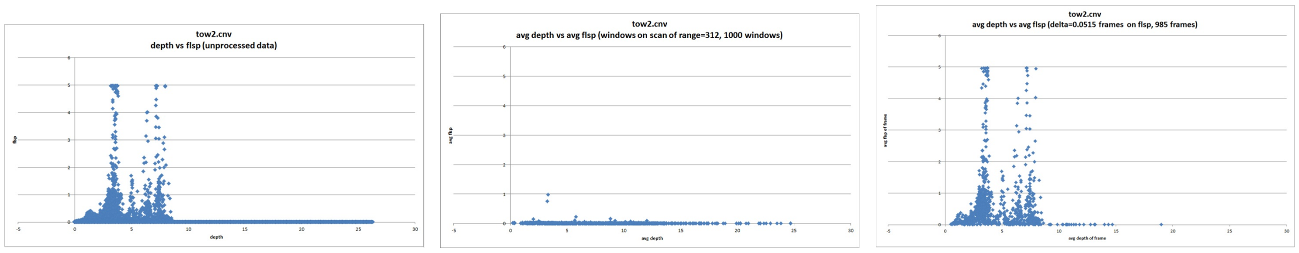
7. RELATED WORK

8. CONCLUSIONS

9. 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 Universitys "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.



(a) Unprocessed Data (b) Windowed Data (c) Framed Data

Figure 2: Comparison of original versus windowed and framed data sets

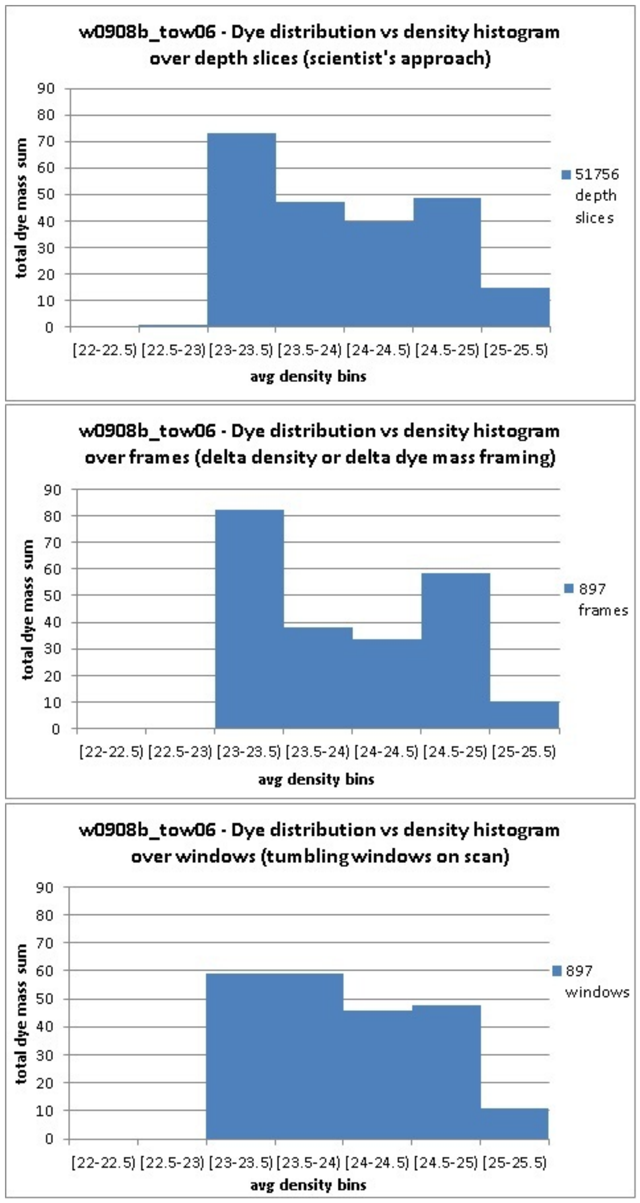


Figure 4: Histogram tow06

10. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient Pattern Matching over Event Streams. In *Proc. Intl. Conference on Management of Data (SIGMOD)*, pages 147–160, 2008.

[3] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[4] S. Babu and J. Widom. StreaMon: An Adaptive Engine for Stream Query Processing. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 931–932, 2004.

[5] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: A High-Performance Event Processing Engine. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 1100–1102, 2007.

[6] B. Chandramouli, J. Goldstein, and D. Maier. High-Performance Dynamic Pattern Matching over Disordered Streams. *Proc. VLDB Endow.*, 3(1–2):220–231, 2010.

[7] C. K. Chui, B. Kao, E. Lo, and D. Cheung. S-OLAP: An OLAP System for Analyzing Sequence Data. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 1131–1134, 2010.

[8] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 647–651, 2003.

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

[10] T. M. Ghanem, A. K. Elmagarmid, P.-A. Larson, and W. G. Aref. Supporting Views in Data Stream Management Systems. *ACM Trans. Database Syst.*, 35(1):1:1–1:47, 2008.

[11] J. Glaz, V. Pozdnyakov, and S. Wallenstein, editors. *Scan Statistics: Methods and Applications*. Birkhäuser Boston, 2nd edition, 2009.

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

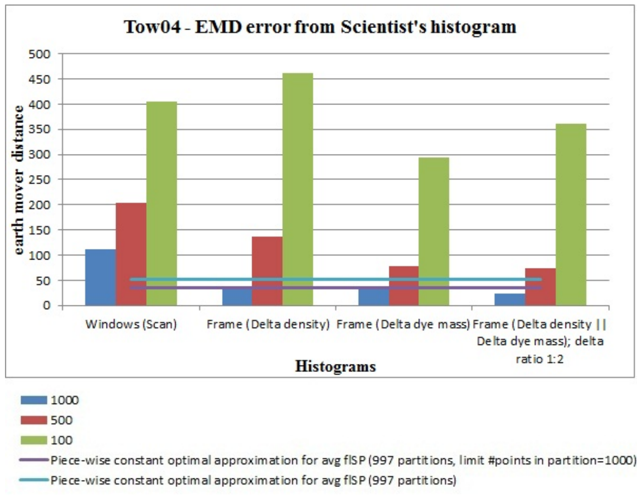


Figure 6: EMD error from Scientist's histogram (tow04)

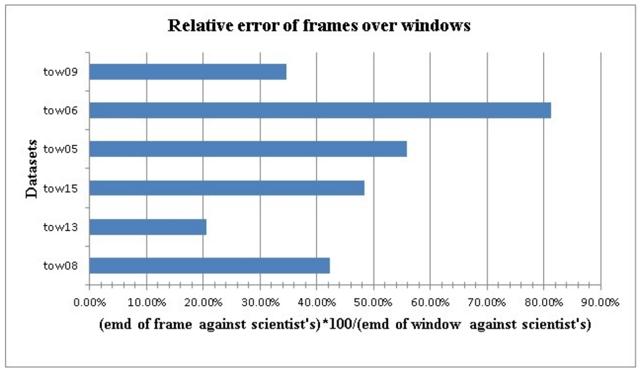
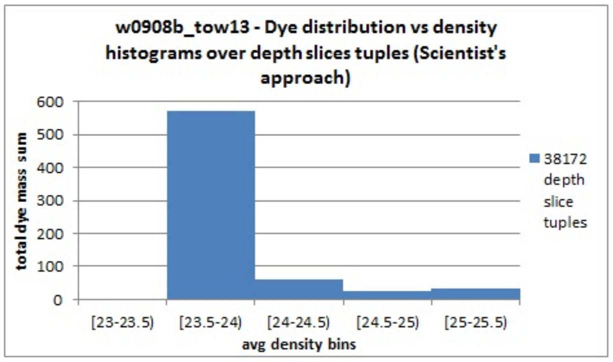
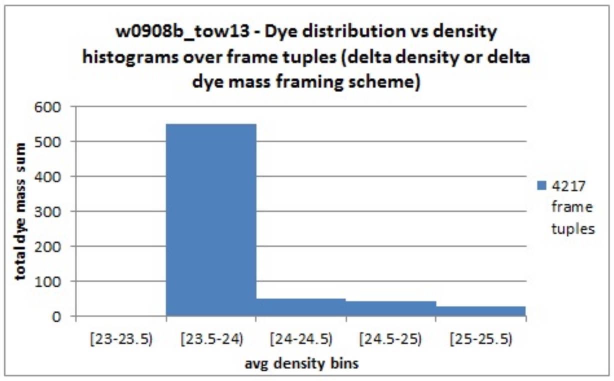
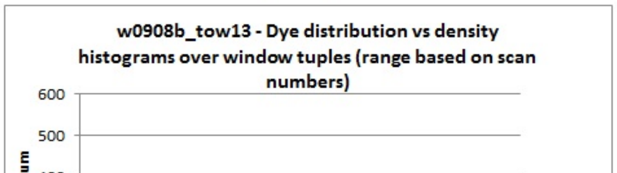


Figure 7: Relative error of frames over windows







Available at http://research.microsoft.com/pubs/70517/tr-2007-158.pdf.

[13] Y. E. Ioannidis and V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *Proc. Intl. Conference on Management of Data (SIGMOD)*, pages 233–244, 1995.

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

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

[16] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. Intl. Conference on Management of Data (SIGMOD)*, pages 311–322, 2005.

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

[18] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-Order Processing:

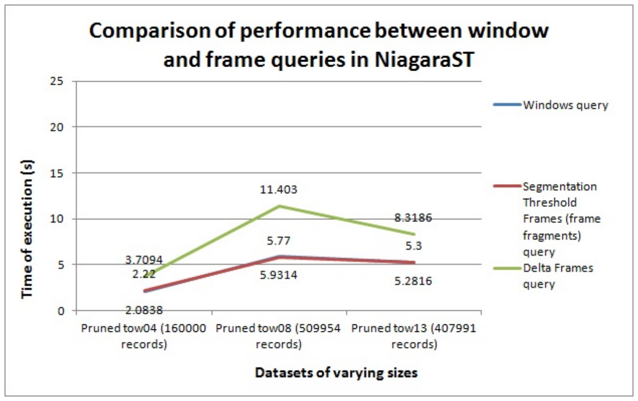


Figure 8: Comparison of performance between window and frame queries in NiagaraST

A New Architecture for High-Performance Stream Systems. *Proc. VLDB Endow.*, 1(1):274–288, 2008.

[19] S. McReynolds. Complex event processing in the real world. Oracle White Paper, September 2007. Available at `http://www.oracle.com/us/technologies/soa/oracle-complex-event-processing-066421.pdf`.

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

[21] J. Naughton, D. Dewitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, and S. Viglas. The Niagara Internet Query System. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[22] D. Preston and P. Protopapas. Searching for events in time series using scan statistics (poster). Initiative in Innovative Computing Open House, Harvard Univesity, 2008. Available at `http://iic.seas.harvard.edu/posters/scanstats-poster.pdf`.

[23] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity. In *Proc. Intl. Conf. on Very large Data Bases (VLDB)*, pages 1353–1356, 2004.

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

[25] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):555–568, 2003.

[26] J. Whiteneck, K. Tufte, A. Bhat, D. Maier, and R. J. Fernández-Moctezuma. Framing the Question: Detecting and Filling Spatial-Temporal Windows. In *Proc. Intl. Workshop on GeoStreaming (IWGS)*, pages 19–22, 2010.

[27] F. Zemke, A. Witkowski, M. Cherniak, and L. Colby. 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`.

APPENDIX

A. CUTTING ROOM FLOOR

Note that some of the local properties imply global properties. For example, requiring intervals to be maximal at the local level will always lead to a disjoint framing scheme as overlapping intervals can be combined into a single interval. Similarly, certain constraints on the starting points of interval will also directly yield a certain framing scheme. For example, to specify an adjacent segmentation, the starting point condition would be defined as $s_i = e_{i-1}$, while the condition $s_i = e_{i-1} + 1$ yields a partitioning of the stream. As a consequence, care has to be taken in order not to specify conflicting conditions.

Furthermore, we allowing multiple possible interval starting points using a complex expression together with a non-overlapping framing scheme, the situation may arise where a starting point occurs before the ending point of the preceding interval. We use policies to indicate which condition will be violated by the framing if such a situation arises. There are three possibilities.

1. Close and report the preceding interval: This policy will lead to a framing that possibly violates the local condition defined by the predicate.
2. Drop and suppress the previous interval: This policy possibly leads to false negatives in terms of the local condition defined by the predicate. It will also possibly violate the global condition of a covering (or partitioning) framing scheme.

Continue the previous interval: This policy does not violate any of the local conditions, but possibly violates a requirement to generate a non-overlapping framing scheme.

B. FRAME DEFINITION (DAVE/CIDR)

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:

$$F : Seq(T) \rightarrow \{(\mathrm{fs} : Time, \mathrm{fe} : Time)\}$$

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 $\langle Time, Temperature\rangle$:

$$\langle 1,30\rangle, \langle 2,31\rangle, \langle 3,33\rangle, \langle 4,34\rangle, \langle 5,30\rangle,$$
$$\langle 6,34\rangle, \langle 7,33\rangle, \langle 8,34\rangle, \langle 9,35\rangle, \langle 10,32\rangle$$

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

F_1. (locally) maximal periods where Temperature > 32. $F_1(S) = \{(\mathrm{fs} : 3, \mathrm{fe} : 4), (\mathrm{fs} : 6, \mathrm{fe} : 9)\}$. Note that $(\mathrm{fs} : 7, \mathrm{fe} : 9)$ is not in $F_1(S)$, because it is not maximal—$(\mathrm{fs} : 6, \mathrm{fe}; 9)$ contains it.

F_2. (globally) maximum period where Temperature > 32. $F_2(S) = \{(\mathrm{fs} : 6, \mathrm{fe} : 9)\}$. Note that $(\mathrm{fs} : 3, \mathrm{fe} : 4)$ is not in $F_2(S)$, since it does not have maximum duration.

F_3. maximal periods where Temperature is increasing over at least three time units. $F_3(S) = \{(\mathrm{fs} : 1, \mathrm{fe} : 4), (\mathrm{fs} : 7, \mathrm{fe} : 9)\}$.

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 F_1 and F_3 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 F_2 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 F_2 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, F_1 above is expressible in our specification, as would be F_1' 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 $(\text{fs} : 10 : 13, \text{fe} : 10 : 20)$ and $(\text{fs} : 10 : 18, \text{fe} : 10 : 23)$, because $(\text{fs} : 10 : 13, \text{fe} : 10 : 23)$ 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.

C. IMPLEMENTATION CONSIDERATIONS (CIDR)

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 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 [25] 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 9, 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.

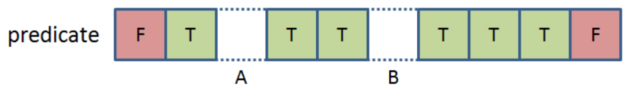


Figure 9: Tuples A and B are missing

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 [12] 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 donesuch as to aggregate together multiple sensors or to clean the datato 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.

D. RELATED WORK (CIDR)

E. SEC:RELATEDWORK-CIDR

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 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.)

Various pattern-matching methodologies have been proposed previously for stream-processing systems, including SASE+ [2], Cayuga [5] and AFAs [6]. 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 [24] and S-OLAP, a flavor of online analytical processing system that supports grouping and aggregation of data based on patterns [7]. 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 [23], StreaMon [4] and AdaptWID [17] 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.

F. NIAGARAST IMPLEMENTATION (CIDR)

As a test framework, we have implemented one version of frames in the NiagaraST [16] 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 [21] 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 used to deal with disordered data.

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 **??**, 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.

G. EVALUATION (CIDR)

We evaluated the performance of our frame implementation in the NiagaraST system over various sizes of input data and with various frame specifications selected specifications 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 and running Java Runtime Environment Version 6 (1.6.0_21), with the `server` option, and with 1.5 GB allocated memory 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.

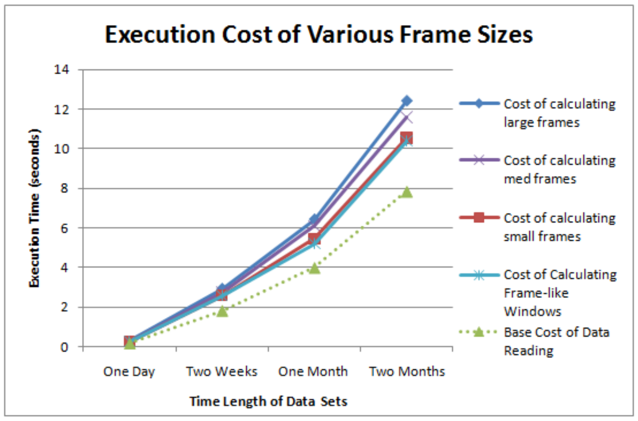


Figure 10: Experiment 1: Frame performance over various data sets and frame specifications

For the first experiment, shown in Figure 10, 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. Note that by density of frames, we are referring to the total % of tuples in all frames, not the density of the tuples within the frames themselves. Table 1 lists the frame specifications as well as the average frame length and percentage of tuples in frames, a measure of frame density. Figure 10 shows the performance of the framing queries for these specifications over the various data sets. The base cost of reading the data is indicated as a dotted line. As the size of the dataset increases the cost of computing frames also increases. This is based on our initial implementation and we believe the cost can be brought down by optimizing the algorithm.

Table 1: Frame Specifications used in Experiments 1 & 2

Frame Specification	# of Frames	Avg Frames	# of Tuples
Short (Speed $\leq$ 30)	302	38	0.042 84
Medium (Speed $\geq$ 50)	3071	39	0.447 19
Long (Speed $\geq$ 30)	866	278	0.898 85

We can create a result similar to a frame result using traditional NiagaraST windows. The basic idea is logically illustrated in three steps: first, create sliding windows of length equal to the minimum frame duration and slide by each time step; second, aggregate over those windows with a predicate derived from the frame predicate; and finally select windows that match the frame predicate. While this process will not create the same output as frames, it does create a similar result. Further processing is required to coalesce windows to create the maximal window as opposed to producing many smaller windows. The cost of calculating a frame-like window is indicated in Figure 10.

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 11 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 11.

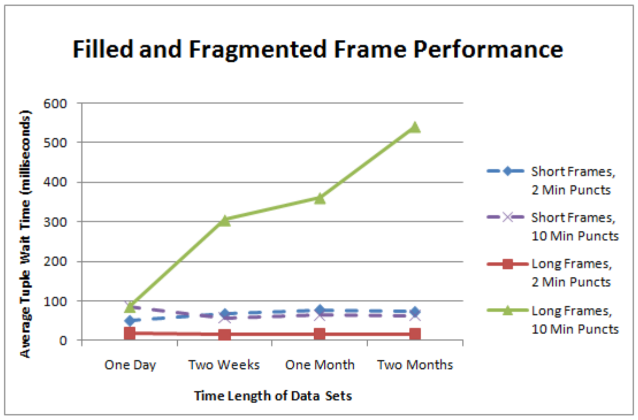


Figure 11: Frame Fragment Size and Latency Performance

H. FUTURE WORK (CIDR)

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 [11] 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 [26], 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.