Data-driven Windows: The Theory and Practice of Frames

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

ABSTRACT
Frames are our friends.

1. INTRODUCTION
Consider Figure 1(a). It shows a scatterplot of a stream of data from an ocean sensor showing intensity of florescence of dye being tracked versus depth. The plot shows more than a half-million stream elements. Figure 1(b) shows average values for these quantities for roughly 3600 *frames*. These frames resemble tumbling windows over the input stream, except rather than extending for fixed intervals, each ends when there is a difference of more that 0.02 intensity units in the florescence. [Need to check the delta.] Figure 1(c) shows roughly the same number of (uniformly sampled?) points from the original data set. Figure 1(d) shows the average values over about 3600 fixed-interval windows.

Just judging by eye, Figure 1(b) gives a much more faithful rendition of the full data stream than either Figure 1(c) or (d). About the only divergence between Figure 1(b) and the original is near the maximum depth, where the values are sparse. (In this region there was essentially no dye, so frames tend to be quite long.) Samples and windows, on the other hand, give a fairly poor depiction of the data stream. While the difference between frames and the other two methods seems self-evident, trying to *quantify* the goodness of representation that the eye detects is an interesting challenge. If we want to score how well the values in one of Figures 1(b)-(d) match the distribution of points in the original stream, we might consider for reach point p in Figure 1(a), how far is it to the nearest point q in the collection in question, and sum up this "error" over all possible p. Surprisingly, this measure does not show a large difference among the three approaches. [Give scores here?] The reason is that the vast majority of points in Figure 1(a) lie near the baseline – only small regions of the ocean actually had any dye in this study. But clearly there are features of the data stream in Figure 1(a) that are not captured well by Figure 1(c) or 1(d).

If one considers what a scientist might want to determine from such a scatterplot, it could be more a notion of "spread" than distribution. That is, he or she might be more concerned with what combinations of dye and depth are showing up than the number of points with each combination. One means to score this aspect is to superimpose a fairly fine grid over each scatterplot, and label a cell with a 1 if there is a stream element in it, and label it 0 otherwise. The divergence, then, between the original stream and any of the other representations is then measured by the number of cells where the label differs. We call this measure the *spread difference*, to contrast it with the distribution difference. This measure seems to correspond better with our visual impression, with the spread difference between plots in Figure 1(a) and 1(b) being much smaller (val) than that for either Figure 1(c) or 1(d) (val and val, respectively).

One of the contributions of our work … [TBC]

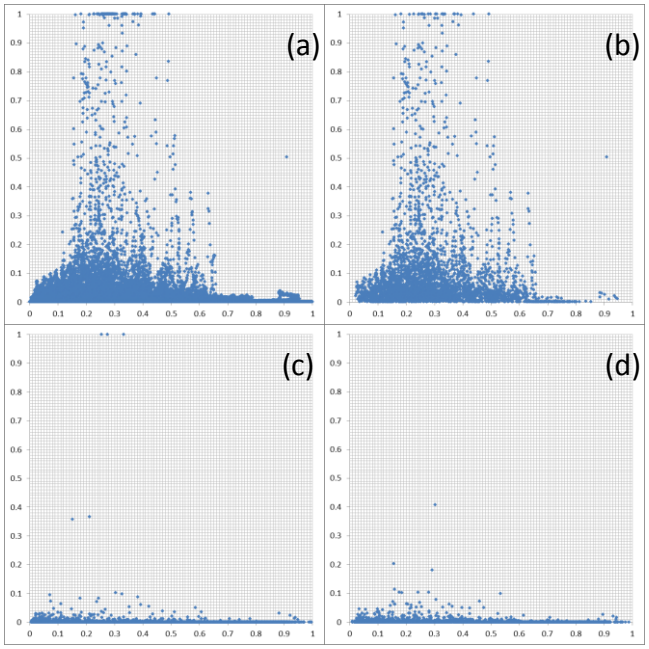

Figure 1. Florescence (y-axes) versus Depth (x-axes) scatterplots, on normalized scales: (a) original data, ~510K points; (b) ~3600 frames; (c) regular sample of ~3600 points; (d) ~3600 equal-sized windows.

2. USE CASES

3. FRAME SPECIFICATION
To give a formal specification of frames, we first define the framing of a data stream and then specify the functions that can be used to define a framing. We also describe how a framing can be used to fill frames with tuples from a stream. Finally, we

introduce example of commonly used framings and describe how they can be expressed based on the given specification.

Definition 1: A *data stream* S is defined as an infinite sequence of tuples $S = [t_1, t_2, t_3, \dots]$. All tuples of a data stream have the same schema. 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.

3.1 Framing of a Data Stream

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

Definition 2: The *possible frames* $F_p(S)$ of a data stream S are given by the infinite set of intervals (or framing) $F_p(S) = \{[s_1, e_1], [s_2, e_2], [s_3, e_3], \dots\}$, such that $\forall [s, e] \in F_p(S) : s, e \in dom(A)$, 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] For an interval $[s, e] \in F_p(S)$, we define an $extent([s, e])$ as the set of tuples $\{t | t \in S \wedge s \leq t.A < e\}$, 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 individually checked for each interval extent contained in the set of possible frames F_p. We distinguish data-dependent and data-independent predicates. For *data-dependent predicates*, expressions of the form $lhs\, \theta\, rhs$ can be used, where θ is a comparison operator, rhs is a constant, and lhs is a sub-expression built from arithmetic operators, aggregates, and universal quantification. The predicate $\mathrm{t.X} > \mathrm{c}$ 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 $|extent([s.e])| \geq n$ (or $|extent([s, e])| \leq n$), where $|\cdot|$ denotes set cardinality. The two conditions are used in 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.

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

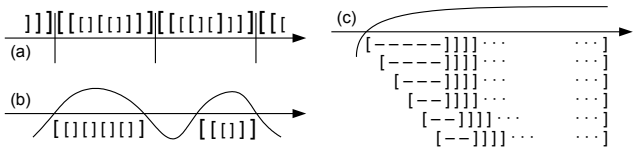

Figure 1: Boundary (a), threshold (b), and delta/aggregate (c) frames.

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 $p_g^1, \dots, p_g^n$ are true, i.e., $F_f(S) = \{[s, e] | [s, e] \in F_c(S) \Leftrightarrow \wedge_{1 \leq i \leq n} p_g([s, e])\}$.

3.2 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 2 summarizes important framing schemes.

a) **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.

b) **Cover.** $\forall\, a \in dom(A) : \exists\, [s_i, e_i] \in F : s_i < a \leq e_i$, where A is the progressing attribute, i.e. $s_i \leq e_{i-1}$. 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 $b_i < b_{i+1} \wedge e_i < e_{i+1}$.

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

d) **Unconstrained.**

3.3 Examples

Coming back to the use cases presented in Section 2, 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

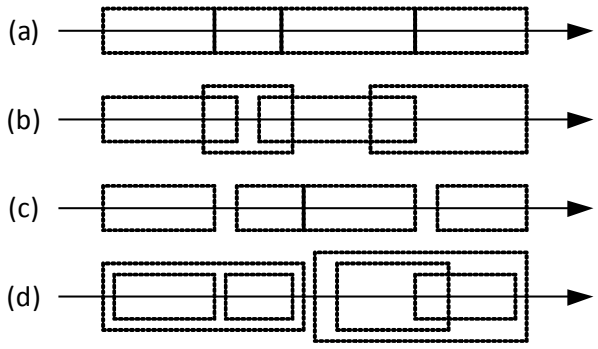

Figure 2: Different types of framings.

by the frame predicate $a > x$ (or $a < x$) as a local condition. Global conditions that apply to this type of frame are maximal and drained to restrict the final frames to only include contiguous episodes of maximal length. 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: any tuple is contained in some candidate frame and full coverage is given by the fact that any single tuple frame satisfies the predicate. The local condition is given by frame predicate for the n[th] frame as $(n-1)x < a \leq nx$. Global conditions maximal and drained can be used in conjunction with this type of frame. 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. The frame predicate is therefore given by $\max(a) - \min(a)\ \theta\ x$ as the local condition, θ is a comparison operator. Depending on the comparison operator, these framings are typically required to be minimal or maximal at the global level. For example, if the delta is required to be smaller than the constant x, it can make sense to only report maximal frames, whereas minimal frames are useful if the delta is required to be larger than the constant x. Depending on whether the framing is set to be drained or saturated, this type of frame produces non-overlapping and overlapping sets of intervals.

Aggregate Frames: This type of frames monitor a predicate over an aggregation of an attribute and reports a new frame if the aggregate value becomes greater than a given constant. The frame predicate is therefore given by $f_{aggr}(a)\ \theta\ x$, where f_{aggr} is an aggregation function and θ is a comparison operator. In terms of global conditions, the same observations as for delta frames apply to aggregate frames. An example of this type of frame is segmenting on the sum of dye mass.

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

Finally, Figure 1(c) summarizes the case of delta and aggregate frames for predicates in which the delta or aggregate is specified to be larger than a given value. As can be seen, a frame can start with any arriving tuple. In contrast to the previous framing schemes, frames can only end at tuples that delimit a region for which the predicate is satisfied. Once that tuple is encountered, any super-interval is typically also a candidate frame, as shown in the figure.

3.4 "Lemmas and Proofs"

Definition 5: A framing has *union closure*, if the union of overlapping candidate frames is also a candidate frame. Note that this is the case for threshold and boundary frames. A framing has *super-interval closure*, if any frame that (fully) contains a candidate frame is also a candidate frame. Vice-versa, it has *sub-interval closure*, if any frame that is fully contained in a candidate frame is also a candidate frame.

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

Proof (by contradiction): If we assume that the framing is not unique then there are at least two different framings. Without loss of generality, we assume that these two framings differ for the interval $[i,j]$, which satisfies the local conditions. The first framing contains the intervals $[i,k]$ and $[k+1,j]$, while the second contains $[i,l]$ and $[l+1,j]$, where $k \neq l$. However, this is in contradiction to the lemma as these two framings are not drained and maximal. Because of candidate frame coverage, both framings can be drained further by substituting $[i,j]$ for the two intervals in each framing. ■

Lemma 2: If a framing has super-interval closure (or sub-interval closure) and the set of final frames is drained, minimal (or maximal), and has coverage of the candidate frames, then the final framing is unique.

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

3.5 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 6: 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$(F_{in}, C_{in}, t, p) \rightarrow (F_{out}, C_{out})$
1: $f_{old} \leftarrow [s_{old}, e_{old}] \mid [s_{old}, e_{old}] \in C_{in}$
2: **IF** $p(t, S_{in})$ **THEN**
3: $F_{out} \leftarrow F_{in}$
4: $f_{new} \leftarrow [s_{old}, ts(t)]$
5: **ELSE**
6: $F_{out} \leftarrow F_{in} \cup \{f_{old}\}$
7: $f_{new} \leftarrow [ts(t), ts(t)]$
8: **END**
9: $C_{out} \leftarrow C_{in} \backslash f_{old} \cup \{f_{new}\}$
10: **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 6) and a new frame is opened (line 7). Finally, the state is updated to reflect the updated or new frame (line 9). The function $ts(\cdot)$ is used to return the value of the progressing attribute of a tuple.

The above framing function can be implemented efficiently in terms of internal data structures required and the overhead to process each tuple as it arrives. The reason for this is that in order to create a partitioning framing scheme, only one open frame has to be maintained at any time. Generally, we are interested in framing schemes that can be computed incrementally by maintaining a finite number of open frames. In Figure 3, we give an example of a boundary framing scheme that needs to keep track of a maximum total of three frames. The added complexity stems from the fact that a frame is also to be reported if the maximum (or minimum) data value does not coincide with a boundary. On the left-hand side of the figure, we show the "normal" case, where the signal crosses first the lower and then the upper boundary. In this case, the full frame is reported. On the right-hand side, the signal crosses the lower boundary twice without ever crossing the upper boundary. In this case, two frames are reported, as shown in the figure. To implement this behavior, the framing function additionally needs to keep track of two additional frames that can be reported if the signal does not cross the next boundary. Note that since the number of open frames is still constant, this framing function has an efficient implementation.

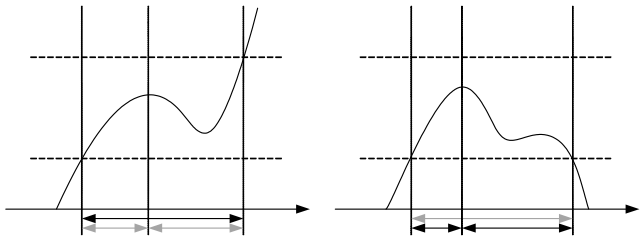

Figure 3: Boundary frames with multiple open frames.

3.6 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 7: A *frame filling* is defined by a function $f := (S', g(F), Q) \rightarrow \hat{F}$, 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.

4. IMPLEMENTATION

5. EVALUATION

We are primarily interested to evaluate frames using task-based performance metrics that quantify the value of frames as a technology to compute a specific data product. We compare the quality of frame-based data products to window-based and manually created data products. Finally, we evaluate the run-time performance overhead of frames with respect to windows.

All experiments were run in NiagaraST, using its existing window implementation and the frame implementation presented in ??. The figures reported in this section were measured on a Dell Optiplex 780 with a 3 GHz CPU and 4 GB of main memory.

The dataset used in the experiments contains measurements from dye track cruise W0908B, conducted by Oregon State University (OSU) in the Pacific Ocean off the coast near Newport from August 26 to September 2, 2009. Dye is measured with a photometer at a sub-second frequency.

5.1 Episode Detection

An important application for frames is the detection of so-called episodes in a stream. In Maier *et al.* [3], we exclusively focus on the use and the evaluation of threshold frames for this use-case. We will not repeat these results here and direct the interested reader to our previous publication.

5.2 Histogram Approximation

We evaluate the use of frames to compute specific data products in the setting of dye mass vs. depth histograms as used by oceanographers at OSU. As a baseline, we compute the exact histogram manually using the same technique as the oceanographers. We then compare this baseline to approximations based on frames and windows. Each of these histograms is computed based on a different technique to segment the data. To reduce noise, the average dye mass of each segment is calculated and multiplied by the Δdepth of the segment. Finally, this value is summed to the corresponding density bin of the histogram.

The *oceanographer's histogram* (baseline) segments the dye dataset into predefined depth slices of a fixed size. The *windows-based approximation* segments the dye dataset based on its progressing attribute, i.e., the scan number into tuple-based windows of a fixed size. We compute *frame-based approximations* based on three types of delta-frames that each uses a different framing predicate. The first type of delta-frame

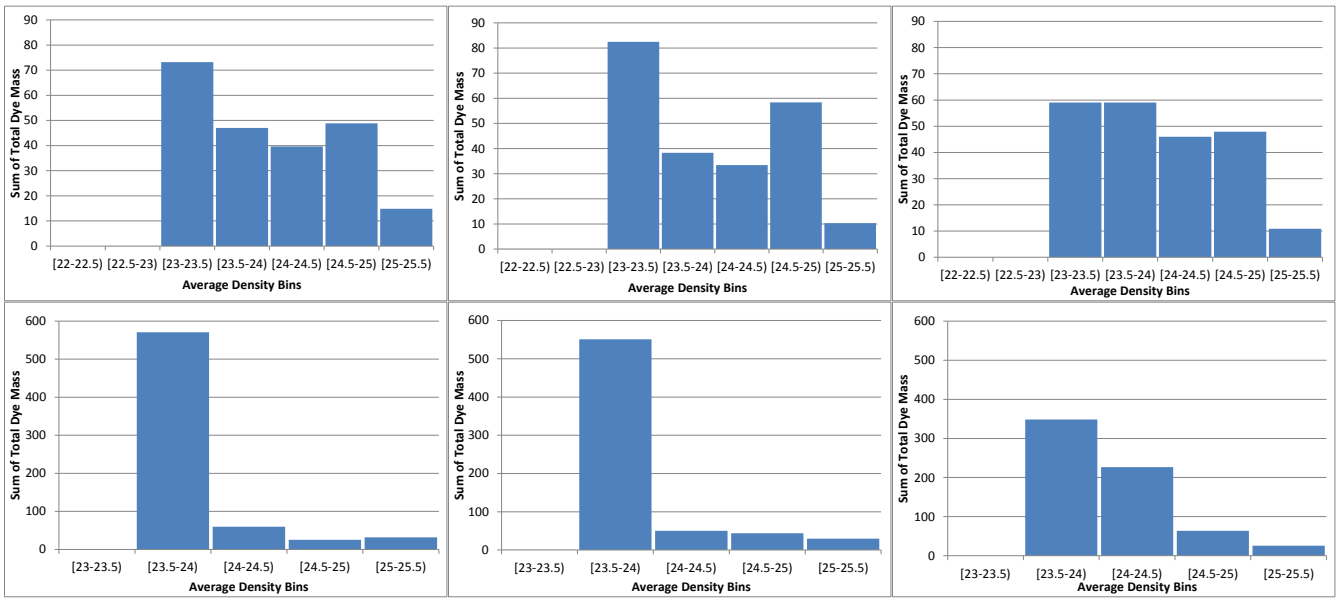

Figure 4: Baseline histogram, frame-based and window-based approximation for tow06 (top) and tow13 (bottom).

starts a new frame whenever it has seen a shift in density (of the water). The second type uses the same approach to look for shifts in dye mass. Finally, the third type of delta-frame applies both predicates and starts a frame if either of them is true. In the last case, we have fixed the ratio between the two deltas to 1:2. Finally, we use dynamic programming to compute the *pair-wise constant optimal approximation* of the oceanographer's histogram in terms of partition boundaries, given a predefined number of partitions.

As a task-based metric to compare the histograms we compute, we use the so-called *earth-mover distance* (EMD). The earth-mover distance measures how many data units have been assigned to a wrong bin in the histogram in terms of how "far" they would have to be moved to be in the correct bin.

In Figure 4, we show the oceanographer's baseline histogram (left) together with the frame-based (center) and window-based (right) approximation. The top line of histograms is based on tow06 from the dye dataset, whereas the bottom line visualizes tow13 from the same dataset. We have chosen these two tows as the relative task-based error of frames with respect to windows is worst on tow06 and best on tow13.

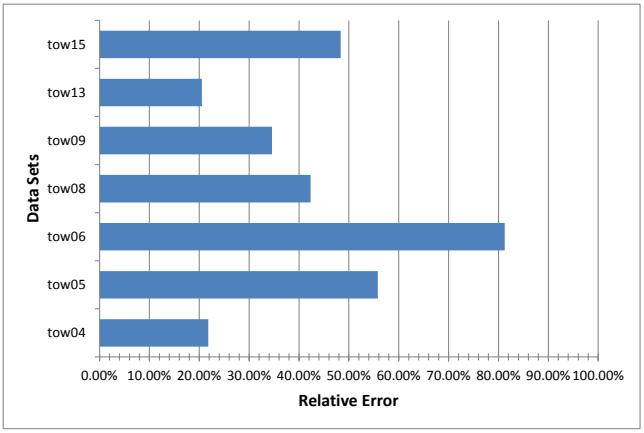

Figure 6: Relative error of windows and frames.

The tow06 data stream consists of a total of 519,998 tuples, which are segmented into 51,756 depth slices for the baseline histogram capturing a total of 225.6 dye units. The frame-based approximation is computed based on 897 frames and has earth-mover distance 19.3 with respect to the baseline histogram. The window-based approximation is computed using 897 windows but has earth-mover distance 23.7.

The tow13 data stream contains 407,992 tuples in total, which require 38,172 depth slices for the baseline histogram capturing a total dye mass of 687.7 units. The frame-based approximation uses 4,217 frames and yields an error of earth-mover distance 60.5. Similarly, the window-based approximation needs 4,207 windows, but has earth-mover distance 294.6, i.e., almost 45% of the total dye mass.

Figure 6 plots the relative error of windows over frames for all six two data streams of the dye data set. The relative error is calculated as $1 - EMD_{frame}/EMD_{window}$.

In Figure 5, we compare the earth-mover distance of histograms computed based on three different window sizes (100, 500, and 1000 tuples) to four different delta framings, which were tuned

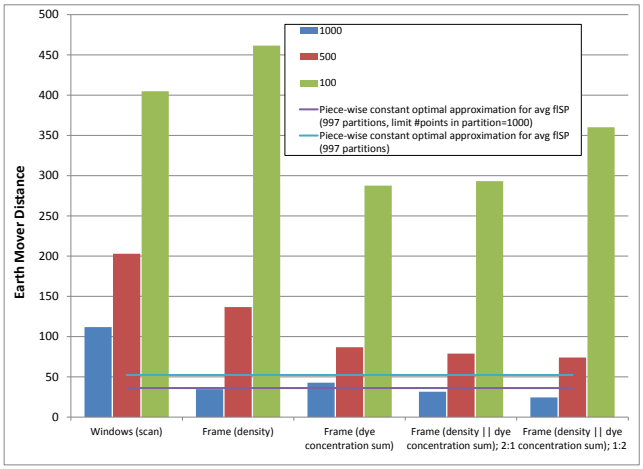

Figure 5: EMD for different frame and window sizes (tow04).

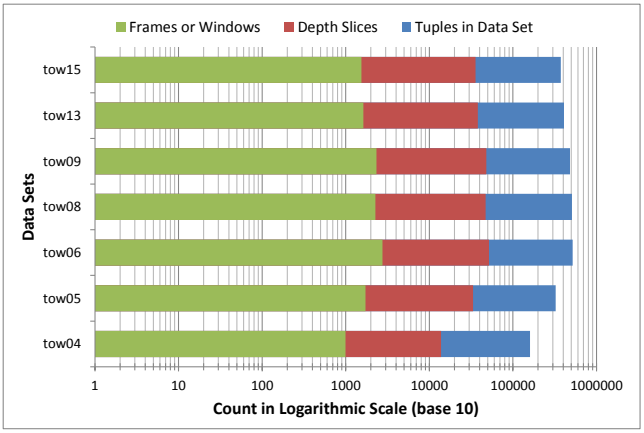

Figure 7: Data points in computation.

(by adjusting the delta constant) to yield the same (or a very close) number of frames. The first framing is segmented using dye density deltas, whereas the second frames on the delta of the sum of dye concentration. The third and the forth framing use a conjunction of the previous delta predicates, with the delta of the two predicates adjusted to 1:2 in the former case and 2:1 in the latter case. As a reference, we have also included the piece-wise constant optimal approximation that uses dynamic programming to find the boundaries of the partitions. We show the values for (about) 1000 partitions once with and once without a limit of 1000 points per partition. We note that the two framings that use a conjunction of predicate seem to perform "super-optimally". This is explained by the fact that the dynamic programming algorithm only considers average fluorescence (density), whereas these framings also factor in dye mass.

One goal of the approaches using frames or windows is to reduce the number of data points in the computation. In Figure 7, we quantify this reduction by comparing the number of data points in the original data sets, to the number of depth slides, and to the number of frames or windows. As can be seen, the use of frames or windows reduces the number of data points that need to be processed by an order of magnitude with respect to the manual approach for all tow data streams.

I'm not sure Figure 8 has anything to do with the histogram experiments. However, Sharmadha included a line-plot based on this data in the collection of figures for the paper. I suspect this is a much more general comparison of windows and different types of frames. Nevertheless, it would be good to have something at this point (or later) demonstrating that the performance of frames is reasonable.

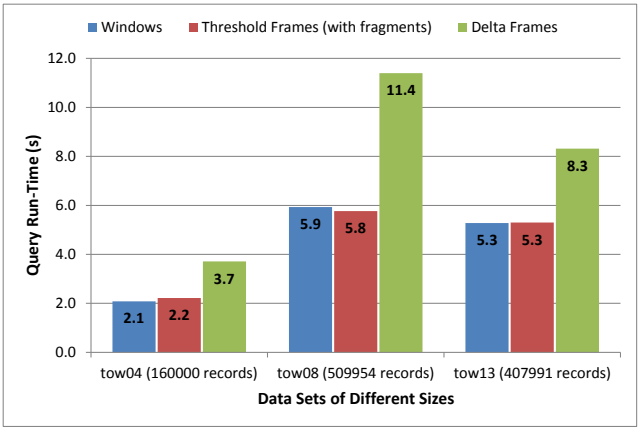

Figure 8: Comparison of run-time performance.

5.3 Dataset Summarization

6. RELATED WORK

Dave?

The proposal to extend XQuery with windows by Botan *et al.* [1] shares some similarities with frames. As frames, the proposed windows for XQuery are not tuple or time-based, but are specified using a mandatory start and end condition as well as an optional where predicate, which have the same effect as the local conditions in our approach. Additionally, the approach distinguishes tumbling, sliding, and landmark windows to control the behavior of the system. This is comparable to our notion of global conditions. We believe, however, that our separation of concerns into local and global conditions is conceptually superior as with XQuery windows there can be interference between window types and start conditions. Finally, the proposed windows for XQuery do not take disorder or very long windows into consideration. In our work on frames, we have addressed both issues with punctuation and frame fragments, respectively.

7. CONCLUSION

8. ACKNOWLEDGMENTS

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

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

9. REFERENCES

[1] I. Botan, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending XQuery with Window Functions. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 75-86, 2007.

[2] P. M. Fischer, A. Garg, and K. S. Esmaili. Extending XQuery with a Pattern Matching Facility. In *Proc. Intl. XML Database Symposium (XSym)*, pages 48-57, 2010.

[3] D. Maier, M. Grossniklaus, S. Moorthy, and K. Tufte. Capturing Episodes: May the Frame Be with You (Invited Keynote). In *Proc. Intl. Conf. on Distributed Event-Based Systems (DEBS)*, pages 1-12, 2012.

10. 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.

a) **Close and report the preceding interval:** This policy will lead to a framing that possibly violates the local condition defined by the predicate.

b) **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.

Columns on Last Page Should Be Made As Close As Possible to Equal Length