Sharmadha Moorthy
Department of Computer Science
Portland State University
Spring 2012

CS506 Report – Framing Systems II

Abstract

The project revisited the implementation of *thresholdframe* and *fillframe* operator in NiagaraST. More aggressive punctuation has been implemented in the *thresholdframe* operator to improve the performance. Threshold plus duration (T+D) frames capture periods of interest (episodes) over a data stream. They are defined such that a predicate condition on one or more attributes holds for a minimum duration. Experiments were performed on dye tracer and traffic datasets to evaluate the performance of threshold plus duration frames.

Implementation in NiagaraST

We have implemented *thresholdframe* and *fillframe* operator to support T+D frames in NiagaraST.

ThresholdFrame operator

The parsed input stream and the names of group-by and timestamp attributes are passed in as arguments to the operator. Group-by attribute is used to identify the substream over which group-wise frames are processed. The timestamp attribute is used to mark the start and end of a frame. Before being passed into the *thresholdframe*, the input stream is processed by the *apply* operator. *apply* computes the predicate condition on one or more attributes of the tuples in the input stream and appends an *appliedattribute* to each tuple. The minimum duration and segmentationflag parameters are also input as an argument to *thresholdframe.* When the segmentationflag is set, *segmentation threshold frames* are generated. This is discussed at the end of this section.

Frames are processed over the input stream only when *punctuation* is received. Meanwhile, the (timestamp, appliedattribute) pairs of the input tuples are buffered in a hash map keyed on the group-by attribute. If the punctuation has value for both group-by and timestamp attribute, the processing is done for the list of tuples hashed on the group-by value. If it has value only for the timestamp attribute, it is performed for all the groups. (This feature currently works for only one group. For the correct implementation, we would have to facilitate generation of separate set of output frames and punctuations for each group.) At punctuation, the list of tuples are sorted on the timestamp attribute and we process frames over tuples whose timestamp is less than or equal to the punctuation timestamp. We maintain state for the frame start and end time and the frame size of the current frame, if any.

We start a frame when the predicate value of a tuple is true and the current frame size is zero. While the predicate values of consecutive tuples are true, the frame end time is updated. When the predicate value of a following tuple is false, we check whether the frame size satisfies the minimum duration condition. If yes, a frame tuple of the form (frameid, group-by value, start time, end time) is generated. The current frame state is reset and the remaining tuples are processed. Thus at each punctuation, a list of new frames are output. If the input punctuation timestamp is greater than the end time of the last frame, the punctuation is passed on. Otherwise, a punctuation set to the end time of the last frame is output. (At the close of a stream, potential frame candidate and punctuation is output without any check for the minimum duration. If no frame is detected, no punctuation is output. As a result of this, in the fillframe operator, the last set of tuples can be held up until the close of the operator.)

Support for frame fragments are also implemented in *thresholdframe*. When the last frame is not closed and if the frame satisfies the minimum duration, a frame tuple is generated with the current frame end time and a long frame flag is set. At the next punctuation, if the frame is continued, the frame id used in the last punctuation is reused. A punctuation with timestamp set to the frame fragment end time is output. On the other hand, if the last frame does not satisfy the minimum duration, we do not generate any frame fragment and a punctuation with timestamp set to the current frame start time is output.

Segmentation threshold frames partition the input stream into on and off frames depending on whether the tuples in each frame satisfy the predicate and minimum duration condition. (The current implementation only works for minimum duration=1. For values greater than 1, it is easier to create the on frames and deduce the off frames from them.). The on/off frames are processed the same way as threshold frames. Frame fragments are also implemented to handle long on/off frames.

FillFrame operator

The stream of (frameid, group-by value, start time, end time) *(frame stream)* output from the *thresholdframe* operator and a parsed input stream to fill the frames *(filling stream)* is input to the operator. In addition to this, the names of the group-by and frame start and end time attributes from the frame stream, and the names of the group-by and timestamp attributes from the filling stream is passed as parameters. (The current implementation supports only non-overlapped frames.)

We maintain a hash map of (group-id, list of tuples) and a hash map of (group-id, list of frame tuples) to buffer tuples from the filling and the frame stream respectively. As the frame tuples are generated by the system, it is safe to assume they arrive in order (on the frame start time attribute). The objective of the operator is to tag the tuples from the filling stream with the appropriate frame-id and output the tagged tuples.

When a tuple T arrives from the filling stream with timestamp, t and group-id G, we search the list of frame tuples corresponding to G for a matching frame (fid, fstime, fetime). A matching frame is one such that t falls within fstime and fetime. We perform a modified binary search through the list to locate the matching frame. T is then tagged with fid and output. If a matching frame is not found or if there is no entry for G in the hash map, we buffer T.

When a tuple F (fid, fstime, fetime) with group-id G arrives from the frame stream, we output all matching tuples T' with timestamp t from the list of tuples corresponding to G. Matching tuples are those such that t fall within fstime and fetime. And then F is buffered in the list of frame tuples.

When a punctuation arrives from the filling stream, we purge frames from the list of frame tuple buffer whose frame endtime is less than or equal to the punctuation timestamp. If the punctuation has value for both group-by and timestamp attribute, the processing is done for the list of tuples hashed on the group-by value. If it has value only for the timestamp attribute, it is performed for all the groups. (This feature currently works for only one group. For the correct implementation, we would have to facilitate generation of separate set of output frames and punctuations for each group.) We also pass on punctuation with the frame id of the last frame purged from the buffer.

When a punctuation arrives from the frame stream, we purge tuples in the list of tuples buffer whose timestamp is less than or equal to the punctuation timestamp. This is because we don't expect any matching frames for these tuples.

Experimental Evaluation

In the first set of experiments, we investigate how windows approximate a set of frames over a data stream. We define two error metrics for this purpose: duration error and existence error. All the data points that are mismatched between the set of frames and windows, that is, data points included exclusively either in the set frames or in the set of windows account for duration error. This includes error due to mismatch in the duration of frames and windows. Existence error comprises of two components: false positives and false negatives. While false positives count the cases when windows signal an episode when there really was none (no corresponding frame), false negatives count the cases when an episode was missed by a window.

We first look into the results of traffic dataset measured at detector 1129 recorded in January 2012. We perform threshold frames where the average speed of reports is within 40mph for at least 3 minutes. We generate windows of ranges varying from 1 to 16 minutes. We apply 3 different selections on the windows: those qualified on minimum speed, average speed and maximum speed within 40mph. Figure 1 shows the duration error graph for the traffic dataset. The total frame duration is represented by the black line. While the minimum and average heuristic yields duration error greater than the total frame duration, the maximum heuristic gives a small duration error.

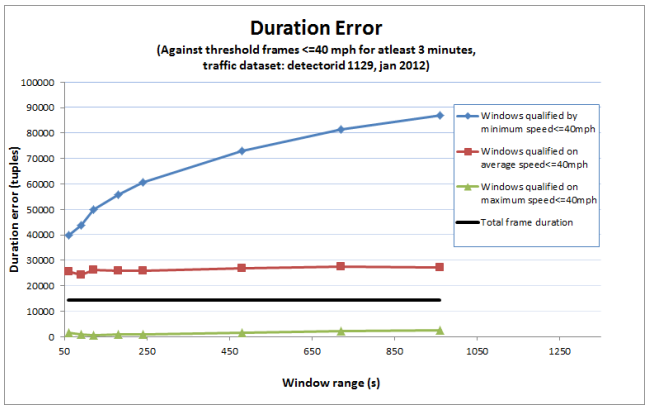

Figure 1

Figure 2 gives the false positives as a percentage of the total qualifying windows. The minimum heuristic results in up to 90% false positives while that in maximum heuristic tapers to 0% after the window range equals the minimum duration of the frame.

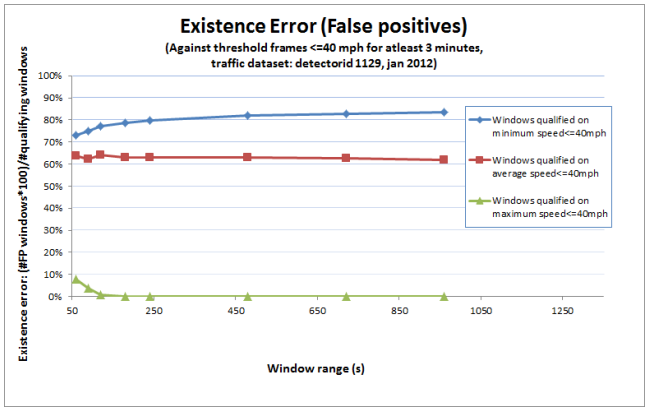

Figure 2

Figure 3 gives the false negatives as a percentage of the total number of frames. The minimum heuristic is conservative and hence yields no false negatives. On the other hand, the maximum heuristic is liberal and reports up to 70% false negatives.

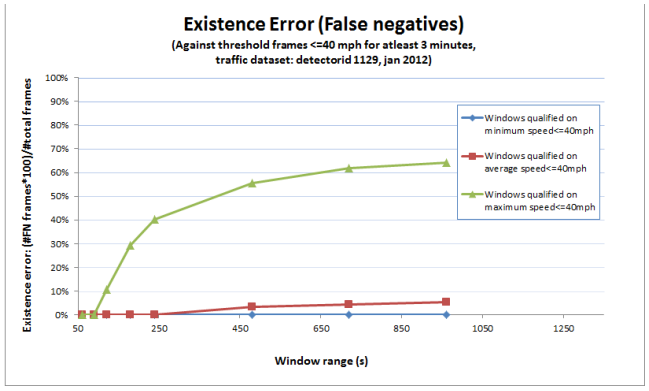

Figure 3

From the results on duration and existence error, it is evident that none of the selection of windows approximates frames consistently. The only candidate emerges to be the maximum heuristic on windows with very small window ranges. Figure 4 gives the query execution time on the traffic data set for the frame and window queries. The execution time for windows of small ranges is about four times that of the frame query.

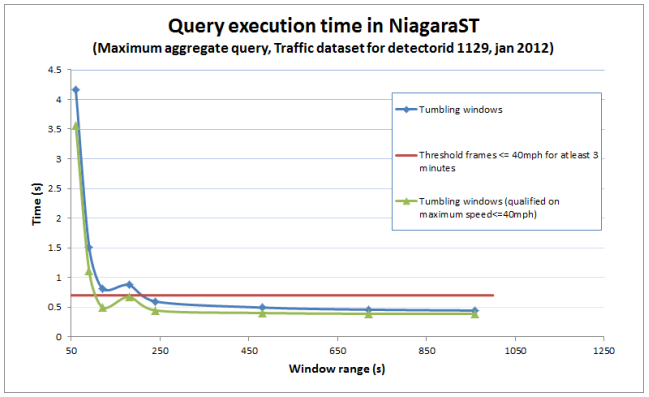

Figure 4

The error and performance experiments were repeated for dye dataset measured using tow4 and tow13 from dye tracer experiments conducted by Oregon State University on August 2009. For the dye datasets, threshold frames greater than 0.2 dye fluorescence for at least 20 tuples are computed. Here, the frame is formed when the value of the attribute is above the threshold which is in contrast to the situation for framing in the traffic dataset. Hence the roles of the minimum and maximum heuristic are switched. The maximum heuristic has greater duration error than the total frame duration (represented by the black line) and the minimum heuristic has small duration error (Figure 5).

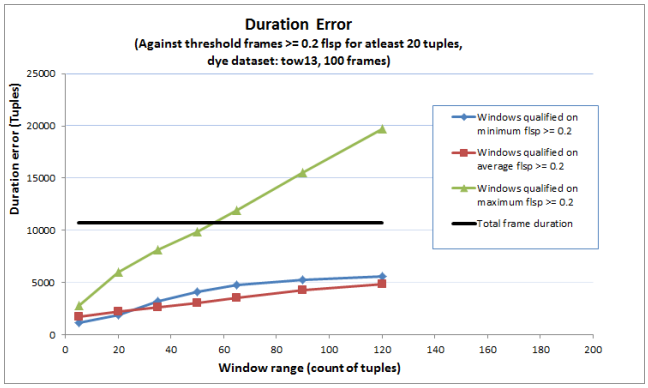

Figure 5

The minimum heuristic has 0% false positives (measured as a percentage over the total number of qualifying windows) for window ranges greater than or equal to the minimum duration value of the frame. The maximum heuristic has up to 30% false positives (Figure 6).

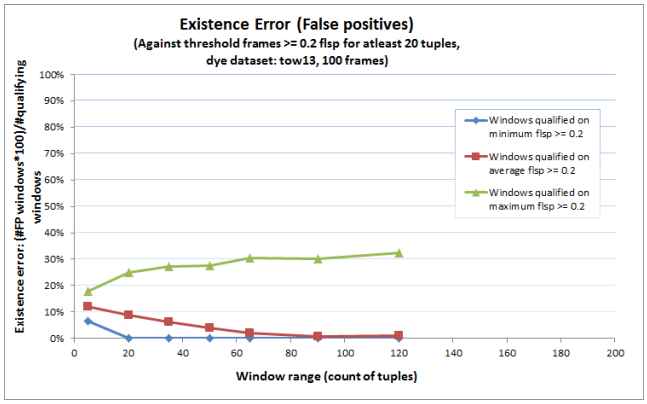

Figure 6

Figure 7 shows that the false negatives (represented as a percentage of the total number of frames) is up to 80% for the liberal minimum heuristic while the conservative maximum heuristic has 0% false negatives. Clearly the errors demonstrate that none of the heuristics consistently approximate frames with minimal error. For the smaller window ranges, the three heuristics have minimum error but have to pay a high performance cost for the query execution time as shown in figure 8.

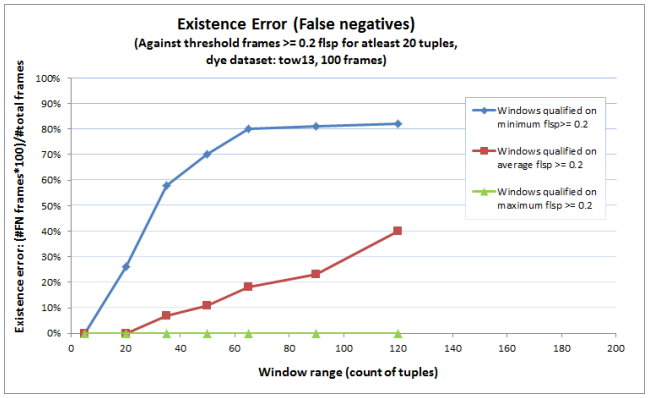

Figure 7

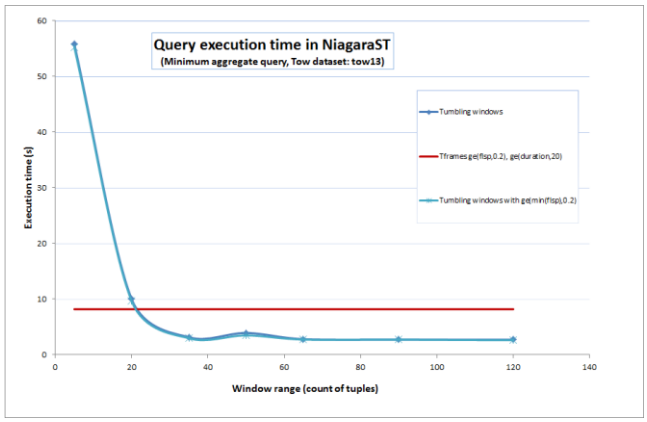

Figure 8

To understand how windows compare to frames in case of episodes without any duration constraint, let us consider the case of segmentation threshold frames of minimum duration 1 tuple. The set of on/off frames cover the entire dataset. Figure 9 shows the results for dataset tow4. As is expected, the duration error is 0 for window range of 1 tuple. But the error increases for all three heuristics for increasing window ranges. The minimum and average heuristic have duration error close to the total frame duration while maximum heuristic has duration error more than the total frame duration. Figure 10 gives the duration error numbers for the threshold frames for minimum duration of 20 tuples for tow4 dataset.

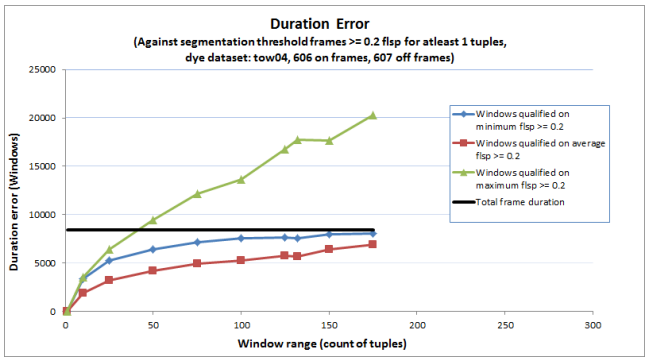

Figure 9

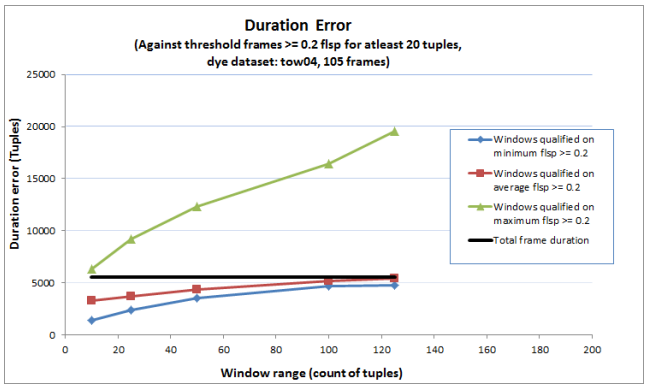

Figure 10

Existence error doesn't make sense for segmentation threshold frames as the frames cover the entire dataset. So selection of windows by applying the various heuristics is not necessary as the set of all windows is necessary to approximate the frames.

In the second set of experiments, we demonstrate the benefit of frame fragments and examine the effect on the execution times of the query. Table 1 gives the maximum and average size of the buffer which holds tuples from the filling stream in the fillframe operator. The tuples are held in the buffer until a matching frame is input or until a punctuation arrives that signal no matching frame is incoming. In the case of segmentation threshold frames for tow13, (T+D frames (T>0.2, D=1), we have a frame which covers over 150K tuples. Without fragments, the fillframe operator only knows about the frame when the frame is closed. This causes the buffer to hold the 150K tuples from the filling stream waiting for the frame. On average, the size is up to 10 times larger for the segmentation threshold frames. For the threshold frames for minimum duration of 20 tuples, the average size of the buffer is comparable while there are cases when the frame fragments reduce the buffer size to at least half the size.

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

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

When frame fragments are emitted for long frames, the buffer is traversed each time in the fillframe operator to check for matching tuples to be output. Table 2 shows that this does not add much cost to the performance of the query as the execution times are comparable and in fact for the segmentation threshold frames, frame fragments saves on execution time.

Table 2: Execution times

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

Conclusion

Frames compute the condition that defines an episode and thus capture episodes accurately. In case of long frames, frame fragments is helpful to notify the user that a frame has started and reduces the processing cost on operators downstream without adding much cost to performance. The experiments demonstrate that neither the liberal nor the conservative heuristics applied in the selection of windows consistently approximate the frames with minimal error. We can thus conclude that frames capture episodes precisely and promptly when compared to windows.

As frames only define the boundaries of the episodes, they are flexible and can be filled with data from a secondary stream or historical data. This is in contrast to windows which are filled with data from the same data stream. Hence, frames is a favorable alternative than windows to capture episodes and also enable combining episodes detected on one stream with data from a secondary or historical data.