“What’s in a Frame?”

The NiagaraST Frame Team

# Abstract

Frames are a new way of segmenting a potentially infinite stream of tuples into finite sets that can be used by the query processor to return results before the whole stream has been seen. Data windows that advance by time interval or tuple count have been previously proposed for the same purpose. In contrast to windows, frames advance based on a predicate that defines a condition required to hold over the tuples contained in the frame. Frames can be seen as “data-dependent windows” for which neither the time interval nor the tuple count that they actually span is known or specified before-hand. In this document, we examine different types of frames, describe their specification and give examples of their use.

# Preliminaries

Before specifying frames in terms of their properties and return values, we need to discuss a couple of preliminary issues.

## Tuple Timestamps

The condition associated with a frame selects a sub-stream of tuples from a stream based on the data values of these tuples. However, frames are returned in terms of the time span over which the condition was true. Therefore, the sub-stream of tuples needs to be mapped to a time interval. In this setting the interpretation of tuple timestamps in an important factor as it will influence the resulting boundaries of frames. In NiagaraST, tuples are stamped with a single timestamp value as opposed to other systems that use (open or closed) intervals. There are several possibilities how this single timestamp value can be interpreted. Furthermore, the exact interpretation is application-dependent and, therefore, our frame engine needs to support more than one in the long term. In the following examples, we present possible mappings of a tuple timestamp **i** to an interval of validity.

* **[i, i]:** The tuple is valid **exactly** at the timestamp with which it was stamped and we assume no knowledge about the period **(i, i+1)**, i.e. the open interval between the current tuple and the next (or previous).
* **(i-1, i]:** The validity of the tuple begins right after the timestamp of the previous tuple and lasts up to (and including) the timestamp of the current tuple. Note that at the implementation level, this definition raises the question what “right after a timestamp” means. Is there some smallest unit of time that can be added to the previous timestamp to define that point in time?
* **[i, i+1):** The validity of the tuple begins at its timestamp and lasts up to right before the timestamp of the next tuple. Apart from the same implementation issue as the definition before, this one implies that the validity of a tuple is only known once the next tuple has been seen.
* **[i-i, i+ i+1]:** The validity of the tuple is defined as an interval that begins somewhere before its timestamp and ends somewhere after it. This, admittedly very general definition, could be used to specify validities that begin and end exactly between two tuples, where “between” is defined as **i = ½ (i - i-1)**.

Of course, this list is not exhaustive in the sense that there might be other, more exotic, definitions. For the being, however, we limit ourselves to these four options.

## Segmenting the Stream

Frames split an unbounded stream of tuples into variable length sub-streams for which a given predicate holds. The exact way of how frames segment the stream depends on the where the next frame starts in respect to where the previous frame ended. The figure below lists possible cases.



* In case (a), the frames **partition** the stream. Every tuple of the stream belongs to **exactly one** frame and *endi-1 = starti*. Note that this segmentation of the stream is similar to tumbling windows. However, frames are still variable in length, whereas windows tend to be of fixed length.
* In case (b), the frames **cover** the stream. Every tuple belongs to **at least one** frame and *endi-1 >= starti*. Note that this segmentation of the stream is similar to sliding windows. However, neither the slide offset nor the frame length is fixed. A special case of this segmentation is the case where frames overlap by exactly one tuple. We will refer to this stream segmentation as **adjacent**.
* In case (c), the frames produce **disjoint** stream segments. Every tuple belongs to **at most one** frame and *endi-1 <= starti*.
* In case (d), the frames produce a **crazy** segmentation of the stream. Every tuple belongs to an arbitrary number of frames and there are no constraints over *endi-1* and *starti*.

## Frame Length and Slide Offset

The above argument (in the most cases) puts some constraints over the start and end points of frames. More precisely, it helps to define the earliest point where the next frame can start in respect where the previous frame ended (“Where do we start looking for the next frame?”). However, it does not contribute to controlling the length or slide offset of frames, in particular if the predicate is true for various overlapping segments of the stream. In this case, a frame specification should define exactly how frames are reported to the client. In order to approach this problem, let us consider the following cases.



Let us assume that t1 is the earliest possible start point for the frame that the system is currently computing. Further, t3 is the point where the predicate evaluates to true for the first time, yielding segment [t1, t3]. However, the predicate is also true for the segments [t2, t3] and [t2, t4]. In this setting there are four options how frames could be reported.

In case (a), the **maximal** frame [t1, t4] is reported. In case (b), the **minimal** frame [t2, t3] is reported. In case (c), the frame [t1, t3] is reported and in case (d), the frame [t2, t4] is reported. The former might make sense if new frames start where previous frames end and should be reported as soon as possible. The latter might make sense if there can be gaps between the frame and we want to report only if the predicates starts being false.

# Frame Specification

We first describe the properties of a frame that need to be specified to produce the behaviors described above. Then we talk about how frames are returned in terms of boundaries with respect to the tuples contained in them.

## Properties

* **Condition:** A logical expression that specifies the predicate that needs to hold over the tuples contained in the frame. An example of such an expression would be **max(x)-min(x) > 5** to denote that the absolute difference between the maximum and minimum values contained in a frame needs to be larger than five.
* **Frame initiation point:** A logical condition that, if true, triggers the start of a new frame. There are several ways how this condition can be expressed. First, it can be specified in terms of the boundaries of the previously detected frame. Second, it can be specified as a data-dependent condition over the tuples in the stream. Third, the start of a new frame can be triggered w.r.t. to external event such as a wall clock.
  1. **Boundaries of the previous frame:** Encompasses the possible scenarios listed above under *“Segmenting the Stream”*. For example, to specify an adjacent segmentation, the initiation point condition would be defined as **cur.start = prev.end**, while the condition **cur.start = prev.end + 1** yields a partitioning of the stream. Note, that in order to make a statement about partitioning or adjacent frames, we assume maximal frames (see below) in both examples.
  2. **Data-dependent:** Used to specify the start of a new frame based on some properties of the data, for example a local minimum or the observation of a certain value or punctuation (e.g. a “synchronization pulse”).
  3. **External Events (Wall-clock):** Triggers the start of a new frame w.r.t. to an external event such as the wall-clock. To start a frame every ten minutes (independent of the data), for example, one could use the condition **clock % 600 = 0**.

Expressions combining all of the above using **and** and **or** are also permitted. For example, to specify that frames start ever fifteen minutes or right after the previous ended, the expression **(clock % 900 = 0) or (cur.start = prev.end + 1)**. Conditions are lazily evaluated from left to right and can thus be used to define precedence of sub-conditions. In the given example, the wall-clock trigger takes precedence over the boundary condition as the latter is only applied if the former evaluates to false.

When allowing multiple possible frame initiation points, the situation may arise where a frame initiation point occurs **before** the current frame has been closed. In this case, the frame specification needs to include an indication what will happen to the current (open) frame. There are three possibilities.

1. **Close the frame:** The frame is closed prematurely and reported. Thus, the tuples contained in it might not fulfill the frame predicate and thus the frame is said to be incomplete. This option leads to a non-overlapping segmentation of the stream.
2. **Drop the frame:** The current frame is dropped and not reported. This option can lead to a non-covering segmentation of the stream.
3. **Continue the frame:** Processing on the current frame continues as usual and the frame is reported when the predicate holds. This option might lead to overlapping frames.

* **Span:** A constant value that describes how frames are reported. In correspondence with the above example under *“Frame Length and Slide Offset”*, permitted values are **maximal**, **minimal**, **left-maximal**, **right-maximal**.

## Results

When reported, the sub-stream of tuples needs to be related to a time interval that captures the validity period of the frame predicate on the stream. The current thinking is to use the **union** of all tuple intervals (see *“Tuple Timestamps”*) and, therefore, mirror the semantics chosen at the tuple level. Note that there is an issue with this approach if the tuple level chooses the exact approach (or any other interpretation where there are gaps between the tuple intervals) as the union of discrete points is a bit unintuitive. An alternative definition would be to take the start of the first tuple and the end of the last tuple as frame boundaries, effectively “interpolating” over the gaps.

**Current Thinking:** At the tuple level, we assume the third interpretation option (right open intervals) for validity timespans as described above, i.e. t1: [1, 2), t2: [2, 3), t3: [3, 4), etc. For frames, however, we assume closed intervals, where the start point and end point are determined by the start point of the first and the last contained tuple, respectively. For example, a frame containing the tuples {t1, t2, t3} is reported to span [1, 3].

# Evaluating Frames

Possible “selling points” for frames include, but are not limited to, the following.

* Arguing that frames give higher “precision” with the same (computational) effort.
* Arguing that frames give the same “precision” with less (computational) effort.
* Arguing that there is an added value to frames as their boundaries and the number of tuples they contain are part of the query result. This is in contrast to windows where one of the two is always part of the specification.

# Examples

Below, we go through a series of frame applications, show how our specification is used to express the frames and graphically present the resulting stream segmentation.

1. **sum(vol) > 20**
2. **max(x)-min(x) > 5**
3. periods of high loss rates for router
4. freezer temp is high, but temp is reported on change (irregular data reporting)
5. minimal spans of 20 different people
6. freezer door openings

# Open Issues

Yeah, currently there are no open issues!

# Notes

## January 20, 2011

### Time vs. Tuple Segmentation

A frame clearly segments a stream. A frame could be considered to segment time (or some other underlying domain) or a frame could be considered to segment tuples. Our position is that frames group tuples and use the grouping of tuples to segment time.

### Context-Dependent Segmentation

We observe that some frame specifications appear to be context-independent vs. others that appear to be context-dependent. The **sum(vol)** example or the **max(x)-min(x)** examples appear to be context dependent. In these examples, the frames produced depend on where an implementation or system begins observing the stream; two implementations given the same stream may indefinitely produce a series of different frames (need picture). In contrast, consider the router loss rate example – two implementations would consistently produce the same frames regardless of when they started observing the stream (note that the first frame could possibly differ, but all subsequent frames would be identical).

### Items Required for a Frame Specification

* Predicate
* Maximal vs. Minimal
* Frame start (re-start) specification

The predicate defines what tuples appear in the frame (i.e. loss rate > 0.3%). ***MICHAEL***: How do you specify the predicate for the **max(x)-min(x)** frames or the **sum(vol)** frames (see examples).

Maximal/minimal is specified in the *“Frame Length and Slide Offset”* section.

Tentatively, frame start (re-start) specification defines where a new frame starts after a previous frame concludes.

We believe that the segmentations proposed in the *“Segmenting the Stream”* section should be consequences of the frame specification.