Abstract—Despite the availability of several data stream processing engines (SPEs) today, it remains hard to develop and maintain streaming applications. Existing SPEs vary widely in their data and query models and capabilities. A lack of standards, and the wide (and changing) variety of application requirements, restrict portability. Users find it difficult to know which system to use, and even to understand the behavior of the system they choose. Our goal in this paper is to propose a formalism that can be used to explain a major subset of these different behaviors. We first provide an in-depth analysis of the heterogeneity problem across three well-known and commonly used stream processing systems (two commercial and one academic), focusing on the execution semantics of sliding window queries. We then propose a simple yet powerful formal model that captures and explains the core differences across these systems. Those systems lead to totally different query results even on queries that look almost the same at the SQL syntax level. We show the power of our formal model by applying it to example queries inspired by the Linear Road Stream Data Management Benchmark. Furthermore, we believe that our proposed model can serve several purposes other than just explaining queries. As an illustration, we describe the use of our formalism as the query execution model in the MaxStream system - a federated stream processing system that we have built at ETH Zurich to integrate multiple SPEs and databases behind a common and easy to use interface.

I. INTRODUCTION (*** LAURA / NESIME)

- Stream processing entering its teenage years
- Many SPEs, lots of heterogeneity => a pain in the neck for everybody (SPE vendors, application developers, and users)
- We need to take a step back and make an effort to create a unified query execution model for streams that is well-defined and well-understood.
- Many potential benefits of such a model: understand and explain how different systems behave, create guiding tools for system users, find out about strengths and limitations of different systems, exploit the power of diversity, help the standardization bodies, ...
- Main contributions of this paper: We realize that the problem is hard. It’s likely that there will be a few proposals evolving in time. This paper takes some initial steps which we think will have major contribution to this process. (TODO: Make sure that we use the right words here)

- This paper is outlined as follows...

II. RELATED WORK

The initial generation of stream processing systems such as Aurora [1], STREAM [2], and TelegraphCQ [3] each proposed its own data and continuous query processing models. This has led to great variety in how important primitives such as stream, time, order, windows, query operators can be modeled in a system. Each proposed model seems to have similar expressive power and each seems to make sense for a given set of applications with proper semantic interpretation, thus, there seems to be no clear winner. A reconciliation of models is clearly necessary, but it is not obvious how this should be done.

STREAM’s CQL provides a formal model that takes its basis from the well-understood relational model [2]. CQL is an extension to SQL:1999. First, it introduces “stream” as a second data type in addition to “relation”. Second, in addition to the “relation-to-relation” operations of the relational algebra, CQL introduces “stream-to-relation” operations for constructing windows on streams as well as “relation-to-stream” operations to convert results of relational operations back into the stream data type. Through these mappings, one can essentially reuse most of the relational algebra semantics in a rather straight-forward way. Finally, CQL has also introduced the notion of time into the relational model, which essentially adds the continuous query execution semantics: time advances from \( t - 1 \) to \( t \), when all data items up to \( t - 1 \) have been processed.

CQL’s basic principles have also been adopted by other systems (e.g., Oracle CEP [4]). However, CQL semantics by itself is not sufficient to explain all the different behaviors that we see from different systems today. As a simple example, not all systems’ have a time-driven continuous query execution model [5].

In the retrospect of the early generation systems, a few recent studies have tried to offer cleaner abstract models without necessarily being tied to a specific system implementation.

Maier et al have proposed reconstitution functions for precise denotation and representation of data streams [6]. This
work essentially generalizes the denotational semantics approach of STREAM CQL. The focus has been only on defining the meaning of a stream itself, and the reconstitution functions idea has not been developed further to capture the execution semantics (in particular window-based processing). The same group of researchers have also proposed another framework (called Window-ID) for defining window semantics [7]. The Window-ID framework decouples the logical content of each window from the corresponding operator implementation and other physical stream properties (e.g., arrival order). Window semantics is defined based on three functions: windows, extent, and wids. Let \( T \) be a set of tuples, and \( S \) be a window specification (i.e., window type + three parameters \{range, slide, window-attribute\}). \( \text{windows}(T, S) \) returns a set of window-id's; \( \text{extent}(w, T, S) \) returns which tuples belong to window \( w \); \( \text{wid}(t, T, S) \) returns the set of wid’s to which tuple \( t \) belongs. In other words, \( \text{wid} \) is the inverse of \( \text{extent} \). This framework basically returns the same window contents regardless of the implementation. The authors also present an implementation of the framework where a "bucket" operator tags tuples with wid ranges, which are sufficient for a windowed operator to process the windows without any further information on window semantics. Thus, window semantics becomes totally independent from the operator behavior. Our work differs from Maier et al’s work in that we not only consider window contents but also other operational issues that influence the query results.

Patroumpas and Sellis have also proposed a formal framework for expressing windows [8]. The model is based on a time-parameterized scope function that specifies a time-based window’s size and progression in time. As in STREAM CQL, the query evaluation takes into account the most recently and completely arrived window looking in a backward direction. Thus, there can at most be one window open at a given time. This work is not based on real system implementations and it has not been tested if the proposed formalism is powerful enough to capture the existing systems’ behaviors.

Most recently, Kramer and Seeger have proposed a model of logical and physical operator algebras for stream operators, applying ideas from temporal databases [9]. This approach is very similar to STREAM CQL, with a few differences. First, every tuple is assigned a time interval showing its validity period instead of a single timestamp. Second, the snapshot reducibility concept from temporal databases is used in finding equivalences for query optimization, however, this concept does not apply to window operators. Finally, the authors describe the physical implementation of their operators in the PIPES system. This paper does not improve our understanding beyond the CQL model.

We are not well-informed about the underlying formal models of the current commercial systems [10], [11], [12], [4], [13]. Furthermore, each of them seems to use a completely different model, and the results are not easy to compare.

A recent paper tried to reconcile the differences across two of these commercial systems, Oracle CEP and StreamBase, in terms of the way the window execution is triggered in their respective models [5]. Though an important first step towards reaching an agreed model, this work focuses on only one aspect of execution behavior.

III. PROBLEM ANALYSIS (**IRINA / ROOZBEH**)

Current SPEs differ highly in their continuous query capabilities. Furthermore, the implementation of a common capability can also vary from one SPE to another, due to the differences in these SPEs’ query execution models. In this section, we will illustrate that the differences among SPEs are beyond just simple syntactical ones; rather they originate from the deeper differences in implementation semantics of the streaming operators and may lead to completely different (and sometimes even unexpected) query results.

After studying a range of academic and commercial SPEs, we have observed that the heterogeneity among the SPEs exposes itself at two levels:

- **Capability heterogeneity:** This type of heterogeneity refers to the differences in support for certain types of queries across different SPEs, and exposes itself at the language syntax level in a way that is directly visible to the application developer.
- **Execution model heterogeneity:** This type of heterogeneity refers to the differences in underlying query execution models across different SPEs, and does not expose itself at the language syntax level. Therefore, it is hidden from and can not be influenced by the application developer.

Next, we will illustrate some of these differences through a few sample queries that we have run on two commercial SPEs that we call System X and System Y hereafter. The queries are simplified versions of the Linear Road Benchmark [14] queries defined on an input stream of car position reports \( \text{InStream}(\text{VID}, \text{Spd}, \text{Time}) \), where \( \text{VID} \) represents the vehicle id, \( \text{Spd} \) represents the speed reported by the vehicle, and \( \text{Time} \) represents the time of the report.

A. Capability Differences

The following two examples show how two SPEs can differ in the query capabilities that they support.

**Example 1: Window Definition on a Non-time Tuple Field**

*Query:* Continuously compute the number of cars that send reports with speeds in a range of 30 mph (i.e., the difference in speed between the first and the last report in a given window should be at most 30).

In System X, we can easily express this query because System X allows windows to be defined on any of the tuple fields; while this is not possible in System Y since it only allows windows to be defined in terms of number of tuples or in terms of time (either system time or tuple timestamp).

**Example 2: Flexible Window Slide**

*Query:* Continuously compute the number of cars that emit position reports in a window of 3 minutes, with a slide of 2 minutes.
In System X, one can define any slide value for advancing to the next window; while in System Y, this is not possible since window slide is assumed to be an implicit value (either 1 tuple/1 time unit, depending on the window type, or the window size).

B. Execution Model Differences

The heterogeneity does not stop at differences in supported capabilities. Queries that look similar in visible capabilities, may produce completely different results due to the differences in SPE execution models. Furthermore, even for a single SPE, running the same query multiple times generates unexpected results, as we illustrate next.

We only show in the timestamp (set by the system) field the seconds, while the input items were sent to the system one second apart.

Example 3: Window Construction

Query: Continuously compute the average speed of cars with position reports arriving in a tumbling window of size 3 seconds.

First run:

Input: \((VID, Speed, Time) = \{(1,10,10), (2,20,11), (3,30,12), (4,40,13), (5,50,14), (6,60,15), (7,70,16), \ldots\}\)

System X Output = \{15\}, \{40\}, \ldots\)

Second run:

Input: \((VID, Speed, Time) = \{(1,10,11), (2,20,12), (3,30,13), (4,40,14), (5,50,15), (6,60,16), (7,70,17), \ldots\}\)

System X Output = \{10\}, \{40\}, \ldots\)

Third run:

Input: \((VID, Speed, Time) = \{(1,10,12), (2,20,13), (3,30,14), (4,40,15), (5,50,16), (6,60,17), (7,70,18), \ldots\}\)

System X Output = \{20\}, \{50\}, \ldots\)

What differs from a run to another are the timestamps of the input tuples. Although we were expecting to get the third result (first three tuples belong to the first window, next three to the second window etc), the example proves us wrong. It seems that the grouping of tuples in a window is related to the input tuples’ timestamps.

Example 4: Change of Window State

As has been the focus of a recent publication by Jain et al [5], change of window state might be triggered by different events in different SPEs, leading to different query execution semantics and results. This paper provides an extensive set of examples showing the problem between the tuple-driven execution model of StreamBase [10] and the time-driven execution model of Oracle [2] 1.

We ran the following query in System Y and System Z.

Query: Continuously output the speed of cars with position reports arriving in a sliding window of size 1 second.

Input = \{(1,10,1), (2,20,1), (3,30,3), (4,40,4), \ldots\}

System Y Output = \{(10), (20), (30), (40), \ldots\}

System Z Output = \{(10), (30), (40), \ldots\} or \{(20), (30), (40), \ldots\}

Where does the difference come from? System Y has a tuple-driven model for window state changes, while System Z has time-driven model for window state change. Since System Z can not make any further temporal distinction between simultaneous tuples, it picks one of them non-deterministically as stated in Jain et al[5].

Example 5: Different Number of Results

We consider again the query in example 3.

Input: \((VID, Speed, Time) = \{(1,10,30), (2,20,31), (3,30,32), (4,40,33), (5,50,34), (6,60,35), \ldots\}\)

System X Output = \{(20), (50), \ldots\}

System Y Output = \{(10), (15), (20), (40), (45), (50), \ldots\}

Where does the difference come from? In Figure 1(a), we depict the window states at each time instant. System Y reports the result of evaluating the average function on the current window state for each input tuple arrival, while System X reports a result only when the window reaches its maximum size.

Example 6: Window Evaluation Strategy

Query: Continuously compute the average speed of cars with position reports arriving in a sliding window of size 5 seconds.

As a side note, Jain et al also state that Coral8 implements the “big switch” approach in that tuple-driven corresponds to their default execution and time-driven can be achieved by the \texttt{OUTPUT EVERY 1 second} clause. We believe that this is a very rough assessment of Coral8, and that one should look a bit more fine-grained to see what other semantic aspects play a role here.
Window Definition

Analyzing the SPEs and their query capabilities, we extracted a set of parameters that influence window definition: CONTENT The content of the window can be defined by:
- The tuples which arrive in the input stream within a certain system time (also called “wall clock time”) interval. The windows for which content is defined in this manner are called “time-based windows”.
- The number of tuples (these windows are called tuple-based or “physical windows” [8]).
- The tuples for which a given (monotonically increasing) field value falls into a certain range (we name these windows “value-based windows”). For System Y a special case is when the field represents timestamps, because the input stream evolution is kept synchronized with the system time based on the tuples’ timestamps and a maximum delay (lag).
- The start and end predicates on certain fields in the tuple (“semantic windows”). An example for semantic windows usage in queries is: the query specifies that a window should be opened on a stream when a car report arrives with Speed > 40 and closed when a Speed < 10 is reported for that car.

SIZE Excepting the semantic windows, which are explicitly defined through their boundaries, for all others we need to specify a size representing the (maximum) number of units (corresponding to the CONTENT type - tuple, time, field value unit) it should contain. For example, a 5 seconds window has a size 5 and is defined on time - either system or timestamp.

DIRECTION (for time-based windows) A window can be defined either forward or backward, meaning that it can be described by one of the boundaries (a moment in time representing the start time or the end time of a window) and its SIZE. That is, the window is defined either as (window start + SIZE) or (window end - SIZE).

SLIDE defines the number of units that expire between two consecutive window states (defines the “distance” between two window states). For example, a query like: every 2 seconds, compute a query’s results on a window of size 5 seconds expresses a SLIDE value of 2. If the query is defined like: continuously compute the results on the last 5 seconds, there is an implicit value for SLIDE and that is 1.

WINDOW DEFINITION START This refers to the time the first window on the input stream is constructed. It can be based on query registration time, first tuple arrival time etc.

EVALUATION defines the unit that determines the change of a window’s state. As we have already presented in Query Example 3, there are three units for window state change: tuple, time and batch.

Window Operations

EVALUATION specifies the frequency for computing (partial) query results. This operation can be triggered by a series of events:
- A logical change in the window state: time moves forward with one unit. A special window state is when it reaches
the SIZE (it is equivalent to window being "closed" in some streaming engines).

- A content change as a result of new tuples being added or of old tuples expiring.
- The operator’s evaluation on the window content could be just triggered periodically.

REPORT determines the strategy (frequency) for reporting query results. This operation is important as it determines what the user sees as the (partial) result of the query. It can be EVALUATION-based, that is, whenever an EVALUATION operation is performed the generated result is immediately presented to the user, or it can skip some results if the user only expects periodic updates.

All these will be presented more formally in the next section. In this paper we concentrate on time-based windows as their semantics is the most challenging and involves the most factors that affect the behavior. The tuple-based windows can be described in the same manner.

IV. THE FORMAL MODEL (*** NIHAL)

- "scope" to capture physical window content
- "tick" to capture window evolution, and extending scope to take care of "tick". Show that we can also accommodate the batch-based model proposed by Jain et al [5].
- "result reporting frequency"
- the effect of evaluation mechanism (in particular for monotonic vs. non-monotonic queries)

A. Basic Definitions

**Definition 1** The Time Domain \( \mathbb{T} \) is considered as an ordered, discrete, infinite set of time instances as in [8].

**Definition 2** A Stream \( S \) is bag of items, where each item \( \langle s, \tau_i \rangle \) in the set is composed of a relational tuple \( s \) with a timestamp \( \tau_i \in \mathbb{T} \) attached to it. Notation \( s.AT \) is used for the timestamp attribute of the relational tuple \( s \). For batch-driven systems, the definition of a stream is extended with batch number \( b_i \), so that each element in the stream is in form of \( \langle s, \tau_i, b_i \rangle \). Similarly, the notation \( s.AB \) is used to represent the batch number attribute of the relational tuple \( s \).

B. The Scope of a window

The Scope of a window at a given time \( \tau_i \) is a set of time intervals *** IF WE HAVE MULTIPLE WINDOWS - DO WE STILL CONSIDER THIS OPTION OR ONLY THE SINGLE WINDOW MODEL? ***. Each interval in this set is represented through the lower and the upper bounds of an active window at a given time \( \tau_i \).

**Definition 3** For \( \tau_i \), Scope\((\tau_i)\) := \( \{[\tau_i', \tau_i] \} \) with \( \tau_i \geq \tau_i' \). We name Size of Scope\((\tau_i)\) and represent as \( |\text{Scope}(\tau_i)| \) the difference \( \tau_i - \tau_i' \).

The scope of a time-based window depends on the window specification (size and slide) as well as stream engine’s window model (window construction direction, single/multiple window model *** DO WE STILL CONSIDER IT? *** , first window construction time) as described in section III. Single and multiple window model are related to lazy or eager window construction strategies, respectively.

The difference between single and multiple window models can only become visible with the help of the other execution model parameters like evaluation and/or reporting. We have not observed a system yet to show this difference such as a system which has multiple window model and window state change based evaluation strategy, so that it reports each tuple in each window. Therefore, for the rest of the paper we will concentrate on single window model, but it is important to be aware of this parameter for completeness.

1) Forward window construction, *** Single window model ***: Figure 3 illustrates the construction of time-based windows in an engine which has *** a single window model**** and the direction of window construction is forward. As stated before, in *****single window model******, there is always one window at a given time (in Figure 3, the gray color in the figure is used to represent the inactive parts of the window). For example, the active window at time \( \tau_0 + 2\beta \) is \( W_1 \) and its scope is \( [\tau_0 + \beta, \tau_0 + 2\beta] \).

Fig. 3. Window construction when direction=forward and *** model=single ***

The following formula gives the scope of a time-based window at a given time instance \( \tau_i \in \mathbb{T} \) when ***** the window model is single ***** and construction direction is forward. \( \omega \) denotes the SIZE of the window, while \( \beta \) represents the SLIDE unit of the window. Let \( \tau_0 \in \mathbb{T} \) be the time instance when then first window of the query is constructed. Let \( n \in \mathbb{N} \) be the number associated with a given window (\( n=0 \) represents the first window, \( n=1 \), the second window and so on).

\[
\text{Scope}(\tau_i) = \begin{cases} 
\emptyset, & \tau_i < \tau_0 \\
\{[\tau_0, \tau_i]\}, & \tau_i < \tau_0 + \omega \\
\{[\tau\text{start}(\tau_i - \tau_0 - \omega) + 1, \tau_i]\}, & \text{otherwise} 
\end{cases}
\]

where, \( \tau\text{start}(n) = \tau_0 + n\beta \)

The formula returns the empty set at time \( \tau_i \) if the first window has not constructed yet *** if \( \tau_i \) is earlier than the first window’s construction time ***. The second condition in the formula defines the scope of the first window of the system. Except these special cases, the formula returns the scope of the most recently opened but not yet closed window. Since window model is single, \( W_n \) exists iff \( W_{n-1} \) is closed. Therefore; first the most recent closed window’s number is calculated as follows: \( 0^{th} \) window closes at time \( \tau_0 + \omega \), whereas \( 1^{th} \) window at \( \tau_0 + \omega + \beta \) and \( n^{th} \) window closes at time
The most recent closed window’s number at time \( \tau_i \) is the maximum value of \( m \) when \( \tau_0 + \omega + m\beta \leq \tau_i \) which is \( \lfloor \frac{\tau_i - \tau_0 - \omega}{\omega} \rfloor \). After that, the next window after the most recently closed window’s scope is returned. This sentence is confusing. Since window construction is forward, a new window is opened at every \( \tau_i \) we are talking about scopes but in the same time opening and closing of windows. I think it should be either one or another because otherwise it becomes confusing.

2) Backward window construction, single window model: Figure 4 illustrates the construction of time-based windows in an engine which has single window model and the direction of window construction is backwards.

![Window construction when direction=backward and model=single](image)

As depicted in Figure 3 and Figure 4, the backward and forward window construction directions are slided versions of each other. The same scope values are obtained for both window constructions directions if the difference between the \( \tau_0 \) values of backward and forward is equal to size modulo slide. The scopes always coincide iff two starting points of backward and forward windows directions coincide at some point in time. Thus,

\[
\exists n, m \in \mathbb{N}, \quad \tau_{0f} + n\beta = \tau_{0b} + m\beta - \omega
\]

\[
\exists p \in \mathbb{Z}, \quad \tau_{0b} - \tau_{0f} = p\beta + \omega
\]

\[
\tau_{0f} - \tau_{0f} = \text{mod}(\omega, \beta)
\]

The window evaluation strategy can be based on different events in the system. For some systems a window is reevaluated whenever its content is changed (\( ev_{\text{content change}} \)), for some of them whenever the window closes (\( ev_{\text{window close}} \)) or its state changes (\( ev_{\text{state change}} \)). These three different conditions are not orthogonal to each other: for example changes in the state of the window might close the window, or arrival of a tuple can both change the state and the content of the window. The evaluation of a window can be explained more formally as follows: but it seems we need to add the condition that the evaluation is executed only if the scope has changed from the previous evaluated one in the window close case:

\[
\text{Eval}(\tau_1, \text{Op}) = \begin{cases} 
\text{Op}(C(\tau_i)), & ev_{\text{window close}} \land \text{Scope}(\tau_i) = \omega \\
\text{Op}(C(\tau_i)), & ev_{\text{content change}} \land C(\tau_i) \neq C(\tau_{i-1}) \\
\text{Op}(C(\tau_i)), & ev_{\text{state change}} \land \text{Scope}(\tau_i) \neq \text{Scope}(\tau_{i-1}) \\
\emptyset, & \text{otherwise}
\end{cases}
\]

In the formula above, each event which triggers evaluation is described more formally. A window closes whenever it reaches its maximum size, \( \omega \). The content of a window changes whenever the tuples it contains at time \( \tau_i \) differ from the ones at \( \tau_{i-1} \). On the other hand, the state of a window changes whenever its scope interval changes. Whenever evaluation is triggered, operator \( \text{Op} \) is executed on the current content of the window.
E. Reporting of a window

Another important factor for query execution is the frequency of reporting the query results. The Reporting strategy of a system can be periodic, like every 5 seconds or whenever the window is evaluated (evaluation-based). It can be explained more formally as follows:

\[
\text{Report}(\tau_i, R_p, \gamma) = \begin{cases} 
\text{Eval}(\tau_i), & R_p = \text{per_time} \\
\text{mod}(\tau_i, \gamma) = 0, \gamma > 0 \\
\text{Eval}(\tau_i), & R_p = \text{eval_base} \\
0, & \text{otherwise}
\end{cases}
\]

In the formula above, \( R_p \) represents the reporting strategy while \( \gamma \) represents the reporting frequency for periodic reporting. Different systems can have different reporting strategies, like reporting with each tuple arrival, or a certain number of tuples. Reporting ensures to return the most recent result to the user. Therefore, evaluation is triggered whenever result is reported. *** I THOUGHT THAT THE THE REPORTING STRATEGY CAN BE EVALUATION-BASED, WHICH MEANS THAT A REPORT OPERATION IS ISSUED WHENEVER AN EVALUATION OPERATION IS EXECUTED, NOT THAT AN EVALUATION IS EXECUTED WHENEVER WE NEED TO REPORT; ***.

V. THE FORMAL MODEL IN ACTION

- Formally explain the queries from Section III. (**NIHAL**)
- Briefly present MaxStream federated architecture as its working use case. (**NESIME**)

A. Example 3

In example 3, we ran the same query three times on the same engine, System X, using the same input items each time. To our surprise, we obtained different results in each run. As we already presented in section III, we were expecting the first three tuples to belong to the first window, the next three to the second and so on. The difference in results comes from how the windows are created. Instead of starting the first window with the first arriving tuple, System X uses a time-division scheme: the time line is divided into discretized subintervals. The arriving tuples are then mapped to the corresponding windows as presented in Figure 5. For our example, this will generate 3 series of results because to the slide value. The start of the first created window (\( \tau_0 \)) can be described using the formula \( \lfloor \frac{\tau_1}{\beta} \rfloor \beta \), where \( \tau_1 \) represents the timestamp of the first tuple.

So, for example, for the input items,

\[
\text{Input( VID, Speed, Time)} =\]

\[
(1,10,10), (2,20,11), (3,30,12), (4,40,13), (5,50,14), (6,60,15) \]

Fig. 5. Time Division for Window Definition

\[
\frac{10}{3} \cdot \beta = \frac{10}{3} \cdot \frac{30}{3} = 10
\]

the start of the first window containing at least one of the input tuples is \( \lfloor \frac{10}{3} \rfloor \beta = \lfloor \frac{10}{3} \beta \rfloor = 9 \). Therefore the first window will have the scope [9, 11], which means that it will contain tuples (1,10,10) (2,20,11), generating result (15).

B. Example 4

C. Examples 5 and 6

The Execution Model specifications of System X and System Y are shown in Table ??.. Both systems have the same first window construction strategy. That is, \( \tau_0 \) is computed for both from the first input tuple’s timestamp: \( \tau_0 = \lfloor \frac{\tau_1}{\beta} \rfloor \beta \). *** On the other hand, while System X has multiple window model, System Y has the single window model.***

As stated in Section IV-B.3, if the window size is a multiple of the window slide, backward and forward window construction will eventually have the same scope values. In System Y, the slide value (\( \beta \)) can be either the minimum time unit or the window size (\( \omega \)). While System X has a backward window construction, we consider System Y as having a forward construction direction here and we will show that this rule holds *** IF WE KNOW THAT SYSTEM X HAS A BACKWARD MODEL, WHY DON’T WE CAN MAKE THE COMPARISON MORE CLEAR! ***. Both systems have tuple-driven evolution and their reporting is based on evaluation. The main difference between the two systems is evaluation strategy: while System X reevaluates the operator on the window whenever it closes (reaching maximum size), System Y reevaluates it whenever the window content changes. *** TABLE ?? NEEDS TO BE RESIZED ***

<table>
<thead>
<tr>
<th>Name</th>
<th>Scope</th>
<th>Evolution</th>
<th>Evaluation</th>
<th>Result Reporting</th>
</tr>
</thead>
<tbody>
<tr>
<td>System X</td>
<td>( \tau_0 = \lfloor \frac{\tau_1}{\beta} \rfloor \beta ), <em><strong>multiple</strong></em>, backward</td>
<td>tuple-driven</td>
<td>window close</td>
<td>evaluation based</td>
</tr>
<tr>
<td>System Y</td>
<td>( \tau_0 = \lfloor \frac{\tau_1}{\beta} \rfloor \beta ), <em><strong>single</strong></em>, ( \omega )</td>
<td>tuple-driven</td>
<td>content change</td>
<td>evaluation based</td>
</tr>
</tbody>
</table>

Tables ?? and ?? show the execution of the Example Query 5 on Systems X and Y, respectively. *** THE SCOPES ARE DIFFERENT. WEREN’T WE SUPPOSED TO GET THE SAME SCOPES? HOW DID WE GET OPEN INTERVALS WHEN THE SCOPE FORMULA CONTAINS ONLY CLOSED INTERVALS???*** The difference in their window model (single/multiple) does not have any affect on the result here, since window is specified as tumbling window in the example and both single and multiple window have one single window at a time ***. \( \tau_0 \) is calculated as 30 for both systems. It is important to note that, we can only see the tuples’ arrivals at a minimum granularity of seconds in System X. Based on our experiments, the tuples are fed to
the system with some delay, not exactly at the exact second tick. Therefore, for example, the first tuple having a timestamp value of 30 arrives later than 30th second and it is not part of the (27 30) interval. However, we do not observe such a behavior in System Y as the tuples are fed into the system strictly according to their timestamp values.

The execution of the Example Query 6 on System X and System Y is shown in Tables ?? and ?? respectively. \( \tau_0 \) is calculated as 0 for both systems. *** The difference in their window model (single/multiple) does not have any affect on the result of this query either because of the window reporting strategy System X has.*** As it can be seen in Table ??, System X’s result reporting is based on its evaluation, which is whenever window closes. *** Though System X has multiple windows, since more than one window can not be closed at the same time, we don’t see its effect on the result and it can be considered as single window, ***

For this query, the slide value (\( \beta \)) is equal to the minimum time unit supported by the system. Therefore, a window is constructed at every time tick and the count of the most recent window \( \lfloor \frac{\tau_i - \tau_0 - \omega}{\beta} \rfloor \) is equal to \( \tau_i - \tau_0 - \omega \). Thus, if the minimum time unit supported by the system is milliseconds, the start of the interval for forward window is \( (\tau_i - 5000 + 1) \times 1 \), which explains why the lower boundaries are not included in execution of the query *** ???? ***. As shown in the execution tables, after the first window both systems have the same scope values, like we expected, because the size of the window is a multiple of the slide. The difference in their results originates from the different evaluation strategies they expose. ***

*** THE SCOPES HAVE TO BE REWRITTEN BECAUSE THEY ARE NOT CONSISTENT WITH THE TIMESTAMPS IN THE EXAMPLE OF SECTION III ***

**VI. Extensions to Unified Model**
- extending the above to binary windowed operations (i.e., join)

*References*