SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems

Irina Botan[1], Roozbeh Derakhshan[1], Nihal Dindar[1],
Laura Haas[2], Renée J. Miller[3], Nesime Tatbul[1]

[1]ETH Zurich, Switzerland [2]IBM Almaden Research Center, USA [3]University of Toronto, Canada

ABSTRACT

There are many academic and commercial stream processing engines (SPEs) today, each of them with its own execution semantics. This variation may lead to seemingly inexplicable differences in query results. In this paper, we present SECRET, a model of the behavior of SPEs. SECRET is a descriptive model that allows users to analyze the behavior of systems and understand the results of window-based queries for a broad range of heterogeneous SPEs. The model is the result of extensive analysis and experimentation with several commercial and academic engines. In the paper, we describe the types of heterogeneity found in existing engines, and show with experiments on real systems that our model can explain the key differences in windowing behavior.

1. INTRODUCTION

Stream computing is passing from the domain of pure research into the real world of commercial systems. Many research projects [8; 11; 17, and others] have shown how data can be processed as it pours into a system from a diversity of sources such as sensors, online transactions, and other feeds. Each system proposed its own set of operators, windowing constructs, and, in some cases, whole new query languages [9, 12]. As these systems have been commercialized [1, 6, 7], they have added features to meet the needs of their own customers. There are no standards today for querying streams; each system has its own semantics and syntax. For the purchaser or user of an SPE, the choices are confusing. Without a clear understanding of features and semantics, applications are not portable, and can be hard to build, even on a given SPE.

The emerging SPEs have different capabilities, and even common capabilities may be expressed differently in different systems. For example, both StreamBase [6] and Coral8 [1] allow time-based windows where a window is defined by an interval size (in units of time) and where different windows are separated by a slide value that specifies how many units of time separate the start of different consecutive windows. To specify such a window in StreamBase, the user has to write "`[SIZE x ADVANCE y TIME]`". In Coral8, the same function is requested with the "`KEEP x SECONDS`" clause. StreamBase allows an arbitrary slide value for a window (specified by the `ADVANCE` clause); Coral8 only permits two values: 1 msec or a slide that is equal to the window size. Worse yet, the underlying semantics of such common features as windows is often radically different. Even if we set window size and slide to the same respective values in Coral8 and StreamBase, we can get different query results due to hidden differences in their query execution models.

Recently, several more abstract models of streams and windows have been proposed [15, 18, 14], for the most part not associated with any existing system. These models tend to define only a portion of the behavior expected of an SPE. While they are useful as guides to future SPE developers, they do little to help users understand existing systems, and even less for comparing or explaining the behaviors of different systems.

In this paper, we propose a general model for describing and predicting the behavior of these diverse systems. Our model is a descriptive model, not yet another execution model. It strives to explain, and to allow the comparison of, the differing behaviors found in existing SPEs. The model is the result of detailed analysis and experimentation with a carefully-chosen set of real commercial and academic systems. We believe that our unique approach of creating a descriptive and explanatory model offers significant benefits to potential users of stream systems, both before and after they have chosen an engine for building their applications.

The next section illustrates the differences in features and semantics of several SPEs. With these differences as motivation, Section 3 presents our proposed model. Section 4 demonstrates, through examples run on different engines, how our model predicts the results that similar queries will generate for the different systems. Related work is covered in Section 5. Finally, we conclude in Section 6 with a discussion of future work.

2. MOTIVATION

Our goal is to create a formal framework which can be used to analyze the execution behaviors of individual SPEs, and to compare the execution behaviors of heterogeneous SPEs. This heterogeneity exposes itself at three levels:

1. Syntax heterogeneity: This type of heterogeneity refers to the differences in the language clauses (keywords) used for the definition of common constructs (e.g., windows). The syntax differences are understandable given the lack of

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

a standard language for stream processing.

2. Capability heterogeneity: This type of heterogeneity refers to the differences in support for certain types of queries across different SPEs, and also exposes itself at the language syntax level. For example, Coral8 offers a clause that controls how often a query result should be emitted, a feature we have not encountered in any other system.

3. Execution model heterogeneity: This type of heterogeneity refers to the differences in the underlying query execution models across different SPEs. It is hidden from and cannot be influenced by the application developer, and is subtler than the other types of heterogeneity, hence potentially more confusing. As a result, we focus on analyzing the execution semantics of SPEs in this paper.

To motivate the need for the type of descriptive model we propose, consider two examples, defined on a simple input stream `InStream(Time, Val)` of tuples. `Time` represents the application timestamp of the tuple in seconds, and `Val`, an integer value, represents the content of the tuple. Our queries compute an average over `Val`, and `OutStream(Avg)` is the output stream containing the results of the query.

Example 1: Window construction in an SPE
Consider a query which computes the average value of the tuples in the input stream using a time-based tumbling window of size 3 seconds.[1] We ran this query on StreamBase [6] three different times, each time feeding it exactly the same input data file. Surprisingly, StreamBase produced different results each time. It seems that the assignment by StreamBase of time values was slightly different for the three runs:

```
InStream(Time, Val) = {(10, 10),(11, 20),(12, 30),(13,40),
                       (14, 50),(15, 60),(16, 70), ...}
StreamBase Output1  = {(15),(40), ...}
InStream(Time, Val) = {(11, 10),(12, 20),(13, 30),(14,40),
                       (15, 50),(16, 60),(17, 70), ...}
StreamBase Output2  = {(10), (30), ...}
InStream(Time, Val) = {(12, 10),(13, 20),(14, 30),(15,40),
                       (16, 50),(17, 60),(18, 70), ...}
StreamBase Output3  = {(20),(50), ...}
```

Intuitively, we expected the result of `Output3` in all runs (i.e., the first three tuples belong to the first window, the next three to the second window, etc.). However, the slightly different time values seem to have led to a difference in window construction, hence different results.

Example 2: Evaluation differences across SPEs
Consider a query which continuously computes the average value of the tuples over a time-based window of size 5 seconds that slides by 1 second. We ran this query in three different SPEs: STREAM [5], Coral8 [1], and StreamBase [6], with the following results:

```
InStream(Time, Val) = {(30, 10),(31, 20),(36, 30), ...}
STREAM Output       = {(10),(15),(20),(30), ...}
Coral8 Output       = {(10),(15),(20),(30), ...}
StreamBase Output   = {(10),(15),(15),(15),(15),(20),(30), ...}
```

StreamBase produced a different result than STREAM and Coral8. Why? In STREAM and Coral8, the average operator is invoked on a window whenever the window's content changes (i.e., when a tuple is added to or expires from the window), whereas in StreamBase, the invocation happens every second, even if the tuple content of the window stays the same. Thus the evaluation strategy used by an engine is another important factor affecting the query results.

These two motivating examples among others show that we need a way to understand, express, and predict the query execution behaviors of different SPEs. Our model, SECRET, takes up this challenge.

3. THE SECRET MODEL

In this section, we present a model for analysis of the execution semantics of continuous queries in SPEs, focusing on time-based windows and single-input query plans (see Appendix B.2 for extensions to other query types). We named our model SECRET, as it captures window-based query execution semantics along four complementary dimensions: ScopE (Section 3.2.1), Content (Section 3.2.2), REport (Section 3.2.3), and Tick (Section 3.2.4). Each of these models a certain aspect of window-based execution all the way from window construction into actual execution and result generation. Thus, our model gives an end-to-end view of what impacts execution semantics from inputs to outputs.

SECRET is **expressive**; it captures the key behaviors of a broad range of stream systems. It is **simple**: designed to be easy to understand, and easy to apply, avoiding complicated and redundant features. We have designed the features to be **orthogonal** to each other, as well. SECRET is **extensible**, offering the ability to add new features if necessary as new SPEs are encountered. Finally, for **clarity**, we want our model to separate the operational aspects of *how* the SPE processes streams from the non-procedural effects of that processing. For example, we can talk about how windows are formed independently of their content; this does not depend on any procedural aspect of a system. By contrast, when the system chooses to evaluate results depends heavily on its processing model. The model should also make a clear separation between data-level issues (e.g., values in a stream), query-level issues (e.g., window size) and system-level issues (e.g., when the engine takes an action).

3.1 Basic Definitions and Assumptions

In this section, we present the definitions for a set of basic stream processing concepts and constructs that we use in our model, together with any relevant assumptions we make.

Definition 1 (Time Domain) *The time domain $\mathbb{T}$ is a discrete, linearly ordered, countably infinite set of time instants $t \in \mathbb{T}$. We assume that $\mathbb{T}$ is bounded in the past, but not necessarily in the future.*

Definition 2 (Stream) *A stream $\mathbb{S}$ is a countably infinite set of elements $s \in \mathbb{S}$. Each stream element $s : \langle v, t^{app}, t^{sys}, bid\rangle$ consists of a relational tuple v conforming to a schema S, with an application time value $t^{app} \in \mathbb{T}$, a system time value $t^{sys} \in \mathbb{T}$, and a batch-id value $bid \in \mathbb{N}$. We use the notation $s.t^{app}$, $s.t^{sys}$, and $s.bid$ to denote the application time value, system time value, and batch-id value of stream element s, respectively.*

In the above definition (as in related work [19]), we have used two different notions of time: "application time" (t^{app}) and "system time" (t^{sys}). These both take values from our time domain $\mathbb{T}$, but carry two different meanings, and therefore, are used for two different purposes in our model. The value t^{app} captures the time information that is associated with the occurrence of the application event that a stream

[1] In a *tumbling* window, the size of the window is equal to the slide.

element represents (usually provided by the data source), and therefore will be used as the basis for query execution over the stream; whereas t^{sys} captures the time information that is associated with the occurrence of the related system event (arrival of the corresponding stream element at the system) and therefore will be used as the basis for reasoning about tuple arrival events in the system and how the system should react to them. Elements in a stream are assigned unique t^{sys} values, but multiple elements can share the same t^{app} value. Therefore, streams are totally ordered by the t^{sys} values of their elements, whereas they are partially ordered by their t^{app} values.

Definition 3 (Batch) *A batch $\mathbb{B}$ of stream elements for a given stream $\mathbb{S}$ is a finite subset of $\mathbb{S}$, where all $b \in \mathbb{B}$ have an identical t^{app}. Each such batch is given a unique batch-id bid $\in \mathbb{N}$ such that, for all $b \in \mathbb{B}$, $b.bid = bid$, indicating that b belongs to the batch that is uniquely identified by bid. For tuples t_1 and t_2 where $t_1.t^{sys} < t_2.t^{sys}$, then $t_1.bid \leq t_2.bid$.*

Batches are used to define a further ordering among simultaneous tuples [13]. By definition, all tuples in a given batch have the same t^{app} value, but that does not mean that *all* tuples with the same t^{app} value are in the same batch. For example, we can have four tuples with $t^{app} = 5$ in two consecutive batches of two tuples each. Therefore, a new batch can arrive without t^{app} advancing. This implies that streams are also partially ordered by their *bid* values.

Definition 4 (Window) *A window W over a stream $\mathbb{S}$ is a finite subset of $\mathbb{S}$.*

Windows can be defined in many ways. In this paper, we will mainly focus on "time-based windows". In time-based windows, stream elements whose t^{app} values fall into a certain t^{app} interval constitute a window. More formally:

Definition 5 (Time-based Window) *A* time-based window *$W = (o, c]$ over a stream $\mathbb{S}$ is a finite subset of $\mathbb{S}$ containing all data elements $s \in \mathbb{S}$ where $o < s.t^{app} \leq c$.*

In general, systems do not process arbitrary sets of windows, but rather require the windows to have a specific relationship to each other defined by two parameters, *size* (ω) and *slide* (β). More formally:

Definition 6 (Window Size and Slide) *The set $\mathbb{W}$ of all time-based windows defined over a stream $\mathbb{S}$ must satisfy the following two constraints:*

1. Size(ω): All windows must be the same size, that is, $\forall\, W = (o, c] \in \mathbb{W}$, $c - o = \omega$.

2. Slide(β): The distance between consecutive windows must be the same. For two windows $W_1 = (o_1, c_1]$ and $W_2 = (o_2, c_2]$, we require that $o_1 \neq o_2$. Furthermore, we say W_1 and W_2 are consecutive if $o_1 < o_2$ and there is no window $W' = (o', c')$ such that $o_1 < o' < o_2$. For all consecutive windows W_1 and W_2 in $\mathbb{W}$, we require that $o_2 - o_1 = \beta$.

At $t^{app} = t$, we say a window $W = (o, c]$ is *open*, if $o < t \leq c$. A window is *closed*, if $c < t$.

3.2 SECRET for Time-based Windows

In this section, we describe SECRET for time-based windows. Given a query's window parameters, ScopE provides information about potential window intervals. Content then helps us map those intervals into actual window contents, for a given input stream. REport states under what conditions those window contents become visible to the query processor for evaluation. Finally, Tick models what drives an SPE to take action on a given input stream. Tick is the actual entry point to the control loop of our model, creating a chain reaction by invoking Report, which in turn invokes Content, which builds on Scope (Tick → REport → Content → ScopE).

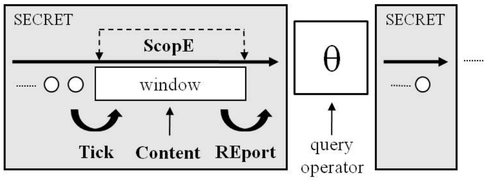


Figure 1: SECRET of a query plan

Figure 1 illustrates how we use SECRET to explain the semantics of a given query plan. SECRET is compositional in the same way a query plan is composed of a sequence of operators. We next present each of the SECRET parameters in detail, in reverse order, from Scope to Tick.

3.2.1 Scope

For a query q, the function *Scope* maps an application time value t to an interval over which q should be evaluated. To define *Scope*, we define the *active* window as the *open window with the earliest start time* at t.

We assume a value $t_0 \in \mathbb{T}$ that denotes the application time instant of the start of the very first window in a given system. Its value is system-specific, and possibly, invocation-specific, since different systems use a different starting point for their application time line depending on environmental factors (recall Example 1 of Section 2). Hence, the initial window (W_0) starts at time t_0, the next one (W_1) starts at time $t_0 + \beta$, and window i (W_i) starts at time $t_0 + i\beta$. Let $W_i = (o_i, c_i]$ be the i^{th} window in $\mathbb{W}$.

We now present a formula that computes n, the index of the earliest open window (i.e., the active window) at time t.

$$n = max(0, \lceil \tfrac{t - t_0 - \omega}{\beta} \rceil)$$

This formula is obtained as follows: W_0 closes at $t_0 + \omega$; W_1 closes at $t_0 + \omega + \beta$; and W_n closes at time $t_0 + \omega + n\beta$. At time t, we are interested in the earliest open window, which is the smallest n that satisfies $t \leq t_0 + \omega + n\beta$ (i.e., $n > (t - t_0 - \omega)/\beta$). Intuitively, $n = 0$ if the first window that opened at t_0 (and to be closed at $t_0 + \omega$) is still open. Otherwise, a new window has been opened every β time units. Then to find n for the earliest open window at t, we need to divide the total elapsed time since the close of the first window (i.e., $t - (t_0 + \omega)$) by β (and round it up to get a whole number).

Hence, the start time of $W_n = (o_n, c_n]$ is $o_n = t_0 + n\beta$, and *Scope* at time t is defined as follows:

$$Scope(t) = \begin{cases} \emptyset & \text{if } t < t_0 \\ (o_n, t] & \text{otherwise} \end{cases}$$

Figure 2 illustrates our *Scope* formulation. As a simple example, assume we have a query q with a window of size 5 seconds and of slide 2 seconds, to be run on a system with

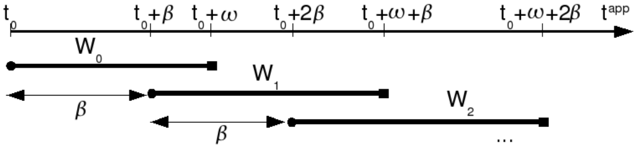


Figure 2: ***Scope*** **of a window**

t_0 of 30 seconds. Then the window scope at $t = 34$ seconds is $Scope(34) = (30, 34]$, since $n = 0$ and $o_0 = 30$.

There are a few important points to note about *Scope*:

1. The scope of a window for a given application time point solely depends on an SPE's t_0 parameter and the query's window parameters (which define $\mathbb{W}$). All these parameters take statically-defined values, and therefore are completely non-operational. In fact, one can hypothetically generate window scopes for arbitrary application time values, even before the actual query processing on a particular input stream starts, as *Scope* does not depend on the input stream nor any other system-specific behaviors.

2. Our Scope formula focuses on the time interval for the earliest open window. This is in fact one of many ways one could define Scope. For example, previous work defines Scope to be the time interval for the most recently closed window [9, 18]. Likewise, it would also be possible to define *Scope* as the set of intervals for all open windows. Our choice is justified by the need for a general and flexible *Scope* definition that could be used as a basis for explaining the behavior of systems that report their results on partial windows as well as those that do so on closed windows only.

3. During our analysis, we observed that systems may interpret the window slide value in two different ways, leading to two different window construction mechanisms. Some construct their windows at every slide in the backward direction, i.e., every new slide signals the end of a window which started ω time units ago [9, 18]. Others construct their windows at every slide in the forward direction, so every new slide signals the beginning of a new window [6, 8]. The window scopes produced for these two alternative interpretations differ only by a fixed amount δ, and therefore, one can choose one of these models and calibrate the starting time of the very first window t_0 by δ in case the other model's behavior is desired. More specifically, in SECRET, we chose to use the forward interpretation of slide in formulating *Scope*, and to simply adjust t_0 in order to model both forward and backward windows as needed.

3.2.2 Content

Scope defines the interval for query evaluation at application time t. A complementary function *Content* specifies the set of elements of stream $\mathbb{S}$ that are in this scope. As such, *Content* makes the mapping from the application time interval representation of a window to a set of data elements. We can formally define the content of a window at application time instant t and system time instant τ as follows:

$$Content(t,\tau) = \{s \in \mathbb{S} : s.t^{app} \in Scope(t) \wedge s.t^{sys} < \tau\}$$

Note that unlike *Scope*, the result of *Content* depends on actual contents of the input stream, which only become available at run time. Therefore, $Content(t, \tau)$ might potentially return different results even if it is called with the same t value, depending on how much of the input stream is already available (determined by τ) when it is invoked.

3.2.3 Report

The Report dimension in our model defines the conditions under which the window contents become visible for further query evaluation and result reporting. SPEs use different reporting strategies as illustrated in Example 2 of Section 2. We have identified four basic reporting strategies.

1. *Content change (R_{cc})*: reporting is done for t only if the content has changed since $t - 1$.

2. *Window close (R_{wc})*: reporting is done for t only when the active window closes (i.e., $|Scope(t)| = \omega$).

3. *Non-empty content (R_{ne})*: reporting is done for t only if the content at t is not empty.

4. *Periodic (R_{pr})*: reporting is done for t only if it is a multiple of λ, where λ denotes the reporting frequency.

Furthermore, some systems use multiple strategies (e.g., the content must have changed and be non-empty). Hence, we will use four boolean variables ($R_{cc}, R_{wc}, R_{ne}, R_{pr}$), each of which can be set to true or false by a system. Note that if all of these variables are set to false, then the report will still return $Content(t, \tau)$. This is the default behavior in our SECRET model. When all four variables are false, the reporting takes place every time it is triggered by the previous step of the model (i.e., *Tick* to be defined below).

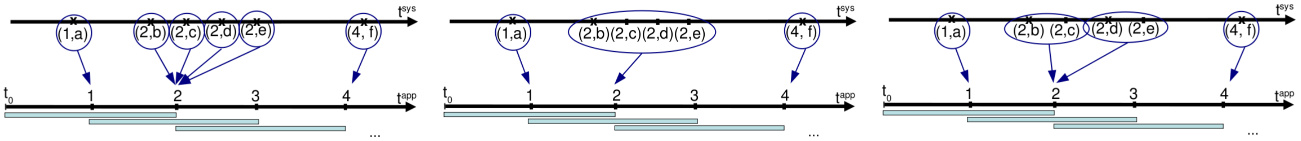
$$Report(t,\tau) = \begin{cases} Content(t,\tau) & \text{if } (\neg R_{cc} \vee Content(t,\tau) \neq \\ & \qquad\qquad Content(t-1,\tau)) \\ & \wedge(\neg R_{wc} \vee (|Scope(t)| = \omega \wedge \\ & t < max\{s.t^{app} | s \in S \wedge s.t^{sys} \leq \tau\})) \\ & \wedge(\neg R_{ne} \vee Content(t,\tau) \neq \emptyset) \\ & \wedge(\neg R_{pr} \vee mod(t,\lambda) = 0) \\ \emptyset & \text{otherwise} \end{cases}$$

3.2.4 Tick

The Tick dimension in our model defines the condition which drives an SPE to take action on its input (also referred to as "window state change" or "window re-evaluation" [13]). Like *Report*, *Tick* is also part of a system's internal execution model. While some systems react to individual tuples as they arrive, others collectively react to all or subsets of tuples with the same t^{app} value. During our analysis, we have identified three main ways that different systems "tick": (a) tuple-driven, where each tuple arrival causes a system to react; (b) time-driven, where the progress of t^{app} causes a system to react; (c) batch-driven, where either a new batch arrival or the progress of t^{app} causes a system to react. [2] These different Tick behaviors are illustrated in Figure 3. We show two time lines for t^{sys} and t^{app}. Tuple arrivals are shown on the time line for t^{sys}, and window scopes are shown underneath, on the time line for t^{app}. Circles around the tuples show the units of tuples that the system will react to at one time, whereas the arrows show to which application time instant those units belong. Note that the tuples are the same in all three figures, and that the four tuples in the middle have the same t^{app} value.

The tick models described above are based on the detection of three events: new tuple arrival, the progress of application time, and new batch arrival. The detection of each

[2] Remember from the definition of batch that a new batch can arrive without t^{app} advancing.



(a) Tuple-driven system (b) Time-driven system (c) Batch-driven system

Figure 3: Tick models

of these events is really based on the detection of new tuple arrival, since both application time information as well as batch-id information are carried in the tuples. Every new tuple arrival can only be uniquely detected if we check whether the stream has a tuple corresponding to every system time instant (for which there can be either one or none).

Before we present the formulas, we would like to note two key ideas that helped us structure our formulation:

1. At every tick, SECRET needs to check the reporting condition. However, since *Tick* operates on system time units and *Report* operates also on application time units, we need a mapping between them. We achieve this mapping with five mapping functions which will be presented shortly (*app*, *prev_app*, *batch*, *prev_batch*, and *prev_tick*). Note that the mapping is purely based on what is observed in the input stream, and not on any synchronization assumption between t^{sys} and t^{app} time lines, although this is something that real systems often do in order not to block the system's progress when the input lags behind (see Appendix B.1.2).

2. Since tick events can only be detected at new tuple arrivals, this will be a basic condition in our formulation. We must be able to account for irregularities in tuple arrival such as simultaneous tuples (i.e., multiple tuples with a common t^{app}) and gaps (i.e., absence of tuples at certain t^{app}). To detect simultaneous tuples, we need to be able to compare the current t^{app} with the previous tick time. To handle gaps, the arrival of a new tuple with t^{app} causes all application time instants between t^{app} and the previous tick time to invoke *Report* so that we do not miss any important application time instants. This is why we need mapping functions that provide us mapping for current instants as well as previous ones, explained next.

First, we define $S(\tau)$ and $S_I(\tau)$ as follows:

$S(\tau)$ denotes the set of tuples in stream S that has arrived through time instant τ.

$$S(\tau) = \{s \in S | s.t^{sys} \leq \tau\}$$

$S_I(\tau)$ denotes the set of tuples in stream S that has arrived at time instant τ. There can be at most one such tuple.

$$S_I(\tau) = \{s \in S | s.t^{sys} = \tau\}$$

We use the following mapping functions to define *Tick*:

app(τ): Given a system time instant τ, returns the application time value of the tuple that has arrived at τ.

$$app(\tau) = \{s.t^{app} | s \in S_I(\tau) \wedge S_I(\tau) \neq \emptyset\}$$

prev_app(τ): Given a system time instant τ, returns the application time value of the most recent tuple that has arrived before τ. If no such tuple exists, it returns t_0.

$$prev\_app(\tau) = max(max\{t_0, s.t^{app} | s \in S(\tau - 1)\})$$

batch(τ): Given a system time instant τ, returns the batch-id value of the tuple that has arrived at τ.

$$batch(\tau) = max\{s.bid | s \in S_I(\tau)\}$$

prev_batch(τ): Given a system time instant τ, returns the batch-id value of the most recent tuple that has arrived before τ. If no such tuple exists, it returns 0.

$$prev\_batch(\tau) = max(0, max\{s.bid | s \in S(\tau - 1)\})$$

prev_tick(τ): Given a system time instant τ, returns the application time value of the most recent tuple that has arrived before τ for which the result of the tick was non-empty. If no such tuple exists, it returns t_0.

$$prev\_tick(\tau) = \{max(t_0, app(max(x | x < \tau \wedge Tick(x) \neq \emptyset)))\}$$

Based on the above, we will now formulate *Tick* for each tick model. All formulas follow a similar structure.

In a tuple-driven system, Tick is triggered under two conditions: (i) if a tuple arrives whose t^{app} is the same as the previous tick time, or (ii) if a tuple arrives whose t^{app} is greater than the previous tick time. The former ensures that the system reacts to each tuple in a simultaneous sequence, whereas the latter ensures that the system also reacts to the application time instants where there might be a gap.

$$Tick(\tau) = \begin{cases} \{Report(app(\tau), \tau)\} & \text{if } S_I(\tau) \neq \emptyset \wedge \\ & \quad prev\_tick(\tau) = app(\tau) \\ \bigcup\limits_{x = prev\_tick(\tau)}^{x < app(\tau)} Report(x, \tau) & \text{if } S_I(\tau) \neq \emptyset \wedge \\ & \quad app(\tau) > prev\_tick(\tau) \\ \emptyset & \text{otherwise} \end{cases}$$

In a time-driven system, there is no need to react to each tuple in a simultaneous sequence separately, therefore, the first condition in the tuple-driven case is skipped. On the other hand, the second condition needs to be triggered if a tuple with a new t^{app} arrives (which means that t^{app} has advanced, to which the system must react).

$$Tick(\tau) = \begin{cases} \bigcup\limits_{x = prev\_tick(\tau)}^{x < app(\tau)} Report(x, \tau) & \text{if } S_I(\tau) \neq \emptyset \wedge \\ & \quad app(\tau) > prev\_app(\tau) \\ \emptyset & \text{otherwise} \end{cases}$$

Finally, a batch-driven system acts like a modified tuple-driven system. We need to check both the condition for simultaneous tuples as well as for a tuple with a new t^{app} arriving. The only difference is that we need to additionally check if the new tuple arrival initiates a new batch by checking if the new batch-id is greater than the previous one.

τ	S(v, t^{app}, t^{sys}, bid)	tuple-driven	time-driven	batch-driven
$\tau = 30$	$(a, 1, 30, b1)$	Report(x, 30), $\forall x \in [t_0, 1)$	Report(x, 30), $\forall x \in [t_0, 1)$	Report(x, 30), $\forall x \in [t_0, 1)$
$\tau = 40$	$(b, 2, 40, b2)$	Report(x, 40), $\forall x \in [1, 2)$	Report(x, 40), $\forall x \in [1, 2)$	Report(x, 40), $\forall x \in [1, 2)$
$\tau = 45$	$(c, 2, 45, b2)$	Report(2, 45)	-	-
$\tau = 60$	$(d, 2, 60, b3)$	Report(2, 60)	-	Report(2, 60)
$\tau = 80$	$(e, 2, 80, b3)$	Report(2, 80)	-	-
$\tau = 90$	$(f, 4, 90, b4)$	Report(x, 90), $\forall x \in [2, 4)$	Report(x, 90), $\forall x \in [2, 4)$	Report(x, 90), $\forall x \in [2, 4)$

Table 1: Tick example

$$Tick(\tau) = \begin{cases} \{Report(app(\tau), \tau)\} & \text{if } S_I(\tau) \neq \emptyset \wedge \\ & \quad prev\_tick(\tau) = app(\tau) \wedge \\ & \quad batch(\tau) > prev\_batch(\tau) \\ \bigcup_{x=prev\_tick(\tau)}^{x<app(\tau)} Report(x, \tau) & \text{if } S_I(\tau) \neq \emptyset \wedge \\ & \quad batch(\tau) > prev\_batch(\tau) \\ \emptyset & \text{otherwise} \end{cases}$$

In order to illustrate how our *Tick* formulation works, in Table 1, we present a sample trace of the model for the scenario shown in Figure 3. The table shows when exactly each of the three models triggers *Report* and with which time values. As expected, the time-driven model invokes *Report* only when time advances, one for each time point including the gap time ($t^{app} = 3$). The tuple-driven model, on the other hand, invokes *Report* at every new tuple arrival. Finally, the batch-driven model invokes *Report* at every new batch arrival as well as time advance ($t^{app} = 3$).

3.3 Discussion

As we will show in the next section, SECRET can successfully explain the execution behavior of three real, representative SPEs which are quite different from each other. Its **expressivity** is the result of careful design choices. For example, Scope is defined for the earliest open window making it easy for SECRET to capture the behavior of systems that report partial window results (e.g., Coral8) as well as full window results (e.g., StreamBase). Our model embraces **simplicity** wherever possible. For example, Scope refers to a single active window and we use the t_0 parameters to adjust window intervals rather than distinguishing between forward/backward window construction, greatly simplifying our Scope formulation. We do not model real-time effects (e.g., timeouts, synchronizing application time with system time, etc.), as this would make it difficult to define a clean, predictable, and repeatable semantics (see Appendix B.1). SECRET's features are **orthogonal** and **extensible**. Each behavior seen in our experiments is explained in a single way by the model, and the various aspects can be combined as needed. For example, different systems may use different combinations of the evaluation strategies for the Report dimension. Meanwhile, more values or more dimensions can be added if we encounter a new SPE with new features that cannot be expressed (see Appendix B). SECRET also is **clear**: it decouples the different levels of concerns from each other and treats them separately. For example, Scope handles query-level issues, Content handles data-level issues, and Report and Tick handle system-level issues. Likewise, Scope and Content capture non-operational effects of query processing, whereas Report and Tick capture the operational ones. Thus, SECRET embodies the characteristics we desired in our model.

4. EXPERIMENTS

We have tested our model with three different systems: the STREAM open-source academic prototype [5], Coral8 Version 5.5 [1], and StreamBase Version 6.4 [6] (see Appendix B.3 for a discussion of this choice and extensions to other systems). Each system's SECRET parameter values are obtained after a careful analysis. Table 2 summarizes our findings; details can be found in Appendix A.1.

We apply our model on three query examples that we ran on the SPEs to show how SECRET can explain the differences in their answers. In these experiments, we compare the query results produced by the SPEs with those predicted by our SECRET model simulator.

Experiment 1: Difference in Window Construction

Example 1 of Section 2 shows that the same query on the same input stream might produce different results when it is run multiple times on a given SPE (StreamBase). The only configuration difference from one run to another seems to be the initial value of the application time attributes of the tuples. Here is how SECRET explains the situation:

First, we found that before each run, StreamBase reassigns application time values to tuples according to the current system time. It takes the tuples' own source-assigned time values as the basis, using exactly the same time differences between consecutive tuples (i.e., only the absolute time values change; their relative values stay the same).

Second, t_0 in StreamBase depends on the absolute value of the application time of the first tuple (denoted by t_{t1}), as formulated in Table 2. The value t_0 affects window scopes and contents, and therefore, the query results.

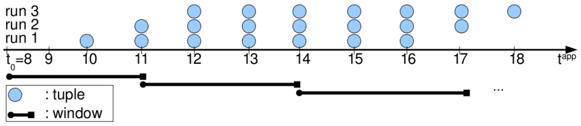


Figure 4: Window contents for Experiment 1

In all three runs, when we plug in the values $\omega = 3$, $\beta = 3$, and t_{t1} into the t_0 formula for StreamBase, we in fact get the same t_0 value of 8 seconds. Accordingly, the scopes of the windows are also the same for these three runs as shown in Figure 4. However, what falls into the scopes from the input (i.e., SECRET's Content) is different. This is why the average results are different. Note that in run 3, the first window is in fact empty. Therefore, the first result we see in Output3 in Example 1 of Section 2 is the average result for the second window, as StreamBase does not operate on and produce results for windows with empty contents.

Experiment 2: Difference in Window Report

Example 2 of Section 2 shows that the same query on the same input stream might produce different results on different SPEs. While STREAM and Coral8 gave similar

SPE	t_0 (Scope)	Report	Tick
STREAM [5]	$t_{t1} - \omega$	window close & content change & non-empty & λ=1	time-driven
Coral8 [1]	$\lceil \frac{t_{t1}-\omega}{\beta} \rceil$ β-1	content change & non-empty & λ=1	batch-driven
StreamBase [6]	$\lceil \frac{t_{t1}-\omega}{\beta} \rceil$ β-1	window close & non-empty & λ=1	tuple-driven

Table 2: SECRET parameters of STREAM, Coral8, and StreamBase

query results for this query, StreamBase gave a different one. We present how SECRET explains this situation.

Step 1. Tick: The input stream in Example 2 does not include any simultaneous tuples. This means that each tuple arrival will correspond to the arrival of a new application time value as well as to the arrival of a new batch. Therefore, all three systems will behave exactly the same in terms of their Ticks (for details, see Appendix A.2.1).

Step 2. Scope and Content: According to Table 2, t_0 equals 25, 24, and 24 for STREAM, Coral8, and StreamBase, respectively. These values are obtained by plugging in the values for $t_{t1} = 30$, $\omega = 5$, and $\beta = 1$ in the t_0 formulas. To calculate the window scopes themselves, SECRET uses the *Scope* formula presented in Section 3.2.1. Figure 5 depicts the scopes and contents of the windows for the engines. Due to their t_0 values, Coral8 and StreamBase share the same window scopes and contents, while STREAM excludes the very first scope. However, since for the given input, the first scope is empty anyway, in practice, there is no difference in the scopes and contents of the three engines.

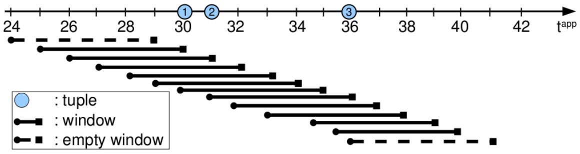


Figure 5: Window contents for Experiment 2

Step 3. Report: Table 3 illustrates how reporting is triggered for the three systems (t_n denotes the n^{th} tuple). StreamBase reports only the contents of non-empty windows when they close, which happens at every second until the last tuple expires. On the other hand, Coral8 reports only the contents of non-empty windows when their contents change, which happens at time 30, 31, 35, 36, and so on. Finally, STREAM reports only the contents of non-empty windows when they close and their contents change, which, in this example, happens at exactly the same time points as for Coral8. Hence, differences in the Report parameters explain why the three systems produce different answers (for details, see Appendix A.2.2).

t^{app}	StreamBase	STREAM	Coral8
30	$\{t_1\}$	$\{t_1\}$	$\{t_1\}$
31	$\{t_1, t_2\}$	$\{t_1, t_2\}$	$\{t_1, t_2\}$
32	$\{t_1, t_2\}$	-	-
33	$\{t_1, t_2\}$	-	-
34	$\{t_1, t_2\}$	-	-
35	$\{t_2\}$	$\{t_2\}$	$\{t_2\}$
36	$\{t_3\}$	$\{t_3\}$	$\{t_3\}$

Table 3: Contents reported for Experiment 2

Experiment 3: Difference in Tick

In this this last experiment, we show that different tick models can yield different results for a given query on a given input stream. For this experiment, we chose a fixed input with simultaneous tuples and a batch-driven system, Coral8, but we configured the batches in three different ways to create the three different tick scenarios. We did this using Coral8's *atomic bundling* mechanism [1]. If each individual tuple is placed in a separate bundle, then Coral8 acts like a tuple-driven system, since it then reacts to every new input arrival (i.e., tuple ∼ batch). On the other hand, if we place all of the simultaneous tuples in a common bundle, Coral8 works like a time-driven system, since it then reacts to all such tuples collectively when the time advances (i.e., time-unit ∼ batch). For all other configurations, Coral8 behaves like a normal batch-driven system.

More concretely, for a fixed input stream of tuples S(v,t^{app}) = {(10, 3),(20, 5),(30, 5),(40, 5),(50, 5),(60, 7),...}, we obtained the following three configurations: STuple (one tuple per batch), STime (all simultaneous tuples in the same batch), and SBatch (two simultaneous tuples per batch). With the batch-id (shown in bold), the input streams are:

$$\begin{aligned}\text{STuple}(v,\, t^{app},\, \text{bid}) = \{&(10, 3, \mathbf{1}), (20, 5, \mathbf{2}), (30, 5, \mathbf{3}), (40, 5, \mathbf{4}),\\ &(50, 5, \mathbf{5}), (60, 7, \mathbf{6}),...\}\\ \text{STime}(v,\, t^{app},\, \text{bid}) = \{&(10, 3, \mathbf{1}), (20, 5, \mathbf{2}), (30, 5, \mathbf{2}), (40, 5, \mathbf{2}),\\ &(50, 5, \mathbf{2}), (60, 7, \mathbf{3}),...\}\\ \text{SBatch}(v,\, t^{app},\, \text{bid}) = \{&(10, 3, \mathbf{1}), (20, 5, \mathbf{2}), (30, 5, \mathbf{2}), (40, 5, \mathbf{3}),\\ &(50, 5, \mathbf{3}), (60, 7, \mathbf{4}),...\}\end{aligned}$$

Then we ran a query on these three input streams, which continuously computes the sum of values over a sliding window of size 4 seconds, yielding the following results:

```
Coral8 Output(STuple) = {(10), (30), (60), (100), (150), ...}
Coral8 Output(STime)  = {(10), (150), ... }
Coral8 Output(SBatch) = {(10), (60), (150), ...}
```

Step 1. Tick: Table 4 shows the execution trace of our batch-driven Tick formula on our example for `STuple`, `STime`, and `SBatch`, each simulating a different Tick value. The tick condition returns true, if the batch-id of the newly arrived tuple is greater than the previous tuple's batch-id. `STuple` ticks at every new tuple arrival, `STime` ticks every time t^{app} changes, and `SBatch` ticks at every new batch arrival.

(tuple, t^{app})	tuple-driven	time-driven	batch-driven
(t_1, 3)	Report(2)	Report(2)	Report(2)

(t_2, 5)	Report(3)
	Report(4)
		Report(3)
		Report(4)
			Report(3)
			Report(4)

(t_3, 5)	Report(5)	-	-
(t_4, 5)	Report(5)	-	Report(5)
(t_5, 5)	Report(5)	-	-

(t_6, 7)	Report(5)
	Report(6)
		Report(5)
		Report(6)
			Report(5)
			Report(6)

Table 4: Ticks for Experiment 3

Step 2. Scope and Content: Given $t_{t1} = 3$, $\omega = 4$, and $\beta = 1$, t_0 can be calculated as -2 seconds for Coral8. Accordingly, Figure 6 depicts the window scopes and contents based on the *Scope* formula of Section 3.2.1.

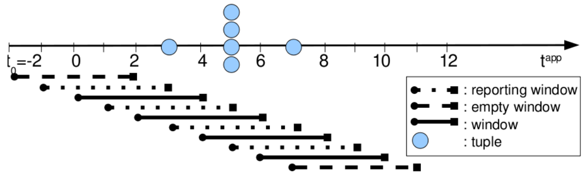


Figure 6: Window contents for Experiment 3

Step 3. Report: In Figure 6, windows which have produced results because of changes in their contents are denoted with fine dots. Table 5 depicts the result of the aggregation run on these windows at the time instants when Report is invoked (Table 4). SECRET correctly models the result for STuple as {10, 30, 60, 100, 150, ...}, for STime as {10, 150, ...}, and for SBatch as {10, 60, 150, ...} (for details, see Appendix A.3).

tuple	Report	tuple-driven	time-driven	batch-driven
t_1	Report(2)	-	-	-
t_2	Report(3)	{10}	{10}	{10}
t_3	Report(5)	{30}	-	-
t_4	Report(5)	{60}	-	{60}
t_5	Report(5)	{100}	-	-
t_6	Report(5)	{150}	{150}	{150}

Table 5: Results reported for Experiment 3

5. RELATED WORK

It is hard to get information about underlying formal models used by current commercial systems [1, 2, 3, 4, 6, 7]. Each system seems to use a different model, and the query results that they generate are not easy to compare. Jain et al. [13] tried to reconcile the differences across two of these commercial systems, Oracle CEP and StreamBase. They only consider the way that window execution is triggered. Though an important first step, this work focuses on only one aspect of execution behavior (i.e., Tick in SECRET), just one of the aspects our model captures and explains.

A few recent studies have tried to offer cleaner abstract models without necessarily being tied to a specific system implementation [14, 15, 16, 18]. We present a comparison of these models to SECRET in Appendix D.

6. CONCLUSION AND FUTURE WORK

The SECRET model describes important differences in the semantics underlying stream processing models. We developed SECRET by studying both academic and industrial SPEs. We have shown, through examples and experimentation, how SECRET can be used to understand, compare, and predict the behavior of diverse SPEs. Our model is unique to date in its comprehensive consideration of common differences in execution models, differences that can lead to surprisingly varied results when even simple stream queries are executed on different engines.

We have focused on queries with time-based windows and unary operators only, excluding real-time effects, and analyzed three representative SPEs. Appendix B discusses some preliminary results on extending SECRET for other query types, SPEs, and system considerations. In addition to exploring and explaining the differences between different engines, SECRET can be used to find equivalences between them. Such equivalences can be used to devise query rewrite and transformation rules (e.g., providing a foundation for query optimization in a federation of SPEs [10]). We are currently pursuing this interesting research direction.

7. REFERENCES

[1] Coral8, Inc. http://www.coral8.com/.

[2] IBM System S. http://www.ibm.com/.

[3] Microsoft SQL Server StreamInsight Technology. http://www.microsoft.com/sqlserver/2008/en/us/R2-complex-event.aspx.

[4] Oracle CEP. http://www.oracle.com/technologies/soa/complex-event-processing.html.

[5] Stanford Stream Data Manager. http://infolab.stanford.edu/stream/.

[6] StreamBase Systems, Inc. http://www.streambase.com/.

[7] Truviso, Inc. http://www.truviso.com/.

[8] D. Abadi and et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), 2003.

[9] A. Arasu and et al. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2), 2006.

[10] I. Botan and et al. Design and Implementation of the MaxStream Federated Stream Processing Architecture. Technical Report TR-632, ETH Zurich Department of Computer Science, June 2009.

[11] S. Chandrasekaran and et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, 2003.

[12] B. Gedik and et al. SPADE: The System S Declarative Stream Processing Engine. In *ACM SIGMOD Conference*, 2008.

[13] N. Jain and et al. Towards a Streaming SQL Standard. In *VLDB Conference*, 2008.

[14] J. Kramer and B. Seeger. Semantics and Implementation of Continuous Sliding Window Queries over Data Streams. *ACM TODS*, 34(1), 2009.

[15] L. Li and et al. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *ACM SIGMOD Conference*, 2005.

[16] D. Maier and et al. Semantics of Data Streams and Operators. In *ICDT Conference*, 2005.

[17] R. Motwani and et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR Conference*, 2003.

[18] K. Patroumpas and T. Sellis. Window Specification over Data Streams. In *EDBT Workshops*, 2006.

[19] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. In *ACM PODS Conference*, 2004.

[20] M. Tsimelzon. On Streaming SQL Standards. http://www.coral8.com/blogs/blog-entry/streaming-sql-standards, September 2008.

APPENDIX

A. EXPERIMENT DETAILS

In this section, we discuss some of the details related to the setup and the experiments of Section 4.

A.1 Setup and Methodology

The result of a query is affected by three main factors: the input, the query, and the system. The success criteria for our model is to find at least one SECRET parameter setting for a given system which can explain the execution behavior of that system for a common set of windowed queries and input configurations. We have done extensive experiments with three systems to find their respective values for each of the four SECRET parameters.

In these experiments, we varied the input data and queries carefully, as they also affect the query results. More specifically, the input stream should have irregularities due to gaps in application time or due to simultaneous tuples with common application times. Queries with time-based windows should be analyzed under three categories: sliding windows with $\beta = 1$, sliding windows with $1 < \beta < \omega$, and tumbling windows with $\beta = \omega$.

Not all systems support all possible input/query configurations (due to capability differences or incomplete implementation) and this might create some limitations. For example, the available release of the STREAM prototype only supports time-based sliding windows with $\beta = 1$. Therefore, we can only test it with this type of query. However, the same system has an associated research paper that describes the more general theoretical model underneath its system implementation [9], which we also used as a reference.

After analyzing the execution of all supported configurations on each system, we obtained the SECRET parameters for these systems as shown in Table 2. Next, we describe how we arrived at these values.

For Scope, the system only influences the choice of the t_0 parameter (i.e., the application time instant for the start of the first window). We obtained the t_0 formulas in Table 2 by running different queries with various size and slide value combinations on a varied set of input streams multiple times.

Report values for the SPE set have been found, again based on the queries that we ran on these SPEs.

For Tick, related work has already revealed the tick models for StreamBase (tuple-driven) and STREAM (time-driven) [13]. Additionally, we found that Coral8 uses a batch-driven model, based on our own experiments, personal communication with members of the Coral8 support team, as well as a blog discussion provided at Coral8's website [20]. In Coral8, batches form automatically depending on how the input adapter feeds tuples into the Coral8 server. Lastly, we note that all tick models behave similarly when the input is regular. To identify the differences, we needed to feed irregular input (with simultaneity in particular) to our sample queries.

A.2 Experiment 2

A.2.1 Tick

All three systems in Example 2 have different tick values: STREAM is time-driven, Coral8 is batch-driven, and StreamBase is tuple-driven. However, differences in the ticks' effect can only be seen when there are simultaneous tuples in the input stream. When each tuple has a unique application time, STREAM, Coral8, and StreamBase will have the same tick behavior; since the arrival of a new tuple corresponds to the arrival of a new application time as well as the arrival of a new batch at the same time. Table 6 illustrates the execution trace of our Tick formula (Section 3.2.4) on Example 2 for STREAM, Coral8, and StreamBase, respectively. One can quickly see that on that input, all three systems "tick" in exactly the same way.

(tuple, t^{app})	STREAM	Coral8	StreamBase
(t_1, 30)	Report(29)	Report(29)	Report(29)
(t_2, 31)	Report(30)	Report(30)	Report(30)

(t_3, 36)	Report(31)
	Report(32)
	Report(33)
	Report(34)
	Report(35)
		Report(31)
		Report(32)
		Report(33)
		Report(34)
		Report(35)
			Report(31)
			Report(32)
			Report(33)
			Report(34)
			Report(35)

...			

Table 6: Tick for Experiment 2

A.2.2 Report

Table 7 displays the execution trace for the report parameter in Experiment 2 at the application time instants when reporting is called. If the reporting condition is true for a window, the scope and the actual content of the window are calculated and the content then becomes visible to the aggregation operator. The table shows that SECRET models STREAM's and Coral8's results as {10, 15, 20, 30, ...}, and StreamBase's results as {10, 15, 15, 15, 15, 20, 30, ...}, which are the actual results produced by those engines.

A.3 Experiment 3

Table 8 presents the execution of the SECRET model for Experiment 3. The first column in the table refers to the arrival of the tuple. For instance when the third tuple (t_3) arrived, STuple ticked since the batch-id of the tuple was greater than that of the previously seen tuple. Consequently, Report was called, and the content of the window (the table shows only the v attribute of the tuples) became visible for the evaluation of the sum operator. On the other hand, STime and SBatch did not tick, since the batch-id of the tuple remained the same. As a result of different Tick models, the SPEs called Report at different time instants, which led to different actual window contents as shown in Table 8.

B. EXTENDING SECRET

As we explained earlier, SECRET captures factors that affect a query result including those that stem from the query (ScopE), the input (Content), and the system (REport, Tick). In this section, we discuss how SECRET can be extended further with respect to these three aspects.

B.1 Input Aspects

B.1.1 System-time-based Windows

In this paper, we have focused on windows which are defined based on on application time. Time-based windows can also be defined using system time, which is the timestamp assigned to the tuples at their arrival to the system. In this case, $t^{app} = t^{sys}$. This case can also be handled by

t^{app}	STREAM			Coral8			StreamBase		
	Report?	Scope	Content	Report?	Scope	Content	Report?	Scope	Content
29	No	-	-	No	-	-	No		
30	Yes	(25 30]	{10}	Yes	(25 30]	{10}	Yes	(25 30]	{10}
31	Yes	(26 31]	{10, 20}	Yes	(26 31]	{10, 20}	Yes	(26 31]	{10, 20}
32	No	-	-	No	-	-	Yes	(27 32]	{10, 20}
33	No	-	-	No	-	-	Yes	(28 33]	{10, 20}
34	No	-	-	No	-	-	Yes	(29 34]	{10, 20}
35	Yes	(30 35]	{20}	Yes	(30 35]	{20}	Yes	(30 35]	{20}
36	Yes	(31 36]	{30}	Yes	(31 36]	{30}	Yes	(31 36]	{30}
...	...	...	...	...	...	...	...	...	...

Table 7: Report for Experiment 2

	tuple-driven (STuple)			**time-driven (STime)**			**batch-driven (SBatch)**		
tuple	Tick?	Report	Content	Tick?	Report	Content	Tick?	Report	Content
t_1	Yes (1>0)	-	-	Yes (1>0)	-	-	Yes (1>0)	-	-
t_2	Yes (2>1)	Report(3)	{10}	Yes (2>1)	Report(3)	{10}	Yes (2>1)	Report(3)	{10}
t_3	Yes (3>2)	Report(5)	{10, 20}	No	-	-	No	-	-
t_4	Yes (4>3)	Report(5)	{10, 20, 30}	No	-	-	Yes (3>2)	Report(5)	{10, 20, 30}
t_5	Yes (5>4)	Report(5)	{10, 20, 30, 40}	No	-	-	No	-	-
t_6	Yes (6>5)	Report(5)	{10, 20, 30, 40, 50}	Yes (3>2)	Report(5)	{10, 20, 30, 40, 50}	Yes (4>3)	Report(5)	{10, 20, 30, 40, 50}

Table 8: Comparing different input batch configurations (i.e., Tick models) in Coral8

SECRET, since in practice, it does not matter whether the timestamps are assigned at the source or by the system.

There are a handful of systems in which time-based windows are constructed based on the system time at the point when the tuples hit each window-based operator. We have excluded this case from our model, since it has a non-repeatable semantics (e.g., the behavior would be sensitive to the operator scheduling policy in the system).

B.1.2 Synchronized Timestamps

In our current model, time information can only be gathered through tuples. This strategy might delay the processing if there is a gap between tuple arrivals. In order to prevent the delay, real systems use various mechanisms to synchronize the application time of tuples with the actual system time (e.g., heartbeats in STREAM [19], MAXDELAY in Coral8 [1], and TIMEOUT in StreamBase [6]). Extending our model to include synchronized timestamps is straightforward: if a maximum delay threshold is known in advance, dummy tuples with punctuations can be injected into the input stream and the application time can be advanced without waiting for the actual delayed tuples to arrive.

B.1.3 Out-of-order Streams

SECRET makes an assumption that tuple arrivals are always totally ordered. This assumption might seem like a limitation, but since SECRET aims to explain the execution model differences among SPEs, it has the finest granularity for tuple order information. Please note that some engines impose a total order on the input data based on arrival (such as StreamBase) before processing. The total order assumption might not be met in practice because of network latencies and distributed data sources. In case of out of order tuples, the system can buffer input tuples for a maximum amount of time and then reorder them [19].

B.2 Query Aspects

B.2.1 Tuple-based Windows

In this paper, we mainly focused on presenting SECRET for analyzing the execution semantics of queries with time-based windows. Our model can also be used to explain the semantics of queries over tuple-based windows. A tuple-based window is defined by tuple arrival, rather than by time units (size and slide) First of all, since window type is a query property, we can reuse Tick and Report (which relate to system properties) without much change. The only change required is that the window close condition in Report should be defined in terms of Content size rather than Scope size, since what counts in the window size is now the actual number of tuples rather than the size of the time interval. Similarly, since Content is an input-related property, we can reuse it for tuple-based windows to a large extent. One exceptional case that arises with Content is that time- and batch-driven systems are known to have an “evaporating tuples” situation when there are more simultaneous tuples around than the window size can accommodate [9, 13]. We address this requirement by adding a tuple selection function into the Content formula that allows tuple selection when necessary. Finally, since Scope is a query-related property, we have reformulated it in the following way. For tuple-based windows, since window size is defined in terms of the number of tuples rather than the size of the application time interval, windows must be constructed on the tuple-id domain instead of the application time domain.

B.2.2 Binary Operators

This paper has only focused on unary operators as they are the foundation for stream queries. We can also extend our model to handle binary operators (such as joins). Join operators are fundamentally different from unary sliding window operators, as they involve two inputs with two windows. Our model can directly explain how each of those windows are populated with input tuples (i.e., Tick, Scope, and Content can be used without any change). However, one additional issue to consider is when to make the windows of the two input streams visible to the join operator. The Report definition must be extended to address this issue.

B.3 System Aspects

As has been shown in earlier sections of this paper, we have tested our model with three different SPEs (StreamBase, STREAM, and Coral8), which are representative in the area and are widely used. StreamBase commercialized the Aurora/Borealis academic prototypes, and therefore, its basic execution model descends from those of these two systems. Similarly, the Oracle CEP engine directly follows STREAM's query execution model [13]. Lastly, Coral8 is a widely used commercial system and its model is also substantially different from the two families of SPEs mentioned above (also see Figure 2). Overall, we believe that we cover a significant variety of SPE models in this paper.

Although we cover a major subset of SPEs and their execution models, it would be interesting to expand our experimental set even further to include other SPEs as well. The first step in analyzing an SPE with our SECRET model is to find out what value each SECRET parameter should take for the given system. If the required knowledge about the system is not readily available, these values can be obtained by executing a set of queries against a range of inputs. The input stream should have irregularities due to gaps in application time and due to simultaneous tuples with common application times. Furthermore, queries should include some with windows that slide each time unit, windows with slide parameters (i.e., having a slide value greater than minimum window unit, but less than the window size), or tumbling windows (i.e., having a slide value of the same size as the window size). By executing different configurations of these input and query properties, the SECRET parameter values can be revealed.

C. POTENTIAL USES OF SECRET

The original motivation behind our SECRET model was to analyze the query execution behavior of SPEs and their differences. However, the model can actually be utilized for several other important purposes beyond this goal.

First, as we briefly mentioned in Section 6, SECRET can also be useful for discovering equivalences among SPEs. These equivalences can then be used for rewrite-based query optimization in integrated stream processing settings (e.g., MaxStream [10]). Likewise, equivalences can also be used to semantically translate a query running on one SPE to enable porting of applications to other SPEs. For instance, if it is known that no simultaneous tuples and gaps can occur in a given input stream, query translation is possible even among SPEs having different Tick values, as in this case, tuple-, time-, and batch-driven SPEs would all behave similarly in terms of their Tick behavior.

Furthermore, SECRET parameters cover all the key aspects of query execution: input, query, and system. Based on different input and query requirements that originate from the applications, different SPEs might be preferred for a given application. For example, if an application has simultaneous tuples (e.g., position reports of cars in dense areas of traffic) and there is no well-defined order among them, using an SPE following a time-driven Tick model might be a better option. On the other hand, if it is possible to define a further ordering among the simultaneous tuples (e.g., based on car types), an SPE following a batch-driven Tick model might be preferable, as it could produce the result faster than a time-driven SPE. Different reporting strategies may also be better suited for different application needs. For instance, for slowly changing datasets and queries with small window sizes to process them, an SPE having window content change as its reporting strategy might be preferred over one with a window close strategy in order to avoid repeated results. On the other hand, for join queries, using an SPE with window close as its reporting strategy might provide more insight in understanding the query results compared to one using a reporting strategy based on window content change.

D. EXTENDED RELATED WORK

In this section, the difference between SECRET and a subset of the previously proposed models is discussed.

As a first generation research system, STREAM's CQL provides a formal model based on the relational model [9]. In addition to introducing the stream data type and mapping operations between streams and relations, CQL has also introduced the notion of time into the relational model, which essentially adds the "time-driven" continuous query execution semantics. However, CQL semantics alone is not sufficient to explain all the different behaviors that we see from different SPEs today (e.g., other Tick models).

In the aftermath 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, as we summarize next.

Maier et al. [16] generalizes the denotational semantics approach of STREAM CQL, focusing on defining the meaning of a stream itself rather than the complete query execution semantics. Li et al. [15] have proposed a framework for defining window semantics based on three functions: *windows*, *extent*, and *wids*, where the window semantics is described independent of the execution model. Our work differs from Li et al. [15], 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 for a CQL-like model [18]. The model is based on a time-parameterized scope function that specifies a time-based window's size and progression in time. This work is not based on real system implementations, and whether the proposed formalism is powerful enough to capture the existing systems' behaviors is not known.

Most recently, Kramer and Seeger have proposed a pair of logical and physical operator algebras for stream operators, applying ideas from temporal databases [14]. Their approach is similar to that of the STREAM team, 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 their PIPES system. This paper covers similar semantic issues as in the CQL model and does not provide constructs to explain the operational aspects of other SPE systems.

E. ACKNOWLEDGMENTS

We would like to thank John Wilkes and Olga Irzak for their helpful comments on earlier versions of this paper; Ghislain Fourny for his input about our formalism; Coral8 and StreamBase's technical support teams for answering our questions; and our former collaborators at SAP China and Korea Labs for useful discussions. This work has been supported in part by the following grants: Swiss NSF NCCR MICS 5005-67322, Swiss NSF ProDoc PDFMP2-122971/1, and ETH Zurich Enterprise Computing Center (ECC) SAP industrial partner grant DE-2008-022. Miller was partially supported by NSERC.