Towards a Streaming SQL Standard

Namit Jain
Shailendra Mishra
Anand Srinivasan
Oracle Corp.
Redwood Shores, CA

Johannes Gehrke
Cornell University
Jennifer Widom
Stanford University

Hari Balakrishnan
Uğur Çetintemel
Mitch Cherniack
Richard Tibbetts
Stan Zdonik
StreamBase, Inc.
Lexington, MA

ABSTRACT

This paper describes a unification of two different SQL extensions for streams and its associated semantics. We use the data models from Oracle and StreamBase as our examples. Oracle uses a time-based execution model while StreamBase uses a tuple-based execution model. Time-based execution provides a way to model simultaneity while tuple-based execution provides a way to react to primitive events as soon as they are seen by the system.

The result is a new model that gives the user control over the granularity at which one can express simultaneity. Of course, it is possible to ignore simultaneity altogether. The proposed model captures ordering and simultaneity through partial orders on batches of tuples. The batching and the ordering are encapsulated in and can be modified by means of a powerful new operator that we call SPREAD. This paper describes the semantics of SPREAD and gives several examples of its use.

1. INTRODUCTION

Stream processing has been around for a while (e.g., [2, 6, 12, 14, 19, 23]). There have been many research prototypes and now there are several industrial products, some of which are derived from their respective academic heritage [15, 19]. Each system supports a stream-oriented query language. They are essentially all SQL extensions that incorporate a notion of a window on a stream as a way to convert an infinite stream into a finite relation in order to apply relational operators.

As the industry matures, there is a need for a single standard language. It is tempting to say that such a language is simply an agreement over simple syntactic differences. About a year ago, Oracle and StreamBase embarked on a project to create a convergence language in which these simple differences would be resolved. What emerged was a realization that there were fundamental differences in the basic model that made convergence difficult. From both sides, there were things that one model could do that the other model could not do. What was needed was a new model that spanned the original two.

This paper describes these semantic differences and proposes a new model that seems to give us the best of both worlds. In fact, we believe that it provides new functionality that neither model could provide heretofore.

While it is not our intension to present a complete language standard in this paper, we believe that by leveling the playing field at the model level, such a standard is much closer.

In this paper, we only present our final solution. It is worth pointing out that several other alternative formulations were proposed along the way, including one that increased the number of window types to cover all combinations of the two systems' semantics. These other proposals, while workable, seemed to be "big switch" solutions. That is to say, by using one subset of features you would get the Oracle model, while using a disjoint subset of features, you would get the StreamBase model. We believe that the current solution is much less of "big switch" in that it cleanly integrates the two, allowing the user to smoothly move from one to the other and to arbitrarily many points in between.

The remainder of this paper is organized as follows. In Section 2, we give an informal overview of some cases neither data model was expressive enough to handle by illustrating them through queries on the widely-used Linear-Road Benchmark. In Section 3, we pinpoint the differences between the models through several abstract examples. In Section 4, we describe the proposed unification model in detail. In Section 5, we present the syntax and semantics of a new operator called SPREAD and illustrate it using examples. In Section 6, we revisit our examples from Section 3 in the light of the proposed model. We illustrate the key components of the new model in the context of a network intrusion detection query in Section 7. We comment on implementation issues in Section 8, discuss related work in Section 9, and present concluding remarks in Section 10.

VLDB '08, August 24-30, 2008, Auckland, New Zealand.

2. BACKGROUND AND MOTIVATION

2.1 A Brief Description of the Two Models

In what follows, we assume a basic understanding of stream query languages and the notion of a window.

The Oracle Model

The Oracle model is based on CQL [1]. In this model, tuples have timestamps, and there is no ordering between tuples with identical timestamps. Tuples with the same timestamp value are said to be *simultaneous*. The processing follows a *time-driven model*: the value of a window is computed at each timestamp by evaluating the history of its input streams as of that timestamp. Furthermore, all processing at a given timestamp happens instantaneously. Thus, all derived tuples also get labeled with the same timestamp value. In this way, each relation has a value as of each timestamp.

The StreamBase Model

In the StreamBase model, tuples also have timestamps, some of which might be identical. The conceptual execution, however, assigns an internal *rank* to tuples based on their arrival order to the system and ensures that an input tuple, as well as all tuples derived from its processing, are processed as far as possible before another input tuple with a higher rank. Unlike the Oracle processing model, which is time-driven, the StreamBase model is *tuple-driven*: each relation (logically) has a value as of each tuple. This value is obtained by evaluating the window on the history of its input stream(s) as of that tuple.

It is important to note that despite StreamBase's tuple-at-a-time execution model, the values of relations need not change on every tuple. In fact, for some window definitions (e.g., time-based windows), the window contents will in general not change on the arrival of each tuple. A time-based window will only change when it has collected all the tuples within the next timestamp range.

2.2 Linear-Road Example

In this section, we will discuss a few queries that are based on the Linear Road Benchmark (LRB) [3]. We show how the current Oracle and StreamBase models each fall short on some of these queries. The purpose of this section is to introduce some of the shortcomings of the two models that we study here. Section 3 will look at these issues in more depth. Here and in the remainder of the paper, we take some liberties with the exact syntax of the two languages, but this should be clear from the context.

A position report in LRB has the form:

(Type, Time, VID, Spd, XWay, Lane, Dir, Seg, Pos).

For ease of discussion, we use an abbreviated version as follows:

(VID, Spd; Time)

Here:

- **Time** (0 . . . 10799) is a timestamp identifying the time at which the position report was emitted,
- **VID** (0 . . . MAXINT) is an integer vehicle identifier identifying the vehicle that emitted the position report, and
- **Spd** (0 . . . 100) is an integer reflecting the speed of the vehicle (in MPH).

Scenario

Let S be the stream of car position reports (as in LRB). Below we show a sample of this data for two time units.

S (VID, Spd; Time) = (1,50;1) (2,50;1)
(1,50;2) (2,50;2) (3,20;2)

Neither time-basis nor tuple-basis works

We would like to find, for each input report on stream S, the average speed of all cars as of the current timestamp. Let P1 be that stream.

Since we want a result every time a new tuple arrives to the system, we expect 5 results on the above data. Computing the average speed (as of each arrival) will drive us to use a row-based window defined as S[rows 1] as in

P1 = **ISTREAM** (**SELECT** avg (Spd) **as** AvgSpd, Time
FROM S[rows 1])

which will produce different results for P1 with the two original models:

Oracle

P1 (AvgSpd; Time) = (50;1) (20;2)

StreamBase

P1 (AvgSpd; Time) = (50;1) (50;1) (50;2) (50;2) (20;2)

Notice that both of these answers are wrong. Oracle does not produce 5 answers because it is forced to evaluate by time. StreamBase produces 5 values, but the "rows 1" window will not allow us to group the tuples by time unit.

Tuple-basis doesn't work; Time-basis works

On the other hand, suppose the query wants to report an answer for each time-interval, both languages will define a window as S[range 1] as in

P1 = **ISTREAM** (**SELECT** avg (Spd) **as** AvgSpd, Time
FROM S[range 1])

which will produce P1 as:

P1 (AvgSpd; Time) = (50;1) (40;2)

Further, suppose that we want a stream of the total number of cars on the road for each time interval. The query that achieves this is

P2 = **ISTREAM** (**SELECT** count(distinct (VID)) **as** Count, Time
FROM S[range 1])

which produces:

P2 (Count; Time) = (2;1) (3;2)

Now, we want to correlate these two derived streams to detect the event at which the average speed is below 45 and the total number of cars is two or more. This requires a JOIN of P1 and P2 as follows:

Q = **ISTREAM** (**SELECT** *
FROM P1[range 1] **as** SpdReport, P2[range 1] **as** NumCars
WHERE SpdReport.AvgSpd< 45 **AND** NumCars.Count >= 2)

Q will produce different results in each of the two models. Oracle joins the results across time intervals giving us the following result:

Q (AvgSpd, Count; Time) = (40,3; 2)

Because StreamBase serializes all tuples and relies on this total order, there exists an arrival order (as given in the earlier definition of S) such that it will return:

Q (AvgSpd, Count; Time) = (40,2; 2) (40,3; 2)

The first tuple in this result is spurious since the speed of 40 and the count of 2 never co-existed. In other words, the implied simultaneity is not expressible.

Tuple-basis works; Time-basis doesn't

Assume you want a report of the average speed of the last two cars that you have heard from (independently of time). StreamBase can do this trivially with the following query:

Q=ISTREAM (**SELECT** avg(Spd) as AvgSpd
FROM S[rows 2])

For the same query, Oracle will again produce only two answers, one for each timestamp:

Q (AvgSpd; Time) = (50;1) (35;2)

Furthermore, the third report (1,50;2) is ignored entirely in these answers since the window of size two at time = 2 skips over it. This is an example of the "evaporating tuples" problem which will be explained in greater detail in what follows.

3. DIFFERENCES BETWEEN MODELS

While there are many differences between the two languages, we will initially restrict our attention to those that revolve around the notion of time, tuple ordering, window evaluation, and the interplay among these concepts. To highlight the differences between the two models, we use some simple examples.

We assume in the examples that tuples have a special attribute called "time" whose value is assigned by the system.

Thus our tuples have the following schema:

(value; time)

We will consider a stream S with the above schema and (possibly sliding) row-based and range-based window queries.

Let us now point out some of the differences between the two models and discuss the implications of these differences.

Difference 1: Change of Window State

Example 1: Identical Results Without Simultaneity

As a first example, consider the following stream:

S1(value; time) = (10;1) (20;2) (30;3) (40;4)

Consider the following query:

ISTREAM (**SELECT * FROM** S [rows 1])

In both the Oracle model and the StreamBase model, this query will return the following result:

(10;1) (20;2) (30;3) (40;4)

However, consider the following stream which contains two tuples with the same timestamp time=1.

Example 2: Difference in Window State Change With Simultaneous Tuples

S2(value;time) = (10;1) (20;1) (30;3) (40;4)

ISTREAM (**SELECT * FROM** S2 [rows 1])

In the Oracle Model, the state of a window always changes at timestamp boundaries. When time=1, one of the two tuples with timestamp 1 will be picked non-deterministically. Thus the result will be one of the following two streams:

(10;1) (30;3) (40;4) or (20;1) (30;3) (40;4)

The reason that one of the two tuples with time=1 seems to "evaporate" is that the two tuples with time=1 arrive simultaneously, since they have the same timestamp, and thus no further temporal distinction is possible between them. Windows get re-evaluated only when timestamps change.

In the StreamBase Model, the state of the window changes whenever a new tuple arrives. Thus, the output of the query in Example 2 in the StreamBase model would be

(10;1) (20;1) (30;3) (40;4)

We would like to emphasize that both models of window state change are meaningful and useful, and it would be best left up to the user to be able to choose when she would like the state of a window query to change.

Difference 2: Temporal Resolution

Both the Oracle and the StreamBase models have temporal resolution at the level of a timestamp. The StreamBase model also provides one refinement of timestamp by totally ordering all tuples in the system. Thus, while in the Oracle model all tuples arriving at a given timestamp are considered unordered, in the StreamBase model, there exists a total ordering between these tuples (which, for example, may be the order of tuple arrival).

To illustrate this difference, consider two streams that we would like to join. Due to the arrival order (call this the *rank*) of tuples, we have an explicit order between these tuples, but we cannot exploit this order in the semantics of the query. Let us illustrate this issue in Example 3.

Example 3: Difference in Order Beyond Timestamps

S2(value; time) = (10;1) (20;1)

S3(value; time) = (100;1) (200;1)

Now consider the following query:

ISTREAM (**SELECT * FROM** S2[rows 1], S3[rows 1])

In the Oracle Model, the output is one of the following tuples, without being able to consider the order across streams:

(10,100;1)

(20,100;1)

(10,200;1)

(20,200;1)

In the StreamBase model, we have to consider the arrival order to determine the output of this query. We give a few example arrival orders (among the 4! possible arrival orders) and their associated outputs. Recall that in StreamBase, the [rows 1] windows always contain the last arriving tuple in each stream.

Arrival order	Output
(10;1) (20;1) (100;1) (200;1)	(20,100;1) (20,200;1)
(20;1) (10;1) (100;1) (200;1)	(10,100;1) (10,200;1)
(20;1) (100;1) (200;1) (10;1)	(20,100;1)(20,200;1) (10,200;1)

While it may seem that the choice of the highest resolution possible (e.g., at the level of a tuple) is always preferable, this is not always the case. Temporal ordering is only useful as long as there is semantic meaning to it. For example, the notion of a total order may be semantically meaningful as a refinement of a timestamp when tuples enter the system and in other special cases; however, in general it is not always clear what assignment of unique ranks to downstream tuples to use. Let us illustrate this issue through two examples

Example 4: No Semantically Meaningful Ordering Across Streams

S4(value;time)=(10;1) (20,2) (30;3) (40;4)

S4′(value;time)=**ISTREAM** (**SELECT * FROM** S4 [rows 1])

S4″(value;time)=**ISTREAM** (**SELECT * FROM** S4 [rows 1])

Now what (total) order should there exist between the tuples in S4′ and S4″, in particular in comparison to each other? Should the tuples in S4′ obtain the lower rank since the definition of S4′ comes earlier in the DDL than the definition of Stream S4″? There does not seem to be a principled approach with a clean semantic meaning of order in this setting.

The inherent problems of always requiring a total order can also be seen in the following example.

Example 5. No Semantically Meaningful Ordering When Joining with Static Relations

S5(value; time) = (10;1) (10;2)

R(value) = {(10), (10)}

ISTREAM (**SELECT ***
FROM S5[rows 1] AS W, R
WHERE W.value = R.value)

In this query, each of the tuples in S5 joins with two tuples from the static relation R. Now what order should exist between the two new tuples? Note that if we were to enforce a total order it would lose its semantic meaning in the output stream; there is still order up to the "equivalence class" in which each of the two output tuples reside, but there is no order between these two tuples. Thus, always requiring a total order between all tuples is also not an acceptable solution.

Example 6. Orders Between Total Order and Timestamp Order

There is one further issue to consider that we will also illustrate through an example. Consider the following stream with attributes "value" and "batch" in addition to the mandatory attribute "time". Let us assume that the semantics of batch are that batch is consistent with time: If t.time < t′.time then t.batch < t′.batch.

Consider the following stream S6 with the window query below:

S6(value,batch;time) = (10,1;1) (20,1;1) (30,2;1) (40,2;1)

ISTREAM(
SELECT *
FROM S6[range 1])

If we would consider S6 as a total order, then the result would contain the four tuples from S6. However, "batch" is a semantically meaningful attribute as a pseudo-refinement of time, and thus a user may want to see the state of the window only change whenever a new batch of tuples arrives or if the window slides. None of the two existing models can capture this semantics; tuples do not get added to the window one by one as in the StreamBase model nor do they get added a timestamp at a time as in the Oracle model.

These examples illustrate that we need to relax simultaneity in order to be able to expose inherent, semantically meaningful order between tuples with the same timestamp. We also need to be able to define when a window gets evaluated in order to avoid the evaporating tuples problem. Next, we will describe a proposal based on equivalence classes that we believe strikes a fine balance between these new demands and the general concerns of staying conceptually clean and keeping a simple processing model.

4. PROPOSED MODEL

We now discuss the key components of the proposed convergence language by giving a precise description of the model and illustrating it with examples.

The key insight is that evaluation of results in both systems is triggered by the arrival of a *batch* of tuples. In the Oracle model the batch is defined by like values of a timestamp. In the StreamBase model, batches are always of size one tuple. By controlling the batching of tuples and the ordering between these batches, we can simulate both models plus many alternatives that neither model can capture.

In what follows, we assume that the reader is familiar with elementary group theory, in particular with the notion of an equivalence relation and the induced quotient set and the notion of a partial order.

We say that an equivalence relation $\sim_1$ is a refinement of an equivalence relation $\sim_2$ if whenever $x\sim_1 y$ holds then $x\sim_2 y$ also holds ($\sim_1$ makes the equivalence classes of $\sim_2$ "finer"). We say that a partial order $<_1$ is a *refinement* of a partial order $<_2$ if whenever $x<_2 y$ holds, $x<_1 y$ also holds.

4.1 Streams

Streams and Timestamps

A stream S has tuples T = (s;t), where t is the timestamp of the tuple. The timestamp is from a discrete domain **TIME** with a total order $<_t$.

There is an unbounded but finite number of tuples with the same timestamp. The timestamps induce an equivalence relation $\sim_t$ over the tuples in S such that two tuples are in the same equivalence class if they have the same timestamp. Formally, for tuples T1, T2, $T1\sim_t T2$ if T1.t = T2.t; we will also write $[T1]_t=[T2]_t$ where $[T1]_t$ denotes the equivalence class of tuple T1 with respect to the equivalence relation $\sim_t$. We will call this equivalence relation the *timestamp equivalence relation*. We call tuples with the same timestamp *timestamp-simultaneous.*

The total order on the timestamp domain induces a total order < in the quotient set of the equivalence relation $\sim_t$: We define that $[T1]_t < [T2]_t$ if T1.t < T2.t. This induces a partial order $<_t$ on tuples from stream S: For any two tuples T1 and T2 from S, we define that $T1 <_t T2$ if $[T1]_t < [T2]_t$. (Note that we have overloaded the symbol < to denote different orders, but its meaning is clear from the context.)

Example: Consider the stream S(value; time) with tuples (a;1), (b;2), (c;2), and (d,4) shown in Figure 1. Tuples (b;2) and (c;2) are timestamp-simultaneous, it holds that $[(b;2)]_t=[(c;2)]_t$.

Timestamps are essential to any steam processing model. However, as the examples in Sections 2 and 3 demonstrated, we need to be able to further distinguish between tuples *with the same timestamp*. We do this by refining the partial order $<_t$ between tuples that is induced by the timestamps even further through the introduction of *batches*.

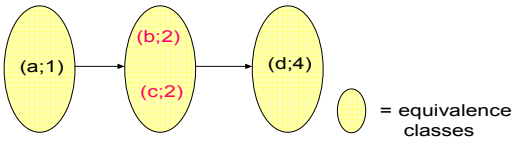

Figure 1 – A Stream with the Timestamp Equivalence Relation

Batches

In addition to the timestamp equivalence relation, a stream S has an associated *batch equivalence relation* $\sim_S$ that is a refinement of the timestamp equivalence relation $\sim_t$. We will call a set of tuples that is equivalent with respect to $\sim_S$ a *batch of tuples* and we will say that two tuples t1 and t2 such that $t1 \sim_s t2$ are *batch-simultaneous*. There also exists a total order $<_S$ in the quotient set of the equivalence relation $\sim_S$ with the property that the induced partial order on the tuples of S is a refinement of the partial order induced by the timestamps. Thus $<_S$ does not only subsume the timestamp order $<_t$, it also defines a partial order on all the timestamp-simultaneous tuples at time t; we will use $<_S$ to denote both the partial order between tuples and the total order in the quotient set of $\sim_S$; its usage will be clear from the context.

We will number the batches of a stream with (t,i), where t is the timestamp of the tuples in the batch and i is the number of the batch within the timestamp. Thus, at time t, the first batch is numbered (t,1), the second batch is numbered (t,2) and so on. Note that this ordering is unique since there is a total order on all the batches on a single stream. We refer to this batch number of a tuple T as $BATCH_S(T)$. The timestamp of a tuple is immutable, but as we will see later, tuples within a timestamp are batched and how tuples are assigned to batches can be changed through the query language. Note that we have so far only defined batches for a single stream; we will extend this notion in Section 4.2.

With batches, two tuples Ti and Tj from the same stream S must have one of the following ordering relationships:

1. $Ti <_t Tj$: Ti precedes Tj (or Tj follows Ti) in the timestamp order

2. $Ti \sim_t Tj$: Ti has the same timestamp as Tj, they are timestamp-equivalent. Now we either have:

   2.1: $Ti \sim_S Tj$: Ti and Tj are timestamp-simultaneous and batch-simultaneous

   2.2: $(Ti <_S Tj)$ or $(Tj <_S Ti)$: Ti and Tj are not batch-simultaneous, and the batch-order on stream S induces a total order between Ti and Tj.

Example (Continued.)

As an example consider the stream S(value; time) = $(a;1) <_S \{(b;2), (c;2)\} <_S \{(d;2), (e;2)\} <_S (f;4)$ shown in Figure 2. There exist four tuples with timestamp=2, they belong to two batches. The first batch numbered (2,1) consists of tuples (b;2) and (c;2) and the second batch numbered (2,2) consists of tuples (d;2) and (e;2).

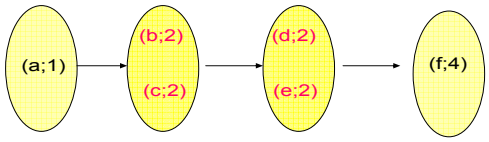

Figure 2 - Multiple Equivalence Classes per Timestamp

4.2 Stream Groups

So far, we have only discussed order within a single stream. The timestamp equivalence relation and its associated total order in the quotient set of the equivalence relation can be extended to a set of streams in a straightforward way since we can always compare the timestamps of any two tuples T1 and T2 regardless of what stream they come from. Thus, for any two tuples T1 and T2, we can always determine that one of the following two conditions holds:

1. $T1 \sim_t T2$: T1 and T2 have the same timestamp and thus are timestamp-equivalent.

2. $(T1 <_t T2)$ or $(T2 <_t T1)$: Either T1 has a smaller timestamp than T2 or vice versa.

However, such an extension is non-trivial for batches since batches within the same timestamp across streams cannot be compared. We address this issue by introducing the novel concept of a *stream group*.

Stream Groups

A stream group $\mathbf{G}=\{S1,S2,\ldots,Sk\}$ is a set of streams with an associated equivalence relation $\sim_{\mathbf{G}}$ and a total order $<_{\mathbf{G}}$ in the quotient set of this equivalence relation. A stream group is *consistent* if for each stream S in **G** the following holds:

> For all pairs of tuples T1, T2 from the same stream S such that T1 and T2 have the same timestamp t:
> $T1 \sim_S T2$ if and only if $T1 \sim_{\mathbf{G}} T2$ and
> $T1 <_S T2$ if and only if $T1 <_{\mathbf{G}} T2$.

In other words, the equivalence relation $\sim_{\mathbf{G}}$ and the total order in the quotient set $<_{\mathbf{G}}$ across all streams is consistent with the equivalence relations and total orders in the quotient set of each individual stream S in **G**. We will only consider consistent stream groups in the remainder of this paper.

Thus a stream group also induces a total order on the batches, and we can assign tuples within a timestamp again a unique batch number; we denote this batch number of a tuple T within a stream group **G** by $BATCH_{\mathbf{G}}(T)$.

4.3 Streams to Relations and Back

4.3.1 Batch-Varying Relations

Our notion of batches defines *batch-varying relations*. In particular, the granularity of relation states is defined as individual batches instead of time or individual tuples. Thus, a

window defined on a stream S will have a new state for each batch of S and each window state is simultaneous with the corresponding batch. This definition naturally generalizes to work with stream groups as well, thus allowing us to reason about the ordering (or simultaneity) of windows defined on the same stream group but on different streams. This is necessary, say, for the execution of a join of two windows defined on the same stream group.

Each table also has a value as of a given batch. However, how a table maps its progression of states into this domain is not part of our language specification.

4.3.2 Relations to Streams

We use two basic relation-to-stream operations, **ISTREAM** (Insert Stream) and **DSTREAM** (Delete Stream). Conceptually, ISTREAM captures tuples that are inserted to a relation (such as a window) over time, whereas DSTREAM captures the tuples that "fall out" of the relation, as used in CQL [1]. While the granularity of captures in CQL is a time tick, our capture granularity is a batch. Without loss of generality, we define these operations on windows as follows:

- ISTREAM of window *W* defined on S as of batch *bi* contains a tuple t if t is in *W* as of batch *bi* but not as of *bj*, where *bj* is the batch that immediately precedes *bi* on S.
- Similarly, DSTREAM of window *W* on S as of batch *bi* contains a tuple t if t is in *W* as of batch *bj* but not as of *bi* where *bj* is the batch that immediately precedes *bi* on S.

One subtle issue that arises with ISTREAM and DSTREAM involves deciding whether two tuples with the same value (identical attribute-value pairs) are to be treated as the same or not. For example, consider a tuple (a;1) to be inserted into window W with state {(a;1), (b;1)}. There are two possible semantics one can assume in this case: we can use set-difference semantics and treat the two (a;1) tuples as identical. With this **difference semantics**, the ISTREAM of W will not contain the new tuple (a;1) as another tuple with the same value already exists in W. We can also define an alternative semantics by assuming that tuples are inherently different regardless of their values, in which case the new tuple will be contained in the ISTREAM. We call this latter semantics **delta semantics**.

Because these two semantics are applicable and useful in different situations, we decide to include both in our proposed language. Specifically, we use the keywords **DIF** and **DEL** along with ISTREAM and DSTREAM to indicate which of the two semantics we intend to use.

4.4 Revising Existing Models

With this notation, we now revisit the two stream processing models that this paper unites.

The Oracle Model.

For a stream S the equivalence relation $\sim_S$ is equal to the timestamp equivalence relation $\sim_t$, i.e., all tuples with the same timestamp are not only timestamp-simultaneous, but also in the same batch. Since the quotient set of $\sim_S$ contains only one batch per time, there is no further ordering $<_S$.

All existing streams form a single stream group with all tuples within the same timestamp across all the streams belonging to one equivalence class of $\sim_{\mathbf{G}}$. Thus there is also no further ordering $<_{\mathbf{G}}$.

The StreamBase Model.

In the StreamBase model, the equivalence relation $\sim_S$ is trivial, i.e., it only contains elements of the form (T,T) and thus each batch consists of one tuple. Thus, the total order $<_S$ orders all the tuples with the same timestamp on a single stream.

In the StreamBase model, all streams together form a single stream group **G** with each tuple again in its own equivalence class of $\sim_{\mathbf{G}}$ and $<_{\mathbf{G}}$ defines a total order across all streams between all tuples with timestamp t (recall that $<_{\mathbf{G}}$ refines $<_t$). Thus in the StreamBase model, there is a total order between all tuples in the system.

4.5 Windows

One of the main constructs in any stream query language is a window. Let us now explain how to form windows. Windows take as input a stream and result in batch-varying relations. There are two fundamental aspects to the semantics of windows.

- Deciding when a new window state is triggered.
- Deciding the contents of that window state.

In our novel model, a window on stream S is triggered *whenever a new batch of tuples arrives*, and thus the relation that is created from the stream varies with each batch of arriving tuples. Let us make this notion a bit more formal and introduce the two basic types of windows that every stream processing language supports in our new model.

In this section, we always assume that the window constructs are part of a larger query, and that all the streams (and time-varying relations) are part of the same stream group **G**. We denote the equivalence relation that is associated with the temporal group **G** simply by $<$ since **G** is clear from the context.

4.5.1 Row-based windows

A row-based window is specified on a stream S as

S [rows n]

The output is a relation that varies with each *batch* of tuples in the input stream. More formally, R(t,i) consists of the n tuples of S with the largest batch numbers that are $\leq$ (t,i). If this set consists of more than n tuples, then we non-deterministically choose among the set of tuples with the smallest batch number in order to reduce the number of tuples to n. Formally:

$$\begin{aligned}\mathrm{Rin}(t,i) = \{(s,t') \text{ in } S : \mathrm{BATCH}_G(s,t') \leq (t,i) \text{ and} \\ |\{(r,t'') \text{ in } S : \mathrm{BATCH}_G(r,t'') \leq \mathrm{BATCH}_G(t,i) \\ \text{and } \mathrm{BATCH}_G(r,t') > \mathrm{BATCH}(s,t')| < n\}.\end{aligned}$$

Let (t',j) be the smallest batch number among all tuples in Rin(t,i). Then we choose n-|Rin(t,i)| additional tuples from the batch before (t,j'); R(t,i) consists of Rin(t,i) union these additional tuples.

When n=infinity, then R(t,i) consists of all tuples in S up to and including batch (t,i).

Example. As an example of how row-based windows work, consider the stream that is illustrated in Figure 2. The states that would be created for a window defined as S[rows 2] would be:

R(1,1) = {(a;1)}

R(2,1) = {(b;2), (c;2)}

R(2,2) = {(d;2), (e;2)}

R(4,1) = {(d;2), (f;4)} or {(e;2), (f;4)}

The last result is non-deterministic because, while (f;4) is certainly the tuple with the largest timestamp in stream S, the second to last tuple could be (d;2) or (e;2) since they are batch-simultaneous. Note that (b;2) and (c;2) are not considered since they belong to an earlier batch even though they have the same timestamp as (d;2) and (e;2). (The SPREAD command that we introduce in the next section allows us to impose an order between these two tuples and thus eliminate the non-determinism.)

4.5.2 Range-based windows

A time-based window is defined as

S [range T]

The output is a time-varying relation that varies with each *batch* of tuples. We denote by t the timestamp of the batch-varying relation and by i the batch number:

$R(t,i) = \{s \mid (s,t') \text{ in } S \text{ and } BATCH_{Gr}(s,t') <= (t,i) \text{ and } (t' >= \max\{t-T,0\})\}$

When T=0, then R(t,i) consists of all tuples (s;t) with timestamp t up and including batch number i. When T=infinity, then R(t,i) consists of tuples obtained from all elements in S up to the i^{th} batch at timestamp t.

Example. If we now form a range-based window over the same stream (Figure 2) as S[range 2], we get the window states that are enumerated below:

R(1,1) = {(a;1)}

R(2,1) = {(a;1), (b;2), (c;2)}

R(2,2) = {(a;1), (b;2), (c;2), (d;2), (e;2)}

R(3,1) = {(b;2), (c;2), (d;2), (e;2)}

R(4,1) = {(f;4)}

Note that R(3,1) has the given value because of the absence of a batch at time t=3. A window state is generated here for the range-based window because timestamp 3 exists even if a tuple was not generated for that timestamp.

4.5.3 Partitioned Windows

A partitioned window is defined as

S [Partition by A1,...,A*k* rows n]

The output is a batch-varying relation that varies with each batch of tuples in the input stream.

Intuitively, this window logically partitions S into different substreams based on equality of attributes A1, A2,…, A*k*, then computes row-based windows of size n independently on each substream, and finally takes the union of the corresponding batches of these windows to produce the output relation.

This is analogous to the way partitioned windows work in CQL [1]; however, unlike in CQL where all partitioned windows get evaluated simultaneously at each timestamp, our windows get evaluated with the arrival of batches on each window. As such, all windows do not necessarily get evaluated simultaneously. We omit a formal description due to space constraints.

4.5.4 Sliding Windows

Most data models have a means for specifying the way that windows move from one state to the next. This is typically referred to as a window slide. We incorporate window slides into our new model in a very simple way. Batches will still trigger new windows, but there will be requirements on the number of rows or the number of timestamps that must pass before a new window will be allowed to form. We will illustrate this by considering row windows and range windows separately.

Row-based slides

A row window can slide by some number of tuples as in S[rows 1 slide 2]. In this case, we are defining a 1-tuple window such that there must be at least two tuples since the last window before a new window state is formed.

Consider the following example. Suppose that the stream S is as follows with curly brackets indicating batches:

S(value;time) = {(a;1) (b;1) (c;1)} {(d;2) (e;2) (f;2)} {(g;3)} {(h;3)} {(i;4) (j;4)} {(k;4)}

The parentheses indicate batches and the integer values represent tuples with the given timestamp. A window defined on stream S as S[rows 2 slide 2] would produce the following three states

[(b;1) (c;1)]
[(e;2), (f;2)]
[(g;3), (h;3)]
……

The first batch of tuples triggers a window and the second and third tuple form the first 2-row window. The window slides and as soon as the next batch is seen, the slide condition is satisfied. That is, two new tuples have been seen since the last window state. At this point, the second and third tuple from the second batch are taken as the next window state. We must see two additional batches before the condition of two new tuples is satisfied, which explains the third window state.

Range-based slide

Here it is still batches that trigger a new window. The basic form for a range-based sliding window on stream S is S[range 2 slide 2]. Only batches that have timestamps, ts, such that (ts mod 2 = 0) will trigger a new window. As an example, consider stream S as before. A window defined on S as S[range 2 slide 2] would generate the following three states.

[(a;1) (b;1) (c;1) (d;2) (e;2) (f;2)]
[(g;3) (h;3) (i;4) (j;4)]
[(g;3) (h;3) (i;4) (j;4) (k;4)]

In the case of a range-based window, we produce new windows as long as the window timestamp range is obeyed. Thus, in this example, we get two windows for timestamp range 3-4.

5. MANIPULATING ORDERS

Given the simple model of streams, stream groups, and windows described above, the goal is now to develop mechanisms to create and manipulate ordering relationships among the tuples, within and across streams. We accomplish this goal through a powerful stream-to-stream operator called SPREAD.

Intuitively, SPREAD allows us to modify the batch equivalence relation $\sim_{\mathbf{G}}$ and the total order between batches $<_{\mathbf{G}}$ for a stream group **G**. Let us start by defining the syntax and semantics of SPREAD.

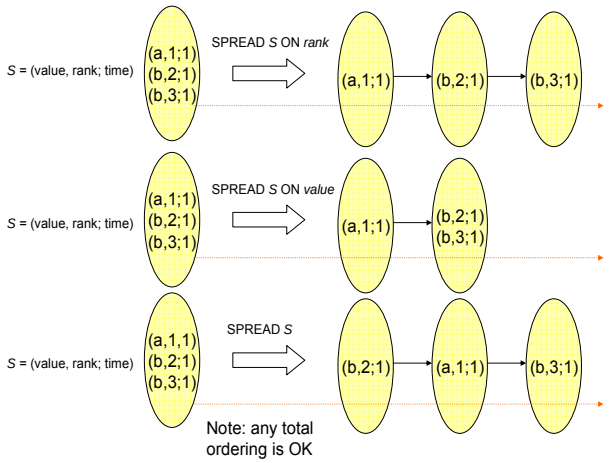

Figure 3 - Single Input Spread Examples

5.1 SPREAD: Syntax and Semantics

For simplicity of exposition, we first discuss the single-stream version of SPREAD that takes a single input stream and produces a single output stream. We will then generalize the description to multiple streams. We assume without loss of generality that the attribute list just consists of a single attribute "Attr"; generalization to a list of attributes is straightforward.

5.1.1 Single-Stream SPREAD

The syntax for the single-input SPREAD is as follows:

SPREAD InStream [ALL] [ON Attr] OutStream

There are two flavors of SPREAD, distinguished by the keyword **ALL**. The first flavor ignores existing orders, whereas the second retains them.

SPREAD ALL. SPREAD ALL orders tuples into batches by constructing a new batch equivalence relation $\sim_{OutStream}$ from the stream InStream. Consider two tuples T1 and T2 such that $T1 \sim_t T2$, i.e., T1 and T2 are timestamp equivalent in InStream. Then the SPREAD above defines the following batch equivalence relation:

$T1 \sim_{OutStream} T2$ if $[T1]_t = [T2]_t$ and T1.Attr = T2.Attr.

The total order $<_{OutStream}$ is defined as follows:

$[T1] <_{OutStream} [T2]$ if

$T1 <_t T2$ or ($T1 \sim_t T2$ and T1.Attr < T2.Attr.

Thus SPREAD orders the batches within a timestamp by the value of column Attr. If T1 and T2 have the same value for Attr, they are batch-simultaneous in OutStream.

Thus, the outcome of SPREAD is independent of any existing batch equivalence class on InStream. If no Attr is specified, then SPREAD creates a random total order. Note that with SPREAD, we can now force a specific total order on tuples in a stream *with the same timestamp*: We simply have to create and maintain an attribute, or a list of attributes, whose values are guaranteed to have the desired ordering properties for tuples with the same timestamp. In some systems such an attribute will be called a rank attribute.

SPREAD. SPREAD creates a new equivalence relation that is a refinement not only of the timestamp equivalence relation but of a possibly existing batch equivalence relation on the input stream. More formally, SPREAD orders tuples into batches by constructing a new batch equivalence relation $\sim_{OutStream}$ on the stream OutStream that is a refinement of any existing batch equivalence relation $\sim_{InStream}$. Consider two tuples T1 and T2 such that $T1 \sim_t T2$ and $T1 \sim_{InStream} T2$, i.e., T1 and T2 are timestamp and batch equivalent in Instream. Then we define the batch-equivalence relation that is induced by the SPREAD command at the beginning of this section as follows:

$T1 \sim_{OutStream} T2$ if

$T1 \sim_{Instream} T2$ and T1.Attr = T2.Attr

The total order $<_{OutStream}$ is defined as follows:

$[T1] <_{OutStream} [T2]$ if

$[T1] <_{InStream} [T2]$ or

$[T1] \sim_{InStream} [T2]$ and T1.Attr < T2.Attr

In other words, if two tuples T1 and T2 are already batch-ordered (T1 < T2 or T2 < T1) in InStream, SPREAD does not have any affect on the relationship between those tuples. However, SPREAD further orders batch-simultaneous tuples in InStream based on Attr.

Propagating ordering information from the inputs to outputs: We use a simple rule to infer the order information for the derived tuples: all tuples that get produced as the result of processing a given batch are simultaneous with that batch. Notice that the basic CQL time-propagation model is a specific instance of this rule.

Figure 3 shows three examples of the use of SPREAD on a single stream. In the top two cases, the tuples are SPREAD on the basis of an attribute value. The bottom SPREAD does not specify any attribute, indicating that any total ordering of the tuples is acceptable.

5.1.2 Multi-Stream SPREAD

Now let's consider multiple input-output streams.

SPREAD InStream_L [ALL] [ON Attr] OutStream_L

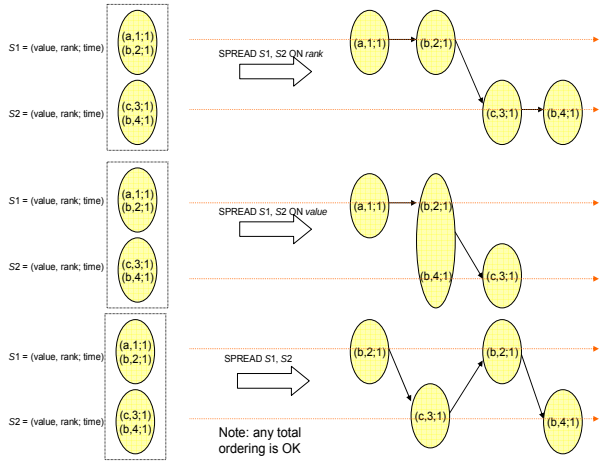

Figure 4 - Multi-Input Spread Examples

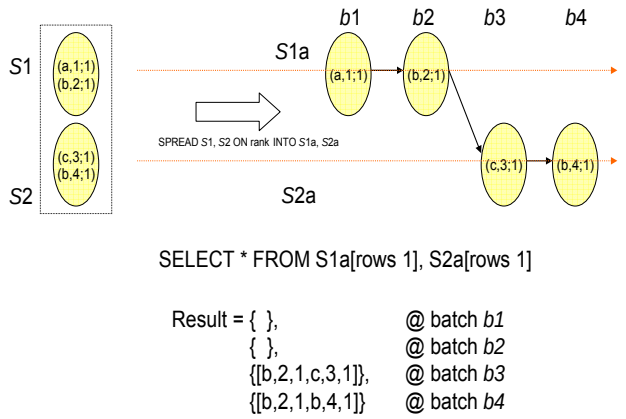

Figure 5 – A JOIN Example Using SPREAD

SPREAD ALL takes the tuples that have the same timestamp on all input streams specified in InStream_L, reorders them according to Attr as described above, and then places them on the corresponding output streams specified by OutStream_L. If no Attr is specified, then the result is a random total order across all output streams.

SPREAD retains the existing batch equivalence relation within the stream group and refines it in the same way as described in Section 5.1.1. Note that it is imperative that all input streams belong to the same stream group as otherwise there may not exist a total order among the batches such that SPREAD can refine this total order.

Figure 4 shows three examples of the use of SPREAD on two input streams. In the top two cases, the tuples are SPREAD on the basis of an attribute value. Note here that multi-input SPREAD creates new equivalence classes and orders among tuples on different streams. The bottom SPREAD does not specify any attribute, indicating that any total ordering of the tuples across the two streams is acceptable.

5.2 A JOIN Example using SPREAD

Consider two streams S1 and S2 as illustrated in Figure 5. Suppose that we first spread these two streams using SPREAD S1, S2 ON rank INTO S1a, S2a and then join S1a and S2a with one-tuple row-based windows. The result is shown at the bottom of Figure 5.

As shown above, we now generate window states as of a batch. Similarly, results are also generated as of a batch.

As a further example of multi-input SPREAD and JOIN, consider the same input streams and a SPREAD S1, S2 ON time INTO S1a, S2a with the same JOIN command as before. The result can be seen at the bottom of Figure 6.

6. DIFFERENCES REVISITED

In Section 3, we saw examples of specific problems that occur with the Oracle and StreamBase languages. The main issue is that neither of the languages has sufficient expressive power to affect the order in which tuples are processed and results are generated. In particular, the Oracle language suffers from a coarse notion of simultaneity, which is exclusively driven by time and cannot exploit additional ordering information. On the other hand, the StreamBase language has no notion of simultaneity and thus has to "artificially" introduce order at times.

The new model described above does not suffer from these problems for two reasons. First, there is inherent support for simultaneity and partial order. Second, the SPREAD operator can be used to introduce order and simultaneity as needed.

We now revisit some of the previous examples to illustrate the utility of SPREAD in addressing the aforementioned issues.

First, let's recall the query from Example 2, which had the evaporating tuples problem, and rewrite it using the partial order notation.

S2(value;time) = {(10;1), (20;1)} < {(30;3)} < {(40;4)}

ISTREAM (**SELECT * FROM** S2 [rows 1])

The evaporating tuples problem arose here because of the simultaneity of the first two tuples. In the new model, this issue can be easily addressed by:

SPREAD S2

which yields as a possibility the following output stream

S2 = {(10;1)} < {(20;1)} < {(30;3)} < {(40;4)}

This is also the output we get as the result of the query. While this example introduced order arbitrarily, it is also possible to leverage additional ordering information not reflected by time. Let's rewrite Example 3 with the modification that tuples now have an additional "rank" attribute whose value reflects the tuple arrival order:

S2(value, rank; time) = {(10,1;1),(20,3;1)}

S3(value, rank; time) = {(100,2;1),(200,4;1)}

ISTREAM(**SELECT * FROM** S2[rows 1], S3[rows 1])

(Note that both the within-stream and the cross-stream order of the tuples is undefined.)

Here we can break the simultaneity and produce a total order consistent with the rank values by applying "SPREAD ALL S1, S2 on rank into S1', S2' ", that leads to the following output streams:

S2'(value, rank; time) = $\{(10,1;1)\}^1 < \{(20,3;1)\}^3$

S3"(value, rank; time) = $\{(100,2;1)\}^2 < \{(200,4;1)\}^4$

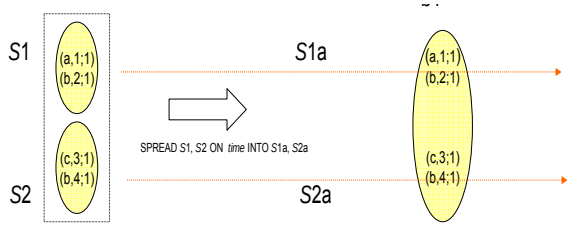

Figure 6 – Same Join, Different SPREAD

(The numbers in superscripts indicate the cross-stream order of the batches.) The output of the query is as follows:

{(10,1,100,2;1)} < {(20,3,100,2;1)} < {(20,3,200,4;1)}

Examples 4 and 5 illustrated the non-intuitive implications of a model that always requires a total order. Since our proposal allows and properly propagates simultaneity, it does the "right thing" here. That is, in Example 4, the simultaneity of tuples across S4' and S4'' are automatically maintained, and in Example 5, the two new tuples created for each S5 tuple are simultaneous. In both cases, the new model obtains the desired behavior by default without any explicit SPREAD operations.

Finally, let us consider Example 6 again using the new notation:

S6(value, batch; time) = {(10,1;1), (20,1;1), (30,2;1), (40,2;1)}

```
ISTREAM(SELECT*
        FROM S6[range 1] )
```

Here if we apply SPREAD S6 on batch, the updated S6 becomes

S6 = {(10,1;1), (20,1;1)} < {(30,2;1), (40,2;1)}

Here, the tuples with the same batch values are placed in the same equivalence class and are thus batch-simultaneous.

7. INTRUSION DETECTION EXAMPLE

We now present a more sophisticated example that exercises the various constructs that we have introduced so far.

Scenario

We consider a simplified scheme for network intrusion detection: the goal is to track those external Internet hosts that initiate connections to hosts within a protected network to identify if they are potentially engaging in port-scanning activities. Port scans are one common way in which an attacker finds vulnerabilities to exploit, because it is a way to find out if a particular service (such as a TCP port) is active or not on a host.

In this case, the input stream, **CStream**, is a stream of connection summaries, where a connection has a source IP address (**srcIP**), destination IP address, destination port number, and a Status field (**status**) indicating whether the connection succeeded or failed. Further, each connection has an arrival number (**arrival_number)**, a unique sequence number applied by the system, as well as a field (**time)** that represents the connection attempt time on the destination system.

A "scanner alert" occurs when there are *X* failed connections out of *Y* successive attempts. Our objective is to raise a high-level alert if, for a specific srcIp, the number of scanner-alerts in the last minute is greater than the number of scanner-alerts in the previous minute.

Query

The connection stream, CStream, might contain many simultaneous connection attempts. Thus, we first need to make sure that we process each attempt; in particular, we need to break the simultaneity to avoid the "evaporating tuples" problem studied earlier. We can accomplish this by *spreading* CStream as follows:

```
SPREAD CStream ON arrival_number INTO COrderedStream;
```

The new stream, COrderedStream, has the exact same tuples as CStream, but its tuples are totally ordered on the basis of their arrival orders. As such no two tuples on COrderedStream are simultaneous.

We now define a new stream, ScannerAlertStream, which contains the sources producing scanner-alerts we described above:

```
ScannerAlertStream =
ISTREAM (SELECT  srcIp
         FROM   COrderedStream[Partition by srcIp rows Y]
         WHERE  status = 'FAILURE'
         GROUP BY srcIP
         HAVING count(*) > X)
```

Now, we need to compute the number of scanner-alerts in the last minute and the last two minutes. Since we need only one entry per minute per srcIp (updated every minute), we first need to "restore" simultaneity back again on the ScannerAlertStream, which we can achieve by spreading on time as follows:

```
SPREAD ALL ScannerAlertStream ON time
        INTO ScannerAlertSimStream;
```

On this stream, we define two relations: Relation MinuteCount:

```
SELECT srcIp,  count(*) as min_count
FROM    ScannerAlertSimStream[range 1 minute slide 1 minute]
GROUP-BY srcIp;
```

Relation TwoMinuteCount:

```
SELECT srcIp,  count(*) as two_min_count
FROM ScannerAlertSimStream[range 2 minutes slide 1 minute]
GROUP-BY srcIp;
```

Notice that by virtue of the automatic propagation of ordering information from the input streams to the derived streams, the states of MinuteCount and TwoMinuteCount are now updated simultaneously every minute. Finally, the desired output stream, AlertStream, can be defined as:

```
ISTREAM(SELECT srcIp
        FROM MinuteCount as M, TwoMinuteCount  as T
        WHERE M.srcIp = T.srcIp and
              2*M.min_count > T.two_min_count)
```

This example illustrates the use of SPREAD to break and recover simultaneity as well as the propagation of ordering/simultaneity information across nested queries.

8. A BRIEF NOTE ON IMPLEMENTATION

The main result reported here is on stream data models and semantics; however, we take this opportunity to say a few words about implementation and extensibility issues.

As in either of the original languages, relations (incl. windows) take on a progression of new states, each one of which will potentially trigger evaluation of the query. Once a new relation state in a given query has been determined, the query is evaluated exactly as it would be in SQL, thus, enabling all standard optimization techniques. This is all still true in this new model with the caveat that relation states are produced as the result of batches.

The batches will be determined by the use of SPREAD. Thus we might ask how we implement SPREAD. Of course, this is up to each vendor that adopts this notion and would likely be proprietary. A naïve implementation would collect all the tuples for a given timestamp and would apply the sorting criteria implied by the SPREAD to create the new batches. An optimization could occur if we knew that all values (within a timestamp) of the SPREAD attribute have arrived, allowing us to create a batch early. This could be accomplished through the use of heartbeat tuples, much as this technique is used in time-based processing.

We also note that our proposal does not require changes to the existing relational operators or preclude adding new ones. This is a direct result of the decoupling between the window semantics and operator semantics. SPREAD only affects the former; as such, standard relational operators as well as new UDFs that work on relations can be immediately used in our framework.

9. RELATED WORK

There have been many stream query languages proposed over the last several years [1, 7, 8, 10, 11, 13]. The goal of this section is to describe the evaluation models of other stream query engines (both academic and commercial), and to report whether they more closely resemble the Streambase model (tuple-driven) or the Oracle model (time-driven).

It is somewhat surprising that it is not common to find language descriptions in this arena that explicitly and precisely discuss the evaluation rules directly. As a result, some of what follows is based on our best guess. We have learned through the activity that we report here how subtle evaluation issues can be and how hard it is to find such a model that satisfies multiple needs. We hope that this paper will encourage others to evaluate their own systems in this light.

In many cases, scheduling policies were discussed (e.g., [5, 9]), but it is important to note that scheduling policies are orthogonal to the evaluation model. For example, [9] discusses a batch scheduling approach that schedules query evaluation when *batches* of input tuples arrive, but this is an optimization meant to preserve the semantics of the tuple-driven model. Similarly, one could imagine that a scheduler for a time-driven model could schedule computation with the arrival of every tuple, taking advantage of lulls between tuple arrivals to partially generate results that will be emitted with the next clock tick.

Both tuple-driven and time-driven evaluation models have historical precedents set prior to the emergence of stream processing systems. The tuple-driven model is the basis for view maintenance [18]; any change in a base table must be reflected in a dependent view for the view to be accurate. Thus, view maintenance occurs in response to the *event* of a tuple being inserted into, deleted from or changed within a base table. The implementation of view maintenance can be lazy, but this is an optimization and does not change the semantics of what is considered to be a correct view. Similarly, database integrity constraints (which can be thought of as views that return Boolean responses to updates (allow or disallow)) and triggers have an execution model that is tuple-based. On the other hand, time-driven evaluation models evolved from early work in continuous query processing (e.g., Tapestry [21]). A parameter of these systems was a time interval that determined the frequency of generating incremental query results.

According to the CCL user manual, [8], CCL supports *both* tuple-driven and time-driven evaluation models. This is similar to the "big switch" model that we mentioned earlier. The default setting is tuple-driven evaluation. Thus, the CCL query,

INSERT INTO R1
SELECT sum (value) as Sum
FROM s
KEEP 3 rows

which windows stream S with a tuple-based window of size 3, is equivalent to the StreamBase query

R1 = **ISTREAM** (**SELECT** sum (value) as Sum
FROM S [rows 3])

in that it allows windows to include tuples with different timestamps. For example, let

S (value; time) = (1;1),(2;1),(3;1),(4;1),(5;2),(6;2),(7;3), …

be the input stream such that time is the system-provided timestamp (expressed as a measure of seconds). Given this stream, both Coral8 and StreamBase queries would return the stream,

R1(Sum; time) = (6;1), (9;1), (12;2), (15;2), (18;3) …

To support time-driven evaluation, CCL provides an optional OUTPUT clause that suppresses outputs except at specified periods. For example, the CCL query,

INSERT INTO R2
SELECT sum (value) AS v
FROM S
KEEP 3 rows
OUTPUT EVERY 1 second

would return results only when an input tuple's timestamp reported that a second had "ticked" since the last output. In this case, the CCL query would return the stream,

R2(Sum; time) = (9;1), (15;2), …

as this result indicates the sum of the last 3 values seen when tuple (5;2) arrives (9 = 2+3+4) and when tuple (7;3) arrives (15 = 4+5+6).

ESL, a part of the StreamMill system [7, 25], supports both internal and external timestamps. Internal timestamps are generated by the system as tuples arrive, while external timestamps come from the environment. These timestamps are used to determine ordering and can be used as a way to understand when that ordering has been violated (out of order tuples), but they are not used as the fundamental unit of evaluation. Instead, tuples are acted on as soon as they become available, much as in StreamBase.

Time-driven semantics as we have discussed in this paper originated with the CQL language [24, 1]. The Oracle language is a direct descendant of CQL.

10. CONCLUSION AND FUTURE WORK

This paper begins the discussion of a SQL-based standard for streaming databases. It discusses some deep model differences that exist between Oracle CQL and StreamBase StreamSQL. We believe that similar differences exist among other prototypes and products. Until fundamental model differences are settled, there is little chance of producing a good standard. We believe that this paper has uncovered differences and proposed a novel way of

resolving them. This proposal has the added benefit that it increases the expressive power of both languages and gives the user fine-grained control over time and simultaneous evaluation.

There are many remaining problems that we could investigate on the road to a complete standard. For example, pattern matching has been shown to be an important part of any language over streams. There have been several proposals [4, 16, 17, 20, 22]. In the same way that we tackled the evaluation model here, it makes sense for a group to perform a similar exercise for pattern matching semantics.

Given the new model that is proposed here, there are some interesting questions regarding how we might efficiently implement such a model. While, of course, this is not the business of a standards group, it nonetheless is an important issue that needs to be addressed. We would need a good way to represent the current batch state of a stream. Given a batch state, we would need a good implementation of window generation and table maintenance.

Acknowledgments. We would like to thank Oracle Corporation and StreamBase, Inc. for the support that we received in doing this work. IBM organized a one day meeting in August of 2006 that was very helpful in getting this discussion started. Dieter Gawlick and Mike Stonebraker had the foresight to put the Oracle/StreamBase team together that led to this report.

11. REFERENCES

[1] A. Arasu, Arvind; S. Babu J. Widom: The CQL Continuous Query Language: Semantic Foundations and Query Execution, VLDB Journal, Vol. 15, No. 2, June, 2006.

[2] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik: Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal (12)2: 120-139, August 2003.

[3] A. Arasu, M. Cherniack, E.F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts: Linear Road: A Stream Data Management Benchmark. VLDB 2004: 480-491

[4] Anonymous: Pattern Matching in Sequences of Rows, SQL standard proposal, http://asktom.oracle.com/tkyte/row-pattern-recogniton-11-public.pdf, March, 2007.

[5] B. Babcock, S. Babu, M. Datar, and R. Motwani: Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. SIGMOD 2003: 253-264

[6] R.S. Barga, J. Goldstein, M.H. Ali, M. Hong: Consistent Streaming Through Time: A Vision for Event Stream Processing. CIDR 2007: 363-374

[7] Y. Bai, H. Thakkar, C. Luo, H. Wang, C. Zaniolo: A Data Stream Language and System Designed for Power and Extensibility. CIKM 2006.

[8] Coral8 Inc., Coral8 CCL Reference, available at http://www.coral8.com/system/files/assets/pdf/5.2.0/Coral8CclReference.pdf

[9] D. Carney, U. Cetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, M. Stonebraker: Operator Scheduling in a Data Stream Manager. VLDB 2003.

[10] S. Chandrasekaran, and M. Franklin: Streaming Queries over Streaming Data. VLDB 2002.

[11] Cherniack, M.: SQuAl: The Aurora [S]tream [Qu]ery [Al]gebra, Technical Report, Brandeis University, Nov 2003.

[12] Codehaus.org, Esper online documentation set, http://esper.codehaus.org/tutorials/tutorials.html, 2007.

[13] Conway, N., An Introduction to Data Stream Query Processing, Slides from a talk given on May 24, 2007, http://www.pgcon.org/2007/schedule/attachments/17-stream_intro.pdf., 2007.

[14] Coral8 Systems, Coral8 CCL Reference Version 5.1, http://www.coral8.com/system/files/assets/pdf/current/Coral8CclReference.pdf, 2007.

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

[16] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M. White: Cayuga: A General Purpose Event Monitoring System. CIDR 2007: 412-422

[17] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, G. Anderson: SASE: Complex Event Processing over Streams. CIDR 2007.

[18] M. Staudt, M. Jarke: Incremental Maintenance of Externally Materialized Views. VLDB 1996.

[19] StreamBase Systems, StreamSQL online documentation, http://streambase.com/developers/docs/latest/streamsql/index.html , 2007.

[20] S. Reza, C, Zaniolo, A. Zarkesh, J. Adibi: Expressing and optimizing sequence queries in database systems. ACM Trans. Database Syst. 29(2): 282-318 (2004).

[21] D. B. Terry, D. Goldberg, D. Nichols, B. M. Oki: Continuous Queries over Append-Only Databases. SIGMOD 1992: 321-330

[22] E. Wu, Y. Diao, S. Rizvi: High-Performance Complex Event Processing over Streams. SIGMOD 2006.

[23] P.A. Tucker, D. Maier, T. Sheard, P. Stephens: Using Punctuation Schemes to Characterize Strategies for Querying over Data Streams. IEEE TKDE. 19(9): 1227-1240 (2007)

[24] Widom, Jennifer: CQL: A Language for Continuous Queries over Streams and Relations, Slides from a talk given at the Database Programming Language (DBPL) Workshop, http://www-db.stanford.edu/~widom/cql-talk.pdf , 2003.

[25] Zaniolo, C., Luo, C., Wang, H., Bai, Y., Thakkar, H.: An Introduction to the Expressive Stream Language (ESL), Technical Report, UCLA, 2006.