Transactional Stream Processing

Irina Botan[1], Peter M. Fischer[2], Donald Kossmann[1], Nesime Tatbul[1]

[1]ETH Zurich, Switzerland [2]Universitï£¡t Freiburg, Germany

ABSTRACT

Many stream processing applications require access to a multitude of streaming as well as stored data sources. Yet there is no clear semantics for correct continuous query execution over these data sources in the face of concurrent access and failures. Instead, today's Stream Processing Systems (SPSs) hard-code transactional concepts in their execution models, making them both hard to understand and inflexible to use. In this paper, we show that we can successfully reuse the traditional transactional theory (with some minimal extensions) in order to cleanly define the correct interaction of a set of continuous and one-time queries concurrently accessing both streaming and stored data sources. The result is a unified transactional model (UTM) for query processing over streams as well as traditional databases. We present a transaction manager that implements this model on top of an existing storage manager for streams (MXQuery/SMS). Experiments on the Linear Road Benchmark show that our transaction manager flexibly ensures correctness in case of concurrency and failures, without sacrificing from performance. Moreover, this model is powerful enough to express the implicit transactional behaviors of a representative set of state-of-the-art SPSs.

1. INTRODUCTION

Stream processing has become the base technology for an increasing variety of applications. These applications often times involve access to multiple data sources, including not only purely streaming ones, but also stored ones, e.g., for correlating streams with historical data or for enriching them with additional metadata. As the scale and complexity of these applications increase, it becomes harder to ensure their correct execution in the presence of concurrent processing or failures over these multiple data sources. This paper investigates this challenge.

To illustrate the problem, let us consider a simple scenario. Suppose that there is a set of devices whose functioning is sensitive to temperature. Each of these devices has a minimum and a maximum temperature specification, defining the interval in which it functions properly. These specifications are stored in a database table (R). Furthermore, a set of sensors take real-time temperature measurements (on a Celsius scale) for these devices and make them available as an input stream (IS). We would like to generate an alert, whenever a temperature reading falls outside the acceptable temperature interval of the corresponding device. A typical Stream Processing System (SPS) would model this with the following continuous query, as shown Figure 1: for each incoming temperature reading event in IS, probe the specifications table R, and raise an alarm whenever a violation is detected (e.g., as in the case of event (D2, 24, 4) which generates alarm (D2, 24)).

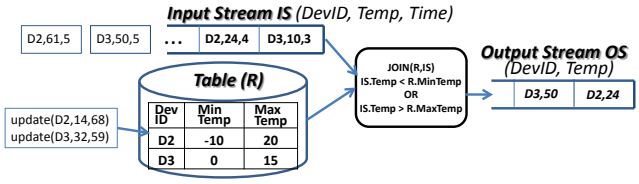

Figure 1: Example Scenario

Now suppose that, after the arrival of event (D2, 24, 4), the temperature readings scale of the sensors was changed to Fahrenheit. Of course, the specifications in R must also be updated to reflect the change in temperature scale; but if the updates on the table do not happen *before* the new sensor measurements arrive, false alarms can be generated. For example, in Figure 1, event (D3, 50, 5) generates a false alarm, while in fact 50 °F represents 10 °C, which happens to be in the accepted range for device D3. In order to obtain a correct execution, an SPS should be able to order the updates on table R and the temperature readings in stream IS. Today, this task cannot directly be achieved by any SPS.

In addition to the update problem illustrated above, failures can lead to incorrect executions, while many streaming applications require that continuous queries be persistent over system crashes. For example, in our scenario, assume that while processing event (D2, 24, 4), the SPS experiences a failure. Since all the temporary state created by this event is lost, the alarm it would have generated in a normal execution scenario, (D2, 24), will never be generated. Therefore, it is important to be able to specify that this event should only be processed to completion (until the results are committed to the output), or reprocessed in case of a failure.

The fundamental problem presented above stems from the differences between the two query processing worlds: stream processing operates on events, while traditional stored data sources work with operations (read/write). While there is an ordering defined among events (e.g., based on timestamps, arrival order, etc.) and a possible ordering among the operations (e.g., defined by a transactional model), there is no well-defined order across events and operations, which makes it impossible to compare them directly. Moreover, traditionally, queries are one-time, while stream processing uses long-running, continuous queries.

EDBT 2012, March 26–30, 2012, Berlin, Germany.
https://doi.org/10.1145/2247596.2247622

Intuitively, there are two alternative solutions to this problem: (i) either non-streaming sources can be transformed into streams (e.g., by creating an event for each DB operation), or (ii) streaming sources can be treated as data sources with regular read/write operations. Most of today's SPSs implement solution (i). The main problem with this approach is that each engine has embedded the transactional properties into their execution models and operator semantics, making them hard to understand and inflexible to use. Furthermore, because of the high degree of heterogeneity in these systems' query execution models [11], each ended up proposing its own implicit "transactional model".

In this paper, we propose a *unified transactional model* (UTM) for streaming and stored data sources based on solution (ii). Our approach reuses decades of research on transactional database theory and exploits the traditional transactional model [29], to define correct executions of both continuous and one-time queries over an arbitrary mix of stored and streaming data sources. The basic insight is that we treat streaming and stored inputs uniformly: they are just data sources to the continuous queries with reads and updates. Moreover, a continuous query is modeled as a possibly infinite sequence of one-time queries. This way, the problem becomes similar to the one solved by the traditional transactional model.

We first analyze the page model [29] regarding its expressive power of allowing the correct execution of streaming applications. We then extend traditional transactions with: (i) *events*, which are represented by write operations, and (ii) *continuous queries*, which are represented by read and write operations on the data sources (including both inputs and outputs). As such, events and/or individual continuous query executions can be grouped together into transactions, thus flexibly defining isolation units not bound to any specific query execution model or operator semantics. We wanted our model to be general enough to be applicable to any query language (e.g., CQL[6], CCL[1], XQuery, StreamSQL [3], etc.), by adding basic primitives such as *commit* and *abort*. Nevertheless, designing the right syntax to define transactions for continuous queries is beyond the scope of this paper.

We propose conflict-serializability to the correct interleaving of operations belonging to these transactions, but similar to the traditional case, weaker constraints on the ordering can be defined. The result is a clean semantics for continuous query execution over streaming and stored data sources even in the presence of failures. Moreover, our analysis of a series of state-of-the-art commercial and academic SPSs showed that their transactional behavior can be described using our model.

In summary, this paper makes the following contributions:

- It defines the space of possible executions of continuous and one-time queries over combinations of streaming and stored data sources under concurrent processing and failures.
- It shows how events and operations can be arbitrarily grouped into transactions, and that conflict-serializability can be readily applied as a criterion for defining correct executions. As such, this paper proposes a unified transactional model (UTM).
- It shows the performance of the first implementation of a unified transaction manager as part of MXQuery/SMS [10], a storage manager for streaming and stored data sources, using the Linear Road Benchmark [5].
- It describes the implicit transactional behaviors of state-of-the-art SPSs expressed in the execution space.

The remainder of this paper is organized as follows: We define the transactional stream processing design space and present our unified transactional model (UTM) in Section 2, and illustrate the practical uses of this model in Section 3. In Section 4, we present the implementation of a transaction manager on top of SMS, a storage management system for streams [10]. After describing the details of our experimental setup, which includes the implementation of the Linear Road Benchmark in MXQuery [9] in Section 5, we present our experimental results in Section 6. Section 7 provides an analysis of the implicit transactional behaviors of five academic and commercial SPSs, while Section 8 compares our work with related work. We conclude the paper in Section 9.

2. TRANSACTIONAL STREAM PROCESSING

In this section, we first present our analysis of the traditional page model [29] with respect to the expressive power that it has in order to meet the requirements of data stream processing applications, and then propose the necessary extensions to turn it into our unified transactional model (UTM).

The main challenge in dealing with streaming and stored (e.g., relational) data sources is that the former operate with *events*, while the basic execution unit for the latter is an *operation* (i.e., either a read (r) or a write (w)). Moreover, continuous queries have different semantics than one-time queries. The problem is how to make these different data and query models comparable, so as to be able to define *correct* executions that involve both.

2.1 The Data Model

In order to apply the page model, we need to first characterize our design space with respect to the data sources (query inputs) and sinks (query outputs), and their corresponding operations.

We propose to treat all the data sources (streaming, relational, stored non-relational, etc.) uniformly: they are all *sets of data items* on which *operations* (reads and writes) are executed, closely following the page model [29]. As a result, a *relation*, which traditionally is defined as an unordered set of tuples (data items) sharing the same schema, is easily integrated into this design space. The operations exposed by a relation are: insert (add new tuples to the relation), delete (remove tuples), value update (replace existing tuples' attribute values), and read (obtain the values of tuples).

As a special type of data source, the *stream* has been previously defined in many different ways. The difference lies in the interpretation of a stream, which depends on the application [23]. Moreover, a stream's definition embeds both the data and update semantics of that stream. In order to apply the page model, we need to abstract away from a stream's semantics. Thus, we adopt the following general definition: a stream is a possibly infinite partially ordered set of events, where an *event* can be interpreted as a tuple with a special ordering attribute. More formally, a stream is defined by the pair $S = (E, \leq_S)$, where E is a set of events and $\leq_S \subseteq E \times E$ is a partial order on the set E.

As a result, a *stream* can be viewed as an unbounded set of tuples (with a special attribute for order) on which the only possible update operation is an *insert*. As such, we convert each event arrival to a write operation (*insert* $\rightarrow$ *write*), thereby allowing the comparison of events with other data access operations. As an example, in Figure 1, the arrival of event (D3, 50, 5) (having 5 as the value of its ordering attribute, $Time$) will be represented by a write on the input stream IS ($w(IS)$).

A stream's state is represented by its content which can be retrieved through a *read* operation. For example, in Figure 1, a read operation on IS after the arrival of event (D2, 24, 4) will return the state of the stream IS as {(D3, 10, 3), (D2, 24, 4)}, while a read operation after the arrival of event (D3, 50, 5) will return the state as {(D3, 10, 3), (D2, 24, 4), ..., (D3, 50, 5)}, and so on.

We consider the most general case in which there are multiple *writers* (processes which try to concurrently append new events to the stream) and multiple *readers* (continuous queries which process the events in the data streams). In the same way, other types of data sources with different sets of supported access operations (which could be represented through read or write operations) can be included in this general design space.

2.2 The Query Model

Another challenge arises when dealing with *continuous queries*. Continuous queries [27] are issued once and run continually, sending newly generated results to the user application which registered them. Given their continuous nature, streaming queries are not a good match for a transactional model, as transactions are closely tied to a one-time query model.

Our proposal is to represent a continuous query as a (possibly infinite) sequence of one-time queries which are fired as a result of the data sources being modified (e.g., arrival of new events, update of existing inputs, etc.) or by periodic execution (e.g., every second). Basically, a one-time execution of a continuous query can be translated into read operations on all its input data sources plus (possibly) a number of write operations, corresponding to the results that it may generate. Such an approach has also been proposed in the context of temporal databases [26].

For example, in the example scenario of Figure 1, suppose that the arrival of event (D2, 24, 4) triggers the query execution. The join execution is then represented by a read on the input stream ($r(IS)$) and a read on the table ($r(R)$), followed by a write on the output stream ($w(OS)$), generating of the output event (D2, 24).

As a result, our design space is composed of one-time queries only, among which the ones corresponding to continuous execution have a predefined structure.

2.3 The Unified Transactional Model (UTM)

A transaction in our design space obeys the traditional definition (a partially ordered sequence of operations), but we adapt it in order to accommodate events and continuous queries. More formally:

Definition 1 (Transaction) *Given a set of data sources and sinks DS={S_i, S_o, ...}, composed of data items DI, a transaction is represented by the pair T=(O_T,$\leq_T$), where O_T is a finite set of operations (of the form* r *or* w*) on the sources and sinks in DS and $\leq_T$ is a partial order on the operations such that:*

- *Each new streaming event e_p, is represented by a write operation on the corresponding stream ($w(S_i)$) and each (one-time) continuous query execution is represented by a read on all the input data sources of the query possibly followed by a sequence of writes on the sink(s) ($w(S_o)$).*
- *$\forall o_p, o_q \in O_T$ which access the same data item and one of which is a write, either $o_p \leq_T o_q$ or $o_q \leq_T o_p$ (i.e., operations on the same data item are ordered).*

For example, in Figure 1, the arrival of event (D2, 24, 4) generates the join operation execution over the stream IS and the relation R which produces event (D2, 24) to the output stream OS as explained earlier. Suppose the application implementing this scenario specifies that all these tasks (arrival of event, query execution, output of results) should be grouped in a single transaction. The resulting transaction will have the following structure: $T_1 = (w_1(IS)\ r_1(IS)\ r_1(R)\ w_1(OS))$. That is, the event's arrival in the input stream ($w_1(IS)$), followed by a one-time continuous query execution ($r_1(IS)\ r_1(R)\ w_1(OS)$).

Traditionally, transactions can end in two ways: successfully or not. To express the termination of a certain transaction (when that state is known), two special operations are used: *commit* and *abort*. As such, a transaction can be in one of three basic states: active (is currently running and has not yet reached a final state), committed or aborted (corresponding to the two possible termination states).

In our design space, Commit is an operation which defines the successful termination of a transaction (all the operations in the transaction have been executed with no error) such that the result generated can be made visible to other transactions. In the case of stored data sources, Commit also specifies that these results are made permanent.

An Abort operation expresses that while processing a transaction, something happened (a violation of consistency, system crash, etc.) which interfered with the normal execution. As aborted transactions may leave the data sources in an inconsistent state, their effect should be undone. As a result, the data in the sources appear as if this transaction had never been executed at all.

Suppose that while executing transaction T_1, the SPS encounters an error, such that operation $w(OS)$ cannot be executed. Nevertheless, the event (D2, 24, 4) is written in the input, and is most probably part of the state of the join operator, but an alert will not be generated. To ensure that no alert is missed, the SPS should return to the state before the triggering event arrived in the stream ($r(IS)$={(D3, 10, 3)}) and possibly re-execute the transaction.

In terms of transaction visibility, many DBMSs relax isolation to improve performance [8]. In this respect, for our UTM, the *streams* need some special consideration: since we restrict the operations on streams to *insert*, data cannot be recalled or modified. In the most strict sense, the use of *streams* calls for serializability. If weaker isolation levels are required, instead of a *stream*, another type of data source that supports cascading deletes and/or compensations should be used. Such data sources would closely resemble those proposed in revision processing [25].

The definitions for (serial) execution history, schedule, conflict, conflict-equivalence, and conflict-serializability can be reused without change [29]. For completeness, we list these definitions here:

Definition 2 (History) *Given T = {T_1, T_2, ..., T_n} a (finite) set of transactions where each $T_i \in T$ has the form T_i = (O_{Ti}, $\leq_{Ti}$), a history is a pair H = (O_h, $\leq_h$), such that:*

- *$O_h \subseteq \bigcup_n^{i=1} O_{Ti} \cup \bigcup_n^{i=1}\{C_i, A_i\}$ and $\bigcup_n^{i=1} O_{Ti} \subseteq O_h$ (the operations belonging to the transactions are all contained in the history plus a control event (C or A) for each transaction)*
- *$\forall i, 1 \leq i \leq n, C_i \in O_h \Leftrightarrow A_i \notin O_h$ (for each transaction there is either a commit or an abort, but not both)*
- *$\bigcup_n^{i=1} \leq_{Ti} \subseteq \leq_h$ (all transaction orders are contained in the history order)*
- *$\forall i, 1 \leq i \leq n, \forall op \in O_{Ti}, op \leq_h A_i$ or $op \leq_h C_i$ (the control operation (A or C) is always the last operation of a transaction)*
- *Every pair (op_r, op_w) of operations, $op_r, op_w \in O_h$ from distinct transactions, accessing the same data item, one of which is a write, are ordered in H in such a way that either $op_r \leq_h op_w$ or $op_w \leq_h op_r$.*

Definition 3 (Conflict) *Two operations on the same data item, so that at least one is a write, are said to be in conflict.*

Definition 4 (Schedule) *A schedule represents a history prefix.*

Definition 5 (Serial History) *$SH = (O_{sh}, <_{sh})$ is a serial history if $\forall T_i(O_i, \leq_i), T_j(O_j, \leq_j), T_i \neq T_j, T_i, T_j \in SH$, $\forall op_{ik} \in O_i$ and $\forall op_{jl} \in O_j$, either all $op_{ik} <_{sh} op_{jl}$ or all $op_{jl} <_{sh} op_{ik}$ (a history SH is serial if for any two distinct transactions T_i and T_j contained in SH, all the operations belonging to T_i are ordered in SH before all the operations in T_j or vice versa).*

Definition 6 (Conflict-Equivalence) *$\forall H_i, H_j, H_i = (O_{hi}, <_{hi}), H_j = (O_{hj}, <_{hj}), H_i \equiv H_j \Leftrightarrow O_{hi} \setminus \{A, C\} = O_{hj} \setminus \{A, C\} = O$ and $\forall o_k, o_l \in O$ s.t. o_k, o_l are in conflict, then $o_k <_{hi} o_l \Leftrightarrow o_k <_{hj} o_l$ (two histories H_i and H_j are said to be conflict-equivalent iff they contain the same operations and they order all the conflicts in the same way).*

Definition 7 (Conflict-Serializability (CS)) *A history H is said to be conflict-serializable if it is conflict-equivalent to a serial history.*

Testing a history for its membership to the conflict-serializable class can be done efficiently by using *conflict graphs*. A conflict graph is a directed graph which contains the transactions as nodes and the edges represent conflicts between the transactions. The orientation of the edges follows the order of conflicting operations in the history. A cycle in the graph indicates that the underlying history is not conflict-serializable.

Before presenting our correctness definition, we also define two notions: *global schedule* and *global conflict-serializability*. In a multi-source environment, a global schedule represents the union of schedules executed at each source. A global schedule is globally conflict-serializable if the conflict-serializability property holds.

2.4 Execution Correctness

Our problem can now be described as follows: given a finite set of data sources and sinks ($DS_i, 1 \leq i \leq n$), an infinite set of updates on the data sources ($U_j, 1 \leq j$), and an infinite sequence of one-time queries accessing the sources, some of which (updates and/or queries) may be grouped into transactions, what defines a correct execution?

In this work, we propose to use conflict-serializability as a criterion to specify correct interleaving of operations. Our main motivation behind this choice is to provide a general definition of this interleaving. Alternatively, we have also considered using other criteria such as snapshot isolation [8] or view / final state serializability, but found problems with these. More specifically, the latter is impractical to implement since schedules are not monotone [29] in that a new operation can transform an illegal schedule into a legal one [24]. The former is not suitable because it is unclear how to define snapshots and assign timestamps to them in a heterogeneous environment such as an SPS. On the other hand, in the context of serializability, there are widely used protocols which generate 'globally' serializable schedules (see Section 4).

As a result, we define correct executions as follows:

Definition 8 (CS-based Correct Execution) *A history of operations belonging to a set of transactions T = $\{T_1, T_2, ..., T_n, ...\}$ generated as a result of executing a set of queries (one-time and continuous) Q over a set of data sources and sinks DS is correct if it is globally **conflict-serializable**.*

Conflict-serializability is an example of how our model can be used to define correct execution. As needed, it may be restricted or relaxed, and as in [30], different correctness *levels* can be defined. For example, processing the events in their arrival order in a stream is important for the correct implementation of many streaming applications. Moreover, the majority of SPSs rely their execution models on the arrival order of the events. As serializability means equivalence to *any* serial execution, the correctness condition should only accept schedules which obey the arrival ordering. As a result, a more constrained level of correctness can be defined (similar to *Strong Consistency* in the above-cited work):

Definition 9 (CS with Arrival Ordering (CSAO)) *A history H is said to be conflict-serializable with arrival ordering if it is conflict-equivalent to a serial history in which the order of transactions generated by sequences of events (stream updates and corresponding continuous queries executions) obeys the events' arrival order in their respective streams.*

Definition 10 (CSAO-based Correct Execution) *A history of operations belonging to a set of transactions T = $\{T_1, T_2, ..., T_n, ...\}$ generated as a result of executing a set of queries (one-time and continuous) Q over a set of data sources and sinks DS is correct if it is globally conflict-serializable and conflict-serializable with arrival ordering.*

2.5 ACID for Streams

Serializability is based on the notion that the transactions obey the ACID (Atomicity, Consistency, Isolation, and Durability) properties. In this subsection, we analyze these from a streaming perspective.

Atomicity and **Isolation** are already covered by our UTM.

Traditionally, **Consistency** refers to maintaining integrity constraints of a database. In the case of streams, Stream Schema [14] can be used to model stream constraints (and validate consistency).

The most interesting property from a streaming point of view is **Durability**. Traditionally, durability specifies that the changes made by a transaction are made persistent if the transaction is committed. As streams are not persistent, it appears that the durability is not relevant for streaming transactions. Nevertheless, if we consider it as stating that operations of a committed transaction survive failures, the durability semantics for streams is then: the events of committed transactions will never be reprocessed or duplicated.

3. THE UTM IN ACTION

We can now analyze an individual processing model (namely STREAM/CQL [6]) and a concrete workload with respect to our UTM. In a first step, we outline how the mapping from a single continuous query to a sequence of one-time queries, as outlined in Section 2.2, can be performed. In a second step, we show how transactions can be defined to achieve the same transactional semantics and correctness guarantees as this processing model.

3.1 Mapping Streaming Operations

As outlined before, a key aspect in our UTM is to represent a continuous query as an equivalent sequence of one-time queries. Since query semantics and execution models differ greatly between streaming systems [11], a single solution for all models is not possible. We therefore show how the most relevant operators on representative models can be translated into one-time query executions. This particularly includes operators which consider more than one item to produce an output item (e.g., window-based aggregation). Other operators (without windows) are typically very similar for continuous and one-time systems and have a trivial translation: pick the last item or timestamp group and perform the operation.

The main challenges in this translation are to achieve equivalent results (same tuples, same order), to ensure termination of

the resulting one-time queries, and to transfer information between different instances of one-time queries for coordination (e.g., instance/slide selection).

There are two complementary approaches to perform this translation:

- Creating a sequence of (finite) snapshots of the stream, treating them as relations and running the query on them (like CQL's stream-to-relation mapping)
- Rewriting the queries with additional predicates/expressions

We are now going to explain how this translation can be applied on STREAM/CQL. The series of one-time queries and the snapshots they are run can be created in an iterative fashion for each timestamp t_i. As a prerequiste, we define R_{i-1} as the set of items that have been read in the previous execution at timestamp t_{i-1}. Then: $R_{i-1} = \bigcup(r(X))$ at execution t_{i-1}, X being any resource accessed by the query. Following the processing model of CQL, each execution is triggered by a new timestamp. The execution then works as follows:

- Create a suffix stream (snapshot/view) including timestamp t_x to t_i, where t_x is smallest $t_n \in R_{i-1+1}$ + *window slide* or 0 if R_{t-1} empty, thus advancing the snapshot by slide timestamps.
- Construct the window according to the (unchanged) formal definition of CQL (all tuples t' between t_i and $t_i - T$).
- This will generate exactly one window per execution, since $t_i - (t_i - T) = T$. In turn, concatenating the window results leads to the same order as the continuous query. Therefore, we have a correct translation which expresses all its coordination needs using the snapshot generation.

Let us illustrate the above translation on a concrete example:

```
SELECT AVG(Temp)
FROM Stream [RANGE 4 SECOND SLIDE 1 SECOND]
```

Consider the following input data:

```
Input(DevID,Temp,Time)={(D3,10,3),(D3,50,5),(D3,25,6),
                        (D3,30,7),(D3,35,8)}
```

For the continuous query, we would expect the following result:

```
Output(Temp,Time) = {(3,10),(5,30),(6,28.3),(7,35),
                     (8,35)}
```

The following snapshots are generated:
$t_i = 3, t_x = 0$; (D3, 10, 3) => (3, 10)
$t_i = 5, t_x = 3$ ($R_1 = \{T1\}$); (D3, 10, 3) (D3, 50, 5) => (5, 30)
$t_i = 6, t_x = 3$ ($R_2 = \{T1, T2\}$); (D3, 10, 3) ... (D3, 25, 6) => (6, 28.3)
$t_i = 7, t_x = 4$ ($R_3 = \{T1, T2, T3\}$) ...

Other stream processing models can be translated along the same lines, but need more complex adaptations in snapshots and queries.

3.2 Defining Transactions

Using our UTM instead of model- or application-specific synchronization primitives provides abstraction and additional flexibility for transaction boundaries, but raises the issue of how to set these transaction boundaries. Continuous queries do not naturally lend themselves to obvious boundaries, and in cases like predicate-based windows, such a boundary may even be unknown in advance.

From a *conceptual* point of view, such flexibility is needed to express the different boundary models of existing SPS, as we show in Section 7. This is not possible with a more restricted model.

From a *practical* point of view, we envision some options which a system designer or application developer could choose from: A system might define its default boundaries, as existing systems do, either in a fixed form or as some kind of "autocommit". When users want to set their own boundaries, aligning with operator semantics (as in Truviso [4]) or punctuations (as CTIs in StreamInsight [2]) would be typical approaches to overcome the problem of boundaries being unkown in advance.

For the window-based query in the previous section, we can place a commit at the end of each query run over a snapshot, and thereby express the synchronization on timestamps in CQL.

For our example scenario of Section 1, we have chosen the simplest transaction grouping that will produce the correct result. Let us present this example in more detail.

As previously assumed, for each new event, the SPS probes the database table for specifications and outputs alerts in case of violations. Suppose the arrival order of the streaming events is the one depicted in Figure 1 by looking at the stream's representation from right to left (event (D3, 10, 3) followed by (D2, 24, 4), followed by (D3, 50, 5), etc).

The application's requirement is that each device temperature measurement has to be compared to the device specifications in the required scale (the assumption we make is that once the first Fahrenheit measurement for a certain device arrives, all the following will have values on the same scale). More specifically, the update on the table has to happen *before* the first Fahrenheit measurement for the respective device arrives and no Celsius measurements should *see* specifications in the Fahrenheit scale.

One way to meet the above requirements is to group the device's specification update and the corresponding first event in the Fahrenheit scale into the same transaction, with the update on the table preceding the processing of the event. This way we obtain the desired visibility of the table update by taking advantage of the ordering of conflicting operations in a transaction. In fact, this grouping also ensures that the specification update and the processing of the event happen atomically. As a result, we obtain four transactions:

$T_1 = (w_1(IS)r_1(IS)r_1(R)C_1)$ (for event (D3, 10, 3))
$T_2 = (w_2(IS)r_2(IS)r_2(R)w_2(OS)C_2)$ (for event (D2, 24, 4))
$T_3 = (w_3(IS)w_3(R)r_3(IS)r_3(R)C_3)$ (for event (D3, 50, 5) and the update of D3's specification)
$T_4 = (w_4(IS)w_4(R)r_4(IS)r_4(R)C_4)$ (for event (D2, 61, 5) and the update of D2's specification)

Processing the events in the order of arrival guarantees that no Celsius measurement will be compared with Fahrenheit specifications. Therefore, the correct histories for our example application will be Conflict-serializable with Arrival Ordering (i.e., any history which is conflict-equivalent to the following serial execution: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$).

4. IMPLEMENTATION

In this section, we present the implementation of the Transaction Manager we built on top of SMS [10], a general-purpose storage manager for Stream Processing Systems.

4.1 Transaction Manager Implementation

The job of a transaction manager is to both order the storage access operations it receives from the streaming system, in order to obtain correct histories (accepted under the correctness definition) as well as to provide rollback execution to aborted transactions. In our case, this translates to designing a transaction manager which generates histories obeying the CSAO-based correctness criterion defined in Section 2.4.

Just like in the traditional case, we designed the Transaction Manager (TM) as an additional component between the query processing and the data access layers, as shown in Figure 2.

We implemented the SS2PL (Strong Strict 2-Phase Locking) protocol in our TM [29]. SS2PL defines an acquiring phase in which

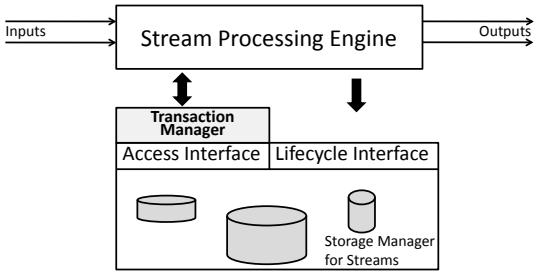

Figure 2: Storage Manager with TM Architecture

a transaction is only allowed to acquire locks for the resources it is about to access. These locks are all released when the transaction commits, as opposed to other 2PL protocol variants which allow early releases. SS2PL is the protocol of choice for many general-purpose database systems, because of its desirable properties: commit ordering which provides global serializability (a property required by our correctness criterion as explained in Section 2) and recoverability (a property which specifies that the resources modified by a transaction can be brought back to the states before the transaction started in case of an abort operation).

We adopt the design presented in [29]. Two types of data structures were used to implement the lock manager: RCB (Resource Control Block) and TCB (Transaction Control Block).

An RCB is defined for each resource being concurrently accessed by transactions. It maintains a list of transactions currently holding a lock on it and the mode of the lock (shared or exclusive). A hash table maps each resource object to its corresponding RCB. When a lock request for a certain resource is received, there has to be a way to check whether another transaction already holds a conflicting lock on this resource. The RCB offers an efficient way to execute this check operation.

The TCB (Transaction Control Block) maintains a list of all resources a transaction has accessed, including the mode (shared or exclusive). This enables efficient lookup and release of the resources of the transaction at termination (commit or abort).

4.2 Transaction Manager Interface

The TM interface exposes five methods:

startTransaction(): Whenever a new transaction is started, this method has to be called before the execution of any operation. Internally, the method creates a new transaction object which will be assigned an ID and added to the list of active transactions. Each transaction object (Transaction Control Block) maintains a list of objects it has locked and in which mode (shared or exclusive).

write(resource R, transaction TID): When calling this method, a transaction attempts to access the resource given as parameter in write mode. First, the RCB corresponding to resource R is obtained. Then, the TM has to check whether this transaction should be allowed (at this point) to access the resource or should be put on hold until the access conditions are met: the resource is not locked by any other transaction and the defined ordering on the transactions, if any, is not violated by the execution of the current operation. If the request permission is granted, then the transaction sets an exclusive lock on the resource (if not already holding it), the RCB of the resource is updated, and the transaction is allowed to execute the update. Otherwise, the operation is put on hold. If an application failure happens, the transaction is aborted and a rollback procedure is executed (see Section 4.4 for details).

read(resource R, transaction TID): A transaction with the given TID attempts to read a resource. First the RCB of this resource is obtained and the TM checks if any of the transactions accessing this resource has an exclusive lock. If so, the request is put on hold until the resource is freed. Otherwise, if executing the read operation still maintains the defined ordering of transactions, then the transaction obtains a read-lock on the store and is allowed to access it (multiple readers are allowed to access the store at the same time).

commit(transaction TID): This method is called by a transaction when it requests a commit operation. At this point all the locks held by this transaction are released and its changes are made persistent (depending on the type of resource, this action means making all the writes persistent, discarding rollback information, etc.). As the resources accessed by this transaction are released, the TM can proceed to schedule the next operations.

abort(transaction TID): The engine specifies that the given transaction should be aborted. The modifications made by this transaction are undone through a rollback operation and all the locks it acquired are released.

4.3 Defining the Transactions

Transactions are defined by the streaming engine given the application requirements. One special characteristic of our TM implementation is the fact that it enables predefined orderings on the received transactions. For example, one can define the order of the transactions (given by the ordering of their conflicting operations) to be the increasing order of the transaction IDs, and the TM will enforce this ordering.

To provide a flexible way of defining individual transactions, we chose to use punctuations to specify transaction boundaries. We injected these punctuations in the streams to be processed by the streaming engine along with the other tuples. When the engine has to process a punctuation, it calls the corresponding method (startTransaction, commit) exported by the TM. If the transactions are not composed of sequences of events (e.g., events with the same value for a certain attribute), then other methods (e.g., adding a special attribute to keep the transaction ID) can be used.

4.4 Recoverability and Recovery

Our model allows to define correct executions in the presence of failures as well. Recoverability is a property of a concurrency control algorithm which defines whether it generates recoverable schedules or not. A recoverable schedule is one in which the effects of an aborted transactions can be undone and the system can be brought back to the state before this transaction has occurred.

Using SS2PL (Strong Strict 2-Phase Locking), a recoverable protocol, ensures that our TM allows correct recovery from failures.

In our TM implementation, for each write operation, the change performed is logged: the modified resource plus the function applied. When a transaction is aborted, the following steps are taken: for each write operation belonging to the aborted transaction, an inverse operation is executed (e.g., if the write operation added value 10 to a column in a table, then the inverse operation will deduct 10). The resources which were only read as part of this transaction are not affected by the abort. After this phase, all the locks held by the transaction are released. The SS2PL protocol is recoverable because the locks are only released at the termination of a transaction and therefore no other transaction will read a *dirty* value.

4.5 Performance Optimizations

Many streaming applications have strict constraints on throughput or response time [5]. As a result, the performance of the Transaction Manager is important for meeting applications' performance requirements.

4.5.1 Increased Parallelism

The page model is general enough to describe many implementation related issues. Nevertheless, its major drawback is that it does not take into account the semantics of the data access on the sources. That is, one could use the access pattern of the queries on the data to implement different locking granularities (a tuple, a group of tuples, etc.) and therefore reduce contention and increase the execution parallelism.

For example, the state of the art in relational database systems is record-level locking in combination with index locking [29]. As the write operations in a stream are actually appends, it results that read operations will practically not interfere with them. As a result, to reduce lock contention, different lock granularities can be defined: e.g., one could change the scope of the read and write operations from the whole stream to individual events. Another solution is to use ranges: events having ids in a certain range are locked for reading or writing (similar to predicate locking [29]).

4.5.2 Low-overhead Recovery

Recovery can be a time-consuming process as all the changes made by aborted transactions have to be undone.

Nevertheless, by using the semantics of the data source the rollback process can be optimized in some cases: when the data sources are streams and the transactions are defined as sequences of inputs, inconsistencies from aborted transactions can only occur from incomplete execution of the last active transaction. As a result, returning the stream to the previous state merely requires undoing a number of append operations.

5. EXPERIMENTAL SETUP

In this section, we present the details of our experimental setup: the benchmark description, the implementation details, the hardware on which we ran the experiments, as well as the metrics used.

The goal of our performance study is two-fold:

- to show that even with the overhead generated by the presence of a transaction manager, we can achieve an acceptable performance, i.e., very close to the one obtained with an implementation with specialized synchronization methods (the kind that other streaming systems use), and
- to show that a streaming application implementation can benefit from the presence of a transaction manager: the correctness requirements of a given application can be easily translated into isolation properties and implemented with very little effort only by specifying transaction boundaries (which can be easily modified). Moreover, the application developer does not have to deal with failures as this task is successfully handled in the transaction manager.

5.1 The Linear Road Benchmark

Our experimental study is based on the Linear Road Benchmark [5]. Linear Road simulates the traffic on a set of highways divided into segments and provides variable tolling depending on traffic statistics and accident occurrences. The input stream is composed of car position reports (each car reports its position every 30 seconds) and queries. An engine that implements the benchmark has to distinguish between the different types of streaming items and execute the queries accordingly.

The benchmark involves all in all, five queries: *Accident Notification* - drivers are notified of accidents in their vicinity when crossing to another segment on the highway; *Toll Notification* - drivers are notified of the toll corresponding to the segment they are entering; *Balance Request* - a driver can request its current balance of assessed tolls; *Daily Expenditure Query* - how much has a driver spent on a particular day in the past 10 weeks; One more query for travel time estimation which is not included in any published implementation so far.

The measure of this benchmark is given by the *Load level*, representing the number of highways that can be handled by the query processing engine. A certain load level is considered to be achieved, when all the queries involved in the benchmark are answered within at most 5 seconds after the request entered the system.

An engine which implements the queries correctly has to generate the exact number of alerts and correct alert values, which are verified using a *validator* [5].

For our experiments, we used the Linear Road Benchmark implementation in MXQuery [9], a Java-based open-source streaming engine which extends XQuery with window functions. One of MXQuery's design decisions was the separation of query processing from storage management. MXQuery uses SMS [10] as its underlying storage manager. We created two versions of the implementation (ADHOC and TM), which will be presented next.

5.2 The ADHOC Implementation

This Linear Road implementation uses a version of SMS with no transactional capabilities, but synchronization methods. We implemented two variants: ADHOC SERIALIZABLE, which orders the operations on the stores so that the execution *behaves* like a serializable one, and ADHOC NOT SERIALIZABLE, where the only restriction is that the writers have exclusive access to the object (i.e., store) they modify.

This implementation contains a set of queries connected by stores as shown in Figure 3 (the rectangles represent the queries, while the cylinders represent the stores). The arrows express the data flow between the different queries. It is important to also note that as a result of the benchmark's requirements, the stores expose high heterogeneity: there are streams (e.g., containing car position reports), in-memory relations (the BALANCE store), a static relation stored in database system (HISTORICAL TOLLS), as well as files in which the results of the queries are written.

Two queries are particularly challenging because of their requirements: *Accident Notification* and *Toll Notification*. The first query generates a notification for each driver who reports a position from a new segment in the vicinity of an accident. The query description specifies that alerts should be sent for all active accidents up to and including the minute before the car position report. As the information about the accidents which happened in the past minute may not be available when a certain car crosses a segment (due to the fact that the events come on different processing paths with different speeds), we need to keep the request waiting until the most recent accident data is available. This logic is implemented through built-in synchronization methods in the stores themselves: a read operation is blocked until a predicate is evaluated to true. In the second query, for computing the toll of a specific segment on a highway, the traffic statistics of that segment are used. Statistics are obtained every minute from analyzing a five minute window worth of traffic information (number of cars, speed, etc.). Again, if the tolls computed for a certain minute are not yet written to the TOLLS store, the Toll Notification query is delayed until that data is available.

The ADHOC implementation requires that the developer has a good understanding of both the application semantics as well as the synchronization primitives. The next implementation shows how the requirements of the benchmark are met by defining transactions and the order among them.

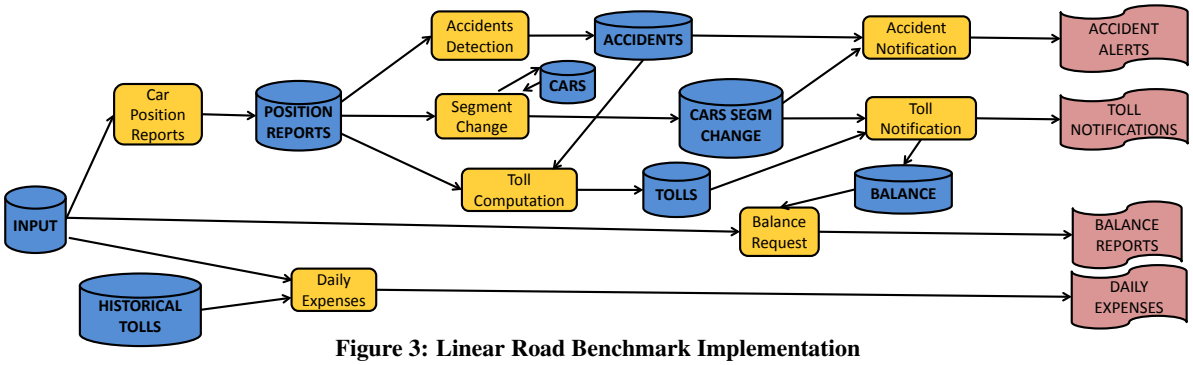

Figure 3: Linear Road Benchmark Implementation

5.3 The TM implementation

For this Linear Road implementation, SMS was enhanced with a transaction manager, as described in the previous section.

In the TM implementation, we basically replace the built-in synchronization methods with a transaction manager which orders the operations according to the correctness requirements of the queries. Moreover, we implemented a mechanism in the storage manager which allows it to recognize the particular events specifying transaction boundaries (start transaction, commit, abort etc.). As in the ADHOC case, we have TM SERIALIZABLE and TM NOT SERIALIZABLE variants.

The Toll Notification query is triggered for every position report sent by a car entering a new segment (each translated to a write operation on the CARS SEGM CHANGE store). As required by the benchmark, we would like that none of the notifications is lost and that the correct toll value is charged to each car. For that, we group all the operations of a one-time Toll Notification query execution in the same transaction, which will then be composed of four operations: read the car position report, r(CAR SEGM CHANGE), extract the toll for that car, r(TOLLS), update the driver's balance with the new assessed toll, w(BALANCE), and generate the toll alert, w(TOLL_ALERTS) . The transaction is committed after the last operation.

Moreover, all the toll values computed for a certain minute are written to the TOLL store as part of the same transaction, whose structure will then be: r(POSITION REPORTS) to read the position reports, followed by looking up the accidents store r(ACCIDENTS) and then writing the toll values w(TOLLS). If the execution of these transactions is serializable, we make sure that all the toll values read by the Toll Notification query are a result of the most recent computation and that there is no mix of old and new values.

We do the same for the Accident Notification query: r(CAR SEGM CHANGE) to get the most recent car position report, followed by r(ACCIDENTS) to select all the accidents in the recently entered segment and possibly w(ACCIDENT_ALERTS) if a notification needs to be sent. Again, all the accident events detected during a minute are grouped in the same transaction: {r(POSITION REPORTS) w(ACCIDENTS)}.

For the Balance Request query, a transaction is composed of a read on the INPUT store to obtain the most recent car position report, r(INPUT), followed by a lookup on the BALANCE store to retrieve the driver's balance, r(BALANCE), and finally generate the balance report, w(BALANCE_REPORTS). While the first two queries require a strict ordering of the transactions, this third query allows a more relaxed ordering: a driver can be notified of her most recent (or the one-minute old) balance of assessed tolls. This property accepts more schedules, permitting the Balance Request query to lower latency.

All other transactions have a simple structure, basically a read on an input store followed by a write on an output store.

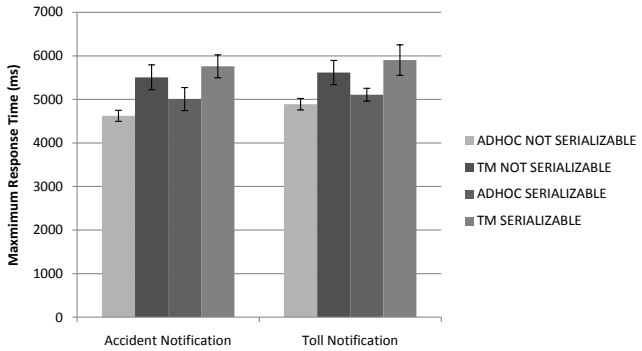

Figure 4: ADHOC vs. TM Maximum Response Time

5.4 Setup Details and Metrics

The experiments were run on 2 x Quad Core AMD Opteron 2376 with 2.3GHz processor machines. We ran Linear Road to completion at load factor L=4.0, which is the highest that can be achieved by the ADHOC implementation.

The experimental results are presented using two metrics: the maximum response time and the correctness of results. We also looked at other measures of performance, such as average response time and the percentage of alerts of the total number that had a response time over the benchmark's limit of 5 seconds.

The response time (latency) of the queries was measured as the difference between the time the generated result is written to the file (or committed in the case of transactions) and the time the request enters the system. We repeated each experiment 15 times and the maximum response time values are presented as an average of those runs, plus an error bar representing the standard deviation.

We chose to measure the latency of the Toll and Accident Notification queries, because they are the ones on the processing paths with the highest load, and transactions they define are complex.

The correctness of results is represented as the number of wrong (which did not have the values expected by the validator), duplicated, or missing results.

6. EXPERIMENTAL RESULTS

In this section, we show the results of running the Linear Road Benchmark on top of the setup presented in the previous section.

6.1 Performance

First, we measured the overhead of the Transaction Manager. We wanted to understand how much the presence of the TM (compared to the ADHOC implementation) and enforcing the serializability property (compared to the not-serializable implementations) increase the latency of the results.

Figure 4 shows the maximum response times when running the TM and ADHOC setups in both the serializable and not-serializable

	ADHOC NOT SERIALIZABLE	ADHOC SERIALIZABLE	TM NOT SERIALIZABLE	TM SERIALIZABLE	Total Number of Alerts
Accidents Notification (missing alerts)	94	0	99	0	9'115'887
Tolls Notification (wrong alerts)	52'618	0	32'226	0	142'433

Table 1: Result Correctness: Number of Wrong/Missing Alerts

versions. Comparing the ADHOC NOT SERIALIZABLE and the TM NOT SERIALIZABLE setups shows the overhead generated as a result of maintaining transactions. As expected, the presence of the TM increases the maximum response time, but the performance penalty is quite low. The overhead of providing serializability can be observed by comparing the TM NOT SERIALIZABLE and the TM SERIALIZABLE setups (as well as ADHOC NOT SERIALIZABLE against ADHOC SERIALIZABLE). The maximum latency is lower for the not-serializable versions (about 250ms on average) as the read operations do not wait for the correct data to be available, but rather return results based on whatever is contained in the store at the time of execution.

Nevertheless, the differences between the ADHOC and the TM setups (the serializable versions) are small and the majority of the alerts have low response times, meeting the requirements of the benchmark. More specifically, in the TM SERIALIZABLE setup, on average, 95.4% of the accident alerts and 96.1% of the toll alerts have response times lower than 5 seconds.

We also measured the average response times, where the relative cost shows the same changes as for maximum response times. We decided to include maximum response times in the paper, as they correspond to the benchmark specification and they provide a better indication of overhead as the system is most stressed. While we have not performed an explicit test for the maximum throughput, the existing results already give clear indications that there is only a small slowdown introduced by the transaction implementation: The chosen Linear Road load factor of 4.0 was the maximum that could be handled in the ADHOC SERIALIZABLE implementation, while a load factor of 4.5 violated the requirements. Given that the LR implementation was well optimized, we had thus been very close to the throughput maximum. As a result, the increase in maximum response times and number of events above the 5-second limit can be attributed to the small throughput reduction.

6.2 Correctness of Results

Although it incurs some performance penalty, serializability is important for providing correct results. To get a better understanding of its importance, we measured the *correctness* of the results. We randomly picked one run of the Linear Road benchmark with the not-serializable implementation (for both the TM and the ADHOC setups) and compared the results with the expected ones (obtained with a serializable implementation). Results are presented in Table 1. As shown, in the not-serializable version, the ADHOC setup misses 94 accident alerts (because of accident information not being available at the time of request), similar to the TM setup which misses 99 accident alerts. Moreover, 52'618 toll alerts are wrong when the ADHOC implementation does not provide serializability, while the TM NOT SERIALIZABLE setup generates 32'226 wrong toll alerts.

6.3 Failure Handling

An important advantage of having a Transaction Manager is that the client application does not have to deal with failures. One of the features that a TM provides is automatic rollback (the objects modified by a transaction are brought back to the state before the transaction started) - a process which is not possible when transaction support is missing. To illustrate this, we present the following scenario: suppose that because of faulty behavior, every 2 seconds a position report (for a car which crosses a segment) is duplicated. As a result, duplicate toll and accident alerts are generated. Moreover, the balance for those cars will be wrong as the same toll will be charged twice. The problem is that, when detecting this error, the ADHOC implementation has no way to rollback the execution of the query generated by the duplicated event. Furthermore, whereas the duplicated alerts may just not be published to the output, the BALANCE store would still contain erroneous values which lead to wrong alerts in the Balance Request query. More specifically, as can be observed in Table 2, comparing the Balance reports obtained when the input has duplicates ('ADHOC SERIALIZABLE (duplicates)' in the table) with the correct ones, we could observe that on average about 8'000 (of a total of 242'458) were wrong.

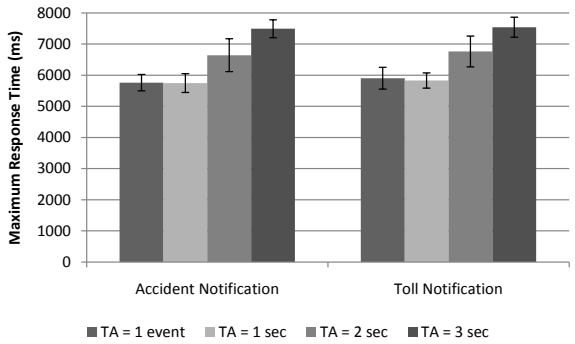

Figure 5: TM SERIALIZABLE: Sensitivity to Transaction Size

This is not a problem for the TM SERIALIZABLE setup: when detecting the duplicate, the current transaction is rolled back and the car balance is restored to the previous value, i.e., before the transaction started. Moreover, this benefit comes with almost no performance penalty as the results in Table 2 show: the maximum response time is practically not affected by rolling back 5'395 Toll Notification transactions. Note that, for the Accident Notification query, the duplicated events had no impact on the output, as the duplicated events did not lead to accident alerts.

6.4 Sensitivity to Transaction Size

In the next experiment, we measured the sensitivity of TM's performance to transaction size, varying the transaction boundaries to not just contain a single event (as before), but span longer periods of times (and thus more events).

As a baseline (and in the same way as the previous experiments), each one-time query execution was part of a distinct transaction (1 event). As a first step, we group all the one-time query executions corresponding to car position reports with the same timestamp (which arrive in the same 1 second, with approximately 850 events). The intuition is that some query executions will not pay the penalty of acquiring the necessary locks, especially for the transactions executed at minute boundaries (right after a commit of new toll values or accident information).

Furthermore, we grouped all queries corresponding to events arriving in subsequent time intervals of 2 or 3 seconds, which resulted in approximately 1'700 and 2'500 events respectively. This

Query	Metric	ADHOC SERIALIZABLE	ADHOC SERIALIZABLE (duplicates)	TM SERIALIZABLE	TM SERIALIZABLE (duplicates)
Tolls Notification	# Duplicated Alerts	0	5'395	0	0
	Maximum Response Time (ms) / Std. Deviation	5'108 / 147.5	4'954 / 301.6	5'902 / 349.9	6'048 / 397.7
Balance Request	# Wrong Balance	0	8'000	0	0

Table 2: Failure Handling: Performance vs. Correctness

coarser-grained grouping was done by moving the punctuations from a per query-level to after every second, every second or third second. Moreover, grouping the queries does not affect the correctness of results as long as the transaction size does not exceed one minute (all one-time query executions corresponding to events with timestamps in the same minute). We did not try other transaction sizes, as the maximum response time exceeded 7 seconds when defining transactions composed of one-time query executions corresponding to alerts arriving in sequences of 3 seconds.

The results are presented Figure 5. As expected, the maximum response time increases with the size of the transaction. There seems to be no statistical difference in the maximum response time when defining the transactions at 1-second boundaries, compared to having one transaction for each event, since in this case, the overhead incurred by acquiring locks was low to begin with.

7. ANALYZING REAL-WORLD SYSTEMS

In this section, we present the result of our analysis of a representative set of state-of-the-art SPSs. As previously noted, for most of these systems, no explicit transactional properties have been defined; therefore, the content we show is based on our interpretation of their behavior, after having experimented with these systems, having read the provided documentation, and discussions with the authors. A summary is presented in Table 3.

When dealing with multiple inputs, the streaming systems offer the application developer a set of primitives which she can use to define *isolation units* as well as ordering among them. These primitives differ among systems and are typically embedded in the processing model or operator semantics, as shown in the *Transactions* and *Synchronization Primitives* columns of Table 3.

Most of SPSs today allow access to non-streaming data sources. For example, when dealing with remotely stored inputs like tables in RDBMSs, the application developer has two options: one option is to use an *adapter* to transform the table into a locally recognized type of data source (e.g., stream), and in combination with other ordering primitives, to implement the desired isolation properties. A second option is to retrieve data from the stored input for each new event in a stream, restricting the isolation unit to a single event.

The reader can observe that what defines a transaction differs from one engine to another. Moreover, the synchronization primitives are embedded in a given engine's processing model (i.e., its execution model [11] plus any other special execution rules that the engine defines) as well as in its operator semantics. Because of the tight dependence of these primitives to the processing models of the engines, very often a change in the isolation properties requires the modification of an entire query plan, or it may even be impossible to implement. The studied systems also offer mechanisms for recovery from failures, though it is not always clear how these mechanisms relate to the defined isolation units.

By reusing the traditional transactional model, our model defines a clean semantics for concurrent access and recovery from failures not bound to any specific model. Being general, it supports all the histories generated by the analyzed streaming engines, thereby allowing a superset.

Next, we present a details of each engine's methods to define isolation units, ordering, and recovery from failures, using our UTM:

Coral8 [1] defines explicit transactional properties for continuous query processing. That is, it specifies that the minimum recovery unit in case of a failure is the *timeslice*. Although timeslices may be processed concurrently, it appears that each timeslice is executed separately and the order between them is preserved. A timeslice represents a row or a sequence of rows having the same timestamp and arriving in the same stream or window. Moreover, the timeslice is different from the engine's execution model unit, which in Coral8's case is a *batch* [11].

If all events with the same timeslice are made part of the same transaction, a history generated by Coral8 is conflict-equivalent to a serial history in which transactions are increasingly ordered by their timestamp values. Moreover, as defined by Coral8's processing model, a transaction on a window having the same timestamp as another transaction on a stream is executed before that transaction.

If some events arrive out of order (e.g., events with timestamp *t-1* arriving after an event with timestamp *t*), they are discarded. This can be expressed as aborted transactions.

When joining a stream and a table located in an RDBMS, Coral8 offers two options: it either restricts the isolation unit to a single event (this way, the situation becomes simple as serializability is guaranteed), or the user could write an adapter which transforms the content of the database and its future updates into a stream. In the former case, the stream (i.e., no window) is the only accepted data source which can be joined with the remote input.

For dealing with failures, Coral8 offers mechanism like *State Persistence* and *Guaranteed Delivery*. State Persistence basically offers the possibility to checkpoint the processing state periodically. The minimum unit is the timeslice. Nevertheless, it does not guarantee that events are not lost in case of failures. Guaranteed Delivery makes sure that an event is received by its destination *at least once*. In our UTM terms, Guaranteed Delivery may violate the Durability property by possibly duplicating events.

STREAM [6] accepts streams and time-varying relations as inputs, and treats them uniformly. A time-varying relation is a relation in the traditional sense, but, in addition, it contains a notion of time. That is, a time-varying relation R represents a mapping from a time domain T to a finite but unbounded bag of tuples belonging to the schema of R. Furthermore, STREAM's processing model is time-driven: time advances to *t* from *t-1* when all the events with timestamp *t-1* have been processed. This behavior translates to a transactional model in which all events with the same timestamp belong to the same transaction, regardless of whether they arrive on the same stream or not. In this context, a history generated by STREAM is conflict-equivalent to a serial history in which transactions are increasingly ordered according to the generating events' timestamps.

StreamBase [3] is a tuple-driven system [11]. In UTM terms, a continuous query is (re)executed whenever a new event arrives in one of the query's streaming inputs.

One guarantee that StreamBase provides is that the order of arrival in a streaming input is the exact order of processing. Nevertheless, there is no predefined inter-stream ordering: if two events are enqueued on two inputs, e.g., on Input1 and then on Input2 in quick succession, it is still possible that the event on Input2 will

SPS	Transactions	Synchronization Primitives	Producible Histories
Coral8	Single event (remote inputs); Timeslice (explicit isolation and recovery unit)	Processing Model; Operator Semantics	Conflict-equivalent to a serial history in which transactions involving events are increasingly ordered by the events' timeslice values (only local inputs)
STREAM	Events sharing the same timestamp	Processing Model	Conflict-equivalent to a serial history in which transactions are increasingly ordered by the events' timestamps
StreamBase	Single event; Group of events	Processing Model; Operator Semantics	Conflict-equivalent to a serial history in which transactions involving events appear in the order of the events arrival in a stream
StreamInsight	Single event (remote inputs); Sequences of events between consecutive CTIs	Processing Model; Operator Semantics	Conflict-equivalent to a serial history in which transactions involving events are ordered by the CTI events' values (only local inputs)
Truviso	Window of events (explicit isolation and recovery unit)	?	?

Table 3: Transactional Properties of Stream Processing Systems

be processed before the one in Input1. Therefore, if an application requires a certain ordering for its input streaming events, the developer should make sure that they are placed on the same stream, or that she uses specialized operators to order them (e.g, the operator which merges two streams).

StreamBase supports both streams and tables as inputs. The tables can be defined locally in the engine and updated through new events arriving in a stream. StreamBase also provides a lock operator so the application developer can do table-level locking. The lock key is an arbitrary expression and it is the responsibility of the programmer to ensure the desired synchronization.

Another type of data source that StreamBase accepts is represented by tables located in relational databases. For example, a stream can be joined with a table residing in a relational database system, in which case for each new event arriving in the stream, a new database lookup is executed.

In terms of the streaming transactional model, in StreamBase, each event defines its own transaction (multiple events can be also logically grouped in transactions by using lock/unlock operators to guard access to tables). In this context, a history generated by a StreamBase engine is conflict-equivalent to a serial history in which transactions generated by events being appended to streams appear in the events' order of arrival. StreamBase also offers high availability features to enable fast recovery from failures.

StreamInsight [2] models streams as changing relations [7]. Each event is composed of a *payload* (i.e., the tuple value) plus application timestamp attributes defining an event's validity interval. StreamInsight reacts whenever it reaches a special event, called CTI (Current Time Increment). Basically, the CTIs allow the engine to *advance* the time by specifying that all events with a timestamp smaller than the CTI have arrived. When reaching a CTI, a query is notified that the results corresponding to that part of the stream (composed of all the events with timestamps smaller than the CTI) will not change anymore and can be safely committed as output.

When a table located in an RDBMS is involved in a join with a stream in StreamInsight, there are two options: either for each event in the stream, a table lookup is executed, or the content of the database table is transformed into a stream by specifying a validity interval for each row. Further updates on the tables will change validity intervals or add new events. By assigning timestamps to rows and updates, StreamInsight can order events and operations.

StreamInsight offers as synchronization primitive the semantics of operators. For example, the join operator matches events from input streams which have overlapping validity intervals.

Given the previous observations, if we consider all the sequences of events between two CTIs as being part of the same transaction, a history generated in the StreamInsight engine is conflict-equivalent to a serial history in which transactions generated by events arriving in a stream between two CTIs (assuming they are assigned the timestamp of the CTI) are ordered by the CTI values.

Truviso [4] is a commercial engine which extends a relational database system. As such, the transactional system of an RDBMS is also extended to continuous processing [15, 22]. In this respect, the authors follow the same approach as ours: streaming and stored data are not intrinsincally different and the traditional transactional model can be reused. One difference to our model is that continuous queries are defined as long-running transactions. Moreover, the authors propose the window as the isolation unit for the interaction of streams and relations [13], which also represents the durability unit for stream archival. We take a step further and formally define the interactions, while allowing the definition of more flexible transactional boundaries. Unfortunately, the information we were able to find about Truviso was not sufficient enough to make any further statements about its transactional behavior.

8. RELATED WORK

Conceptually, the transactional stream processing model we propose in this paper relates to the previous work in two main areas:

Traditional database and data warehousing systems. The similarity between stream processing and materialized view maintenance [18] has long been recognized. Materialized views are like continuous queries since a view has to be updated in response to changes in its base relations. By treating streaming and stored inputs uniformly, our problem becomes similar to materialized view maintenance. We found three lines of work that are closest to ours:

First, Zhuge et al propose a set of algorithms to maintain the consistency of a data warehouse at various levels of correctness, when the view computation is disconnected from the updates at the sources [30]. While we focus more on capturing isolation properties as a definition of correctness, this work focuses on the algorithms to ensure a predefined set of consistency levels.

Second, Chen et al propose a transactional model that defines the view maintenance process as a special transaction consisting of two parts which can commit separately: the update at the source and the view maintenance query [12]. In this case, the warehouse update anomaly problem can be rephrased as the serializability of these transactions. Our work is more general: the update and the continuous query may or may not be part of the same transaction.

Third, Jagadish et al propose the *chronicle data model* which extends materialized views to include chronicles – a form of data streams [21]. A chronicle algebra over relations and streams includes a join operator that synchronizes each tuple from the chronicle with the relation version which existed at the temporal instant of that tuple. The focus of this work is on ensuring that the views defined in the chronicle algebra can be maintained incrementally, without having to store the chronicles.

Stream processing systems. There is only a handful of related work on transactional concepts for stream processing:

The semantic difference between relations and streams as well as unclear role of relations in continuous queries were previously pointed out by Golab and Ozsu [17]. Their work proposes to model relations as look-up time-varying relations and defines the following order for events and updates: any update on the relation at time t will affect only the stream events which arrive after t.

A follow-up study by the same authors focuses specifically on the concurrency control problems that arise in SPSs when window of a long-running query may slide while being accessed by another on-demand snapshot query, resulting in a read-write conflict [16]. This work considers sub-windows to be atomic units of data access, shows that conflict serializability is not strong enough to guarantee correct and up-to-date query results for the considered SPS access patterns, and proposes two stronger isolation levels. These are implemented as part of a transaction scheduler which minimizes the number of aborted transactions. Different from this work, we not only consider streaming data sources (and periodic window queries over them), but relations (and general continuous queries) as well, and accordingly provide a more general transactional model.

Transactional concepts for streams have also arisen in the context of fault tolerance and high availability in distributed stream processing systems [19, 20]. Previous work has defined some properties (e.g., repeatable / deterministic operators) and recovery methods, which can be used to implement recovery protocols in case of failures, but no general transactional stream processing model.

More recently, Wang et al. have proposed a stream-oriented transactional model for providing active rule support to complex event processing systems [28]. In this work, active rules monitor continuous query outputs and issue reads and updates to a table, while other continuous queries are concurrently reading from this shared table. These concurrent read and write operations may in turn lead to anomalies (read-too-late, write-too-late). This work defines a stream transaction as a sequence of system state changes that are triggered by a single input event and redefines the corresponding ACID properties. Correct concurrent executions are achieved through special scheduling algorithms that enforce a timestamp-based notion of correctness, which requires the execution order to follow operations' application timestamps. Our work mainly differs from this work in its goal to define a common and general transactional model for concurrency control and failure recovery over streams and relations by reusing the traditional foundations with minimal extensions. As such, the concurrency scenario that Wang et al. focus on is one of the many that our model can capture.

9. CONCLUSIONS

In this paper, we presented a unified transactional model for streaming applications over an arbitrary mix of streaming as well stored data sources, where serializability can be used as a criterion to define correct executions. As shown experimentally, our model relieves the application developer from the task of dealing with concurrency control and failure recovery, and all this with very modest performance overhead. Our model is also general enough to express the implicit transactional behaviors of real-world SPSs.

10. REFERENCES

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

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

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

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

[5] A. Arasu et al. Linear Road: A Stream Data Management Benchmark. In *VLDB Conference*, 2004.

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

[7] R. S. Barga et al. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR Conference*, 2007.

[8] H. Berenson et al. A Critique of ANSI SQL Isolation Levels. In *SIGMOD Conference*, 1995.

[9] I. Botan et al. Extending XQuery with Window Functions. In *VLDB Conference*, 2007.

[10] I. Botan et al. Flexible and Scalable Storage Management for Data-Intensive Stream Processing. In *EDBT Conference*, 2009.

[11] I. Botan et al. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. In *VLDB Conference*, 2010.

[12] J. Chen et al. A Transactional Model for Data Warehouse Maintenance. In *ER Conference*, 2002.

[13] N. Conway. Transactions and Data Stream Processing. `http://neilconway.org/docs/stream_txn.pdf`, April 2008.

[14] P. M. Fischer et al. Stream Schema: Providing and Exploiting Static Metadata for Data Stream Processing. In *EDBT Conference*, 2010.

[15] M. J. Franklin et al. Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In *CIDR Conference*, 2009.

[16] L. Golab et al. On Concurrency Control in Sliding Window Queries over Data Streams. In *EDBT Conference*, 2006.

[17] L. Golab and M. T. Özsu. Update-Pattern-Aware Modeling and Processing of Continuous Queries. In *SIGMOD Conference*, 2005.

[18] A. Gupta and I. S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*, chapter Maintenance of Materialized Views: Problems, Techniques, and Applications. MIT Press, 1999.

[19] J.-H. Hwang et al. High-Availability Algorithms for Distributed Stream Processing. In *ICDE Conference*, 2005.

[20] J.-H. Hwang et al. A Cooperative, Self-configuring High-Availability Solution for Stream Processing. In *ICDE Conference*, 2007.

[21] H. V. Jagadish et al. View Maintenance Issues for the Chronicle Data Model. In *PODS Conference*, 1995.

[22] S. Krishnamurthy et al. Continuous Analytics over Discontinuous Streams. In *SIGMOD Conference*, 2010.

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

[24] R. Normann and L. T. Ostby. A Theoretical Study of Snapshot Isolation. In *ICDT Conference*, 2010.

[25] E. Ryvkina et al. Revision Processing in a Stream Processing Engine: A High-Level Design. In *ICDE Conference*, 2006.

[26] A. Tansel et al. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Pub. Co., 1993.

[27] D. Terry et al. Continuous Queries over Append-only Databases. In *SIGMOD Conference*, 1992.

[28] D. Wang et al. Active Complex Event Processing over Event Streams. In *VLDB Conference*, 2011.

[29] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.

[30] Y. Zhuge et al. The Strobe Algorithms for Multi-Source Warehouse Consistency. Technical report, Stanford InfoLab, 1996.