Batch Composite Transactions in Stream Processing

K. Vidyasankar(✉)

Department of Computer Science, Memorial University,
St. John's, Newfoundland A1B 3X5, Canada
vidya@mun.ca

Abstract. Stream processing is about processing continuous streams of data by programs in a workflow. Continuous execution is discretized by grouping input stream tuples into batches and using one batch at a time for the execution of programs. As source input batches arrive continuously, several batches may be processed in the workflow simultaneously. Ensuring correctness of these concurrent executions is important. As in databases and several advanced applications, the transaction concept can be applied to regulate concurrent executions and ensure their correctness in stream processing. The first step is defining transactions corresponding to the executions in a meaningful way. A general requirement in stream processing is that each batch be processed completely in the workflow. That is, all the programs triggered by the batch, directly and transitively, in the workflow must be executed successfully. Then, considering each program execution as a transaction, all the transactions involved in processing a batch can be grouped into a single *batch composite transaction*, abbreviated as BCT, and transactional properties applied to these BCTs. This works well when a batch is processed individually and completely in isolation. However, when the batches are split, merged or overlapped along the workflow computation, the resulting BCTs will have some transactions in common and applying transactional properties for them becomes complicated. We overcome the problems by defining *nonblocking* BCTs that have disjoint collections of transactions. They satisfy some properties analogous to those of the database transactions and facilitate (i) defining correctness of concurrent executions in terms of equivalent serial executions of composite transactions and (ii) processing each batch either completely or not at all, and rolling back partially processed batches without affecting the processing of other batches. We also suggest an appropriate roll back mechanism.

1 Introduction

Stream processing is about processing continuous streams of data arriving from external sources by programs in a workflow. Continuous execution is discretized

This research is supported in part by the Natural Sciences and Engineering Research Council of Canada Discovery Grant 3182.

A. Hameurlain et al. (Eds.): TLDKS XXXIV, LNCS 10620, pp. 13–32, 2017.
https://doi.org/10.1007/978-3-662-55947-5_2

by grouping (input) stream tuples into batches and using one batch at a time for the execution of programs. As source input batches arrive continuously, several batches may be processed in the workflow simultaneously. Ensuring correctness of these concurrent executions is important.

As in databases and several advanced applications, the transaction concept can be applied to regulate concurrent executions and ensure their correctness in stream processing. To do this, the first step is defining transactions corresponding to the executions in a meaningful way. A general requirement in stream processing is that each batch be processed completely in the workflow. Each batch will trigger a set of programs in the workflow. Considering each execution of a program as a transaction, all the transactions involved in processing a batch can be grouped into a composite transaction, called *batch composite transaction*, abbreviated as BCT, for that batch. Then, transactional properties can be applied to the BCTs.

In the database context, a transaction is a partially ordered set of operations, but any such set does not qualify as a transaction. Certain properties and conventions are followed in the definition and execution of database transactions. We look at applying these properties to BCTs. In this paper, we focus on the following four properties, denoted *Transaction Properties*, abbreviated as TPs.

TP1. A transaction is a partially ordered set of operations such that *any two conflicting operations are ordered.*
TP2. The operations of each transaction are distinct. That is, no two transactions have any operations in common.
TP3. Each transaction can be executed independently of other transactions.
TP4. Partial execution of a transaction can be rolled back without affecting other transactions.

Denoting the operations of a transaction T as $op(T)$ and partial order as $\prec_t$, TP1 states that conflicting operations in T are ordered by $\prec_t$. Two operations are non-conflicting if their effects are the same in whichever order they are executed; they are conflicting otherwise. For example, a read and a write of a data item are conflicting. TP2 states that $op(T) \cap op(T')$ is empty for any two transactions T and T'. Note that both T and T' may have similar operations like $read(x)$, for the same data item x, but the operations are different, not shared by both the transactions.

We now look at applying these properties to BCTs. They can be stated as follows, replacing 'transaction' by 'BCT' and 'operation' by 'transaction' in the above.

BP1. A BCT is a partially ordered set of transactions such that *any two conflicting transactions are ordered.*
BP2. The transactions of each BCT are distinct. That is, no two BCTs have any transactions in common.
BP3. Each BCT can be executed independently of other BCTs.
BP4. Partial execution of a BCT can be rolled back without affecting other BCTs.

We note that conflicts among the transactions of a BCT are to be determined by the semantics of the operations in the transactions and the data items accessed by them in the application.

In many applications, all processing pertaining to an input batch is done *in isolation*. That is, if a transaction T (which is an execution of a program P) takes as input a batch a and produces as output a batch a', and the output is fed to another transaction T' (an execution of program P'), then a' constitutes the input batch b for T'. In such cases, BCTs defined as consisting of all transactions triggered by the individual batches satisfy the above properties. However, when the batches are merged or overlapped along the workflow computation, the resulting BCTs may not satisfy the above properties. For example, in the case of a merge, when b contains tuples from the outputs of two executions of P, on two source input batches, the BCTs of both batches will contain T' and so will not satisfy BP2-BP4. In this paper, to overcome this problem, we propose a new notion called *nonblocking* BCTs (NBCTs) which satisfy the properties BP1-BP4 and, in addition, the following requirements for processing batches.

B1. *Completion*: Each batch must be processed completely. If it is not possible, then partial processing, if any, must be rolled back *non-intrusively*, that is, without affecting the processing of other batches.

B2. *Monotonic execution*: At any time, for each batch, the amount of processing done on that batch should be a prefix of the workflow.

We describe a procedure for composing NBCTs, that is, figuring out the transactions of each NBCT, in a simple manner. We also describe a non-intrusive roll back mechanism.

With the new notion, the correctness of concurrent executions of the batches can be described in terms of equivalent serial executions of their NBCTs. Rolling back the executions pertaining to a batch can be accomplished by rolling back the NBCTs that process the batch.

The transaction concept was introduced first in the context of (centralized) database systems, characterized by ACID (Atomicity, Consistency, Isolation and Durability) properties, and then adopted in various advanced database and other applications, for example, in transactional processes [12], Web services [17], and electronic contracts [16]. In all these applications, the composite/nested transactions defined corresponding to the executions satisfy the properties BP1-BP4. There have been several studies on the application of the transaction concept in stream processing, including [2,3,6,9,18]. We elaborate the approaches in the Related Works section. Some of them define composite transactions for batches consisting of single tuples or batches executed in isolation. To our knowledge, none of them define composite transactions when the batches are split, merged or overlapped along the workflow computation.

We start with core definitions of compositions and transactions in stream processing environments in Sect. 2. We study the executions involving splits, merges and overlapping of batches and arrive at the definition of the NBCTs in Sect. 3. A recovery mechanism that supports BP4 and B1 for the NBCTs is

given in Sect. 4. We initially consider only one source input stream. Inputs from multiple source streams are considered in Sect. 5. Concurrent execution of BCTs is dealt with in Sect. 6. We discuss related work in Sect. 7 and conclude in Sect. 8.

2 Executions in Stream Processing

A stream processing workflow is a composition of programs. Formally, a *composition* $\mathcal{C}$ is $(\mathcal{P}, \prec_p)$, where $\mathcal{P}$ is a set of *transaction programs* $\{P_1, P_2, \ldots, P_n\}$, simply called *programs*, and $\prec_p$ is a partial order, called *program order*, among them. The partial order consists of dataflow order (of the streams) and control order. We also include *conflict order*. We discuss conflict order in Sect. 6. We call the (acyclic) graph representing the partial order the *composition graph* $\mathcal{GC}(\mathcal{C})$. Stream data are sequences of tuples. Streams coming from outside the composition are called *source streams*. The output streams (of any program) are called *derived streams*. Each execution of a program yields a *transaction*.

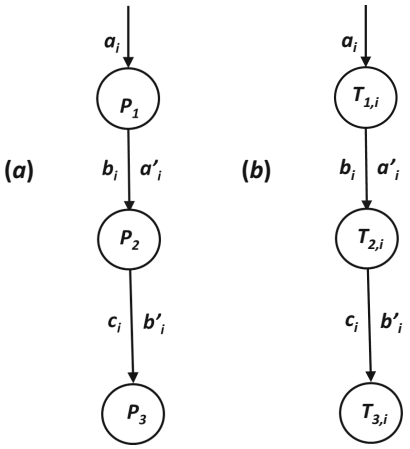

Fig. 1. A schema example

We use the simple composition, shown in Fig. 1, to illustrate the definitions. It is a workflow consisting of a sequence of three programs P_1, P_2 and P_3. Input batches will be denoted by unprimed variables x_i and the corresponding outputs by primed variables x'_i. Stream inputs/outputs for P_1, P_2 and P_3 will be denoted by a, b and c, respectively. The sequence of input batches for P_1 is $a_1, a_2, \ldots$, and the executions are transactions $T_{1,1}, T_{1,2}, \ldots$ (the first index is that of the program and the second index is that of the input batch), producing the output sequence $a'_1, a'_2 \ldots$.

The processing of a source input batch will involve executions of some of the programs in the workflow, resulting in a set of transactions with a partial order $\prec_t$, called *transaction order*. We call this a *batch composite transaction*, BCT,

denoted as $\mathcal{T} = (\{T_1, T_2, \ldots, T_m\}, \prec_t)$. We denote $\{T_1, T_2, \ldots, T_m\}$ as $set(\mathcal{T})$. The graph representing $\prec_t$ is called *transaction graph* $\mathcal{GT}(\mathcal{T})$. The transaction graphs are acyclic. We note that each T_i is an execution of some program P_j. It is possible that $\mathcal{T}$ has more than one execution of some P_j. The transaction partial order $\prec_t$ reflects the program partial order $\prec_p$, that is, if T_i is an execution of P_j, T_k is an execution of P_l and $P_j \prec_p P_l$, then $T_i \prec_t T_k$. In addition, $\prec_t$ will contain triggering relationships, if any. (We note that, in this paper, we use the term 'transaction' exclusively to denote some T_i; a $\mathcal{T}$ always denotes a 'batch composite transaction', that is, BCT.) We denote the BCT that is executed for source input batch b as $\mathcal{T}(b)$. In the execution shown in Fig. 1, the BCT for batch a_i, $\mathcal{T}(a_i)$, is $\{T_{1,i}, T_{2,i}, T_{3,i}\}$ (omitting the transaction order for brevity).

Stream input batches arrive in sequence, for example, as $b_1, b_2, \ldots$. The batch order is denoted $\prec_b$. The batch b_2 and a few more batches may arrive before all the transactions in $\mathcal{T}(b_1)$ are completely executed. Thus many BCTs may be executed concurrently.

General requirements for concurrent executions of BCTs can be stated as follows [14].

1. *Unit of atomicity*: Each BCT is executed either completely or not at all. That is, the entire $\mathcal{T}$ is an *atomic unit* for each $\mathcal{T}$.
2. *Serializability*: The execution is equivalent to a serial execution of the BCTs.
3. *Transaction order*: The effective execution order of the transactions of $\mathcal{T}$ should obey the partial order $\prec_t$. That is, for any i, j, if $T_i \prec_t T_j$, then T_i should precede T_j in the serial execution.
4. *Batch order*: The serial execution should reflect the batch order $\prec_b$. That is, for $i < j$, (all the transactions in) $\mathcal{T}(b_i)$ should precede (the transactions of) $\mathcal{T}(b_j)$ in the equivalent serial execution.

We define nonblocking BCTs in the next section.

3 Batch Composite Transaction Model

Batch composite transactions are initiated by arrival of batches of tuples from source streams. Batch sizes vary. A batch may contain all the tuples with the same timestamp (time-based) or a certain number of tuples (count-based). A program may process one tuple at a time (as in selection and projection operations) or all the tuples in the batch together (as in join). We stipulate only that *each* execution of a program is a transaction. Therefore, with the all-or-nothing atomicity property, the result of the execution will be known only after the entire batch is processed. The intermediate results and states of the program will not be available. In general, smaller batches will reduce latency while larger ones, resulting in fewer executions of the program, may improve efficiency. Batch sizes may also be different for different programs, and even for different executions of the same program.

Batches may be split, merged or overlapped along the workflow computation [4,7]. For example, splitting may occur for processing the batches in parallel.

Subsequently, the resulting output batches may be merged. Merging and overlapping will also occur in aggregates computation. We assume arbitrary splitting, merging and overlapping in this paper. We consider several examples and come up with a definition of nonblocking BCTs and an execution model underlying the definition.

We first consider only one source input stream. (Note that in many applications where multiple input streams are involved, the tuples from the different streams are combined and input as one stream.) We consider multiple source streams in Sect. 5.

Details of the model are itemized with label M. Though we are dealing with concurrent processing of the batches and hence concurrent executions of their BCTs, we assume in this paper that:

M1. Each program in the workflow is executed serially.

It follows that each transaction in a BCT is executed atomically, akin to each operation in a database transaction being executed atomically.

We have identified the composite transaction *to be executed* for batch b as the BCT $\mathcal{T}(b)$. Suppose b is input to transaction T. Then we define $\mathcal{T}(b)$ as the union of $\{T\}$ and all the transactions triggered directly or indirectly by T in the composition, with the corresponding partial order. Suppose T_i precedes T_j. If the precedence is due to dataflow order, the execution of T_j will start only after the execution of T_i finishes. The same can be assumed for control order. We also assume an implementation such that if T_i triggers T_j, the triggering is done only after T_i commits. Then, in all cases, for T_i preceding T_j, the execution of T_j starts only after the execution of T_i finishes. This is true whether T_i and T_j are conflicting or not. If T_i and T_j are executions of the same program, then the assumed serial execution of programs (M1) induces an ordering between the two transactions. Thus, BP1 will be satisfied for $\mathcal{T}(b)$. In the following, we look at the properties BP2-BP4 for various cases. We use the composition shown in Fig. 1 to illustrate the cases.

M2. We model the dataflow, from an output stream of one program P_i to an input stream of another program P_j, with a FIFO (first-in-first-out) queue $Q_{i,j}$; P_i enqueues its output into $Q_{i,j}$ and P_j dequeues its input from that queue. Both enqueueing and dequeueing a batch are assumed to be done atomically.

In the execution shown in Fig. 1, the dataflow between P_1 and P_2 is such that $b_i = a'_i$, that is, P_2 empties the queue $Q_{1,2}$ (in a serial execution of the batches), and similarly P_3 empties the queue $Q_{2,3}$ resulting in $c_i = b'_i$. Here, the BCT for batch a_i, $\mathcal{T}(a_i)$, is $\{T_{1,i}, T_{2,i}, T_{3,i}\}$. Rolling back partially executed $\mathcal{T}(a_i)$ involves rolling back the corresponding transactions in this set. Clearly, the BCT $\mathcal{T}(a_i)$, for each i, satisfies all the properties BP1-BP4.

We note that, in a serial execution of the batches, all the queues are empty before the processing of a_i starts, and all of them are empty after the processing

is completed. This property captures the notion of *the batch being processed in isolation*.

In the following, we consider splits, merges, and overlapping of batches.

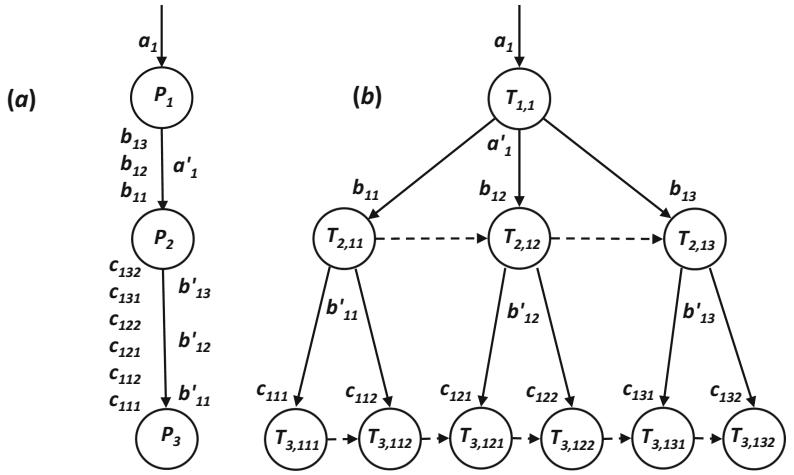

Fig. 2. Splitting of the batches

(a) *Splits*: Consider the following with respect to our composition example, depicted in Fig. 2. (In all the figures, horizontal edges denote batch order.)

- Input batch a_1 for P_1 results in execution of $T_{1,1}$, producing output batch a'_1.
- The batch a'_1 is split into three batches b_{11}, b_{12}, b_{13}, and each b'_{1j} is split into two batches c_{1j1} and c_{1j2}.
- Then the corresponding executions of P_2 are $T_{2,11}, T_{2,12}, T_{2,13}$. The batch order among the three batches translates to $T_{2,11} \prec_b T_{2,12} \prec_b T_{2,13}$.
- The executions of P_3 are $T_{3,111}, T_{3,112}, T_{3,121}, T_{3,122}, T_{3,131}, T_{3,132}$.

Here, $\mathcal{T}(a_1)$ consists of all the transactions listed above. Again, to satisfy BP1, all conflicting transactions must be ordered. Imposing batch order on the split batches will guarantee this property. (We assume that any two executions of the same program are conflicting.) The other three properties, BP2-BP4, are clearly satisfied. We note that, here also, (again in a serial execution of the batches) all the queues are empty before the processing of a_1 starts and are empty after the processing is complete. That is, the batch a_1 is processed in isolation.

(b) *Merges*: Merging of the batches is depicted in Fig. 3:

- Input batches $a_1, a_2, \ldots, a_6$, for P_1, result in executions of $T_{1,1}, T_{1,2}, \ldots, T_{1,6}$, producing output batches $a'_1, a'_2, \ldots, a'_6$, respectively.

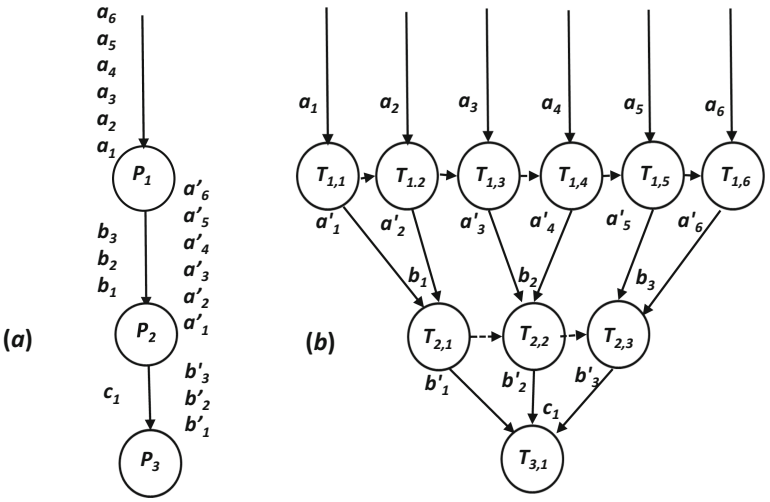

Fig. 3. Merging of batches

- Batch b_1 is $a'_2 \cdot a'_1$, b_2 is $a'_4 \cdot a'_3$, and b_3 is $a'_6 \cdot a'_5$ and the executions of P_2 are $T_{2,1}, T_{2,2}, T_{2,3}$. Here "." indicates concatenation, of the batches in the order of their arrival, that is, from right to left.
- Batch c_1 is $b'_3 \cdot b'_2 \cdot b'_1$, and the execution of P_3 yield $T_{3,1}$.

Here, $\mathcal{T}(a_1)$ is $\{T_{1,1}, T_{2,1}, T_{3,1}\}$ and $\mathcal{T}(a_2)$ is $\{T_{1,2}, T_{2,1}, T_{3,1}\}$. We note that these two sets have $\{T_{2,1}, T_{3,1}\}$ in common. Thus BP2 is not satisfied.

To satisfy BP2, keeping in mind the completion and monotonic execution requirements, B1 and B2, of the batches, we choose an appropriate prefix[1] of $\mathcal{T}(b)$ as a composite transaction for batch b. We can interpret that in the executions of a program where merges occur, on arrival of an earlier batch, the program *waits* (*blocks*) for further batches. For instance, in Fig. 3, after a'_1 arrives, P_2 waits for a'_2 for execution of $T_{1,2}$. We take this as the execution of the BCT for a_1 is complete when it cannot proceed any farther by itself, that is, once a'_1 is sent to P_2. Therefore, eliminating waiting for batches from other transactions, we can close the composition of the BCT with $\{T_{1,1}\}$. We call this *nonblocking* BCT, abbreviated as NBCT and denoted $\widetilde{\mathcal{T}}(a_1)$. Arguing along the same lines, the nonblocking batch composite transactions for other batches will be as follows.

- $\widetilde{\mathcal{T}}(a_2)$ is $\{T_{1,2}, T_{2,1}\}$.
- $\widetilde{\mathcal{T}}(a_3)$ is $\{T_{1,3}\}$.
- $\widetilde{\mathcal{T}}(a_4)$ is $\{T_{1,4}, T_{2,2}\}$.
- $\widetilde{\mathcal{T}}(a_5)$ is $\{T_{1,5}\}$.
- $\widetilde{\mathcal{T}}(a_6)$ is $\{T_{1,6}, T_{2,3}, T_{3,1}\}$.

[1] A subgraph H of an acyclic graph G is a prefix of G if all the edges from H to the rest of the graph are outdirected.

This approach is captured in the following definition.

Definition 1. For a batch a, the *nonblocking batch composite transaction* $\widetilde{\mathcal{T}}(a)$ is the maximal prefix of $\mathcal{T}(a)$ that is executed without inputs from subsequent batch composite transactions.

We note that with this definition, referring to the above example, the properties BP1-BP3 will be satisfied. Now, BP4 requires that roll back of any partial execution can be done without affecting other NBCTs. We describe a non-intrusive roll back mechanism that will accomplish this in the next section. As an example, in Fig. 3, suppose $T_{1,1}$, $T_{1,2}$ and $T_{1,3}$ have been executed by P_1, and P_2 has not yet executed $T_{2,1}$. Now, to roll back $\widetilde{\mathcal{T}}(a_2)$, the state of P_1 is rolled back to the one before $T_{1,2}$ and a_3 is processed again. That is, we roll back the processing of subsequent batches also and then reprocess them. We note that the completion requirement for each batch is fulfilled jointly by the NBCTs of all batches.

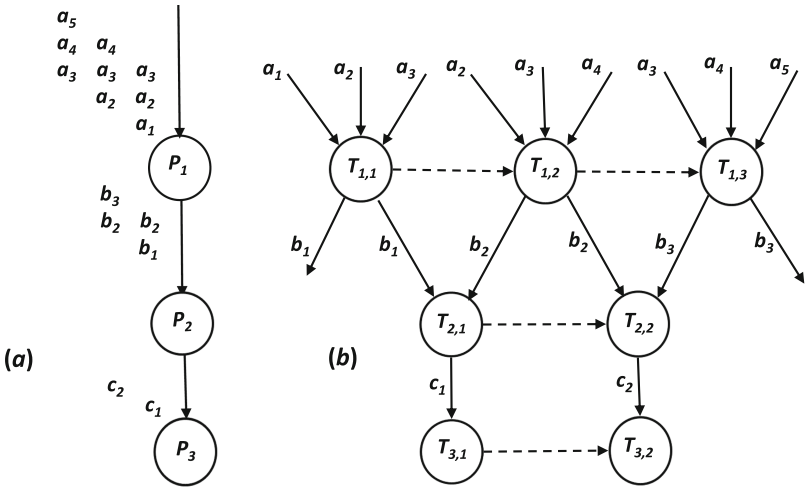

Fig. 4. Overlapping batches

(c) *Overlapping batches*: In the previous examples, the batches input to the executions of a program are disjoint. In some applications, the batches may overlap. For example, in the problem of computing an aggregate function every 5 min where the batch consists of the tuples received in the preceding 10 min, every two consecutive batches will overlap. Figure 4 depicts overlapping batches in our composition example. The transactions and batches used for them are:

– Input batches of $T_{1,1}, T_{1,2}$ and $T_{1,3}$ are $a_3 \cdot a_2 \cdot a_1$, $a_4 \cdot a_3 \cdot a_2$, and $a_5 \cdot a_4 \cdot a_3$; the respective output batches are b_1, b_2 and b_3.

- Input batches of $T_{2,1}$ and $T_{2,2}$ are $b_2 \cdot b_1$, and $b_3 \cdot b_2$; the respective output batches are c_1 and c_2;
- Input batches of $T_{3,1}$ and $T_{3,2}$ are c_1 and c_2, respectively.

Here, we can interpret as (i) an input batch is made up of several smaller batches and (ii) each such batch is input multiple times in the executions of a program. We can then consider BCTs of the smaller batches.

We extend our execution model to accommodate overlapping input batches as follows.

M3. With each program, for each stream input, we associate a *local* FIFO window. At the beginning of an execution, the input batches used for the computation are dequeued from the queues and placed (enqueued, let us say on top) in the respective windows. In the next execution of the program, either all or a part of the content (always from the bottom) is used. At the end of that execution, a part of the content, again from the bottom, is dequeued and discarded if and only if it is not used in any further executions. Then, in an overlapping window, some contents are removed and some are added, dequeued from the queue. It is also possible that an entire window content is used for the next computation, without adding a batch from the queue.

M4. The main point is that any batch (tuple) is dequeued from the queue (just once), but kept in the window for using in one or more executions of the program.

Considering the example in Fig. 4, P_1, to execute $T_{1,2}$, will have a_2 and a_3, concatenated as $a_3 \cdot a_2$ in its window, dequeue a_4 from its input queue to form the input batch $a_4 \cdot a_3 \cdot a_2$. The program P_2 would have b_1 in its window, dequeue b_2 after the execution of $T_{1,2}$, and merge them to get $b_2 \cdot b_1$ for the execution of $T_{2,1}$, and then clear b_1 but keep b_2 for the next execution $T_{2,2}$.

Here, by Definition 1, $\widetilde{\mathcal{T}}(a_4)$ will have $\{T_{1,2}, T_{2,1}, T_{3,1}\}$ and $\widetilde{\mathcal{T}}(a_5)$ will have $\{T_{1,3}, T_{2,2}, T_{3,2}\}$. Clearly, BP1-BP3 are satisfied. Let us consider BP4. Suppose after the execution of $T_{1,2}$ and $T_{2,1}$, but before the execution of $T_{3,1}$, it is decided that $\widetilde{\mathcal{T}}(a_4)$ needs to be rolled back. Assume that $T_{1,3}$ and $T_{2,2}$ have been executed with a_5, and also, $T_{1,4}$ has been executed with a_6 (not shown in the figure). With the roll back of $\widetilde{\mathcal{T}}(a_4)$, P_1 has to be reset to its state before $T_{1,2}$, and P_2 to its state before $T_{2,1}$.

M5. We define the states of the programs to include the states (that is, contents) of their windows also.

Thus, when P_1 is reset, its window will contain $a_3 \cdot a_2$ and when P_2 is reset, its window will have b_1. Then, we want P_1 to process a_5 again, and then a_6 again. With the resulting outputs, P_2 should execute (new) $T_{2,1}$, and then (new) $T_{2,2}$. This way, we achieve non-intrusive roll back. Here also, the completion requirement of each batch is satisfied jointly by several NBCTs.

4 A Roll Back Mechanism

In general, we assume that splits, merges and overlapping of batches could occur arbitrarily in an execution. The source input batches that transactions process can be kept track of as follows.

M6. 1. Index the source input batches serially.
2. For a batch b, we denote the source input batch set from which b is derived as *sb-set*, denoted $\psi(b)$. If b itself is a source input batch, then $\psi(b)$ contains just b.
3. We define sb-sets for transactions also. For a transaction T, let $\psi(T)$ denote the source input batch set that T processes. It will be the union of $\psi(b)$ of all batches b input to T.
4. Each of the output batches of T will have sb-set equal to that of $\psi(T)$.

We observe that, for a batch b_i, $set(\mathcal{T}(b_i))$ is the set of all the transactions T whose sb-set contains b_i. An alternate definition of $\widetilde{\mathcal{T}}(b_i)$ is the following.

Definition 2. For a batch b_i, $\widetilde{\mathcal{T}}(b_i)$ is the subgraph of $\mathcal{T}(b_i)$ with all the transactions T such that i is the largest index of the batches in $\psi(T)$.

We note that if b_i is in $\psi(T)$ of a transaction T, then b_i is in $\psi(T')$ of all descendents T' of T also. Therefore, it follows that $\widetilde{\mathcal{T}}(b_i)$ is a prefix of $\mathcal{T}(b_i)$.

The roll back mechanism is as follows.

Preliminaries:

M7. 1. (a) We denote the completion of a BCT by *committing* it once all its transactions are successfully executed.
(b) A source input batch b can be *committed* when all the BCTs processing b have been committed, that is, all the transactions in $\mathcal{T}(b)$ (*not* $\widetilde{\mathcal{T}}(b)$) have been executed successfully.
(c) The batches are committed serially, in the order of their indices.
2. For each transaction T of program P, we denote the state of P before the execution of T as $prev(P, T)$. Rolling back T amounts to resetting P to this state. Resetting will also roll back changes, if any, made by T to objects in persistent storage.
3. We require each program P to remember $prev(P, T)$ for Ts corresponding to all the BCTs that are currently executed and not yet committed in the workflow. These are for transactions T whose $\psi(T)$ contains an uncommitted batch; if all the batches in $\psi(T)$ are committed, then the $prev(P, T)$ can be discarded. Since we consider serial execution of the programs, a sequence of previous states can be kept corresponding to the executions.
4. In addition, we require P_1, the first program that processes source input batches, remembering (temporarily) all the uncommitted batches themselves. A batch is kept until it commits.

Mechanism:

1. Rolling back an NBCT $\widetilde{\mathcal{T}}(b_i)$, of batch b_i is done by resetting the programs of each of the transactions in that NBCT to the previous states corresponding to the *first* execution of that batch. This amounts to rolling back all the subsequent transactions of that program too, that is, a cascade roll back.
2. A subsequent batch might have been processed by a program that does not process b_i. Therefore, all the programs that have executed a transaction whose sb-set contains a batch with index greater than or equal to i are rolled back.
3. The program P_1, after rolling back its state to the one prior to the first execution of the batch b_i, resumes executions of the successor batches, one by one. All other programs simply roll back their states to the ones prior to the first execution of that batch. Then they wait for normal execution.
4. Some derived batches of the ones that are rolled back may arrive to the other programs in the mean time. They should be ignored. To facilitate this, the reprocessed batches could be given new indices (that are greater than any previous index).
5. Each transaction could have modified different variables and thus state of the program differently. All these changes have to be rolled back, in reverse order.

A Stream Processing Engine (SPE) can regulate the executions.

M8.
1. The SPE will index the source input batches.
2. It will keep track of the transactions executed by programs, and their sb-sets.
3. Therefore, the SPE can figure out the composition of the NBCTs, that is the set of transactions in each NBCT. Note that this set can be completed only after the sb-sets of all transactions have been received.
4. It will determine commitment of the batches.
5. When a decision to roll back an NBCT is made, the SPE will figure out the programs that need to be rolled back and inform them. The actual roll back will be done by the programs themselves. Likewise, the previous states will also be maintained by the programs themselves.
6. The SPE, instead of P_1, could keep the uncommitted source batch sets.

We note that since $\widetilde{\mathcal{T}}(b_i)$ is a prefix of $\mathcal{T}(b_i)$, rolling back $\widetilde{\mathcal{T}}(b_i)$ amounts to rolling back $\mathcal{T}(b_i)$ itself, and thus rolling back the batch b_i, and it is done non-intrusively.

Stream tuples are usually not stored persistently. They are used in the computation and then discarded. Typically, as we have assumed, the tuples arriving from a source or derived by some transaction are written into a queue and read by the next program (transaction) in the workflow. Once the tuples are used, they are not available anymore. However, many recovery considerations require that the tuples are *available for a while* [8]. The duration of their availablity may vary from (i) only until they are used by the successor program(s), (ii) until a certain amount of downstream computations have been carried out, (iii) until

the corresponding BCT commits, or (iv) until some time later or when a certain number of subsequent batches have been processed. Some of the (source or derived) batches may even be stored persistently as part of a checkpoint or for archival purposes. The recovery mechanism described above assumes the availability of source input batches until they are committed, and the availability of the previous states of transactions executed by programs. The previous states can be stored in terms of before-images of the changes each transaction makes. Then, resetting to a previous state of a transaction would amount to installing before-images of all the transactions up to that transaction in reverse order.

In some applications, source input streams, also called *raw* streams, should only be processed (by edge devices) and not stored anywhere, for example, for privacy reasons. In such cases, the derived batches at some downstream level can be stored for reprocessing. For example, in our composition schema, instead of remembering a_i's at P_1, b_i's can be remembered at P_2. Then, when a_j, for some j, needs to be rolled back, b_j can be rolled back instead (and subsequent b_i's reprocessed from P_2). This would amount to dropping a_j, literally after P_1 but semantically at P_1 itself.

We discuss another semantic adjustment in the following. In many applications, a source input batch is processed for several functionalities. We may find that a batch should be rolled back with respect to some functionality, but used for others. This will redefine the NBCT of that batch. We illustrate with an example.

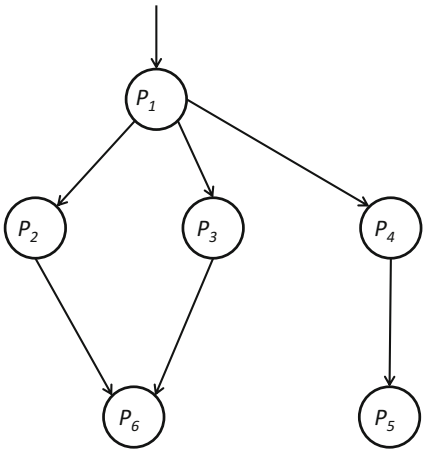

Fig. 5. A composition example

A simple composition is shown in Fig. 5, and its execution on input batch a_1, that is, the NBCT $\widetilde{\mathcal{T}}(a_1)$, is shown in Fig. 6. In this example, $T_{1,1}$, the execution of P_1 on input a_1, produces two stream outputs b_1 and c_1. The batch c_1 is input to P_3. It is also input to P_4 split into two batches c_{11} and c_{12}. The outputs

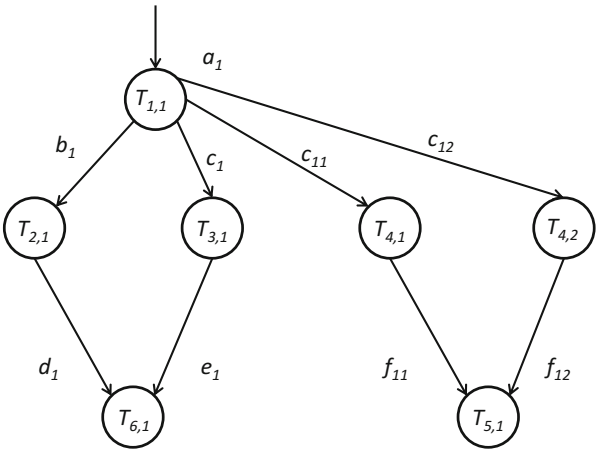

Fig. 6. A batch composite transaction

from the two executions of P_4, namely, $T_{4,1}$ and $T_{4,2}$, are merged and fed to one execution $T_{5,1}$ of P_5. The programs P_2 and P_3 process b_1 and c_1 respectively, and produce d_1 and e_1 which are processed together by P_6. Suppose that after $T_{4,1}$ and $T_{4,2}$ are executed but before $T_{5,1}$ is executed, it is decided to drop c_{11}. Then, P_4 has to be reset to $prev(P_4, T_{4,1})$ and it has to reprocess c_{12}. Then P_5 could execute just with (new) f_{12}. It is possible that c_{11}, and hence c_1, is processed for different functionality by P_3 and $T_{3,1}$ is still valid. Thus, wherever reprocessing is allowed, the batches that are processed must be stored until their source input batches are committed.

5 Multiple Source Streams

So far, we have considered only one source input stream. We now consider multiple streams. We start with an example with two streams to illustrate the problem. Consider the composition and one of its executions shown in Fig. 7. Here, program P_1 processes batches a_1 and a_2, in $T_{1,1}$ and $T_{1,2}$, P_2 processes batch b_1 from a different source, and their outputs are processed by P_3 as shown.

In some applications that have inputs from multiple source streams, it may be appropriate to define NBCTs for combinations of batches of different source inputs. For the example shown, $\widetilde{\mathcal{T}}(a_1, b_1)$ and $\widetilde{\mathcal{T}}(a_2, b_1)$ would be appropriate. However, when the batch a_2 arrives at the source input level, we may not know whether it will be processed downstream with b_1 or some other batch b. This problem arises even when batches from both sources are input to the same (first) program. Irrespective of how the NBCTs are identified, we would like to compose them as per Definitions 1 and 2. We resolve the issue as follows.

M9. We introduce a hypothetical program P_0 and let batches from all source streams be input to this program. This will be a *filter* program sorting out batches to be fed to the original source programs.

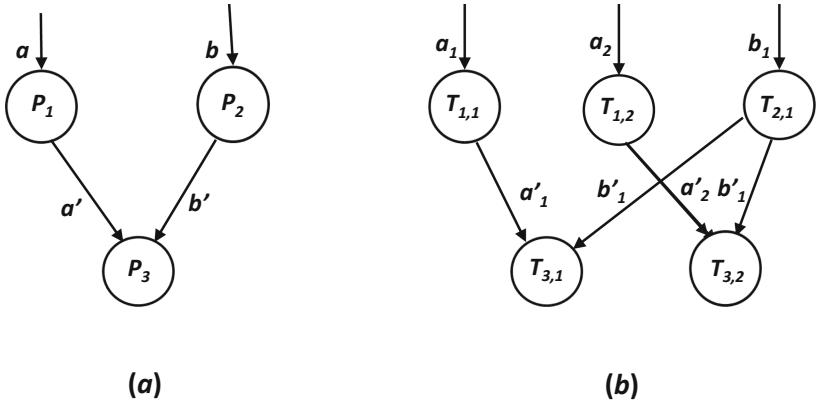

Fig. 7. Inputs from multiple source streams

The construction for the composition in Fig. 7 is shown in Fig. 8 with the hypothetical program in grey box. We identify the executions of P_0 first with a_1 and b_1, and then with a_2. In Fig. 8, we extend the execution in Fig. 7 with input b_2 next and then with a_3 and b_3. Now we can identify the NBCTs with the sets of *new* batches used in the executions of P_0; as per our notion, they will contain all the transactions triggered directly or transitively, but executed *without waiting for* subsequent batches. The NBCTs and their transactions are shown in the figure in part (i), omitting the transactions (hypothetically) executed by P_0. (The first and the third NBCTs are in one line, and the second and the fourth in the next line.) We apply this idea for any number of source streams. We stipulate only that each execution of P_0 will have a new batch that is dequeued from the appropriate queue, from at least one source. It may have new batches from any number of sources. Also, once dequeued into its window, a batch could be used for any number of executions.

Again, for each a_i and b_j, $\widetilde{\mathcal{T}}(a_i, b_j)$ would contain all transactions T such that its sb-set contains a_i but not a_k, for $k > i$, and similarly for b_j. Note that in $\widetilde{\mathcal{T}}(a_i, b_j)$, P_0 is executed with a_i and b_j, and hence all the transactions in the NBCT will have both a_i and b_j. Hence, $\widetilde{\mathcal{T}}(a_i, b_j)$ could as well be identified as $\widetilde{\mathcal{T}}(a_i)$, and similarly as $\widetilde{\mathcal{T}}(b_j)$. The execution of the hypothetical program P_0 can be managed by the SPE.

We note that each of the source input batches can be rolled back; the NBCT of that batch as per Definition 2 will be rolled back, with the result that the batch is not used for any NBCT at all. The roll back mechanism described in the last section is applicable here also. When the NBCT has several (new) batches in the execution of P_0, any number of those batches can be rolled back, and subsequent batches of the respective streams are reprocessed. We note that the reprocessing may produce different NBCTs, compared to the execution without roll back. For instance, in the example of Fig. 8, suppose a_1 is rolled back. Then we might end

up with $\widetilde{\mathcal{T}}(a_2, b_1)$ with the transactions $T_{1,2}$, $T_{2,1}$ and $T_{3,2}$, which would not be present if a_1 is not rolled back.

We have illustrated the application of the idea of *composing a batch composite transaction with all the transactions triggered by arrival of new batches and executed independently without waiting for subsequent batches* to different executions where batches are not processed in isolation. Part (ii) in Fig. 8 displays another interesting case. Here, each time, new pairs of batches from the two source streams are processed. However, P_3 accumulates the previous batches and uses them with the new batches for the next execution. This type of processing is described in [1]. The NBCTs are shown in the figure.

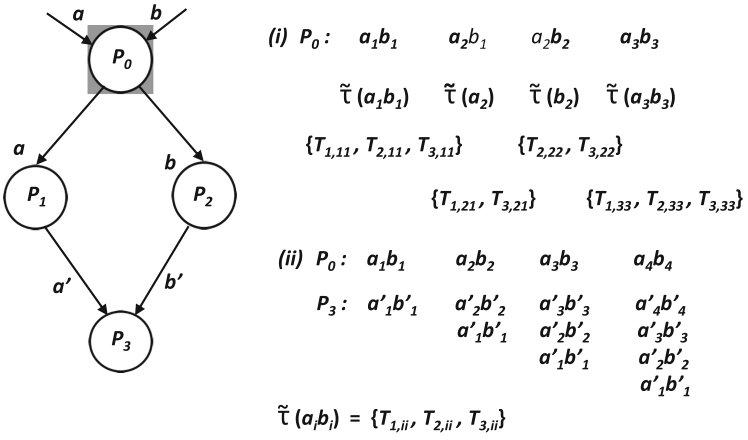

Fig. 8. Nonblocking batch composite transactions for multiple source streams

6 Concurrent Executions

In this section, we consider concurrent executions of NBCTs. Let $\mathbf{T}$ be a set of NBCTs. We define an *execution graph* $\mathcal{GE}(\mathbf{T})$ as the graph whose vertex set is the union of $set(\widetilde{\mathcal{T}})$'s of all $\widetilde{\mathcal{T}}$ in $\mathbf{T}$, and edges for the following:

- the transaction partial order $\prec_t$ of each $\widetilde{\mathcal{T}}$ in $\mathbf{T}$;
- the serial order among the transactions of the same program, for each program in the workflow; and
- the conflict order among the transactions, as described below.

We associate conflicts between programs in a composition. Conflicts are to be determined based on the semantics of the operations executed by the programs and the data items that are operated on. In general, the execution order is important for conflicting operations and irrelevant for non-conflicting ones. We assume that the conflicts between programs carry over to their executions. For

example, suppose programs P_i and P_j, $i < j$, are conflicting. Then, we assume that every execution of P_i conflicts with every execution of P_j. These executions may be for the same source input batch or different source input batches. We also assume that any two conflicting programs in a composition are related by the program order $\prec_p$. Then, since the program order will yield transaction partial order, conflicts between executions of the programs for the same input batch, that is, transactions of the same NBCT, are taken care of by the transaction order edges. For example, for executions $T_{i,m}$ and $T_{j,m}$ for the same source input batch a_m, a conflict edge from $T_{i,m}$ to $T_{j,m}$ need not be added since $T_{i,m} \prec_t T_{j,m}$ and hence the corresponding transaction order edge will be added. However, for different source input batches a_m and a_n, $m < n$, the edges from $T_{i,m}$ to $T_{i,n}$, as well as from $T_{j,m}$ to $T_{i,n}$ need to be added. The former edges are already included in the serial order of the transactions of the same program; only the latter edges need to be added.

We note that the graph obtained as above is acyclic. There are no edges directed from a transaction of an NBCT of a batch b_j to that of batch b_i, for $j > i$. (This is certainly true for conflict edges and those between the transactions of the same program. Consider a transaction order edge of $\widetilde{\mathcal{T}}(b_j)$, say from an execution T of P_k to an execution T' of P_l, for $k < l$. Now, T' could be in $\mathcal{T}(b_i)$, but it must be waiting for a batch derived from b_j. Then, by definition, T' is not in $\widetilde{\mathcal{T}}(b_i)$. Therefore, by contracting the subgraphs generated by the set of vertices of NBCTs $\widetilde{\mathcal{T}}(b)$ into single vertices, we will indeed get a graph consisting of a single directed path whose vertices are the NBCTs and edges correspond to the batch order, that is, a serial execution of the NBCTs in $\mathbf{T}$, according to the batch order.

Again, note that the vertices of the execution graph $\mathcal{GE}(\mathbf{T})$ are the transactions, not the NBCTs, in $\mathbf{T}$. Therefore the graph can be constructed without waiting to know the composition of the NBCTs.

We note that, in the above discussions, we have insisted on the serial order of the NBCTs to be the same as the batch order among the batches. With multiple source input streams, the batch order will be the lexicographic order of the set (of indices) of the batches as they arrive for processing.

7 Related Work

Composite transactions have been defined in different ways in different heterogeneous distributed environments depending on the required/relaxed ACID properties appropriate to the applications. In most cases, they are defined as *saga*s and then the properties of the transactions constituting a *saga* are explored. With respect to atomicity (all-or-nothing property of composite transactions), an early proposal [10] was the schema $c^*[p]r^*$ denoting a partially ordered set of *compensatable* transactions followed by at most one *pivot* (which is non-compensatable) and then followed by a partially ordered set of *retriable* (*assured*) transactions. This schema has been extended, allowing multiple pivots, for Transactional processes [12], and then for Web services [17], Electronic contracts

[16] and recently for Internet of Things services [13]. In the latter applications, nested transactions were considered. The properties BP1-BP4, as applicable to sagas, were satisfied by the (high level) transactions. Our NBCTs are also nested transactions but treated as non-nested ones, capturing the top level of the nesting only.

The works in the context of stream processing include the following. A unified transaction model, called UTM, is proposed in [2]. It treats events also as transactions. It discusses splitting continuous executions into transactions. Isolation and atomicity properties are relaxed. Events and triggers in the context of Complex Event Processing over Event Streams are discussed in [18]. They also define *stream* ACID properties for transactions. The stream atomicity notion requires "all operations stimulated by a single input event should occur in their entirety". In S-Store [9], the unit of atomicity is the entire composite transaction. The batches are executed in isolation. In [6,11], entire read-only composite transactions reflecting "continuous queries reading updatable resources" are taken as units of atomicity. Such considerations are very useful especially in IoT environments, where monitoring and actuations are predominant and monitoring should be consistent. Reprocessing upstream batches is also considered in [11]. Other papers discussing stream transactions and compositions include [3,5]. None of these papers deal with executions arising with splitting, merging or overlapping of the batches and defining composite transactions satisfying properties considered in this paper. Splitting batches for parallel execution and merging them later have been considered in the literature, for example in [4,7].

This paper is closely related to [15]. There, the source input batches b for which $\mathcal{T}(b)$ satisfies BP1-BP4 and B1-B2 are called *atomic* batches. (The properties BP1-BP4 are not brought out explicitly in that paper.) As observed in this paper, when splits, merges and overlapping of the batches occur in an execution, some source input batches may not be atomic. The contribution in [15] is showing that several source input batches can be grouped into a single atomic batch. In contrast, the goal in this paper is to define an NBCT for each source input batch individually to satisfy the properties BP1-BP4, at the expense of satisfying the completion requirement of batches jointly by several NBCTs and achieving non-intrusive roll back by rolling back some subsequent batches also and reprocessing them.

8 Conclusion

In stream processing, input stream tuples are processed in batches by programs in a workflow. Several batches are processed concurrently and the batches may be split, merged or overlapped along the workflow. In this paper, we have identified the executions corresponding to the batches in terms of nonblocking batch composite transactions (NBCTs) that satisfy some basic transactional properties BP1-BP4.

When BP1 is satisfied, the conflicting transactions in each NBCT are ordered. If the transactions that are ordered are executed strictly serially (as can be

expected in stream processing), then all conflicts during the execution are between transactions of different NBCTs, and not between those of the same NBCT. Therefore, conflict-serializability of the NBCTs can be checked with a conflict graph consisting of nodes corresponding to NBCTs and directed edges representing conflicts among them. That is, there is no need to construct a graph with individual transactions of the NBCTs as vertices. Therefore, management of conflict graphs, and concurrency control, will be simpler.

In this paper, we have considered roll back of partially processed batches only. We have not considered compensation of the BCTs, after their commitment, at a later time. As in sagas, we can consider compensating BCTs. Further, these BCTs could be of *compensating* batches. Then, these BCTs could be executed like any other BCTs. The availability of compensating batches will be application-dependent.

References

1. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R.J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proc. VLDB Endow. **8**(12), 1792–1803 (2015). http://dx.doi.org/10.14778/2824032.2824076
2. Botan, I., Fischer, P.M., Kossmann, D., Tatbul, N.: Transactional stream processing. In: Proceedings of the 15th International Conference on Extending Database Technology EDBT 2012, pp. 204–215. ACM, New York (2012). http://doi.acm.org/10.1145/2247596.2247622
3. Conway, N.: Transactions and data stream processing. In: Online Publication, pp. 1–28 (2008). http://neilconway.org/docs/stream_txn.pdf
4. De Matteis, T., Mencagli, G.: Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. Int. J. Parallel Prog., pp. 1–20 (2016)
5. Golab, L., Özsu, M.: Issues in data stream management. ACM SIGMOD Rec. **32**(2), 5–14 (2003)
6. Gürgen, L., Roncancio, C., Labbé, S., Olive, V.: Transactional issues in sensor data management. In: Proceedings of the 3rd International Workshop on Data Management for Sensor Networks (DMSN 2006), Seoul, South Korea, pp. 27–32 (2006)
7. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Comput. Surv. **46**(4), 46: 1–46: 34 (2014). http://doi.acm.org/10.1145/2528412
8. Hummer, W., Satzger, B., Dustdar, S.: Elastic stream processing in the cloud. Wiley Interdisc. Rev. Data Min. Knowl. Disc. **3**, 333–345 (2013)
9. Meehan, J., Tatbul, N., Zdonik, S., Aslantas, C., Cetintemel, U., Du, J., Kraska, T., Madden, S., Maier, D., Pavlo, A., Stonebraker, M., Tufte, K., Wang, H.: S-store: streaming meets transaction processing. Proc. VLDB Endow. **8**(13), 2134–2145 (2015)
10. Mehrotra, S., Rastogi, R., Silberschatz, A., Korth, H.F.: A transaction model for multidatabase systems. In: Proceedings of the 12th International Conference on Distributed Computing Systems 1992, pp. 56–63. IEEE (1992)

11. Oyamada, M., Kawashima, H., Kitagawa, H.: Continuous query processing with concurrency control: reading updatable resources consistently. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing SAC 2013, pp. 788–794. ACM, New York (2013). http://doi.acm.org/10.1145/2480362.2480514
12. Schuldt, H., Alonso, G., Beeri, C., Schek, H.: Atomicity and isolation for transactional processes. ACM Trans. Database Syst. **27**, 63–116 (2002)
13. Vidyasankar, K.: Transactional properties of compositions of internet of things services. In: 2015 IEEE First International Smart Cities Conference (ISC2), pp. 1–6, October 2015
14. Vidyasankar, K.: A transaction model for executions of compositions on internet of things services. In: Procedia Computer Science, pp. 195–202. Elsevier (2016)
15. Vidyasankar, K.: Atomicity of batches in stream processing. J. Ambient Intell. Humanized Comput. (2017)
16. Vidyasankar, K., Krishna, P.R., Karlapalem, K.: A multi-level model for activity commitments in e-contracts. In: Meersman, R., Tari, Z. (eds.) OTM 2007. LNCS, vol. 4803, pp. 300–317. Springer, Heidelberg (2007). doi:10.1007/978-3-540-76848-7_20
17. Vidyasankar, K., Vossen, G.: Multi-level modeling of web service compositions with transactional properties. Database Manag. **22**(2), 1–31 (2011)
18. Wang, D., Rundensteiner, E.A., , Ellison III, R.T.: Active complex event processing over event streams. In: Proceedings of the VLDB Endowment, pp. 634–645. ACM Press (2011)

http://www.springer.com/978-3-662-55946-8

Transactions on Large-Scale Data- and Knowledge-Centered Systems XXXIV
Special Issue on Consistency and Inconsistency in Data-Centric Applications
Hameurlain, A.; Küng, J.; Wagner, R.; Decker, H. (Eds.)
2017, IX, 185 p. 34 illus., Softcover
ISBN: 978-3-662-55946-8