Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 98 (2016) 72 – 79
Procedia Computer Science

The 7th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2016)

On Atomic Batch Executions in Stream Processing

K. Vidyasankar

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

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. A general requirement 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. Executing only a prefix of the workflow amounts to dropping (discarding) the batches that were derived by the executed part and were supposed to be input to the rest of the workflow. In some cases, such partial executions may not be acceptable and may have to be rolled back, amounting to dropping the source input batches that were processed by the partial execution. We refer to this property of processing the batches either completely or not at all as *atomic execution* of the batches. We also attribute the property to the batches themselves, calling them *atomic batches*, meaning that the property applies to the set of transactions that are executed due to that batch. If batches are processed in isolation in the workflow, preserving atomicity is fairly straightforward. When batches are split or merged along the workflow computation, the problem becomes complicated. In this paper, we study issues relating to the atomicity of batches. We illustrate that, in general, preserving atomicity of some batches may affect the atomicity of some other batches, and suggest trade-offs.

Peer-review under responsibility of the Program Chairs
Keywords: Stream processing; transactions; atomic batches; compensation.

1. Introduction

Stream processing is about processing continuous streams of data. Stream data arriving from external sources are processed 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. The programs may generate stream data which may be input to subsequent programs in the workflow. As source input batches arrive continuously, several batches may be processed in the workflow simultaneously. In addition some OLTP (OnLine Transaction Processing)

* K. Vidyasankar. Tel.: +1-709-864-4369; fax: +1-709-864-2009.
E-mail address: vidya@mun.ca

Peer-review under responsibility of the Program Chairs
doi:10.1016/j.procs.2016.09.013

transactions may also be executed concurrently in the workflow. Ensuring correctness of these concurrent executions is important.

Concurrency issues have been studied widely in database context. The transaction concept has been extremely helpful to regulate as well as ensure the correctness of concurrent executions in database applications. The concept was introduced first in the context of (centralized) database systems, and then adopted in various advanced database and other applications, for example, in Web services[1], electronic contracts[2] and transactional memory[3]. Transactions are characterized by ACID properties: Atomicity, Consistency, Isolation and Durability. While these properties are considered very strictly for database operations and memory operations[3], they are relaxed in other applications, depending on the semantics and constraints of the application environments.

The earliest and most universally applied relaxation is with atomicity and isolation in the definition of *saga*s[4]:

- A transaction is said to be correct and to preserve consistency if it is executed completely or not at all;
- A higher level transaction can be split into, and executed by, several lower level transactions;
- Then, isolation is relaxed from the entire high level transaction to the individual lower level transactions;
- For atomicity, all the lower level transactions must be executed successfully, or none at all;
- If some of them are executed successfully, but others cannot be executed successfully, then the earlier ones need to be compensated, to achieve overall null execution; and
- The compensation can only be logical and should take into account that other transactions might have observed and used the results of the successfully executed low level transactions.

Stream processing involves continuous execution. As mentioned earlier, this is discretized by grouping (input) stream tuples into batches and using one batch at a time for the execution of programs. Each batch may trigger a set of programs in the workflow. Different batches may trigger different sets of programs, depending on the tuples in the batches and the semantics of the application. A general requirement is that each batch be processed completely in the workflow. That is, all the programs triggered by the batch, directly and transitively, must be executed successfully. Executing only a prefix of the workflow amounts to dropping (discarding) the batches that were derived by the executed part and were supposed to be input to the rest of the workflow. In some cases, such partial executions may not be acceptable and may have to be rolled back, amounting to dropping the source input batches that were processed by the partial execution. We refer to this property of processing the batches either completely or not at all as *atomic execution* of the batches. We also attribute the property to the batches themselves, calling them *atomic batches*, meaning that the property applies to the set of transactions that are executed due to that batch.

In many applications, all computations pertaining to an input batch are 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, atomicity of batches can be guaranteed in a straightforward manner. When batches are split or merged along the workflow computation (for example, when b consists of only a part of a' or it contains tuples from the outputs of several executions of P, on different batches), the problem gets complicated. In this paper, we study some issues related to atomicity of batches.

There have been several studies on the application of the transaction concept in stream processing. We elaborate the approaches in the Related Works section. To our knowledge, none of them address the atomicity property of the batches. In several applications, each input batch consists of just one tuple and it is processed in isolation. Here, the atomicity property follows trivially. We start with core definitions of compositions and transactions in stream processing environments in Section 2. We study the atomicity properties related to batches in Section 3. Some complex situations are illustrated in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. Executions

A stream processing workflow is a composition of programs. Formally, a *composition* C is $(\mathcal{P}, <_p)$, where $\mathcal{P}$ is a set of *transaction programs* $\{P_1, P_2, \ldots, P_n\}$, simply called *programs*, and $<_p$ is a partial order among them. We call the (acyclic) graph representing the partial order the *composition graph* $\mathcal{GC}(C)$. Each execution of a program yields a *transaction*. A transaction may have some stream and/or non-stream inputs, and may produce some stream and/or non-stream outputs. 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*.

In an execution of a composition, some of its programs will be executed, resulting in a set of transactions with a partial order $\prec_t$. We call this a *composite transaction*, 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 (like in Meehan et al.[5]). The partial order $\prec_t$ is compatible with $\prec_p$, 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 $T_i \prec_t T_k$.

The partial order in the composition graph includes (i) workflow order of the streams, (ii) the order defined between stream processing transactions and OLTP transactions, and among OLTP transactions, (referred to as *control order* in this paper) and (iii) the triggering relationships. Unless explicitly distinguished, we refer to all of these collectively as triggering relationships. Composite transactions inherit the ordering relationships from the composition. Thus, in a composite transaction, a transaction T_i may precede another transaction T_j due to any of the above three partial orders.

Executions of a composition may be triggered either by the arrival of a batch of stream input or by a OLTP-type invocation in the traditional sense of composition execution. We call the former as *stream composite transactions* (also, *batch composite transactions*) and the latter as *OLTP composite transactions*. We denote the composite transaction executed with batch b as $\mathcal{T}(b)$. 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 some more batches may arrive before all the transactions in $\mathcal{T}(b_1)$ are completely executed. Thus many stream composite transactions may be executed concurrently. Some OLTP composite transactions may also be executed concurrently.

General (strict) requirements for correct concurrent executions of composite transactions can be stated as follows[6].

1. *Unit of atomicity*: The atomicity requirement is that each composite transaction 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 composite transactions.
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 serial execution. OLTP composite transactions may occur in any order in the serial execution, relative to the stream composite transactions.
5. *Completion*: For each $\mathcal{T}$ equal to $(\{T_1, T_2, \ldots, T_m\}, \prec_t)$, all T_i's, for $1 \le i \le m$, must be executed. And, if **T** is the set of composite transactions under consideration, all of them must be executed.
6. *Monotonic execution*: At any time, the executed schedule must be such that, for any composite transaction $\mathcal{T}$, its projection on the completed transactions of $\mathcal{T}$ should be a prefix[1] of the transaction graph $\mathcal{GT}(\mathcal{T})$.

In most applications, the transactions will be executed in distributed fashion. Satisfying the above requirements would be impossible or, at the very least, will yield very poor performance. The semantics of the application may be such that many of those requirements could be relaxed. In this paper, we consider the following relaxations.

Like in sagas, we take the individual transactions (that is, individual executions of programs in the composition) as atomic units. That is, the atomic units are T's, not $\mathcal{T}$'s. Then, serializability is with respect to the atomic units, namely, individual transactions. If some inconsistency can be tolerated, the transaction order need not be followed for some transactions. Batch order may not be important for some $\mathcal{T}$'s, and even for some T's within a composite transaction $\mathcal{T}$. The completion requirement is that, for each composite transaction $\mathcal{T}$, all its constituent transactions should be executed, that is, the entire transaction graph $\mathcal{GT}(\mathcal{T})$ should be executed.. Relaxation of this requirement amounts to execution of only a prefix of the transaction graph. The monotonic execution property, that, at any time, the parts of the composite transactions that have been executed successfully should be prefixes of their respective transaction graphs, follows from the requirement that the transaction order should be followed in the execution. It also implies that compensation, if required, should be done in reverse order.

[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.

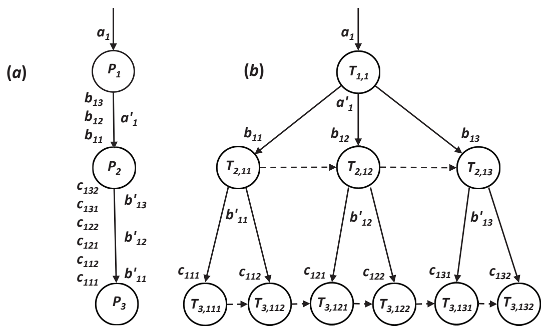

Fig. 1. Splitting of the batches

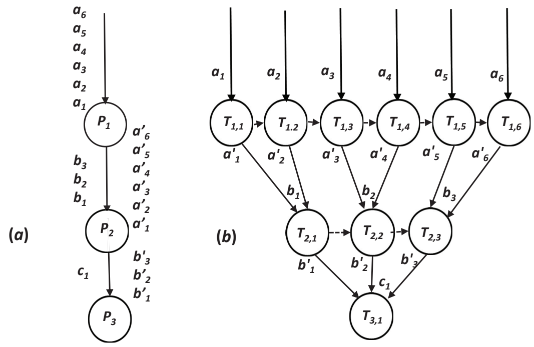

Fig. 2. Merging of batches

We show in the next section that, in certain circumstances, it may not be possible to satisfy both completion and monotonic execution properties independently.

3. Atomicity of Batches

Consider a composite transaction $\mathcal{T}$ and a transaction T in $set(\mathcal{T})$. Let b be a batch input to T. The batch composite transaction $\mathcal{T}(b)$ consists of $\{T\}$ union all the transactions triggered directly or indirectly by T in $\mathcal{T}$ (as per our general usage of the term 'triggering'). This definition applies to both source and derived batches.

We consider a simple example of processing stream inputs in 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 $a'_1, a'_2 \ldots$. In the case where each batch of P_i is executed in isolation, $a'_i = b_i$, and similarly $b'_i = c_i$. Then, for batch a_1, $\mathcal{T}(a_1)$ is $\{T_{1,1}, T_{2,1}, T_{3,1}\}$. Compensation of the batch a_1 involves compensating all the three transactions in this set.

For batch b_1, we have $\mathcal{T}(b_1)$ as $\{T_{2,1}, T_{3,1}\}$. Compensating b_1 will involve compensating $T_{2,1}$ and $T_{3,1}$. The compensation amounts to dropping the tuples in the batch b_1 at the level of executing P_2. As mentioned earlier, it is also possible that when a need for compensating b_1 arises, even the source batches from which b_1 was derived need to be compensated. In this example, the corresponding source batch is a_1 and hence the transaction $\mathcal{T}(a_1)$ also (that is, $T_{1,1}$ also) needs to be compensated.

Definition: For a set of batches B, a *source covering batch set*, denoted $scover(B)$, is a set of source input batches from which the batches in B are derived.

In the current example, $scover(b_1)$ is $\{a_1\}$. Note that when B contains a single batch, $\{b_1\}$ here, we drop the curly brackets for notational simplicity. We now consider the cases where a batch is not executed in isolation. First, we consider splits alone, then merges alone, and finally both of them occurring in the execution.

(a) *Splits*: Consider the following with respect to our composition example, depicted in Fig. 1. (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 $T_{2,11}, T_{2,12}, T_{2,13}$.
- Now, 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 $T_{3,111}, T_{3,112}, T_{3,121}, T_{3,122}, T_{3,131}, T_{3,132}$.

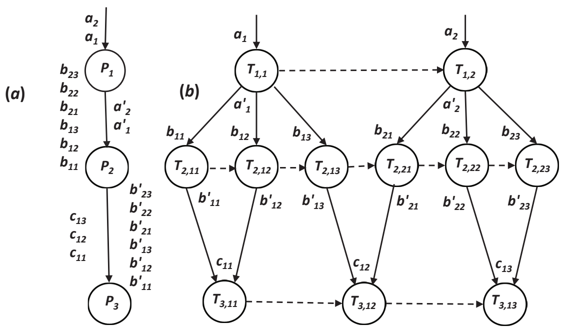

Fig. 3. Splitting and merging of batches

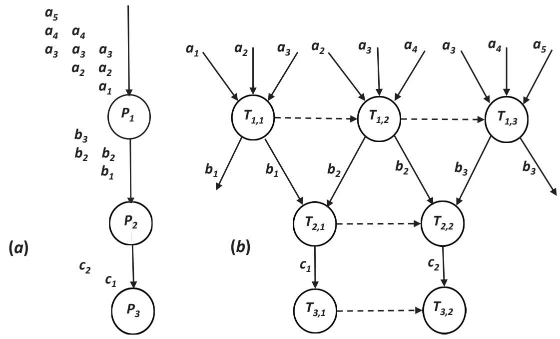

Fig. 4. Overlapping batches

For compensating b_{11}, the transaction $\mathcal{T}(b_{11})$ consisting of $\{T_{2,11}, T_{3,111}, T_{3,112}\}$ needs to be compensated. And, *scover*(b_{11}) is $\{a_1\}$. Compensating a_1 amounts to compensating all the transactions listed above.

(b) *Merges*: Merging of the batches is depicted in Fig. 2:

- 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 $a'_1, a'_2, \ldots, a'_6$ as output batches, respectively.
- Batch b_1 is $a'_2 \cdot a'_1$, b_2 is $a'_4 \cdot a'_3$, and b_3 is $a'_6 \cdot a'_5$ (where "·" indicates concatenation, of batches in the order of their arrival), and the executions of P_2 are $T_{2,1}, T_{2,2}, T_{2,3}$.
- 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 $\{T_{1,2}, T_{2,1}, T_{3,1}\}$. Compensation of $\mathcal{T}(a_1)$ involves compensations of $T_{2,1}$ and $T_{3,1}$ also. This compensates part of $\mathcal{T}(a_2)$ also. We discuss three ways of handling this.

(a) Compensate $T_{1,1}$ only. Then, the monotonic execution requirement, namely, that the completed transactions of $\mathcal{T}(a_1)$ should be a prefix of its transaction graph, will be violated.

(b) Compensate all the three transactions $\{T_{1,1}, T_{2,1}, T_{3,1}\}$. This amounts to dropping the batches a'_2, b'_2 and b'_3. This affects the completion requirements of $\mathcal{T}(a_i)$'s, for i from 2 to 6.

(c) Compensate *all* the transactions in this example. Note that the batch b_1 is derived from both a_1 and a_2. Similarly, c_1 is derived from all the six source batches of P_1. Compensating all the transactions amounts to dropping (that is, compensating) all the six batches $\{a_1, a_2, \ldots, a_6\}$. This will not affect the completion and monotonic execution requirements of any other batches.

We identify a few properties.

Definition: Let b be a source input batch.

- Compensation of $\mathcal{T}(b)$ is *interfering* if it affects the completion requirements of any other batches.
- The batch b is *independent* if the compensation of $\mathcal{T}(b)$ is non-interfering.
- A *non-intrusive* compensation of $\mathcal{T}(b)$ is compensation of (the transactions in) a prefix of $\mathcal{GT}(\mathcal{T}(b))$ that is non-interfering.

To distinguish from non-intrusive compensation, we sometimes use the term *full* compensation for compensating *all* the transactions in $\mathcal{T}(b)$. A non-intrusive compensation of a batch may affect the monotonic execution of that batch. Referring to the three options mentioned above in our current example, option (a) describes a non-intrusive compensation, option (b) is an interfering compensation and Option (c) describes an *scover* that is independent (whose full compensation is non-interfering). Note that the sets of transactions executed for any two independent batch sets will not have any transactions in common.

(c) *Splits and merges*: Figure 3 depicts both splits and merges.

- Input batches a_1 and a_2 for P_1 results in execution of $T_{1,1}$ and $T_{1,2}$ producing output batches a'_1 and a'_2.
- Batch a'_1 is split into three batches b_{11}, b_{12}, b_{13}, and similarly a'_2 is split into three batches b_{21}, b_{22}, b_{23}, for P_2, resulting in executions of $T_{2,11}, T_{2,12}, T_{2,13}$ and $T_{2,21}, T_{2,22}, T_{2,23}$, producing output batches $b'_{11}, b'_{12}, b'_{13}$, and $b'_{21}, b'_{22}, b'_{23}$.
- Batch c_{11} is $b'_{12} \cdot b'_{11}$, c_{12} is $b'_{21} \cdot b'_{13}$, and c_{13} is $b'_{23} \cdot b'_{22}$. The executions of P_3 are $T_{3,11}, T_{3,12}, T_{3,13}$.

Here, $\mathcal{T}(a_1)$ and $\mathcal{T}(a_2)$ are, respectively, $\{T_{1,1}, T_{2,11}, T_{2,12}, T_{2,13}, T_{3,11}, T_{3,12}\}$ and $\{T_{1,2}, T_{2,21}, T_{2,22}, T_{2,23}, T_{3,12}, T_{3,13}\}$. The two batch composite transactions have $T_{3,12}$ in common. Thus, neither a_1 nor a_2 is independent.

Full compensation of a_1, that is, full compensation of $\mathcal{T}(a_1)$, results in compensating $T_{3,12}$ also. This will amount to dropping the batch b'_{21}, thus affecting the completion requirement of $\mathcal{T}(a_2)$, while preserving the monotonic execution property of both batches. An independent *scover* of a_1, and also of a_2, is $\{a_1, a_2\}$.

The above examples suggest the following straightforward way of computing independent *scover*s for batches b. Here, we extend the transaction graph notation to a set of (composite) transactions.

- Let T be the transaction for which b is input.
- Let D_1 be the set of all transactions to which there is a directed path from T in the transaction graph $\mathcal{GT}(\mathbf{T})$.
- Let U_1 be the set of transactions from which there is a directed path to some transaction in D_1.
- Let D_2 be the set of transactions to which there is a directed path from some transaction in U_1.
- Continue building up the sets D_i and U_j this way until U_k, for some k, such that U_k equals U_{k-1}.
- Let s_i be the set of source batches that are input to U_i.
- Then an independent *scover*(b) is s_k, which is the set of source batches that are input to U_k.

.

We note that the above computation for *scover* will terminate at some point in the cases discussed above. We will see, in the following section, that this may not be true in some other situations.

4. Complex batches

In this section, we consider some complicated compositions of batches.

(a) *Overlapping batches*: So far, we assumed that batches input to the executions of a program are disjoint. In practice, the batches may overlap. For example, in the problem of computing an aggregate function every 5 minutes where the batch consists of the tuples received in the preceding 10 minutes, 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.

Here, to compensate the batch a_3, all the transactions listed above (and a few others like those of P_2 for which b_1 or b_3 are input) need to be compensated, resulting in other batches contributing only partially at different levels of execution. For example, the batch a_4 will be used only in the next batch $a_6 \cdot a_5 \cdot a_4$, and similarly a_5 will be used in the next two batches. We say that a_4 and a_5 are *partially dropped*. If the execution pattern continues as in the figure, partial drops are unavoidable; the iterative computation of *scover*s will not terminate and so an independent *scover* will not be obtained. Hence, a suitable *scover* can be chosen to compensate either fully or non-intrusively.

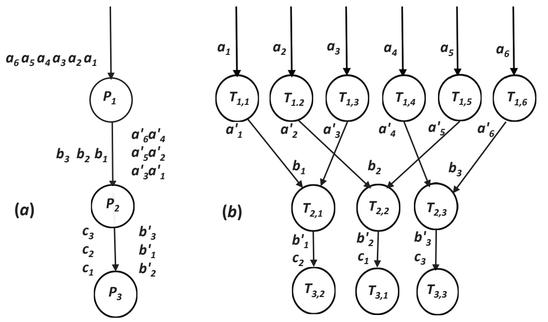

Fig. 5. Merging with relaxed batch order

(b) *Relaxing batch order*: As mentioned in Sec. 2, the batch order could be relaxed for some programs in the composition. That is, the batches need not be processed in the order they arrive. Then, they could even be processed in parallel, by different copies of the program. For instance, in our example composition, P_1 may be executed in parallel. Then, the output batches of P_1 may arrive at P_2 in an order which is different from the order in which their corresponding input batches arrive. This really does not affect the batch composite transactions for different batches when they are executed in isolation; each of them will still have one transaction of the program P_1. However, when splits and merges of the batches are involved down the workflow, things get complicated.

Figure 5 illustrates an execution of our example composition where batch order is relaxed for P_1 and P_2. The outputs of P_1 for batches $\{a_1, a_2, a_3, a_4, a_5, a_6\}$ are merged in the executions of P_2 and then the outputs of P_2 arrive for P_3 sequentially, both in the order shown. They are not merged in the executions of P_3. The important point to note is that merging of non-consecutive derived batches occurs at P_2. By our definition, $\{a_1, a_3\}$ will be an $scover(b_1)$. If we would like the *scover* to be a consecutive set of batches, then we should add a_2 to this set. And, for not affecting completion and monotonic execution requirements of other batches, we end up expanding *scover* to $\{a_1, a_2, a_3, a_4, a_5, a_6\}$, to get an independent set.

5. Related Work

In addition to the papers mentioned in Sec. 1, discussing transactional properties in different environments some other works include the following.

- Discussion of transactional stream processing[7] and the proposal of a unified transaction model, called UTM, that treats events also as transactions. Atomicity and isolation properties for transactions in this model are discussed in detail in the paper.
- Discussion of events and triggers in the context of Complex Event Processing over Event Streams[8]. They also define *stream* ACID properties for transactions: *s*-Atomicity, *s*-consistency, *s*-Isolation and *s*-Durability. The *s*-Atomicity notion requires “all operations stimulated by a single input event should occur in their entirely”. That is, a triggering transaction as well as all transactions triggered by them form a single unit of atomicity. In contrast, all transactions (including triggering and triggered ones) are individual atomic units in our paper.
- Transactional execution of stream composition in S-Store [5]. In that paper, the unit of atomicity is the entire composite transaction. They also use the term “atomic batch”. The batches are executed in isolation.
- Treating entire read-only composite transactions reflecting “continuous queries reading updatable resources” as the unit of atomicity in[9]. Such considerations are very useful, especially in IoT environments where monitoring and actuations are predominant, and monitoring should be consistent.
- Other papers discussing stream transactions and compositions[10,11,12].

6. Conclusion

After seeing the benefits of transactional properties to argue correctness of concurrent executions in database applications, these properties have been applied in several non-database contexts. They have been investigated in stream processing also. Concurrent executions in stream processing are data oriented whereas they are operation oriented in databases. In addition, stream executions are continuous. They appear to require additional transactional properties that are not relevant in database applications. In this paper, we have identified one such property, namely, atomic executions of batches of stream tuples.

The notion of atomicity of batches is that they must be processed either completely or not at all. Partial execution amounts to dropping some derived tuples in the middle of the workflow execution. To roll back partial execution, the source input batches that derived the tuples under consideration need to be compensated. When batches are processed in isolation, such compensation is straight-forward. However, when output batches of a program are split into smaller batches and/or merged with other batches for input to subsequent programs in the workflow, the compensation may not be independent, that is, it will affect the completion requirements of some other batches. To avoid the latter, non-intrusive compensation may be done. This will affect monotonic execution property of the current batches. We have illustrated these properties with several examples in this paper. We argue that, in practice, some trade-off between independence and intrusiveness in compensation is inevitable. Note that in a failure-free execution, each batch will be processed completely, by one or more composite transactions. Thus, the above mentioned trade-off may come into picture only during compensation.

While processing, various factors may determine whether batches are to be split or merged at different stages. The study in this paper suggests that atomicity is another factor that could be considered. Obviously, isolated execution (without splitting or merging) at any level enhances independent execution and compensation properties.

The *scover* is a covering source batch set for a given batch. We can also define covering batch sets in intermediate levels. These batches may be compensated when the initial prefix of the execution cannot be compensated. Covering batch sets in intermediate level might also help to identify programs that produce “bad” outputs and replace or rectify them.

Acknowledgment

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

References

1. K. Vidyasankar, G. Vossen, Multi-level modeling of web service compositions with transactional properties, Database Management 22 (2) (2011) 1–31.
2. K. Vidyasankar, P. R. Krishna, K. Karlapalem, A multi-level model for activity commitments in e-contracts, in: Proceeding OTM Conferences, 2007, pp. 300–317.
3. S. Peri, K. Vidyasankar, Correctness of concurrent executions of closed nested transactions in transactional memory systems, Theoretical Computer Science 496 (2013) 125–153.
4. H. Garcia-Molina, K. Salem, Sagas, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, 1987, pp. 249–259.
5. J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, H. Wang, S-store: Streaming meets transaction processing, Proc. VLDB Endow. 8 (13) (2015) 2134–2145.
6. K. Vidyasankar, A transaction model for executions of compositions on internet of things services, in: Procedia Computer Science, Elsevier, 2016, pp. 195–202. doi:10.1016/j.procs.2016.04.116.
7. I. Botan, P. M. Fischer, D. Kossmann, N. Tatbul, Transactional stream processing, in: Proceedings EDBT, ACM Press, 2012.
8. D. Wang, E. A. Rundensteiner, R. T. E. III, Active complex event processing over event streams, in: Proceedings of the VLDB Endowment, ACM Press, 2011, pp. 634–645.
9. L. Gürgen, C. Roncancio, S. Labbé, V. Olive, Transactional issues in sensor data management, in: Proceedings of the 3rd International Workshop on Data Management for Sensor Networks (DMSN'06), Seoul, South Korea, 2006, pp. 27–32.
10. L. Golab, M. Özsu, Issues in data stream management, ACM SIGMOD Record 32 (2) (2003) 5–14.
11. A. I. Luigi Atzori, G. Morabito, The internet of things: A survey, Computer Networks 54 (15) (2010) 2787–2805.
12. N. Conway, Transactions and data stream processing, in: Online Publication, http://neilconway.org/docs/stream_txn.pdf, 2008, pp. 1–28.