Continuous Query Processing with Concurrency Control: Reading Updatable Resources Consistently

Masafumi Oyamada  Hideyuki Kawashima  Hiroyuki Kitagawa
University of Tsukuba, Japan
masa@kde.cs.tsukuba.ac.jp, {kawashima, kitagawa}@cs.tsukuba.ac.jp

ABSTRACT
A recent trend in data stream processing shows the use of advanced continuous queries (CQs) that reference non-streaming resources such as relational data in databases and machine learning models. Since non-streaming resources could be shared among multiple systems, resources may be updated by the systems during the CQ-execution. As a consequence, CQs may reference resources inconsistently, and lead to a wide range of problems from inappropriate results to fatal system failures. We address this inconsistency problem by introducing the concept of transaction processing onto data stream processing. We introduce CQ-derived transaction, a concept that derives read-only transactions from CQs, and illustrate that the inconsistency problem is solved by ensuring serializability of derived transactions and resource updating transactions. To ensure serializability, we propose three CQ-processing strategies based on concurrency control techniques: two-phase lock strategy, snapshot strategy, and optimistic strategy. Experimental study shows our CQ-processing strategies guarantee proper results, and their performances are comparable to the performance of conventional strategy that could produce improper results.

Categories and Subject Descriptors
H.2.4 [Database Management]: Systems—Query Processing

General Terms
Algorithms, Theory

Keywords
Continuous Query, Data Stream Processing, Transaction, Concurrency Control

1. INTRODUCTION
Data stream processing has been gaining notable attention for several years with the rise of real-time data such as highly-frequent access logs, massive amount of messages from micro-blogs, and reports from wireless sensor nodes. A recent trend shows a new kind of data stream processing pattern with continuous queries (CQs), in which a wide variety of non-streaming resources (e.g., relational data in database systems and model data in machine learning systems) is referenced by data stream management systems (DSMSs) along with streaming data to conduct advanced analysis [5, 6, 11, 12, 14, 19–21]. However, such resources tend to be shared and updated by other systems (Figure 1), and conventional DSMSs could produce improper results in such situation. Surprisingly, in the field of data stream processing, most of the research about the use of external resources do not allow systems to update resources [5, 19–21], and so far only a few attempts have been made for such situations [4, 7]. In this paper, we consider situations where systems are allowed to share and update resources, and address the problem inherent in these situations.

Figure 2 illustrates a problem inherent in the situation, where resources referenced by DSMSs are updatable by other systems. In the figure, incoming events $e_1$, $e_2$, $e_3$, and $e_4$ are respectively combined with a resource $x$ by a join operation, and the results of the join operation are finally aggregated every four events. In midstream, the resource $x$ is updated to $x'$ by an application. The figure shows three timing of the resource update: (a) after the arrival of...
e_2$, (b) after the arrival of $e_4$, and (c) before the arrival of $e_1$. In (a), the aggregation result $q_1$ includes both $x$ and $x'$, that is, referenced resources are inconsistent. In contrast, referenced resources are consistent in (b) and (c).

Inconsistently referenced resources epitomized by (a), hereafter referenced-resource inconsistency, could lead to a wide range of problems from inappropriate results to fatal system failures. For instance, if a relational table is referenced as a resource, and the schema of the table is updated in midstream, then aggregation of events that have referenced multiple schema may produce improper results or cause system failures. Therefore, preventing the inconsistency problem is a key aspect for advanced DSMS that heavily reference shared resources and conduct aggregation.

If a CQ is as simple as Figure 2, preventing referenced-resource inconsistency is not a difficult problem. Since we know that join results are aggregated every four events, (1) we prevent the resource from being updated until the resource is referenced four times, (2) release the lock and allow applications to update the resources, and then repeat from the step (1). However, in real systems, CQs are highly-complex, and aggregation operations might be entangled. In such case, we cannot employ the straightforward approach described above.

In this paper, we introduce a solution to referenced-resource inconsistency problems, which works for highly-complex CQs successfully. By introducing the concept of transaction processing onto data stream processing, we describe inconsistency problem in terms of transaction processing. We show a way to derive read-only transactions from CQs, and illustrate that the inconsistency problem is solved by ensuring serializability of transactions. We propose three CQ-processing strategies that ensure serializability of transactions, and measure their performance in a real system.

The rest of the paper is organized as follows. In Section 2, we describe basic concepts related to data stream processing. In Section 3, we show an idea that derives transactions from CQs, and define the problem clearly. We propose three CQ-processing strategies based on concurrency control techniques in Section 4 and Section 5. In Section 6, we present an experimental evaluation of proposed strategies in a real system. Then, we cover the related work in Section 7. Finally, we conclude in Section 8.

2. PRELIMINARIES

In this section, we describe basic concepts of data stream processing concisely.

2.1 Data Stream

A data stream is a sequence of indefinitely arriving events. To represent data streams formally, a number of models have so far been proposed [2,3,18]. In this paper, we use the following model, which originates from a widely received model based on relational algebra [2].

Definition 2.1 (Data Stream) A data stream $S$ is a set of an event $e: (s,t)$, where $s \in S\text{cema}$ is a tuple that belongs to $S$’s schema $S_{\text{cema}}$ and $t \in T$ is the timestamp of the event.

2.2 Continuous Query and Operator Tree

To process a massive amount of streaming data in on-line fashion, data stream management systems (DSMSs) [1,2,12,15] have been actively studied and developed so far. DSMSs receive continuous queries (CQs) written in human-readable query languages [2,10], and translate them into internal physical plans called operator trees. An operator tree consists of leaf nodes that represent input data streams, and non-leaf nodes that represent relational operators. Each operator has at least one input stream and at least one output stream. For leaf nodes, input streams are raw data streams (input streams for CQs). For non-leaf nodes, input streams are output streams of other operators.

When an event from a data stream arrives at a DSMS, the event is routed to a corresponding leaf node. Then, the event is forwarded to the parent operator of the leaf node and processed by that operator. The operator pushes the result into its output stream, and if the parent operator of the operator exists, the parent operator takes the result from its input stream, and process it in turn. By repeating this process, events climb up the operator tree, and the root operator output events continuously.

For a result of an operator, events used by the operator to produce the result are formally represented by the following function.

Definition 2.2 (Dependent Event Function) Let $S$ be a set of all data streams appears in a continuous query. The dependent event function $\text{DEP}: S \rightarrow 2^S$ for an event $e \in S \subseteq S$ is defined by

- $\text{DEP}(e):= \bigcup_{k=1}^{n} S_{\text{dep}}^{e}$, if $S$ is an output stream of an operator whose input streams are $S_{1}^{\text{in}}, \ldots, S_{m}^{\text{in}}$ and $e$ is produced by the operator using $S_{\text{dep}}^{e} \subset S_{1}^{\text{in}}, \ldots, S_{m}^{\text{in}}$.
- $\text{DEP}(e):= \emptyset$, if $S$ is a raw input stream for the CQ ($S$ is a leaf node of an operator tree).

2.2.1 Window-Based Operator

Operators that conduct computation over ranges of events from data streams are called window-based operators, which include JOIN, SORT, AVERAGE, MIN, and MAX. A window-based operator buffers a range of events called window, and conduct a computation over the window. Windows for a data stream clip events is determined by two parameters: range $r$ and SLIDE $s$ [2,17]. Range $r$ determines the number of events in a window, and slide $s$ determines the shift between consecutive windows. If $s < r$ for windows, they are referred to as overlapping windows. Figure 3 shows an example of non-overlapping windows and Figure 4 shows an example of overlapping windows.

2.2.2 Reference Operator

Operators that reference non-streaming external resources in order to produce their results are called reference operators, which include stream-relation join and data stream classification using external classifiers. For a result of an operator, resource reference operations conducted in order to produce the result are formally represented by the following function.

\[^{1}\text{In this paper, we assume windows are event-based ($r$ and $s$ are specified by the number of events) for brevity, but our approaches can also be applied to time-based windows ($r$ and $s$ are specified by the amount of time).}\]
Definition 2.3 (Dependent Reference Function) Let $S$ be a data stream and $X$ be a set of all external resources possibly referenced in a CQ. The dependent reference function $REF : S \rightarrow \{X \times T\}$ for $e \in S$ is defined by:

- $REF(e) := \{< x_1, t_1 >, \ldots, < x_n, t_n >\}$, if $S$ is an output stream of a reference operator and each external resource $x_k \in X (1 \leq k \leq n)$ is referenced at the time $t_k \in T$ in the process of $e$ at the reference operator.
- $REF(e) := \emptyset$, otherwise.

3. CQ-DERIVED TRANSACTION

In this section, we introduce CQ-derived transaction, a concept that derives read-only transactions from CQs. We also define the problem we address in terms of CQ-derived transaction.

3.1 Consistency Problem Revisited

In terms of transaction processing, we revisit the three situations illustrated in Figure 2, where events from a data stream are combined with an external resource $x$, and the results are aggregated by a window operation ($r = 4$) in a DSMS. Here, we regard successive resource reference operations whose results are involved in the same aggregate operation as a transaction, and the result output process of the aggregate operation as the commit operation of the transaction; we use $r_1(x)$ to denote a reference operation to the resource $x$ made by the transaction, and $c_1$ to denote its commit operation. Then, we consider that the resource $x$ is updated by a transaction from a system (not the DSMS); we use $w_2(x)$ to denote the update operation, and $c_2$ to denote the commit operation of the transaction. By using these notations, three situations can be represented as schedules of the two transactions: a non-serializable schedule represented in Example 3.1, and serializable schedules represented in Example 3.3 and Example 3.2.

Example 3.1 (Referenced $x$ and $x'$)

$r_1(x), r_1(x), w_2(x), c_2, r_1(x), r_1(x), c_1$

Example 3.2 (Referenced $x$)

$r_1(x), r_1(x), r_1(x), r_1(x), c_1, w_2(x), c_2$

Example 3.3 (Referenced $x'$)

$w_2(x), c_2, r_1(x), r_1(x), r_1(x), r_1(x), c_1$

The goal of this paper is to ensure that a CQ consistently references non-streaming resources when the CQ produces a result that depends on the resources. That is to say, our objective is to prohibit problematic schedules, as shown in Example 3.1, whose execution orders lead to inconsistent results. In transaction processing, such schedules are referred to as non-serializable schedules, and a number of concurrency control techniques for transactions have so far been studied. In order to prevent problematic schedules, we consider a running CQ as consecutive read-only transactions, and evaluate operators in strategies based on concurrency control techniques.

We show how transactions are derived from a CQ in Section 3.2, and we describe when derived transactions are regarded as committed in Section 3.3. Then, by using these concepts, we enunciate the problem in Section 3.4.

3.2 Deriving Transactions

Since a CQ needs to consistently reference a resource to produce a consistent result, we have to evaluate the sequence of reference operations conducted to produce the result transactionally. Thus, we consider a sequence of reference operations in reference operators used to produce a final result of a CQ as a transaction, hereafter referred to as a CQ-derived transaction. To represent a CQ-derived transaction formally, we employ the function $REF$ introduced in Section 2.2.2 and introduce a function that maps a result of an operator to a set of all reference operations conducted for the result, which may include operations in upstream operators, as follows.

Definition 3.1 (Transaction Deriving Function) Let $S$ be a data stream and $X$ be a set of all external resources possibly referenced in a CQ. The transaction deriving function $TXN : S \rightarrow \{X \times T\}$ for $e \in S$ is recursively defined as follows.

$$\text{TXN}(e) := \bigcup_{e^\text{dep} \in \text{DEP}(e)} \text{TXN}(e^\text{dep}) \cup \text{REF}(e)$$

3.3 Commit Operator

The commit operation for a CQ-derived transaction can be conducted safely after the root operator of the CQ computes the result that corresponds to the CQ-derived transaction, since all reference operations related to the results are conducted transactionally, and there is no possibility the CQ produces an inconsistent result. However, the commit operation for a CQ-derived transaction can be conducted earlier for most CQs. For instance, in the operator tree represented in Figure 5, the root projection operator does not modify (e.g., adds more reference operations) CQ-derived transactions, and thus commit operations for the CQ-derived transactions can be conducted safely after evaluations of the previous aggregation operator. An operator that ensures the subsequent downstream operators do not modify CQ-derived transactions is referred to as a commit operator, and concisely represented as follows.

Definition 3.2 (Commit Operator) The commit operator for an operator tree is the topmost operator in the set of window-based operators and reference operators in the tree.

3.4 Problem Definition

Here, we define the problem clearly: given CQ-derived transactions and update transactions from other systems, a schedule of the transactions must be mono-version [23] and conflict- or view-serializable to suppress the referenced-resource inconsistency sufficiently. The following two sections show three approaches to this problem.

4. PESSIONISTIC STRATEGIES

In this section, we propose two CQ-processing strategies called two-phase lock strategy and snapshot strategy, which are based on pessimistic concurrency control protocols.
Table 1: Lock compatibility in S2PL. In the table, + indicates compatible (request is not blocked) and - indicates not compatible (request is blocked).

<table>
<thead>
<tr>
<th>Requested</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Held</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>-</td>
</tr>
</tbody>
</table>

4.1 Two-Phase Lock Strategy

First, we introduce two-phase lock strategy, which is based on strict two-phase lock (S2PL), a widely used concurrency control protocol. In two-phase lock strategy, a reference operator takes a read-lock when it tries to reference a resource, and keeps the read-lock until the corresponding CQ-derived transaction commits. By enforcing the lock compatibility shown in Table 1 for both CQ-derived transactions and update transactions, the two-phase lock strategy ensures that until a transaction commits, no other transactions are able to update resources referenced in the transaction. In doing so, this strategy ensures conflict serializability of transactions.

Running times of CQ-transactions are unpredictable and could be very long, since they highly depend on data stream rates. Thus, transactions in two-phase lock strategy could retain read-locks for a long time, preventing resources from being updated by other systems. Therefore, two-phase lock strategy is not suitable for applications where resource update throughput is important.

4.2 Snapshot Strategy

Second, we introduce snapshot strategy. In contrast to two-phase lock strategy, CQ-derived transactions in this strategy do not retain locks for resources during their running time. Instead, when a CQ-derived transaction begins, the transaction copies all resources it will reference as snapshots. Then, the snapshots are referenced by reference operators in the transaction instead of the current resources. When a CQ-derived transaction commits, all snapshots taken for the transaction are expired and removed. By referencing snapshots at the same point in time in each CQ-derived transaction, this strategy ensures conflict serializability of transactions.

4.3 Supporting Overlapping Window

Even in the above two pessimistic strategies based on concurrency control, some conventional techniques for overlapping windows could lead to non-serializable schedules. One of such techniques is synopsis [1]. A synopsis belongs to an overlapping window-based operator and caches past input events that may be used by future window operations in the operator. By doing so, synopses suppress re-evaluation of upstream operators. While the use of synopses enhances performance of CQ-processing, under our concurrency control strategies, it corresponds to reusing results of transactions among concurrent transactions. In this case, if an update transaction from another system is executed between the consecutive transactions, a generated schedule could be non-serializable (Window 2 in Figure 4).

To prevent non-serializable schedules, we introduce a technique that ensures serializable schedules even with overlapping windows. The main idea of this technique is re-evaluating limited operators in order to prevent reuse of transaction results while leveraging synopses.

4.3.1 Redo Queue

We prepare auxiliary queues called redo queues that back up input events at certain input streams: (1) input streams of a reference operator that has window-based operators in its downstream (ancestor operators in the operator tree) and does not have reference operators in its upstream, and (2) input streams that do not have reference operators in its upstream and belong to a window-based operator that has at least one reference operator in its upstream. If an event arrives at an input stream that has a redo queue, the event is added to the redo queue. The events backed up might be used in re-evaluation of operators.

Algorithm 1: Setup-Redo-Queues(op, has-win_i)

Output: A boolean value that indicates whether upstream operators of op have reference operators or not

1 if Is-Leaf-Node(op) then
2 return false
3 has-win_i ← Is-WINDOW(op) \ has-win_i
4 has-ref_i ← false
5 S_ref ← 0
6 foreach op_up ∈ op.children do
7 /* Setup upstream operators first, and record whether reference operators exist in upstream of op_up */
8 has-ref_i ← Setup-Redo-Queues(op_up, has-win_i)
9 has-ref_i ← has-ref_i \ has-ref_i
10 if has-ref_i then
11 /* Record streams that have reference operators in their upstream */
12 S_ref ← S_ref ∪ op_up.output-streams
13 if Is-REFERENCE(op) \ has-win_i \ ~has-ref_i then
14 /* (1) Setup redo queues for all input streams of a qualified reference operator */
15 foreach S ∈ op.input-streams do
16 /* (2) Setup redo queues for all qualified input streams of a window operator */
17 if has-ref_i \ S ∉ S_ref then
18 /* Arrange the redo queue for S */
19 return has-ref_i \ Is-REFERENCE(op)

Figure 5 represents an example of redo queue placement for an operator tree. To conduct such redo queue arrangement efficiently, we use the procedure Setup-Redo-Queues shown in Algorithm 1 in our DSMS implementation. For an operator tree rooted by root, we call the procedure with the two arguments root and false, and then the redo queues for the operator tree are placed efficiently, since the procedure is essentially a DFS algorithm.
we introduce the following Low Water-Mark (LWM) an example of the redo area is represented in Figure 5. Let S be a data stream. Optimistic concurrency control consists of three phases: read phase, validate phase, and write phase. Since CQ-derived transactions are read-only, we can omit the write phase. The detailed steps of our optimistic strategy can be represented as follows.

- **Read phase**: Each reference operator references resources.
- **Validate phase**: When a commit operator is ready to output a result, if read operations in the CQ-derived transaction are not interleaved by write operations from update transactions, commit the CQ-derived transaction. Otherwise, abort and redo the CQ-derived transaction.

To implement optimistic strategy, we have to consider two topics: (1) a way to achieve validation and (2) a way to achieve abort and redo of transactions. In this paper, we present simple approaches, and use them in our DSMS implementation.

### 5.1 Validation
To achieve validation, we assume that a resource holds the timestamp of the latest update operation on the resource, and associate the timestamp with a read operation that references the resource at the time. Such a timestamp is referred to as a read operation’s referenced-resource timestamp. In the validation phase of a CQ-derived transaction, we compare the transaction’s beginning time with read operation’s referenced-resource timestamps \( t_1, \ldots, t_n \). If there exists \( k \) such that \( t_k > t_0 (1 \leq k \leq n) \), then another transaction must have updated the resource in the middle of the CQ-derived transaction, and the CQ-derived transaction turns out to be invalid. Otherwise, the CQ-derived transaction is valid, since it can be a member of a conflict-serializable schedule.

### 5.2 Abort and Redo
Since CQ-derived transactions are read-only, they do not make any changes to resources. Therefore, the process for aborting and redoing a transaction is simply canceling the commit operator’s operation and re-inputting the events the transaction depends on. To achieve this process, we apply the redo process mechanism of pessimistic concurrency controls described in Section 4.3.2 without any modification.

### 6. EXPERIMENTS
We perform real-machine experiments to capture characteristics of our proposed CQ-processing strategies.

#### 6.1 Experimental Setup
We have conducted experiments with our DSMS to evaluate proposed continuous query processing strategies. The DSMS is written in C++ and implements the three CQ-processing strategies proposed in this paper. To compile the DSMS, we used GNU g++ 4.6.3 and specified -O2 as the optimization level. Experiments have been conducted on a machine equipped with a quad-core Intel Core2 Quad Q9400 CPU (2.66 GHz, 6M L2 Cache, 1333MHz FSB) and 4GB DRAM running Ubuntu Linux with 3.2.0 kernel.

To simulate situations where resources referenced by CQs are updatable by other systems, we register a simple CQ to the DSMS, and redo of transactions. In this paper, we present simple approaches, and use them in our DSMS implementation.

### 4.3.2 Redo and LWM
When the commit operator in an operator tree produces a result, the corresponding CQ-derived transaction for the result commits and finishes. From the viewpoint of serializability, events in synopses, which may contain results of past read operations, could be improper to be reused in the consecutive transaction. For this reason, a CQ-derived transaction commits, we clear events in synopses that may contain improper events and re-input events in redo queues into corresponding input streams. Synopses to be reset are limited to the ones that belong to the subtree whose root node is the commit operator and leaf nodes are operators with redo queues. Such a subtree of an operator tree is called a redo area, and an example of the redo area is represented in Figure 5.

To decide events that will be used in the consecutive transaction, we introduce the following Low Water-Mark (LWM).

**Definition 4.1 (Low Water-Mark)** Let S be a data stream. Low Water-Mark for an event \( e \in S \) is

- the timestamp of \( e \), if \( S \) is a raw input stream for a CQ
- the lowest LWM for the events \( \text{DEP}(e) \), if \( S \) is an output stream of an operator.

By using LWM, the redo process after the commit operation of a CQ-derived transaction is conducted as follows. First, the biggest LWM is selected from events in overlapping windows involved in the transaction, as represented in Figure 6. Second, we reset synopses that belong to operators in the redo area, and re-input events whose timestamps are equal to or higher than the selected LWM.

The LWM of an event indicates that events whose timestamps are smaller (older) than the LWM are not needed to compute the event. Thus, when a CQ-derived transaction commits, we can remove the events whose timestamp are smaller then the LWM of the result from redo queues safely to suppress resource consumption.

### 5. OPTIMISTIC STRATEGY
In this section, we propose a CQ-processing strategy based on optimistic concurrency control [16]. Optimistic concurrency control consists of three phases: read phase, validate phase, and write phase. Since CQ-derived transactions are read-only, we can omit the write phase. The detailed steps of our optimistic strategy can be represented as follows.

- **Read phase**: Each reference operator references resources.
- **Validate phase**: When a commit operator is ready to output a result, if read operations in the CQ-derived transaction are not interleaved by write operations from update transactions, commit the CQ-derived transaction. Otherwise, abort and redo the CQ-derived transaction.

To implement optimistic strategy, we have to consider two topics: (1) a way to achieve validation and (2) a way to achieve abort and redo of transactions. In this paper, we present simple approaches, and use them in our DSMS implementation.
attribute. To make windows overlap, range \( r \) is varied from experiment to experiment, and slide \( s \) is set to \( \lfloor \frac{r}{2} \rfloor \).

While the DSMS processes the above operator tree continuously, the relational table referenced by the equi-join operation is updated by an application that dispatches update transactions periodically.

In the following experimental results, “Naive” means conventional no concurrency control support strategy, “S2PL” means two-phase lock strategy described in Section 4.1, “Snapshot” means snapshot strategy described in Section 4.2, and “Optimistic” means optimistic strategy described in Section 5.

6.2 Sensitivity to Update Transaction

First, to see how our CQ-processing strategies are interfered by update transactions, we vary dispatch rate of update transactions from 1 (txn/s) to 100,000 (txn/s), and measure two metrics in the DSMS: (a) proportions of consistent results of the CQ, and (b) throughput of the CQ. Window range \( r \) for the aggregation operator of the CQ is set to 5.

6.2.1 Consistent Result Ratio

Figure 7(a) represents proportions of consistent results of the CQ for each CQ-processing strategy. While all results of the CQ are consistent in our CQ-processing strategies, the proportion of consistent results in conventional strategy degrades with the increase of update transaction’s arrival rate. Note that even in the case where the arrival rate of update transactions is minimum (1 tps in this experiment), there are several inconsistent results. This means that in applications where no inconsistent results are permitted, conventional strategy cannot be employed.

6.2.2 CQ Throughput

Figure 7(b) shows throughput of the CQ in each CQ-processing strategy. When the arrival rate of update transactions is low, optimistic strategy shows high throughput comparable to the throughput of conventional strategy. However, with the increase of update transaction’s arrival rate, the throughput of optimistic strategy rapidly decreases and becomes the worst one. This is because frequent updates to resources increase the probability that CQ-derived transactions are aborted in the validation phase.

6.3 Sensitivity to Window Range

Second, to see the impact of window range in each CQ-processing strategy, we vary the window range \( r \) of the aggregation operator from 5 to 640, and measured two metrics in the DSMS: (a) throughput of update transactions, and (b) throughput of the CQ. In this experiment, dispatch rate of update transactions is fixed to 100,000 (txn/s), and effective throughput of update transactions are measured.

6.4 Discussion

Experimental results showed our proposed strategies always produce proper results that reference resources consistently as described in Section 6.2.1, and their throughput are comparable to the throughput of naive strategy in certain conditions. Since our proposed strategies are not optimal, we should investigate characteristics of an application and carefully choose proper strategies at the time of system deployment. We plan to study automation of such task and adaptive switching of strategies as future work.

![Figure 7: Impact of update transactions on consistency and CQ throughput.](image)

![Figure 8: Impact of window range on update transaction throughput and CQ throughput.](image)
7. RELATED WORK

Our work relates to isolation issues in data streams. Conway studied the concept of transactions in data stream processing and proposed to use a window as the isolation unit [7]. There are several systems that ensure isolation in a certain unit: each event [8], events who share the same timestamp [1], and events in the same window [9, 15]. However, these systems only ensure isolation in the level of an operator, whereas our work ensures isolation in the level of an operator tree, that is, CQ-level.

Transactional issues in data stream processing are discussed by Gurgen et al [13]. They noticed that a CQ-specific meta-information such as selection condition can be modified in the middle of CQ-execution and could lead to false alarms. They considered a CQ as a nested transaction, and proposed a concurrency control technique based on optimistic priority-based concurrency control protocol. We believe that our work can be incorporated into their work, since how a CQ is decomposed into nested sub-transactions is a bit unclear in their paper, whereas our work shows clear way to derive transactions from a CQ using windows.

A general transaction model for data streams is studied by Botan et al [4]. They treated both streaming data and non-streaming data equally as objects to which read and write operations are defined. Their work focused on the general transaction model and the amount of discussions about implementation issues is relatively small. In contrast, our work focuses on real systems and shows several important topics for the implementation.

Wang et al. tackled with transactional issues in Active Complex Event Processing (ACEP), a novel CEP concept that can treat states in CEP queries to conduct advanced event pattern matching [22]. In ACEP, multiple queries reference and update a state concurrently, and as a consequence, a query may read the state inconsistently and detect false patterns. To prevent such a problem, they proposed several pessimistic concurrency control techniques. In contrast to their work where the isolation unit is event patterns, we regard whole CQs as the isolation unit, which potentially include entangled windows.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we address the consistency problem inherent in the situations where resources referenced by CQs are updatable by other systems. We have described the problem in terms of transaction processing, and introduced CQ-derived transaction, a concept that derives read-only transactions from CQs. Based on the concept, we have shown that the consistency problem is solved by ensuring serializable schedules of CQ-derived transactions and update transactions. To ensure serializable schedules, we have proposed three CQ-processing strategies: two-phase lock strategy, snapshot strategy, and optimistic strategy. Experimental study showed our proposed strategies ensured consistent results, and the performance of the proposed strategies was comparable to naive strategy that generated inconsistent results.

Since each proposed strategy is not suited for all situations, we plan to study a way to select proper strategies for a target application as described in Section 6.4. We also plan to study runtime switching of CQ-processing strategies to adapt to dynamically changing characteristics of data streams.

9. ACKNOWLEDGEMENTS

The work is partially supported by NICT (New generation network R&D program for innovative network virtualization platform and its applications) and KAKENHI (#24500106). We thank Mateus Cruz and Wang Yan for their comments on the paper. We also thank the anonymous reviewers for their insightful comments during the review process.

10. REFERENCES