Uninterruptible Migration of Continuous Queries without Operator State Migration

Thao N. Pham, Nikos R. Katsipoulakis, Panos K. Chrysanthis, Alexandros Labrinidis
Department of Computer Science, University of Pittsburgh, USA
{thao, katsip, panos, labrinid}@cs.pitt.edu

ABSTRACT

The elasticity brought by cloud infrastructure provides a promising solution for a data stream management system to handle its incoming workload, which can be highly variable: the system can scale out when heavily loaded, and scale in otherwise. In such a solution, the efficiency of the mechanism used to migrate a query from one node to another is very important. Generally, a stream application requires real-time outputs for its continuous queries, and downtime is not acceptable. Moreover, the migration should not add considerable processing cost to a node that could have been already overloaded. In this paper, we present our migration protocol, named UniMiCo, which satisfies those requirements. We implemented UniMiCo in a *DSMS* prototype and experimentally show that the protocol preserves correctness, while introducing no noticeable changes in the response time of the continuous query being migrated.

1. INTRODUCTION

Today, the ubiquity of sensing devices as well as mobile and web applications leads to the generation of huge amounts of data, which take the form of streams. Those data streams are typically high-volume, high-velocity (fast) and have high-variability (bursty). Data stream management systems (*DSMS*s) have become the popular solution to handle data streams, by efficiently supporting continuous queries (*CQ*s), which process data as they arrive *on the fly*.

The bursty incoming workload can overload the *DSMS* during its peaks. As a result, output is delayed and fails to meet the real-time requirements of monitoring applications and of emerging "Big Data" applications [8]. Most modern cloud infrastructures provide *elasticity*, which can be used to handle overloading situations [9]. Flux [12] was one of the early attempts to introduce a monitoring and load detection operator in a query network, and provided a state migration protocol to move *CQ*s across different machines. Another solution uses backup Virtual Machines (*VM*s) for periodically storing state [3]. In the event of load imbalance, the migrated *CQ*s restore the state from the backup *VM*s and apply incremental changes before resuming execution. Similarly, the operation migration mechanism in [10] follows the state migration paradigm. The efficiency of the migration mechanism is crucial, and no system downtime is acceptable since it translates to loss of data (hence the term "live" in previous work).

As part of the effort to scale-up/-down AQSIOS [4], our *DSMS* prototype, we implemented our own query migration protocol, named UniMiCo (**Uni**nterruptible **Mi**gration of **Co**ntinuous Queries). UniMiCo has the ability to (i) migrate stateful *CQ*s without the need to transfer any state, and (ii) do the migration in a "live" fashion (i.e., no downtime).

Our approach on *CQ* migration generalizes the idea of the *Window Recreation Protocol* (*WRP*) presented in [7] in two functional ways: First, while *WRP* was proposed to handle the migration of a sub-query with only one stateful operator, the UniMiCo protocol allows migrating a query with multiple stateful operators, each of which could have a different window specification. Second, in contrast to *WRP* that considers only time-based windows, UniMiCo's protocol has been designed in a general way to handle both time-based and tuple-based windows. A minor difference between *WRP* and UniMiCo is that UniMiCo does not involve the upstream data source in synchronizing the migration point, otherwise the two protocols share the same performance advantages and limitations. Both migration protocols are equally effective in migrating operators without state migration and query downtime, yet they might not be suitable when the window is too large (e.g., 24 hours [7]) since they may prolong an overloaded situation at the originating node.

We make the following *contributions* in this paper:

- We present the complete UniMiCo protocol that migrates a CQ with multiple stateful operators from one node to another.
- We experimentally show that UniMiCo migrates a *CQ* correctly without incurring any noticeable "hiccups" in its response time.

2. SYSTEM MODEL

We assume a system consisting of multiple shared-nothing nodes, connected by a reliable, high-speed network. One node serves the role of the coordinator, while the others are peers and each one of them runs one instance of AQSIOS. AQSIOS is our experimental *DSMS*, extended from the STREAM source code [2]. Our extensions include new operator implementation [6], optimization schemes [5], new scheduling policies [13], load shedders [11], and UniMiCo, our protocol to transfer a *CQ* from one node to another.

Based on the workload of each node reported by AQSIOS's load manager, the coordinator initializes a *CQ* migration when necessary. For a specific *CQ* migration between two nodes, we refer to the node which is running the *CQ* as *originating node*, and the node which is going to receive the *CQ* as *target node*. The migration can be materialized either through direct communication between the *originating* and *target node*s, or through indirect communication via the coordinator. In this paper we assume the former, but UniMiCo can work equally well with the latter.

AQSIOS supports a *CQ* execution model similar to Borealis and Apache Flink. Each AQSIOS node keeps a copy of the whole query network, but, only a subset of it is active on the node. A node only connects to the stream sources that are necessary for the active queries in the node. Data streams, coming from (possibly) different sources, are received by the *source operators*, which are the most upstream operators in a *CQ*. Figure 1 is an example of our system model with two AQSIOS nodes. The *CQ*s comprised of dark operators are those active at the node. The dashed lines represent network connections among the nodes.

In this paper we consider the whole query as the migration unit. However, the protocol can also be used to migrate only a segment of a *CQ*: the operator(s) right before the migrated segment becomes the stream source(s) for that segment. and their downstream operators act as source(s) in the corresponding *CQ*s.

Window-based operators

There are two types of operators in a *CQ*: *stateless* and *stateful* operators. A stateless operator, such as selection (σ), produces an output tuple based solely on the current input tuple. Conversely, a stateful operator, such as join or aggregation, needs to refer to values from previous input tuples. Due to the fact that input streams are infinite, *DSMS*s use either *tumbling* or *sliding windows*, to limit the state of operators. Sliding windows allow the output to be continuously computed based on the most recent "portion" of the stream data. In addition, a sliding window is specified through a length (or range) l, and a slide s, which can be either time interval or tuple count.

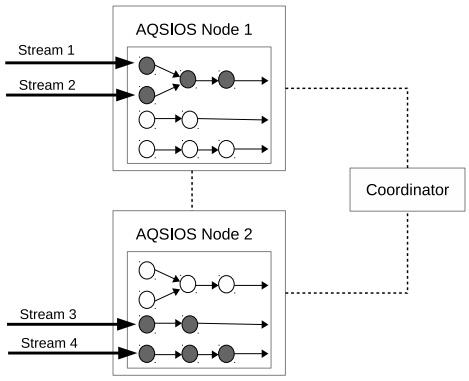

Figure 1: System model

These two types of windows are called *time-based* and *tuple-based windows*, respectively [2].

While most *DSMS*s embed the window definition into the stateful operator, some systems treat it as a separated operator (e.g., [2]). In this paper, when the semantics of the stateful operator are not important, we refer only to the window aspect of it as if the window is a separate operator. UniMiCo works the same way no matter whether the window operator is physically merged to the corresponding aggregate/join operator or not.

3. UNIMICO

The key goal of UniMiCo is to avoid transferring state during the migration of a *CQ* containing stateful operators. To achieve this, UniMiCo migrates a *CQ* at a window boundary, meaning that the *originating node* continues processing until it completes the last in-progress window, while the *target node* starts processing from the first tuple of the next window. Given that two consecutive sliding windows overlap, the tuples belonging to the overlap of the two windows are processed by both the originating and the target nodes. This way, the state of the operator is reconstructed at the *target node* so there is no need to migrate it.

We illustrate this strategy in Figure 2. In this example, the sliding window of a stateful operator (e.g., aggregate) has a size of 4 seconds and a slide of 2 seconds, with input rate 1 tuple/second. The number in

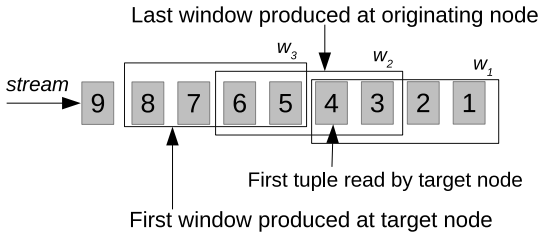

Figure 2: UniMiCo's migration strategy

each stream tuple is its timestamp, which is assumed to monotonically increase over time (i.e. *in-order* processing of tuples). By the time the migration process starts, the most recent window produced is w_1, whose start timestamp is 1. In addition, the first tuple received by the target node after it connects to the stream has a timestamp of 4. UniMiCo determines that (1) the originating node will continue processing until w_2 expires, which happens to be the last window with start timestamp less than 4, and (2) the corresponding *CQ* at the *target node* will start processing tuples with timestamp greater or equal to 5 (w_3).

3.1 Migration timestamp

The migration timestamp marks a *CQ* hand-off from the *originating* to the *target node*. That timestamp is used to synchronize the stop of the last window at the *originating node* and the start of the next window at the *target node*.

Definition The *migration timestamp* is the start timestamp of the last window to be processed at the *originating node*.

In the example in Figure 2, the start timestamp of w_2, which is 3, is the migration timestamp.

3.1.1 Calculating the migration timestamp

The exact calculation of the migration timestamp depends on the implementation details of the window operation. In this section we present how to calculate the migration timestamp on both time-based and tuple-based cases. In all the equations below, s denotes the slide of the window.

Time-based, single-input window: Assuming a time-based window of length l and slide s, let ts_{start} denote the timestamp of the first input tuple the stream source at *target node* was able to read after connecting to the stream. Furthermore, ts_{last_w} is the timestamp of the most recent window processed. The migration timestamp, denoted ts_{mi} is calculated as follows (note that now s is in number of tuples):

$$ts_{mi} = \begin{cases} ts_{last\_w} & \text{if } ts_{start} \leq ts_{last\_w} \\ ts_{start} - \delta & \text{otherwise} \end{cases}$$
$$\text{where } \delta = \begin{cases} s & \text{if } (ts_{start} - ts_{last\_w})\%s = 0 \\ (ts_{start} - ts_{last\_w})\%s & \text{otherwise} \end{cases} \quad (1)$$

Tuple-based, single-input window: For tuple-based windows, the calculation is the same in the case when $ts_{start} \leq ts_{last\_w}$. When $ts_{start} > ts_{last\_w}$, UniMiCo needs to wait until a tuple t comes to the window operator, whose timestamp is equal to or greater than ts_{start}. This way, UniMiCo is aware of the number of tuples with timestamps between ts_{last_w} and ts_{start} (let that number be N). The migration timestamp can be calculated by the following equation:

$$ts_{mi} = timestamp(\delta^{th} \text{ tuple preceding } t)$$
$$\text{where } \delta = \begin{cases} s \text{ if } (N+1)\%s = 0 \\ (N+1)\%s \text{ otherwise} \end{cases} \quad (2)$$

Multiple-input window: The most common example of window-based operator with multiple inputs is a binary join. For time-based windows, Equation 1 can be used, with $ts_{start} = max(ts_{start_i})$, where ts_{start_i} is the timestamp of the first input tuple the stream source i at *target node* was able to read. For tuple-based window, the number of tuples N_i coming between ts_{start_i} and ts_{last_w} is calculated separately for each input i. Afterwards, Equation 2 is applied with $N = max(N_i)$.

Multiple window operators: A *CQ* can have multiple window-based operators with different window specifications (i.e., length and slide), such as a query with an aggregation on top of a join. For these cases, we introduce the concept of the *controlling window operator*.

Definition The *controlling window operator* is the last window operator of the *CQ*. The *controlling window operator* handles the calculation of the migration timestamp, as well as controlling the start and stop of the migrated query at the *target* and *originating node*s.

For simplicity, we assume that the timestamp of an output tuple of a window-based operator is the *earliest* timestamp of input tuples involved in the calculation of that output tuple (we discuss later how this assumption is relaxed). When the aforementioned condition holds, we know that all the original input tuples, contributing to the result produced by the farthest window of start timestamp ts, have timestamps greater than or equal to ts. Therefore, only the farthest window operator (i.e., the *controlling window operator*) in the *CQ* needs to be involved, and the calculation is the same as in the case of single window. Note that the previous assumption is not required for the *controlling window operator*.

Figure 3 shows an example of a *CQ* consisting of two window-based operators: a binary join, whose window has length of 4 seconds and slide 2 seconds, followed by an aggregation, whose window has length of 3 tuples and size of 2 tuples. For each tuple its timestamp is shown on the upper and its join key on the bottom part. For the controlling window, the most recent window being produced is w_{21}, whose start timestamp is 1 (i.e., $ts_{last\_w} = 1$). In addition, assume that out of the two first tuples read from S and T by the target node, the latest timestamp ts_{start} equals 5. In this case, the migration timestamp is calculated as if there is only the *controlling window operator* (i.e, the aggregation) with two inputs S and T. Because the *controlling window operator* is tuple-based, UniMiCo has to wait until tuple

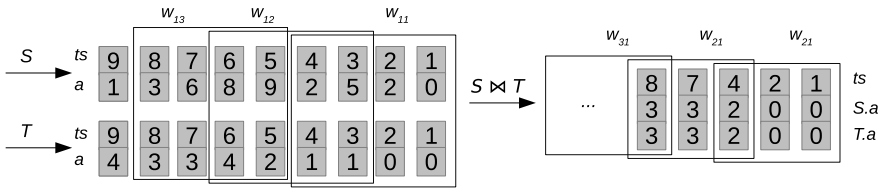

Figure 3: Calculating migration timestamp with two consecutive windows

t of timestamp 7 arrives to know that there are 3 tuples whose timestamps are between 1 and 5, i.e., $N = 3$. Applying the calculation from Equation 2 for the case of tuple-based window, UniMiCo decides that the migration timestamp is that of the tuple preceding t, which is 4. That is, the last window produced at target is w_{21}.

When the previous condition on output tuples' timestamps of preceding window operators does not hold, ts_{start} is measured as the timestamp of the first tuple arriving at the *controlling window operator* on the *target node*. Note than when this condition holds, ts_{start} is the timestamp of the first tuple coming to the source operator, i.e., it can be captured earlier. With the new ts_{start}, all of the above calculations of the migration timestamp are still applicable. Note that in this case if ts_{start} is smaller than ts_{last_w}, there will be some wasted processing at the target to process tuples from source up to the controlling window between ts_{start} and ts_{last_w}. Since migration happens when the target is lightly loaded, it is expected that processing at the *target node* will be at least as fast as that at the *originating node*, hence the wasted processing, if any, would be small.

3.2 Stoping and resuming *CQs*

3.2.1 *Stopping the query at the* originating node

Once the migration timestamp is determined, stopping the query at the *originating node* is relatively straightforward: all operators in the *CQ* continue to process normally until they receive the signal from the *controlling window operator* to deactivate themselves, unless they are shared with other *CQ*s. This happens when the *controlling window operator* has consumed its last window, i.e., the window started with the migration timestamp. Shared operators are not deactivated at the originating node and continue to process normally to serve the remaining queries. Upon stopping, an operator cleans up all its queues.

When the *controlling window operator* is associated with a join, a minor adjustment is needed in order to avoid duplicate outputs between the *originating* and *target node*s. Normally, when there is a match between a tuple t of one input and t' of the other, the join tuple tt' is produced only once, even if both t and t' fall in the overlap of two (or more) consecutive windows. If we start migrating from one of the windows, the join tuple tt' will be produced once at the *originating node*, and again at the *target node*. In the latter case, the production of a duplicate tuple is avoided by suppressing the production of the join result at the *originating node*. Note that when two matching tuples have their timestamps in the window overlap, the previous adjustment is needed only if the join is the last window-based operator in the query. In the event that a join is followed by another window operator, the duplicated intermediate output tt' is needed, as it is an input for the subsequent window at the *target node*.

3.2.2 *Starting the query at* target node

The operators of a migrated *CQ* can be activated at the *target node*, as soon as the migration is initialized. However, full activation is attained by controlling the flow of tuples based on the migration timestamp. That process is different for time- and tuple-based windows, as we describe below.

Time-based *controlling window operator*: If the *CQ* has a time-based *controlling window operator*, the stream source operator(s) calculate(s) the activation timestamp as migration timestamp increased with the slide of the window. Then, the stream source operator discards any input tuples, which carry timestamps less than the activation timestamp. In addition, it starts producing tuples with timestamp equal to or greater than the activation timestamp. With tuples being outputted from the stream source(s), the query is fully activated.

Tuple-based *controlling window operator*: In this case, the stream source operator(s) start(s) producing results from tuples with timestamps greater than the migration timestamp. But, the *controlling window operator* will discard all first $(s - \delta)$ tuples, where s is the slide of the window and δ is calculated from Equation 2 by the *originating node*.

For both types of windows, if the output timestamp of the preceding window-based operator is not the window's start timestamp, the *controlling window operator* has the only authority to decide when to output tuples. Thus, the source operator cannot do any early filtering.

Algorithms 1 and 2 outline the UniMiCo protocol executed at *target* and *originating node*, respectively.

Algorithm 1 UniMiCo protocol at *target node*

1: **BEGIN**
2: Receive(originating_node, *migrate*(Q))
3: **for** $i = 0; i <$ Q.num_streams; $i++$ **do**
4: connect(Q.streams[i])
5: ts_{start}[i] = read(Q.streams[i])
6: **end for**
7: Send(originating_node, ts_{start})
8: Receive(originating_node,ts_{mi})
9: Resume Q based on ts_{mi}
10: **END**

Algorithm 2 UniMiCo protocol at *originating node*

1: **INPUT**: Query Q to be migrated
2: **BEGIN**
3: Send(target_node, *migrate*(Q)
4: Receive(target_node, ts_{start})
5: ts_{mi} = calculate_migration_timestamp
6: Send(target_node, ts_{mi})
7: Finish_processing(Q, ts_{mi})
8: **END**

4. EXPERIMENTAL EVALUATION

While UniMiCo enhances *WRP*'s functionality, at the same time it inherits from *WRP* both its performance advantages and its limitations as stated in Introduction. Since these were experimentally shown in [7], our experimental evaluation focused on showing that UniMiCo migrates *CQ*s with single and multiple stateful operators correctly without impacting their response time.

We implemented and evaluated UniMiCo in a distributed setup of AQSIOS. As mentioned earlier, the window operator in AQSIOS is a separate operator, which receives stream tuples as input, and injects *minus* tuples to the stream to mark the boundary of a window [1]. Windows can have either time-based or tuple-based length, but the window slide is always 1 tuple. Therefore, window-based operators, such as join or aggregation, will rely on those minus tuples to perform their window-based processing. With the separation of the window operator, each input to a join operator can have a window of different length and type. In this paper, we assume that join inputs have windows of the same length and the same type, however, our design can be extended to heterogeneous window environments.

We ran two types of experiments: (1) simple *CQ*s with a single window operator and (2) a complex *CQ* consisting of two window operators. Given our focus on correctness and not performance, window size is not an important parameter in our experiments. We ran each *CQ* twice, under the same settings, and changed only if a migration took place. Then, we compared *CQ*s' outputs and response times around the migration point.

4.1 Simple *CQ* migration (Figures 4 & 5)

We used UniMiCo to migrate a *CQ* with a join operator (Q1), and another one with an aggregate operator

Output with migration

```
[10051579000]:+:location03, 3, 98, location03, 3, 98
[10051589000]:-:location03, 3, 98, location03, 3, 98
[10052632000]:+:location09, 30, 56, location09, 30, 56
[10052642000]:-:location09, 30, 56, location09, 30, 56
----------------------------------------------------------
[10053685000]:+:location16, 2, 77, location16, 2, 77
[10053695000]:-:location16, 2, 77, location16, 2, 77
[10054737000]:+:location20, 43, 21, location20, 43, 21
[10054747000]:-:location20, 43, 21, location20, 43, 21
```

Output without migration

```
[10051579000]:+:location03, 3, 98, location03, 3, 98
[10051589000]:-:location03, 3, 98, location03, 3, 98
[10052632000]:+:location09, 30, 56, location09, 30, 56
[10052642000]:-:location09, 30, 56, location09, 30, 56
[10053685000]:+:location16, 2, 77, location16, 2, 77
[10053695000]:-:location16, 2, 77, location16, 2, 77
[10054737000]:+:location20, 43, 21, location20, 43, 21
[10054747000]:-:location20, 43, 21, location20, 43, 21
```

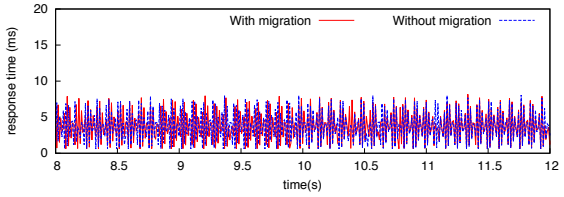

Figure 4: Results and response time of Q1 around the migration point at 10^{th} sec. The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration adds no noticeable delay

(Q2). These two queries written in CQL [2] are:

Q1

```
SELECT *
FROM S [Range 10 seconds],
     T [Range 10 seconds]
WHERE S.1 = T.1;
```

Q2

```
SELECT sum(m)
FROM S [Rows 5];
```

where S and T are input streams. Q1 is associated with time-based windows with size 10 seconds (i.e., [Range 10 seconds]) whereas Q2 is associated with a tuple-based window of size 5 (i.e., [ROWS 5]).

Figures 4 and 5 show the results of Q1 and Q2 around the migration point, respectively. In Figure 4, the top plot is the result under migration, in which the rows above the dashed line are the last output tuples at the *originating node*, and those below are the first output tuples at the *target node*. The middle plot shows the result without migration, which is exactly the same as the concatenation of the two parts of the top plot. Similar observations can be made in Figure 5 for Q2. As one can see, the correctness of the output is maintained by using UniMiCo, and its protocol succeeds in performing the hand-off without losing any data.

The bottom plots in Figures 4 and 5 show the response time of queries Q1 and Q2 two seconds before and after the migration point of about the 10^{th} second. As can be seen in both figures, there are no noticeable "hiccups" in the response time of the queries throughout the

Output with migration

[10054323000]:+:92
[10054323000]:-:46
[10054323000]:+:87
[10054323000]:-:92

[10055771000]:+:102
[10055771000]:-:87
[10055771000]:+:99
[10055771000]:-:102

Output without migration

[10054323000]:+:92
[10054323000]:-:46
[10054323000]:+:87
[10054323000]:-:92
[10055771000]:+:102
[10055771000]:-:87
[10055771000]:+:99
[10055771000]:-:102

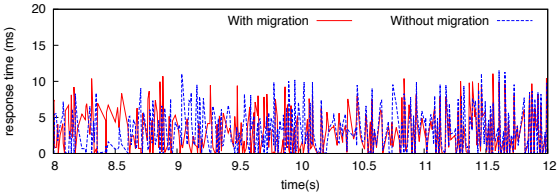

Figure 5: Results and esponse time of Q2 around the migration point at 10^{th} second. The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration adds no noticeable delay

Output with migration

[10022574000]:+:109
[10022574000]:-:104
[10022574000]:+:104
[10022574000]:-:109

[10028529000]:+:107
[10028529000]:-:104
[10028529000]:+:70
[10028529000]:-:107

Output without migration

[10022574000]:+:109
[10022574000]:-:104
[10022574000]:+:104
[10022574000]:-:109
[10028529000]:+:107
[10028529000]:-:104
[10028529000]:+:70
[10028529000]:-:107

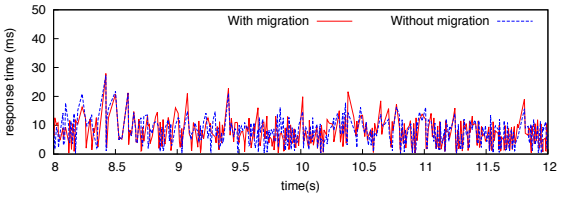

Figure 6: Results and response time of the complex query Q3 around the migration point at 10^{th} second. The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration adds no noticeable delay

migration. For Q1, the average and standard deviation of the response time in this period without migration is 3.751 ms and 3.99 ms, respectively, while under migration they are 3.750 ms and 3.97 ms. For Q2, the corresponding numbers are 3.155 ms and 3.923 ms without migration, and 3.101 ms and 3.836 ms with migration. In both cases, the difference is negligible.

4.2 Complex *CQ* migration (Figure 6)

In this experiment we migrated a more complex query Q3, consisting of a join and an aggregate operator, each using a different window definition as below:

```
Q3: SELECT sum(S.m)
    FROM ISTREAM (SELECT *
                  FROM S [Range 10 seconds],
                       T [Range 10 secon
                  WHERE S.1 = T.1 ) [ROWS 5];
```

In this case, the last window, which is the tuple-based window of size 5 (i.e., [ROWS 5]) associated with the aggregation, plays the role of the *controlling window*.

Figure 6 shows the output tuples and the response time of the query Q3 around the migration point, compared with the run when there is no migration. Similar to the cases of the simple queries, the query output is preserved and the cost of migration is not noticeable. The average and standard deviation of the response time without migration are 6.568 ms and 6.133 ms respectively, while those with migration are 6.658 ms and 6.217 ms.

5. CONCLUSIONS

We presented UniMiCo, a general migration protocol for *CQ*s, used in distributed *DSMS*s. UniMiCo achieves migration without the need to transfer state or stop processing input tuples during *CQ* hand-off. UniMiCo is more general than previous work by being applicable to *CQ*s with different window semantics and with multiple stateful operations. Our experimental evaluation demonstrated UniMiCo's feasibility, by implementing it in a full-fledged prototype *DSMS* (AQSIOS). Our experiments showed its correctness and that it does not incur any noticeable delays in the *CQ*'s response time.

6. REFERENCES

[1] A. Arasu et al. Stream: The stanford data stream management system. Technical report, Stanford InfoLab, 2004.
[2] B. Babcock, S. Babu, et al. Models and issues in data stream systems. In *PODS '02*.
[3] R. Castro Fernandez et al. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD '13*.
[4] P. K. Chrysanthis. AQSIOS - Next Generation Data Stream Management System. *CONET Newsletter*, 2010.
[5] S. Guirguis et al. Optimized processing of multiple aggregate continuous queries. In *CIKM '11*.
[6] S. Guirguis et al. Three-level processing of multiple aggregate continuous queries. In *ICDE'12*.
[7] V. Gulisano et al. Streamcloud: An elastic and scalable data streaming system. *IEEE TPDS*, 2012.
[8] H. V. Jagadish et al. Big data and its technical challenges. *CACM*, Jul 2014.
[9] N. R. Katsipoulakis et al. Ce-storm: Confidential elastic processing of data streams. In *SIGMOD*, 2015.
[10] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD '15*.
[11] T. N. Pham, P. K. Chrysanthis, and A. Labrinidis. Avoiding class warfare: Managing continuous queries with differentiated classes of service. *VLDBJ*, 2016.
[12] M. A. Shah et al. Flux: an adaptive partitioning operator for continuous query systems. In *ICDE'03*.
[13] M. Sharaf et al. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM TODS*, 2008.