Efficient Invocation of Transaction Sequences Triggered by Data Streams

Masafumi Oyamada, Hideyuki Kawashima and Hiroyuki Kitagawa
Graduate School of Systems and Information Engineering, University of Tsukuba
masa@kde.cs.tsukuba.ac.jp, {kawasima, kitagawa}@cs.tsukuba.ac.jp

***Abstract*—Streaming media such as GPS, wireless sensor nodes, RFID and electric commerce are spreading over our lives. To process data streams from such streaming media, DSMS has been developed and widely used. While DBMSs are rarely used to process data streams directly, there is a growing need to integrate DSMS and DBMS, which means to use DBMS as a source of information related to data streams and to use DBMS to archive data streams. In such integration of DSMS and DBMS, transactions are triggered by data streams. Since arrival rate of data streams can be extremely high, it is necessary to invoke those transaction sequences efficiently to keep performance of DSMS.**

In this paper, we model invocation of transaction sequences and propose three schemes named *Synchronous Transaction Sequence Invocation (STSI)*, *Asynchronous Transaction Sequence Invocation (ATSI)* and *Order-Preserving Asynchronous Transaction Sequence Invocation (OPATSI)*. We built a prototype system which implements those schemes and conducted experiments to evaluate their performance. Our primary experiments showed that we experienced an order of magnitude higher throughput in transaction processing by using ATSI compared with STSI while breaking the order of transaction results. The results of experiments also showed that throughput of OPATSI is comparable to that of the ATSI, though it preserves the order of transaction results.

I. Introduction

Streaming media such as GPS, wireless sensor nodes, RFID and electric commerce are spreading over our lives. These media generate data frequently and continuously. Such a kind of data are referred to as data streams. Because of its characteristics, data streams are inclined to be processed by data streaming processing systems (DSMSs) rather than database management systems (DBMSs) which are traditionally used to handle huge amount of data.

While DBMSs are rarely used to process data streams directly, there is a growing need to integrate DSMS and DBMS [14]. An example is integrating an outlier data stream object with detailed information which is stored in DBMS. When an outlier is detected, DSMS obtains detailed information about the detected outlier from DBMS, combines an outlier with the detailed information, and outputs the pair. Another example is data stream archiving. To archive data streams, DSMS requests DBMS to store tuples from a data stream. Since DSMS itself does not manage stored data, such an integration of DSMS and DBMS is essentially important.

As seen above, transactions of DBMS are triggered by the arrival of data streams in the context of integration of DSMS and DBMS. Since arrival rate of data streams could be extremely high, it is necessary to invoke these transactions efficiently to keep performance of DSMS.

In this paper, we model invocation of transaction sequences and propose three schemes named *Synchronous Transaction Sequence Invocation (STSI)*, *Asynchronous Transaction Sequence Invocation (ATSI)* and *Order-Preserving Asynchronous Transaction Sequence Invocation (OPATSI)* which invoke transaction sequences triggered by data streams. We build a prototype system which implements those schemes and conduct experiments to evaluate their utility. As for DBMS, we choose VoltDB which is a distributed in-memory DBMS. We show the performance of each scheme by experiments.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 proposes three schemes to invoke transaction sequences. Section 4 describes analytical models for the schemes. Section 5 describes experiments about performance. Finally Section 6 concludes this paper and indicate future direction.

II. Related work

Integration of DSMS and DBMS is one of the important topics on stream processing. In this section, we show several approaches which attempt to accomplish the integration by grouping them into (1) embedding DBM into DSMS, (2) extending DBMS as DSMS and (3) using functions of external DBMS from DSMS.

A. Embedding DBM into DSMS

Aurora [1] and its successor Borealis [3] use a popular database manager (DBM) Berkeley DB [2] as a relational storage. In these systems, incoming tuples from data streams can trigger queries for stored data in Berkeley DB. Their commercialization StreamBase [12] can also trigger queries in a similar fashion.

Some algorithms such as Mesh Join [10], Partition-Based Join [6] and Semi-Streaming Index Join [4] are proposed. They allow DBM to join data streams and relational data stored in a DBM.

B. Extending DBMS as DSMS

TelegraphCQ [15] is based on PostgreSQL [11] which is a well known traditional DBMS. TelegraphCQ itself does not aim to integrate DSMS and DBMS. On the other hand, its commercialization Truviso [16] can handle data streams and relational data in an integrative way.

Botan developed a DSMS named MaxStream [5] by extending a MaxDB [9] which is a traditional DBMS. Their ultimate goal is the integration of heterogeneous DSMSs, DBMSs and any other storages, which includes the integration of DSMS and DBMS.

C. Using external DBMS functions from DSMS

StreamSpinner [13] is able to archive tuples from data streams with external DBMSs [18] while reducing the amount of data to be written by sharing the result of similar queries [19].

Coral8 [7] achieves integration of DBMS and DSMS by allowing users to use functions of DB2, not limited to archiving.

III. Invocation of transaction sequences

To integrate a DBMS and a DSMS, we cannot avoid invocation of transaction sequences triggered by data streams. In this section, we propose three schemes named synchronous transaction sequence invocation (STSI), asynchronous transaction sequence invocation (ATSI) and order-preserving asynchronous transaction sequence invocation (OPATSI) for invocation of transaction sequences.

A. Synchronous Transaction Sequence Invocation (STSI)

Synchronous transaction sequence invocation (STSI) is a naïve scheme which invokes transaction sequences triggered by data streams. Algorithm 1 shows the process of STSI. In STSI, transactions triggered by tuples of data streams are invoked one by one. DSMS invokes a transaction when a tuple arrives, and blocks processes for the following tuple while waiting for the result of the transaction. This scheme guarantees the order of tuple-transaction pairs which arrives at client, while degrading the throughput of the sequential transactions by blocking processes.

Algorithm 1 *STSI*

loop
  $tuple \Leftarrow GetNextTuple()$
  $trx \Leftarrow InvokeTransaction(tuple)$
  $result \Leftarrow WaitForTransactionResult(trx)$
  $SendResult(tuple, result)$
end loop

B. Asynchronous Transaction Sequence Invocation (ATSI)

STSI degrades throughput due to the process blocking when the rate of a data stream is very high. We propose asynchronous transaction sequence invocation (ATSI) which eliminates process blocking and achieves higher throughput than STSI. This scheme can be said as an application of well known Thread-Per-Message pattern in multi-thread programming.

Algorithm 2 shows the main thread process of ATSI. Main thread invokes a transaction triggered by a tuple from a data stream, then it does not wait for its result. Instead, main thread starts waiting thread and delegates the task to the waiting thread. When transaction result is returned from a DBMS in each waiting thread, tuple-transaction pair is sent to a client as shown in Algorithm 3. In this scheme, a DBMS can process multiple transactions simultaneously. Since execution time of each transaction is not fixed, the order of outgoing results may not equal to the order of incoming tuples from a data stream. On the other hand, this scheme could be more efficient than STSI due to the elimination of process blocking.

TABLE I
Functions used in *STSI*

Name	Description
$GetNextTuple()$	Get next tuple from data stream.
$InvokeTransaction(tuple)$	Invoke transaction for $tuple$ and returns transaction ID.
$WaitForTransactionResult(trx)$	Wait for result of transaction which ID is trx.
$SendResult(tuple, result)$	Send tuple and transaction result to client.

Algorithm 2 *ATSI*: Main thread

loop
  $tuple \Leftarrow GetNextTuple()$
  $trx \Leftarrow InvokeTransaction(tuple)$
  $StartWaitingThread(tuple, trx)$
end loop

Algorithm 3 *ATSI*: Waiting thread

$result \Leftarrow WaitForTransactionResult(trx)$
$SendResult(tuple, result)$

TABLE II
Functions additionally used in *ATSI*

Name	Description
$StartWaitingThread(tuple, trx)$	Start new thread which waits for transaction result.

C. Order-Preserving Asynchronous Transaction Sequence Invocation (OPATSI)

In the stream processing, the order of tuples from data streams sometimes has significant meanings. An example is stock market prediction which involves the monitoring of stock price variation. If we adopt stream warehouse which integrates a DSMS and a DBMS, and applied ATSI to invoke transaction sequences, the order of outgoing results may be out of ordered and the result of the monitoring of stock price variations may be incorrect. Therefore, in such stream

processing where order of tuples has significant meanings, it is required to guarantee the order of outgoing results.

We propose order-preserving asynchronous transaction sequence invocation (OPATSI) which guarantees the order of outgoing results by using priority queue. Algorithm 4 shows the process of the main thread of OPATSI. In OPATSI, we attach an arrival number to each incoming tuple. Then, we invoke transaction for the tuple asynchronously and wait for its result in separate thread. When the result is returned from DBMS, we check whether corresponding tuple arrival number is equal to N_{next} or not. If arrival number is equal to N_{next}, the pair of result and corresponding tuple is sent to the client, and repeat (1) N_{next} is incremented and (2) send pair of which arrival number is N_{next} in the queue to the client, while it is possible as described in Algorithm 5. Otherwise, the pair is pushed into the queue.

Algorithm 4 *OPATSI*: Main thread

```
global queue ⇐ empty
global N_next ⇐ 0
N_current ⇐ 0
loop
  tuple ⇐ GetNextTuple()
  trx ⇐ InvokeTransaction(tuple)
  StartOPWaitingThread(tuple, trx, N_current)
  N_current ⇐ N_current + 1
end loop
```

Algorithm 5 *OPATSI*: Waiting thread

```
result ⇐ WaitForTransactionResult(trx)
Enqueue(queue, Tuple(tuple, result, N_current))
while QueueHasTupleNumber(queue, N_next) do
  t, r ⇐ DequeueTuple(queue, N_next)
  N_next ⇐ N_next + 1
  SendResult(t, r)
end while
```

TABLE III
FUNCTIONS ADDITIONALLY USED IN ORDER PRESERVING ASYNCHRONOUS INVOCATION

Name	Description
StartOPWaitingThread(tuple, trx, N)	Start new thread which waits for transaction result and control the order of results.
Enqueue(queue, elem)	Put $elem$ into the $queue$.
QueueHasTupleNumber(queue, N)	Returns $True$ if the $queue$ has the tuple which tuple-number is N.
DequeueTuple(queue, N)	Remove tuple from the $queue$ and returns it if the tuple tuple-number is N.

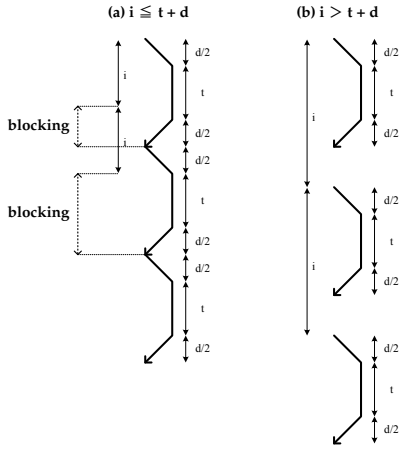

Fig. 1. Graphical model of synchronous invocation

IV. Analytical Model

In this section, we model invocation of transactions by grouping them into synchronous invocation (STSI) and asynchronous invocation (ATSI and OPATSI). In those models, t means the time to complete a transaction in DBMS, d means the round-trip time between DSMS and DBMS and i means the arrival interval of data streams.

A. Model of Synchronous Invocation

Figure 1 represents synchronous invocation of transaction sequences graphically. In synchronous invocation, DSMS cannot process following tuples from a data stream while waiting for the result of current transaction. Such blocking of process affects to the time to complete all transactions only if $i \leq t+d$. Thus, time to invoke N transactions synchronously can be represented as Equation (1).

$$T_{sync} = \begin{cases} N(t+d) & (i \leq t+d) \\ (N-1)i + (t+d) & (i > t+d) \end{cases} \quad (1)$$

B. Model of Asynchronous Invocation

Figure 2 represents asynchronous invocation of transaction sequences graphically. In asynchronous invocation, the result of transactions are waited in separate thread. Therefore, there is no blocking and time to complete a transaction does not affect time to complete all transactions. Therefore, time to invoke N transactions asynchronously can be represented as Equation (2).

$$T_{async} = (N-1)i + (t+d) \quad (2)$$

C. Performance Comparison between Synchronous Invocation and Asynchronous Invocation

By following these models, temporal difference between T_{sync} and T_{async} can be represented as Equation (3).

$$T_{sync} - T_{async} = \begin{cases} (N-1)(t+d-i) & (i \leq t+d) \\ 0 & (i > t+d) \end{cases} \quad (3)$$

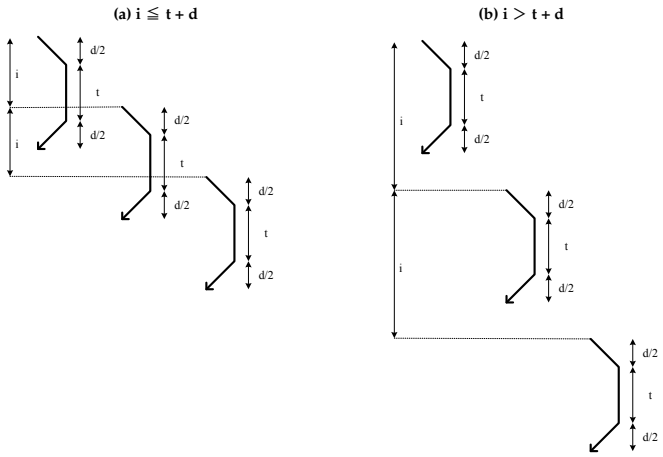

Fig. 2. Graphical model of asynchronous invocation

Equation (3) indicates that asynchronous invocation performs better than synchronous invocation when arrival interval of tuples is shorter than the time to complete a transaction. Therefore, asynchronous invocation is especially effective when arrival rate of data streams is extremely high.

This model can be used to switch invocation schemes adaptively. When i becomes greater than $t+d$, temporal difference between synchronous invocation and asynchronous invocation will vanish, which allows us to avoid needless asynchronous invocation. We are planning to implement such adaptive switching of invocation schemes and measure its feasibility in the future.

V. Evaluation

To evaluate the performance of proposed schemes, we built a prototype system which invokes transaction triggered by incoming tuples from a data stream, and conducted experiments which measure throughput of proposed schemes. Figure 3 shows an overview of the prototype system we built. We used VoltDB [17] as a transaction manager of the prototype system. We expect high throughput on prototype system due to the features of VoltDB[1]. A cluster is consisted of three nodes of which specs are described in Table IV and Table V. In each node, VoltDB runs three threads to execute transactions.

TABLE IV
Spec of machine 1 and 2

CPU Name	Intel(R) Xeon(R) CPU E5503
Clock frequency	2GHz
Number of cores	2
Memory capacity	4GB

[1]VoltDB, the commercialization of H-Store [8], is a distributed in-memory RDBMS which performs extremely high throughput in OLTP by enforcing the use of stored procedures and eliminating collisions in concurrency control.

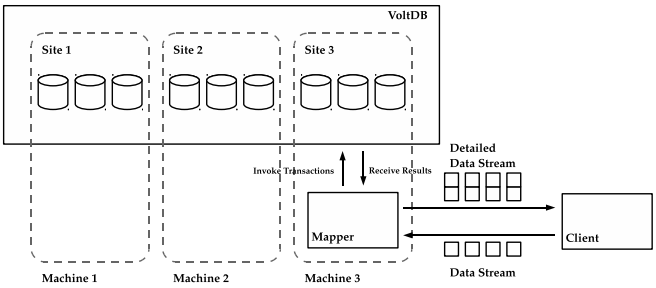

Fig. 3. Overview of the prototype system

A. Effective Queue Implementation

In OPATSI proposed in Section III-C, queue management can be a bottleneck. In this subsection, we conduct simulations to investigate the effective queue implementation. The queue used in OPATSI has following manipulations.

- EnqueueWithArrivalNumber(tuple, n): Put $tuple$ into the queue with arrival number n
- HasArrivalNumber(n): Check if queue has the tuple which has arrival number n
- DequeueWithArrivalNumber(n): Remove the tuple which has arrival number n from queue and returns the tuple

Such queue can be directly implemented by using sorted doubly-linked list. On the other hand, it is notable that the arrival number of the tuple to be dequeued is always N_{next}. This characteristic allows us to implement this queue by using hash table. Therefore, (1) sorted doubly-linked list (search from head), (2) sorted doubly-linked list (search from tail) and (3) hash table can be used as queue implementation for OPATSI.

Time complexities of such implementations are shown in Table VI. In the sorted doubly-linked list, we should keep elements sorted in ascending order of arrival number to make the tuple which has arrival number N_{next} is always on the head. Therefore, in the sorted doubly-linked list, enqueue needs linear-search of which time complexity is $O(N)$. On the other hand, dequeue is done by one step so that the time complexity is $O(1)$. In case of hash table, enqueue and dequeue are done by $O(1)$. Therefore, from the viewpoint of time complexity, it can be said that we should use hash table to implement the queue for OPATSI.

To check if the hash table is efficient than other queue implementations for real, we conducted experiments by simulations. Simulation program is implemented in Java, and *java.util.LinkedList* is used for doubly-linked list and

TABLE V
Spec of machine 3

CPU Name	Intel(R) Core(TM)2 Quad CPU Q9400
Clock frequency	2.66GHz
Number of cores	4
Memory capacity	4GB

java.util.HashMap is used for hash table. We measured the throughput from the time to complete N transactions in the prototype system. In this simulation, we made *GetNextTuple()* to return one tuple without any delay and emulate *WaitForTransactionResult()* behavior by sleeping thread using *Thread.sleep()*. This sleeping time is corresponds to $t+d$ described in IV. By changing N from 100 to 20000 by 100, we measured throughput total of 200 times for each queue implementation. We also measured throughput for ATSI, which performs transaction sequences asynchronously without queue management, for comparison.

We added two parameters to this simulation. First, we varied execution time of transactions[2] from 10 to 11 msec (small variation) and from 10 to 800 msec (big variation) to simulate various transaction types used in the real world. Second, we controlled transactions triggered by the tuple which arrived foremost to complete lastly. This control swells queue size to N and rises overhead of queue management, which amplifies performance differences between queue implementations.

Figure 4 shows the throughput transition when transaction time varies 10 from 11 msec i.e., there is the same kind of transactions. We can see big differences between queue implementations. In Figure 4, sorted doubly-linked list (search from head) performs poor throughput as N increases. Other queue implementations, that is, sorted doubly-linked list (search from tail) and hash table perform pretty good throughputs while hash table performs better throughput than sorted doubly-linked list (search from tail) averagely.

Figure 5 shows the throughput transition when transaction time varies 10 from 800 msec i.e., there is a variety of transactions. Differences between queue implementations are smaller than that in Figure 4, while hash table performs higher throughput than other queue implementations.

To summarize the result of simulations, hash table performs the best throughput of queue implementations for OPATSI. Therefore, we decided to use hash table to implement the queue for OPATSI. We are planning to compare our queue implementation with other priority queue implementations e.g, *java.util.concurrent.PriorityBlockingQueue* in the future.

B. Throughput of Proposed Schemes

To evaluate the performance of our proposed schemes, we measured the throughput of each schemes in the real prototype system. In this experiment, we used the prototype system described above and measured throughput from the time to complete N transactions triggered by data streams. A transaction used in this experiments is a simple query shown in Figure 6. We varied arrival intervals of tuples from 0 (this means that when you request a tuple from a data stream, there is always a tuple without wait) to 30 msec and measured throughput for each arrival interval. The number of stored tuples in table `INFO_DB` is changed for each experiment: 1000 for first and second experiments, 20000 for third and fourth experiments.

TABLE VI
TIME COMPLEXITIES OF QUEUE IMPLEMENTATIONS

Name	Enqueue	Dequeue
Sorted doubly-linked list	$O(N)$	$O(1)$
Hash table	$O(1)$	$O(1)$

[2]Transaction is simulated by sleeping the thread as mentioned above.

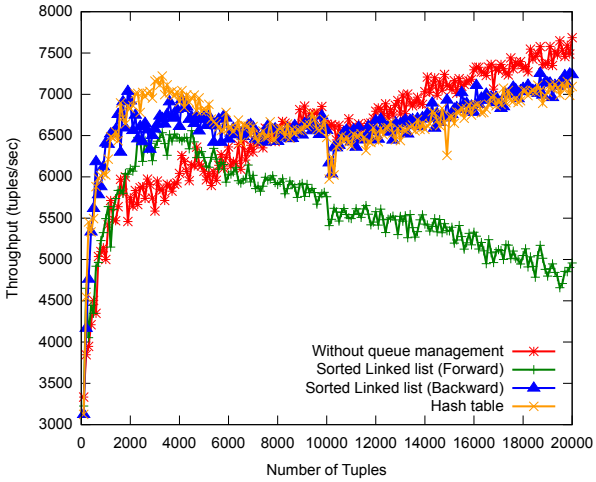

Fig. 4. Throughput transition when transaction time varies 10 from 11 msec

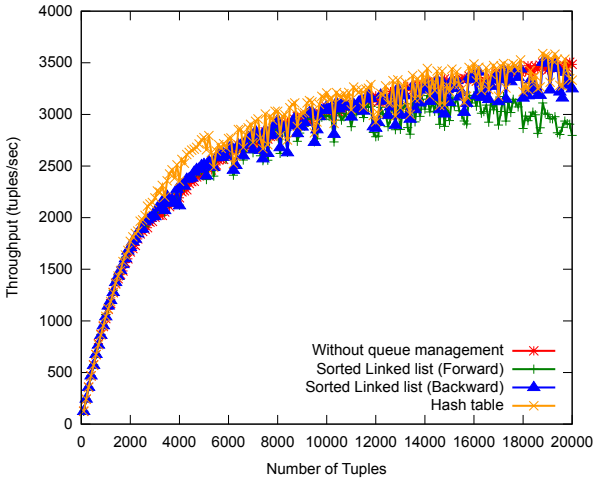

Fig. 5. Throughput transition when transaction time varies 10 from 800 msec

```
SELECT *
FROM   INFO_DB
WHERE  INFO_DB.ID = TUPLE.ID
```

Fig. 6. Query to DBMS

1) Fetching data from 1000 tuples: Figure 7 shows throughput transition for proposed schemes, in case of selecting 1000 tuples from 1000 tuples. In this case, time to complete a transaction which selects one tuple from 1000

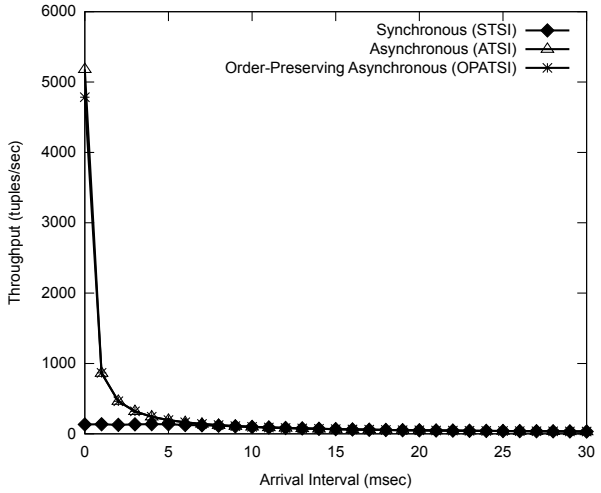

Fig. 7. Throughput transition of transaction which selects 1000 tuples from 1000 tuples

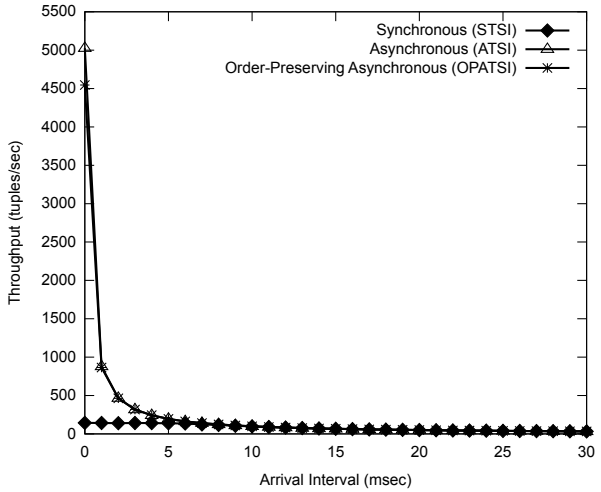

Fig. 8. Throughput transition of transaction which selects 1000 tuples from 20000 tuples

tuples in VoltDB is between 5 msec and 6 msec. We can see that the throughput differences diminishes by arrival interval increases and disappears when arrival interval becomes about 6 msec, which is consistent with the model described in IV. When arrival interval of tuples is extremely small, ATSI performs 35 times higher throughput than STSI by breaking the order of transaction results. OPATSI performs comparable throughput to naïve ATSI while preserving the order of transaction results.

2) Fetching data from 20000 tuples: Next, we increased number of tuples stored in VoltDB to 20000, to check how throughput of prototype system is affected by the data size stored in database. The result of selections for 1000 tuples is shown in Figure 8. As in the case of fetching data from 1000 tuples, we can see that ATSI performs better throughput than STSI and OPATSI.

3) Summary of experiments on prototype system: Summarizing the experiments on the prototype system, when arrival rate of data streams is extremely high, we can experience an order of magnitude higher throughput in transaction processing by using ATSI rather than STSI, while breaking the order of transaction results. The results of experiments also showed that throughput of OPATSI is comparable to that of the ATSI, though it preserves the order of transaction results. While our experiments illustrated the characteristics of proposed schemes, further experiments are needed to investigate detailed behaviors. We are planning to arrange large data and conduct experiments which include invocation of write queries in the future.

VI. Conclusions and Future Work

In this paper, we proposed three schemes which invoke transaction sequences triggered by data streams named synchronous transaction sequence invocation (STSI), asynchronous transaction sequence invocation (ATSI) and order-preserving asynchronous transaction sequence invocation (OPATSI). Our primary experiments showed that by using ATSI, we can experience an order of magnitude higher throughput in transaction processing compared to STSI while breaking the order of transaction results. The result of experiments also showed that throughput of OPATSI is comparable to that of ATSI, though it preserves the order of transactions.

For future work, we plan to implement the adaptive switching of invocation schemes which allows us to avoid needless asynchronous invocation as mentioned in Section IV-C. In addition, we plan to execute complex queries including heavy multi-way join queries so that we can observe the behavior of OPATSI more in detail.

Acknowledgement

This work is partially supported by KAKENHI(#22700090, #21240005).

References

[1] Aurora. http://www.cs.brown.edu/research/aurora/.
[2] BerkeleyDB. http://oracle.com/technetwork/database/berkeleydb/.
[3] Borealis. http://www.cs.brown.edu/research/borealis/public/.
[4] M. Bornea *et al*. Semi-Streamed Index Join for Near-Real Time Execution of ETL Transformations. In *ICDE*, 2011.
[5] I. Botan *et al*. Design and Implementation of the MaxStream Federated Stream Processing Architecture. In *Technical Report TR-632*, 2009.
[6] A. Chakraborty, A. Singh. A Partition-based Approach to Support Streaming Updates over Persistent Data in an Active Data Warehous. In *IPDPS*, 2009.
[7] Coral8. http://coral8.com/.
[8] R. Kallman *et al*. H-store: a high-performance, distributed main memory transaction processing system. In *PVLDB*, 2008.
[9] MaxDB. http://www.sdn.sap.com/irj/sdn/maxdb.
[10] N. Polyzotis *et al*. Supporting Streaming Updates in an Active Data Warehouse. In *ICDE*, 2007.
[11] PostgreSQL. http://www.postgresql.org/.
[12] StreamBase. http://www.streambase.com/.
[13] StreamSpinner. http://www.streamspinner.org/.
[14] N. Tatbul. Streaming data integration: Challenges and opportunities. In *NTII*, 2010.
[15] TelegraphCQ. http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/.
[16] Truviso. http://www.truviso.com/.
[17] VoltDB. http://voltdb.com/.
[18] Y. Watanabe *et al*. Integrating a Stream Processing Engine and Databases for Persistent Streaming Data Management. In *Proc. DEXA, LNCS 4653*, pages 414–423, 2007.
[19] S. Yamada *et al*. An Optimization Method for Multiple Persistency Requirements on Stream Management System. In *DEWS*, 2007.