Query Result Caching for Multiple Event-Driven Continuous Queries

Yousuke Watanabe [a] Hiroyuki Kitagawa [b,c]

[a] *Global Scientific Information and Computing Center, Tokyo Institute of Technology, Oh-okayama, Meguro-ku, Tokyo, 152-8552 Japan*

[b] *Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, Tennohdai, Tsukuba, Ibaraki, 305-8573 Japan*

[c] *Center for Computational Sciences, University of Tsukuba, Tennohdai, Tsukuba, Ibaraki, 305-8573 Japan*

Abstract

With the increasing demands for advanced use of streaming data, efficient execution of continuous queries is an important research issue. This paper focuses on event-driven continuous queries that are activated by foreign events such as data arrival and the progression of time. Existing approaches to multiple continuous query optimization decide the optimal query plan by extracting common subexpressions from the given queries. Event-driven queries containing the common subexpressions may produce many common intermediate results when they are activated within a small interval, but may produce only disjoint data when activated at completely different timings.

This paper proposes an efficient data stream processing scheme for multiple event-driven continuous queries. In the proposed approach, we introduce query result caching to achieve a flexible way to share common operators among queries activated

by unpredictable events. When a query is activated, an intermediate result generated for the query is stored into the cache area if it is expected to be reused by other queries. When other queries including the same operator are activated, they reuse the cached result if the cache includes reusable data. Efficiency of the proposed scheme is validated by intensive experimental evaluations.

Key words: data stream, continuous query, optimization, cache

---

1 Introduction

Beyond traditional information sources such as files and databases, a new type of information source called data streams has been gaining popularity. These streams are used to provide up-to-date information such as online news, weather forecasts, stock prices, and sensor data, in which new information is continuously generated over time. Continuous queries [4,1,6,8,15] are often used to satisfy users' information needs over data streams. A continuous query is continuously executed and generates query results using arriving data items.

Although there are many variants of continuous queries, they are generally triggered by events such as data arrival and time progress [8,9,13,18]. In advanced data stream processing environments, query specifications usually involve specifications for triggering events and temporary data selection conditions like windows. Explicit event specifications are required by users who want to get query results at the times they request, not the original arrival times. For example, in network monitoring applications, we want to monitor not only

Email addresses: `watanabe@de.cs.titech.ac.jp` (Yousuke Watanabe), `kitagawa@cs.tsukuba.ac.jp` (Hiroyuki Kitagawa).

each packet, but also periodical (e.g. hourly or minutely) network statistics. Moreover, when a suspicious access is detected on a server, we would like to analyze recent packets sent to the server. In these cases, we require continuous queries with windows and event specifications using timers and foreign events.

Another advantage of event specifications is that systems can delay processing arrival data until specified events occur. For queries triggered by rare events, huge data become obsolete while waiting for the events; we can then remove them from input queues without any query processing. This contributes to reduce system load and to improve total performance. Without event specifications, it implies that arrival data have to be evaluated immediately. Thus the systems require more CPU power, because it has to work more frequently.

Data stream processing systems must process a voluminous number of these continuous queries. Efficient methods that can process a huge amount of user requirements are especially needed. Multiple query optimization techniques [16,17] are usually used to achieve efficient query processing. The basic idea is to extract operators that commonly appear in multiple queries and to derive query plans in which intermediate results of these common operators are shared among the queries. With multiple query optimization, we can improve total performance, because optimization can contribute to eliminating duplicate query processing.

A difficult problem arises in applying multiple query optimization for event-driven continuous queries. Operators of queries with different triggers are executed at different timings. Queries having the same operators may share many intermediate results when they are activated at the same or close instants, but may involve only disjoint data when activated at completely different instants.

Query execution timing affects amounts of data actually shared by queries. It is a key to deciding an efficient query execution plan as well as common subexpressions. In most existing optimization methods, two naive strategies are used. The first is extracting common operators from all given queries by ignoring event specifications [1,2,6,8,9,15]. The second groups queries guaranteed to be activated by the same events; it then extracts common operators in each group of queries [13,18]. However, neither strategy is optimal and both are inefficient except for special cases. For instance, all queries are triggered by the same events.

Our proposed strategy constructs groups of continuous queries activated within a small interval. Based on the strategy, common operators of queries activated at close instants are shared, and ones activated at completely different instants are separately processed. However, foreign events triggering queries are generally unpredictable and not fixed. It is hard to determine such groups exactly, and they may change over time. To achieve the goal, we need a flexible and adaptive query processing framework that can adapt to uncertainness of query execution patterns.

This paper proposes a new query processing scheme for event-driven continuous queries taking execution patterns into account. The originality of our approach is twofold: (i) cache mechanism to share intermediate results of operators triggered at different time instants, and (ii) the adaptive method on query execution patterns. We introduce query result caching to continuous query processing to implement the proposed strategy. When all queries are triggered by the same events, the proposed system behaves like existing methods. Otherwise, it holds intermediate results in caches and reuses them when other queries are evaluated. More concretely, our optimizer first creates query

plans by extracting common operators based on query specifications; it then starts to process query plans. It dynamically identifies queries whose common operators are expected to produce reusable results, and determines which queries should use the caches. These decisions are based on cost estimates and scanning cache gains.

The remainder of this paper consists of the following parts: Section 2 introduces the event-driven continuous query considered in this paper. Section 3 discusses problems with existing approaches and the main contributions of this paper. We explain the proposed query processing scheme with query result caching in Section 4. Section 5 proposes criteria to decide who produces/reuses caches. Experiment results are shown in Section 6. Related work is summarized in Section 7. Finally, Section 8 presents conclusions and introduces future research issues.

2 Event-driven continuous query

In this paper, we model data streams as unbound relations. We assume that a delivery unit from the data stream is a tuple belonging to a relation and has an embedded attribute TS to record its arrival timestamp.

We introduce the event-driven continuous query. An event-driven continuous query is continuously activated by some events, then operators contained in the query are evaluated. In each evaluation, differential results from previous evaluations are returned, usually generated by new tuples.

A query specification consists of MASTER–SELECT–FROM–WHERE clauses (Figure 1). MASTER clause gives *master information sources*. We can spec-

```
MASTER   master_source, ...
SELECT   attribute_1, ...
FROM     source_1{ [windowsize_1] }, ...
WHERE    conditions
```

Fig. 1. Syntax of continuous query

ify arbitrary multiple streams as master information sources. Data arrivals from any of the master information sources trigger activations of the query. SELECT–FROM–WHERE clauses are almost the same as SQL except for window specifications. Windows are specified by "[...]" after each source description in a FROM clause. We consider time-based windows [4] in this paper, but our optimization method is also applicable to tuple-based windows. Users can specify different window sizes for each data stream. If window sizes are not given, they are regarded as having infinite windows. When a new tuple has arrived from a master information source, the continuous query is triggered and corresponding operators process tuples arriving within the window. We assume not only streams but also traditional relations in a FROM clause. In this paper, we focus on selection, projection, and join operators. For simplicity, we assume predicates in a WHERE clause are connected by AND. We can easily extend our method to handle WHERE clauses including OR.

Figure 2 shows an example of a continuous query: "Every noon, deliver tuples integrating the streams S and R with tuples that have arrived within the last 8 hours." where master "Clock_12" is a clock stream that delivers a constant alarm every noon. Clock streams are used to execute queries periodically and are provided by data stream processing systems. Here, "hour" is a constant value expressing one hour.

```
MASTER   Clock_12
SELECT   *
FROM     S [8 hour],  R [8 hour]
WHERE    S.A = R.A
```

Fig. 2. Example query: Q1

3 Problems and our proposed optimization strategy

Multiple query optimization methods generally derive query plans in which intermediate results of common operators are shared among queries. First, we explain problems with applying multiple query optimization to event-driven continuous queries. We then introduce naive optimization strategies. Finally, we describe contributions of our approach.

3.1 Impact of execution timings in multiple query optimization

To illustrate characteristics of optimizing multiple event-driven queries, we show an example. Suppose we have the three queries $Q1, Q2, Q3$ to be executed at different instants Clock_12, Clock_13, Clock_0 in Figures 2, 3 and 4. They have common join operators on Streams S and R. For simplicity, we assume all queries have the same size window for each stream. Figure 5 depicts windows of three queries executed at noon, 13 o'clock and midnight. The vertical (or horizontal) axis expresses arrival timestamps of data delivered from S (or R). $8hours \times 8hours$ rectangles correspond to data needed to produce the results of the queries. For queries executed within a small interval, like $Q1$ and $Q2$, the largest part of their windows overlap (the gray rectangle in Figure 5). We expect that much of the data generated by $Q1$ can be reused by $Q2$. However, $Q1$ does not need any results of $Q2$. On the other hand, for queries executed at

```
MASTER   Clock_13
SELECT   *
FROM     S [8 hour],  R [8 hour]
WHERE    S.A = R.A
```

Fig. 3. Example query: Q2

```
MASTER   Clock_0
SELECT   *
FROM     S [8 hour],  R [8 hour]
WHERE    S.A = R.A
```

Fig. 4. Example query: Q3

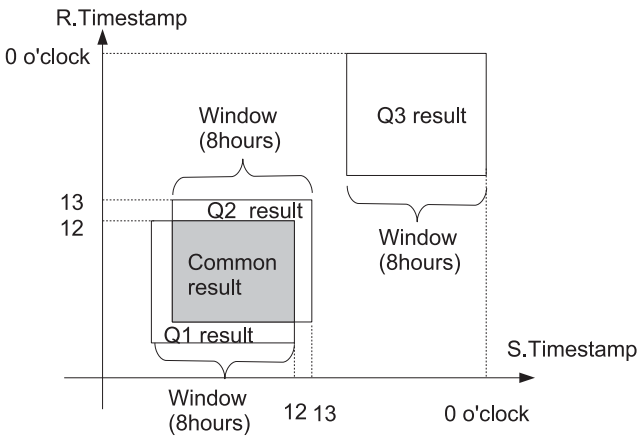

Fig. 5. Overlap of windows and common results

completely different instants, like $Q2$ and $Q3$, there is no intersection of their rectangles. Data generated by $Q2$ become obsolete while waiting for activation of $Q3$.

To summarize, we encounter the following problems when we apply the multiple query optimization method to event-driven continuous queries.

- *Dependency on execution timings:* Queries having the same operators may share many intermediate results when their windows have large overlaps. However, they may involve only disjoint data when their windows do not

overlap.

- *Asymmetric relationships:* Whether or not results of a query are reusable also depends on execution sequences of other queries. Even if $Q2$ can reuse the results of $Q1$, it does not imply that $Q1$ reuses the results of $Q2$.
- *Uncertainness of event occurrences:* Except for queries triggered by constant timers, we cannot exactly know execution patterns of continuous queries triggered by foreign events.

Naive approaches shown in Section 3.2 do not take into account the above three points.

3.2 Naive approaches for sharing query results

We introduce two naive approaches for sharing results in multiple query optimization methods.

- **ALL**: Sharing all common operators among queries by ignoring execution timings
- **TRIGGER**: Sharing common operators among queries guaranteed to be executed by the same triggers

The ALL approach is used in NiagaraCQ [9,8]. Continuous queries in some recent stream processing systems such as Aurora [1], Borealis [2,11], TelegraphCQ [6], STREAM [15] do not contain explicit event specifications. However, we can regard these queries as those triggered by data arrivals from streams referred to in the queries. Since these systems do not consider query execution timings in their optimization processes, they are classified into systems using the ALL approach.

OpenCQ [13,18] groups continuous queries based on trigger conditions. Since it shares evaluation results of queries that have the same types of trigger conditions, it can be classified as a TRIGGER approach.

3.2.1 ALL: Sharing results of all common operators by ignoring execution timings

To explain the ALL approach, we still use the example in Section 3.1. Since three example queries contain common join operators $S \bowtie R$, results of the operators are shared by them. Figure 6 presents a sketch of query processing based on the approach. Here, we assume a system consisting of operators, input/output queues and windows.

When an alarm arrives from Clock_12, the operator is activated for $Q1$ and produces the result from input queues and windows of S and R. The result is then put into output queues corresponding to three queries. The data in $Q1$'s output queue are immediately delivered to the user. On the other hand, the data in the queues of $Q2$ and $Q3$ should be maintained until the events specified by these queries arise. When an alarm from Clock_13 activates $Q2$, the operator generates differential results using new data that arrived after the last activation of $Q1$. The new result is put into the three queues and merged with the result produced by the last activation of $Q1$. The relevant data in $Q2$'s queue are then selected and delivered to the user.

The problems with this approach are as follows.

- *Obsolete tuples in output queues*: Results of a shared operator are put into all output queues of the operator, but much of the data in the output queues

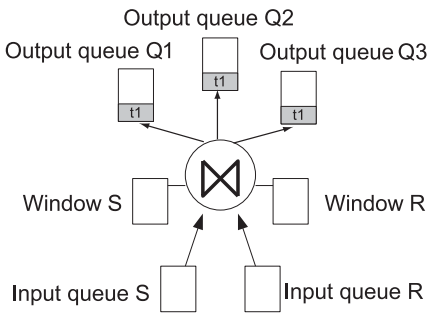

Fig. 6. Sharing results while ignoring execution timings

have fallen out of windows while waiting for the events. We must remove such obsolete data from the queues to avoid delivering unnecessary data. In the example, the output queue for $Q3$ includes data produced by $Q1$ and $Q2$. They will become obsolete when $Q3$ is triggered.

- *No consideration for asymmetric result sharing*: This approach implicitly assumes that results of queries are symmetrically shared. In the example, $Q2$ can reuse the results of query $Q1$, but $Q1$ never needs any results of $Q2$. $Q1$ gets unnecessary data from $Q2$, thus we must remove them from $Q1$'s output queue.

3.2.2 TRIGGER: Sharing results of common operators guaranteed to be executed by the same triggers

The system groups queries proved to be activated by the same triggers; it then extracts common operators in each group. This approach avoids the problem caused by mixing queries triggered by different triggers. Operators shared by multiple queries are activated at the same instants. However, this approach has the following problem.

- *Redundant query processing*: This approach does not focus on sharing common operators activated at close instants. It causes duplicate tasks for the

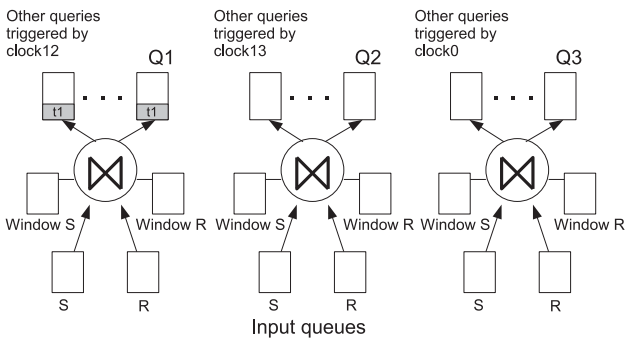

Fig. 7. Sharing results among queries activated at the same times

same stream data. In the example, $Q1$, $Q2$, $Q3$ are grouped into three groups separated because of their different triggers. No operators are shared among the three queries in this example (Figure 7). Therefore, separate processing of $Q1$ and $Q2$ produces a lot of duplicate results.

3.3 Proposed strategy: sharing results of common operators based on query execution patterns

We now show our proposed optimization strategy, which differs from the above two strategies. Our proposed strategy shares common operators based on the execution timings of queries.

In the example, it is a good idea for $Q2$ to reuse the results of $Q1$, but the results of $Q2$ are not required by $Q1$. $Q3$ does not need any results of $Q1$ and $Q2$, and vice versa. Results generated by a query should be reused when the window of the query has a large overlap with the window of the next activated query.

To achieve more flexible optimization, we designed an advanced query processing system with cache mechanism. The proposed system architecture differs from that assumed in Figures 6 and 7. It supports continuous query process-

ing with query result caching. An operator has a cache area to store results generated by the previous activation. The system can control which queries should produce/consume cache data. If it determines most results of a query are reusable for other queries, then the result of the query is stored into the cache area. When a cache may contain a reusable result for a query, the result of the query is generated by reusing the cache data.

Since the example queries are triggered by constant timers, we can decide the optimal query plan based on their execution patterns. However, when queries are triggered by unpredictable foreign events, we cannot decide the optimal query plan for such queries. In our scheme, the system dynamically decides which is better, processing their operators reusing results in caches, or processing separately. Based on the decisions, the system flexibly switches query plans.

4 Continuous query processing with query result caching

This section describes our proposed continuous query processing scheme. It uses a caching mechanism for sharing common intermediate results.

4.1 Architecture

First, we describe the continuous query processing system assumed in this paper. As shown in Figure 8, the architecture of the system follows the mediator/wrapper model. The main modules in the system are Executor, wrappers, clock streams, Query Analyzer and Optimizer. A wrapper is in charge of a corresponding information source such as data streams and traditional databases.

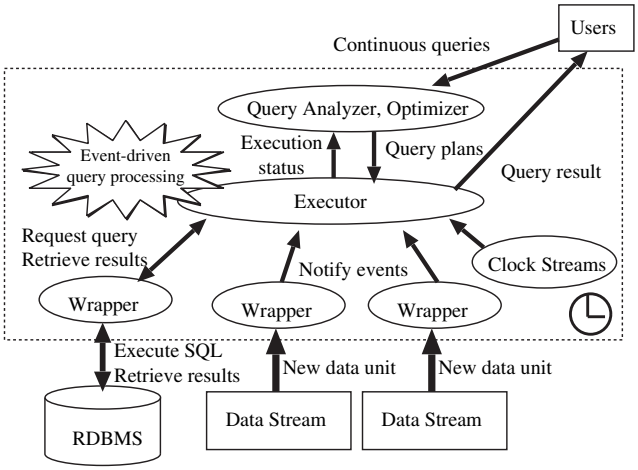

Fig. 8. System architecture

It detects arrivals of new data units, and transforms them into an internal data model. We model data streams as unbound relations as described in Section 2. Clock streams, which give alarms in constant intervals, are provided by the system. Wrappers and clock streams notify Executor of events. When Executor receives an event, it evaluates continuous queries corresponding to the event. A continuous query given by the user is analyzed and transformed into a basic query plan by Query Analyzer. Optimizer applies the proposed multiple query optimization method to a set of basic query plans, and derives query plans for query result caching. The internal mechanism in Executor is presented in Section 4.2.

4.2 Execution Mechanism in Executor

Executor holds query plans generated by Optimizer and processes them when events are notified. The main components in Executor are operators, queues and caches (Figure 9). Before describing them in detail, we summarize a behavior of Executor corresponding to one event. Triggering events are given by master information sources written in the MASTER clause of the query.

(1) When Executor receives an event notification from a wrapper, it appends the arriving tuple into the input queues. Then it loads query plans for relevant queries to be triggered by the event.

(2) Executor evaluates each query plan, which is a directed acyclic graph of operators. The order of activating operators is based on a bottom-up approach. Executor activates operators from the bottom of each query plan. The following steps are repeated for each operator.

   (a) Based on the query's window, the operator detects obsolete tuples in its queues. Tuples included in the window are processed by the operator.

   (b) The operator reads input tuples from input queues, and generates its result. Without the caching mechanism, the result is generated from scratch. If the caching mechanism is available, there is another choice: reusing cache data generated by other queries triggered previously. Whether or not cache data should be reused is decided dynamically.

   (c) The operator appends the result to its output queue. Its cache area is then overwritten by the result if it is expected to be reused by other queries. Update of the cache area is also decided dynamically. We assume cache area is maintained in main memory.

(3) After all evaluations, tuples detected to be obsolete for all queries are removed from queues by Executor.

The remaining part of this section explains details of operators, queues and caches.

4.2.1 Query Plans and Operators

A query plan is represented as a directed acyclic graph of operators. When Executor receives an event notification, it evaluates query plans corresponding to the event. Operators are activated from the bottom of the query plan, and query results are delivered to users at the top of the query plan. Each operator has parameters needed by its evaluation such as master information sources, windows, predicates, id of the query and so on. Operators considered in this paper are projection, window-join [12], and grouped selection [8]. Window-join is a popular operator in processing unbounded data streams. It generates its result using input tuples included in a range of the window. In this paper, we consider binary window-join. Join operations for multiple streams are achieved by a tree of binary-join operators. Grouped selection is an operator to group multiple selection operators. [8] implements it as a combination of join and split operators, but we assume one operator providing the same functions. It is used to efficiently evaluate multiple selection predicates given by multiple queries. Optimizer groups predicates based on signature type, which are triplets expressed in "*attribute op* constant". For example, $S.A > 10$ and $S.A > 20$ are classified into the same group, because their signatures are $S.A > constant$.

Operators shared by multiple queries have a set of parameters associated with all the relevant queries. An activated operator generates output tuples, and it appends them to the output queue, which is an input queue of the upper operator. If the operator is shared by multiple queries, multiple output queues are prepared to receive the output tuples. Unlike the ALL strategy in Section 3.2.1, the result of an activation is put into only the output queues corresponding to the queries triggered by the event.

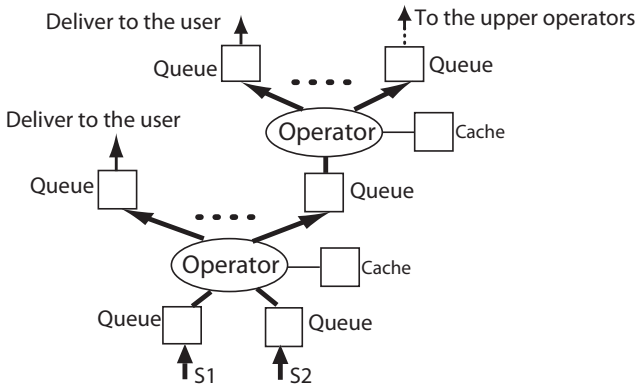

Fig. 9. Operators, queues, and caches

4.2.2 Queues

Queues are associated to an operator and hold tuples that have arrived from wrappers or have been generated by the lower operators. Tuples from the sources written in the FROM clause are sequentially appended to the corresponding queue. We do not allocate queues for master information sources not written in the FROM clause, because their tuples are used only for giving events.

Specifically, a queue is separated into two areas: a delta part and window part. A delta part holds new tuples that are not yet processed, and a window part holds tuples that have been processed once and are still included in the range of the query's window. When an operator is activated, it extracts tuples in the delta part of its input queues, and appends a result to output queues. We assume that generated tuples inherit all timestamps of original tuples. A window part is used by a window-join operator. Tuples in the delta part are moved to the window part once they are processed by the window-join operator. We need to maintain whole tuples included within the query's window to produce the results of window-join. If no master delivers tuples for a long term, tuples from other sources (not master information sources)

drop out from range of the window. Such obsolete tuples are detected from both the delta part and window part before the operator is evaluated. After evaluating operators, tuples not needed by any queries are actually removed from queues. Generally, it is decided by the size of the largest window in the system.

A queue is implemented using a list containing references of tuples. Tuples themselves are never copied, even if they are appended to multiple queues. A difference from ordinary queues is that our queue maintains a set of execution states, each of which is a pointer to the last tuple processed by the query sharing the operator. This helps Executor to identify new tuples that arrived after the last activation. Figure 10 shows a case of sharing an operator and its input queue on stream S among queries $Q1$ and $Q2$. $last\_exec(Q1)$ and $last\_exec(Q2)$ denote the last tuples processed by $Q1$ and $Q2$, respectively. $win(Q1)$ and $win(Q2)$ express the oldest tuples included within the windows of Q1 and Q2, respectively. This figure shows that $Q1$ has processed all tuples in the queue, and $Q2$ has not yet processed s7, s8, or s9. Tuples that arrived later than s3 are included within the range of the windows of $Q1$ and $Q2$. Hence, delta parts and window parts corresponding to $Q1$ and $Q2$ are as follows:

$$\Delta_{S,Q1} \quad : \quad \phi$$

$$W_{S,Q1} \quad : \quad \{ \text{ s3, s4, s5, s6, s7, s8, s9 } \}$$

$$\Delta_{S,Q2} \quad : \quad \{ \text{ s7, s8, s9 } \}$$

$$W_{S,Q2} \quad : \quad \{ \text{ s3, s4, s5, s6 } \}.$$

As exemplified by s1 and s2, Executor removes obsolete tuples not included

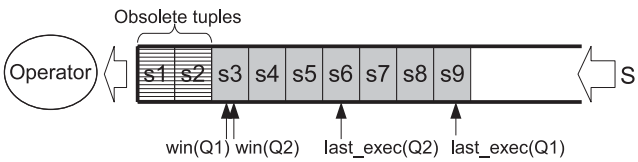

Fig. 10. Queue

in any windows.

4.2.3 Caches

An operator shared by multiple queries has a cache, which is a relation on the main memory. It also stores metadata such as the creation time, number of tuples, id and window size of the query that produced the cache data. Queries sharing the operator play two roles: *producer* and *consumer*. When the operator is activated for a producer query, Executor stores the result in the cache. Upon activation of a consumer query, Executor extracts the result from the cache and reuses it. The cache data is overwritten when the operator is activated for another producer query.

Generally, one operator has multiple producers and consumers, and the roles are not fixed. A query being a producer in this activation may become a consumer in the next activation. Section 5 describes how Optimizer determines producers and consumers from a set of queries sharing the operator.

4.3 Query optimization

Optimizer performs the proposed multiple query optimization method. It consists of the following two steps.

(1) Optimizer finds common operators from the set of basic query plans.

The query plan for each query is selected so that the number of shared common operators is maximized.

(2) In the running system, Optimizer dynamically determines which queries should produce/consume cache data by estimating the costs and the gains of scanning caches. Because of uncertainness with event occurrences, the decision is made for each activation of an operator. The operator generates output tuples according to the Optimizer's decision.

The first step is basically the same as traditional multiple query optimization methods [16,17]. They construct candidate query plans from each query, then find common operators among all query plans. Once common operators are found, the corresponding query plans are merged to share the operator. We regard join operators with the same join predicates as common operators. Projection operators extracting the same attributes are regarded as common operators. Grouped selection operators having the same type of signatures are merged.

For example, we suppose two queries $A \bowtie B$ and $A \bowtie B \bowtie C$. The first clearly has one candidate query plan $A \bowtie B$, and the second logically has three candidates: $(A \bowtie B) \bowtie C$, $(A \bowtie C) \bowtie B$, and $(B \bowtie C) \bowtie A$. Here, the pair of $A \bowtie B$ and $(A \bowtie B) \bowtie C$ has common join operators; Optimizer then outputs the query plan for two queries by merging them.

The second step is applied for operators potentially triggered by different events. If all operators are triggered by the same events, query result caching is not performed.

Multiple query optimization methods generate globally optimal plans. For some queries, individual processing cost could possibly be higher than the

cost of the plan without multiple query optimization. But, costs of many consumer queries are reduced by reusing cache data, thus we can expect to improve total performance. Since total performance is quite important in multiple query processing, our objective is minimizing the sum of processing costs of all queries.

4.4 Query result caching scheme

We now describe continuous query processing with query result caching. Here, we present how it works on evaluating window-join and grouped selection operators. Query result caching is applicable to projection operators; it is omitted here because it is rather straightforward.

4.4.1 Computation in a binary window-join operator

We show an example of query processing using a cache for window-join operator (Figure 11). As described in Section 4.3, we regard join operators with the same join predicates as common operators. Suppose $Q1$ and $Q2$ share a common window-join operator on streams S and R, and Optimizer designates $Q1$ as a producer query and $Q2$ as a consumer query. For simplicity, we also suppose $Q1$ and $Q2$ have the same window size. Even if they differ, query result caching is applicable. Here, we assume $Q1$ was executed in the past, thus the result of $Q1$ is already stored in cache C. The delta parts and window parts of the queries in Figure 11 are given in Table 1.

Without the cache, Executor evaluates the following expression when Q2 is triggered.

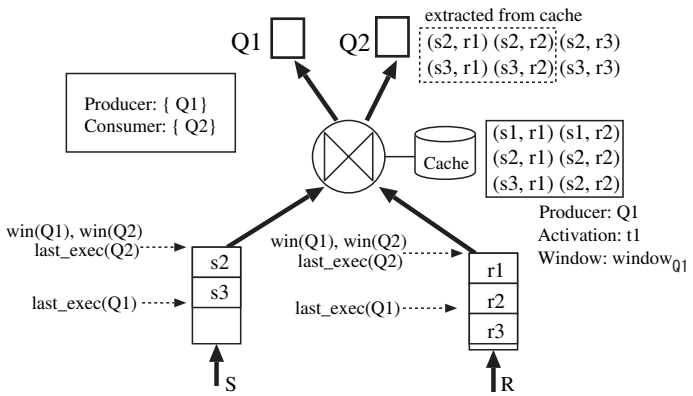

Fig. 11. Computation in a binary window-join operator

$$
\begin{aligned}
(S \bowtie R)_{Q2} = & (\Delta_{S,Q2} \bowtie \Delta_{R,Q2}) \\
& \cup (\Delta_{S,Q2} \bowtie W_{R,Q2}) \\
& \cup (W_{S,Q2} \bowtie \Delta_{R,Q2})
\end{aligned}
$$

We then obtain six tuples as the result (To keep things simple, the join predicate is not considered). Note that this result contains the duplicate tuples in the result generated by $Q1$.

With the cache, a different computation is made to get the same result as follows:

$$
\begin{aligned}
(S \bowtie R)_{Q2} = & (\Delta_{S,Q1} \bowtie (\Delta_{R,Q2} \cup W_{R,Q2})) \\
& \cup ((\Delta_{S,Q2} \cup W_{S,Q2}) \bowtie \Delta_{R,Q1}) \\
& \cup \sigma_{S.TS \in window_{Q2} \wedge R.TS \in window_{Q2}}(C)
\end{aligned}
$$

This means that it extracts tuples included within the $Q2$'s window ($window_{Q2}$) from cache C and computes only a partial result relevant to new tuples that have arrived after the last activation of $Q1$. In Figure 11, tuple r3 in R arrived after the last activation of $Q1$. Thus, two tuples are newly generated using r3, and the other four tuples are extracted from C. Obsolete tuples generated from s1 are filtered out when the operator scans the cache data.

Table 1

Delta and window parts in Figure11

	Q1	Q2
$W_{S,Q}$	{ s2, s3 }	ϕ
$\Delta_{S,Q}$	ϕ	{ s2, s3 }
$W_{R,Q}$	{ r1, r2 }	ϕ
$\Delta_{R,Q}$	{ r3 }	{ r1, r2, r3 }

4.4.2 Computation in a grouped selection operator

We show query result caching for grouped selection operator (Figure 12). Suppose $Q1$ and $Q2$ share a common grouped selection operator on streams S. Note that selection predicates of $Q1$ and $Q2$ differ, and $Q2$ has a stricter predicate than $Q1$'s ($Q1$: $S.A > 10$, $Q2$: $S.A > 20$). In an actual system, we use an index structure like predicate index [14] to evaluate a large number of predicates efficiently. To handle queries triggered by different master information sources, we group predicates by master information sources, and construct an index for each group. Here, again for simplicity, we do not go into detail about the index structure.

From the relationship between two predicates, $Q2$ may reuse some part of the result produced by $Q1$ when they are activated within a small interval. (Conversely, $Q1$ never reuses the result of $Q2$.) As in the previous example, suppose Optimizer designates $Q1$ as a producer and $Q2$ as a consumer, and the previous activation result of $Q1$ is already cached. The delta parts in Figure 12 are $\Delta_{S,Q1} = \{$ s3 $\}$, $\Delta_{S,Q2} = \{$ s1, s2, s3 $\}$. The grouped selection operator does not require window parts.

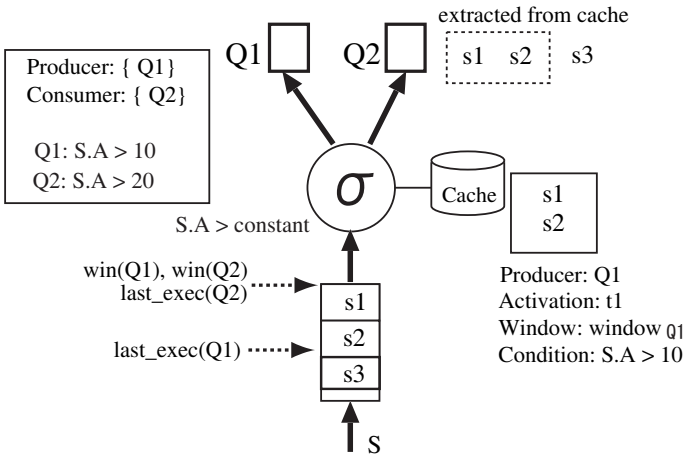

Fig. 12. Computation in a grouped selection operator

Without the cache, Executor evaluates the following expression when $Q2$ is triggered.

$$\sigma_{Q2}(S) = \sigma_{S.A>20}(\Delta_{S,Q2})$$

We then obtain three tuples as the result. (For simplicity, the selection predicate is not considered.) With the cache, it extracts tuples included within the $Q2$'s window ($window_{Q2}$) from cache C and computes only a partial result relevant to new tuples that arrived after the last activation of $Q1$.

$$\sigma_{Q2}(S) \;=\; \sigma_{S.A>20}(\Delta_{S,Q1}) \;\cup\; \sigma_{S.TS\in window_{Q2}\wedge S.A>20}(C)$$

In Figure 12, tuple s3 in S arrived after the last activation of $Q1$. Tuples in the cache area are checked again using $Q2$'s predicate. In Figure 12, s1 and s2 satisfy the condition. Obsolete tuples generated are also filtered out.

Table 2

Symbols related to cache production and reuse

symbol	description
Q	A query triggered by new event
P	The last query that produced cache data before Q's execution
t_Q (or t_P)	Execution time of Q (or P)
W_Q (or W_P)	Window size of Q (or P)
N_Q (or N_P)	Number of tuples generated by Q (or P) at t_Q (or t_P)

5 Decision for cache production/reuse

In the system, Optimizer dynamically determines which queries should produce/consume cache data. Optimizer makes the decision for each activation of an operator. We explain estimations of costs and gains of scanning caches and criteria of cache production and reuse. As described in Section 4.2.3, we call queries producing cache data *producers*, and call queries reusing cache data *consumers*. Symbols used in this section are summarized in Table 2.

5.1 Production of cache data

Optimizer designates a query as a candidate for a producer if its window size is bigger than a given threshold. Namely, results from queries having small window sizes are not cached, since their results will be too small to reuse.

When a triggered query is a candidate for a producer, the system computes gain of the outputs. It also computes loss of overwriting the current cache data

generated by the last activation of the producer. If the gain is greater than the loss, then the outputs are stored in the cache area. More concretely, gain and loss for a candidate query Q and the last producer query P are computed by the following formulas.

$$Production\_Gain_{Q,P} = \begin{cases} N_Q \cdot \frac{t_Q - t_P}{W_Q} & if\ t_P > t_Q - W_Q \\ N_Q & if\ t_P \leq t_Q - W_Q \end{cases}$$

$$Production\_Loss_{Q,P} = \begin{cases} N_P \cdot \frac{(t_Q - W_Q) - (t_P - W_P)}{W_P} & if\ t_P > t_Q - W_Q \\ N_P & if\ t_P \leq t_Q - W_Q \end{cases}$$

N_Q and N_P are the numbers of tuples generated by Q and P, t_Q and t_P are the last execution timestamps of the queries, and W_Q and W_P are sizes of windows of the queries (Figure 13). These formulas compute how many tuples are added or discarded by updating the cache area. Production_Gain and Production_Loss are computed every time Q's outputs are generated. We obtain actual values of N_Q and N_P from Q's outputs and cache data, respectively. If $t_p \leq t_Q - W_Q$, all cache data produced by P is obsolete and should be overwritten by Q's outputs. Thus, we assume that Production_Gain of Q is high (=N_Q).

The outputs generated by Q are stored if the following condition is satisfied.

$$Production\_Gain_{Q,P} / Production\_Loss_{Q,P} > \alpha$$

Note that α is a parameter giving the threshold for cache production. With large α value, the system rarely updates cache area. $\alpha \leq 1$ is better to overwrite cache area adequately. Section 6.5 presents experiment results about changing

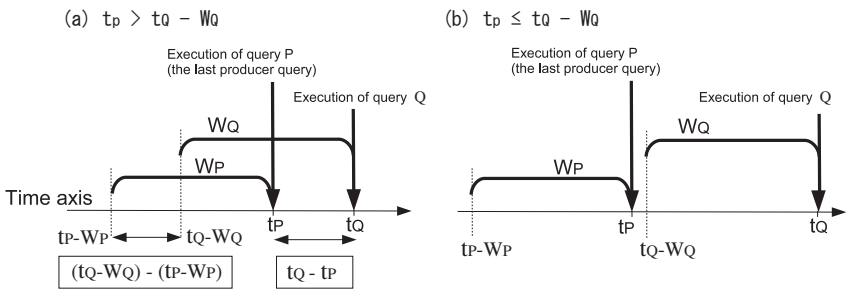

Fig. 13. Gain and loss computation for cache production

α.

5.2 Reuse of Cache Data

When a cache contains many reusable tuples, the cache is being used efficiently. However, if the cache includes many obsolete tuples and few reusable tuples, a query should be executed without using the cache. Therefore, it is important to estimate how many tuples are reusable. Optimizer estimates cache hits by analyzing recent query execution history.

When query Q is activated at t_Q, the number of reused tuples in the cache is estimated by the following formula.

$$Reuse\_Gain_{Q,P} = \begin{cases} N_P \cdot \frac{t_P - (t_Q - W_Q)}{W_P} & if\ t_P > t_Q - W_Q \\ 0 & if\ t_P \leq t_Q - W_Q \end{cases}$$

N_P means the number of tuples generated by the last activation of producer query P, t_Q and t_P are the last execution timestamps of the queries, and W_Q and W_P are sizes of windows of the queries (Figure 14). The cost of the scanning cache area equals the number of stored tuples.

$$Reuse\_Cost_{Q,P} = N_P$$

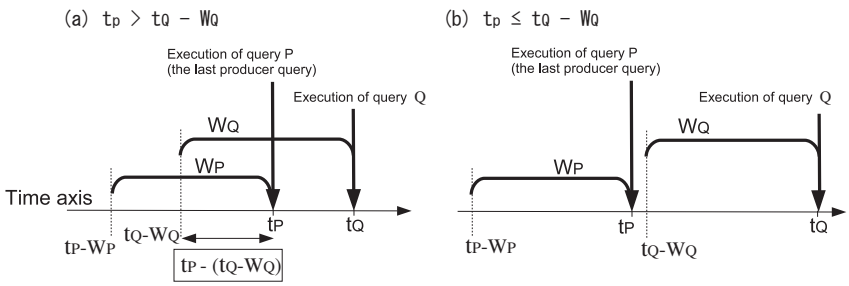

Fig. 14. Gain and cost computation for cache reuse

If the following condition is satisfied, the operator extracts tuples in the cache for query Q.

$$Reuse\_Gain_{Q,P}/Reuse\_Cost_{Q,P} > \beta$$

β is a parameter giving a threshold for cache reuse.

Figure 13 and Figure 14 are examples of 1-dimensional windows. For a window join of two streams, we have to consider an overlap of 2-dimensional windows like Figure 5. Their gain and loss are computed from a ratio of overlapping area.

6 Experiments

Efficiency of the proposed scheme is validated by intensive experimental evaluations.

6.1 Experiment environment

First, we describe an experiment environment and data sets. Experiments are made in the environment shown in Table 3. Our prototype system is written

in Java. Synthetic data are used in the experiments. Continuous queries are generated from the templates presented in Figure 15 and Figure 16. We have two sets of queries: queries including the common join operators, and queries including the common selection operators. Each set contains 1000 queries. In the templates, S is a data stream, which delivers $N(50 \leq N \leq 200)$ tuples per second. R is a relation in RDBMS containing 100 tuples. A generated query includes a window join operator with a 1-minute window on stream S. We can control the selectivity λ by changing S's properties ($0.2 \leq \lambda \leq 0.8$). Except for Section 6.5, parameters for query result caching are α=0.0, β=0.0 to observe performance when the system is optimistic about production/reuse of caches.

Ten clock streams are used to be master information sources. We prepare three sets of ten clock streams (Figure 17).

- *type1*: 10 clock streams with a 5-minutes cycle are prepared. They have different delivery timings. An interval between one clock and the next clock is 30 seconds. Thus, the windows of two queries triggered by two adjacent clocks overlap for 30 seconds (50% of the window).
- *type2*: This set includes 10 clock streams with a 1-minute cycle. An interval between one clock and the next clock is 6 seconds. Query executions with type2 are more frequent than those with MS1. Two queries triggered by two adjacent clocks have a 54-second overlap (90% of the window).
- *type3*: MS3 consists of 10 clock streams with a 5-minute cycle. Queries are triggered to create two types of window overlaps (10% or 90%) in turn.

Each clock triggers 100 queries in each query set. In Section 6.6, to evaluate performance in uncertain execution patterns, we use 10 master information sources that randomly change intervals of data arrival.

Table 3

Experiment Environment

CPU	Pentium 4 3.2GHz
RAM	1.5GB
OS	Windows XP SP2
Lang.	Java (J2SE 6.0)
RDBMS	HSQLDB 1.8.0

```
MASTER   Clock_i
SELECT   *
FROM     S [1 min],  R
WHERE    S.A >= R.A
```

Fig. 15. A template for a query containing a join operator

The proposed scheme is compared with the following two approaches.

- ALL: As described in Section 3.2.1, the system does not concern itself with query execution patterns, hence common operators are shared by all queries. Outputs generated by these operators are distributed to all output queues of the relevant queries.
- TRIGGER: As described in Section 3.2.2, the system constructs groups of queries activated by the same events. In each group, common operators are shared. Outputs generated by these operators are delivered to output queues of the relevant queries in the same group.

```
MASTER   Clock_i
SELECT   *
FROM     S [1 min]
WHERE    S.A = constant
```

Fig. 16. A template for a query containing a selection operator

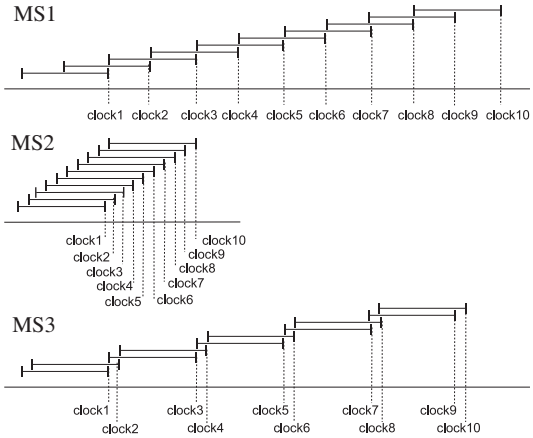

Fig. 17. Sets of clock streams in experiments

6.2 Operator selectivity

The efficiency of query processing with varying operator's selectivity is investigated. In this experiment, α and β are fixed at 0. An arrival rate of S is fixed at 50 tuples/sec for join queries. The rate for selection queries is fixed at 200 tuples/sec.

Query execution is performed for 30 minutes and we measure the processing time corresponding to each activation. The results are shown in Figures 18 and 19. The horizontal axis expresses selectivities and types of clock streams; the vertical axis expresses the average of processing times. Processing time is an interval between the arrival time of a master data unit and the completion time of operator activations corresponding to the master.

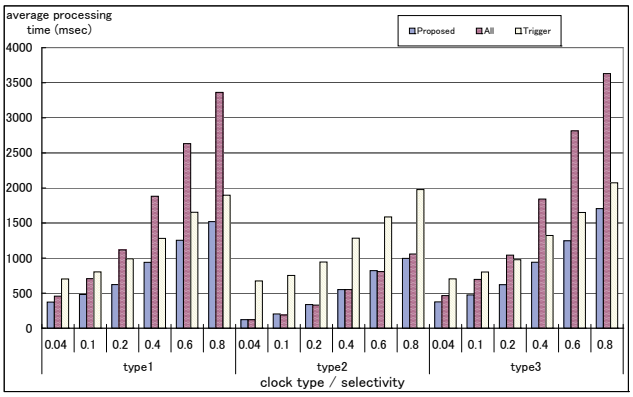

Fig. 18. Average processing time of join queries for several selectivities

Figure 18 is the result of join queries. It indicates that the proposed method is clearly faster than ALL and TRIGGER with clock type1 and clock type3. The difference between the proposed method and others increases when selectivity is high. The reason is that much useless garbage is generated in ALL. In TRIGGER, the system must generate many redundant tuples. With clock type2, the proposed method is as fast as ALL. Here, queries are frequently executed and their windows have large overlaps. Therefore, most intermediate results are reused by all queries. Clock type2 is advantageous to ALL, because ALL makes the system directly distribute outputs to all relevant queues without any decisions.

Figure 19 is the result of selection queries. Differences between the three strategies become small, because selection is a lighter operation than join. The proposed method outperforms ALL and TRIGGER with type1 and type3. With clock type2, ALL is the best, for the reason given above.

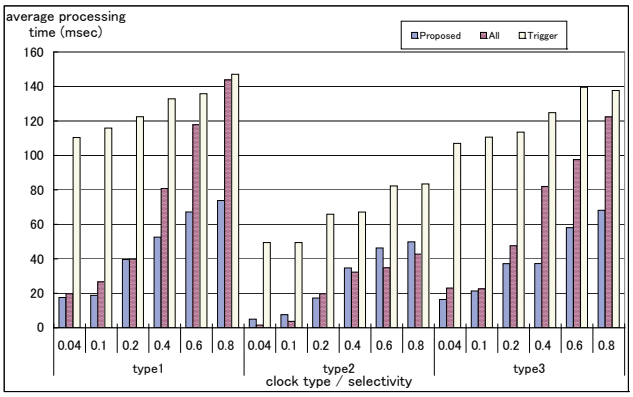

Fig. 19. Average processing time of selection queries for several selectivities

6.3 Rate of input tuples

We measure average processing time with several arrival rates from stream S (50, 100, 150, 200 tuples/sec). Selectivities of join and selection operators are fixed at 0.2. α and β are fixed at 0.

Results are shown in Figures 20 and 21. The horizontal axis shows arrival rates and the vertical axis shows the average of processing times. Processing times linearly increase according to the increase in arrival rates.

Figure 20 shows that our method is best. With clock type1 and type3, our method is distinctly more efficient than ALL and TRIGGER. With type2, the difference between the proposed method and ALL is very small. It is for the same reason described in Section 6.2.

In Figure 21, our method is also good. But, it is not absolutely dominant. The reason is the fewer number of intermediate results. A selection operator with low input rate produces few outputs. Thus, the efficiency that results from using caches decreases.

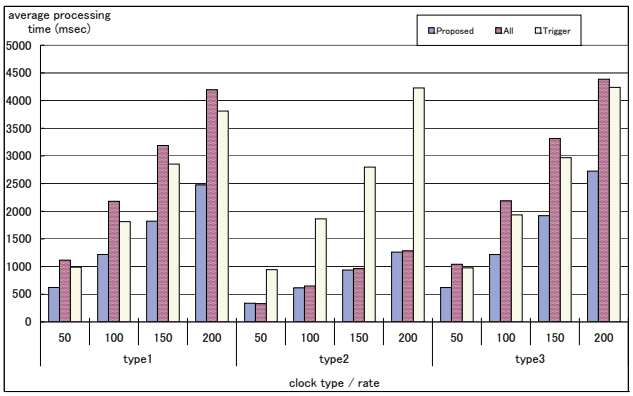

Fig. 20. Average processing time of join queries for several input rates

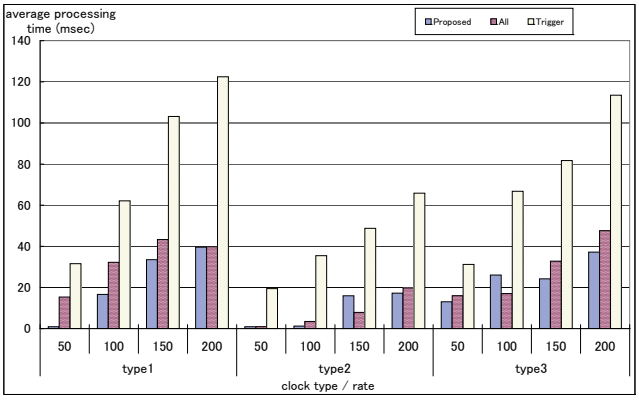

Fig. 21. Average processing time of selection queries for several input rates

6.4 Memory usage

This experiment measures the amount of memory used by the system. An arrival rate of join is fixed at 50, and that of selection is fixed at 200. Selectivity α and β are 0.2, 0 and 0, respectively.

To measure memory usage, we use the *jstat* command, which is included in Java Development Kit. Jstat periodically monitors memory usage at a 5-second interval while the system is executing queries. Figure 22 is an execution result with clock type1. The horizontal axis expresses time progress and the vertical axis expresses the amount of memory used. We investigate memory usage for all clock types as shown in this figure.

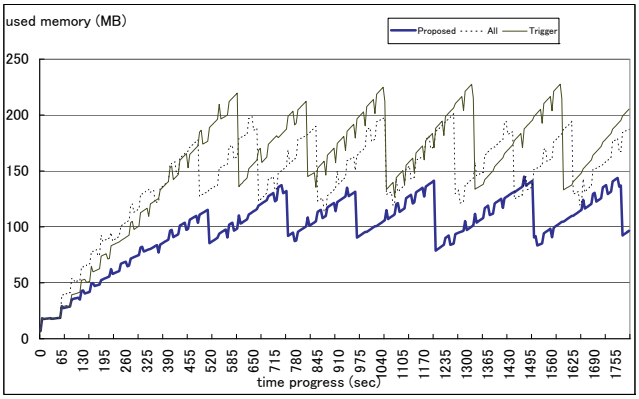

Fig. 22. Memory usage of join queries (type1)

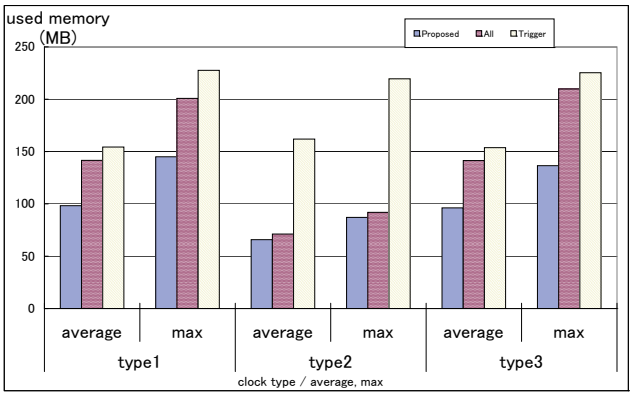

Fig. 23. Memory usage of join queries

Figures 23 and 24 show the results of join queries and those of selection queries, respectively. We show the average and maximum of memory usage in each case. The results reveal that the amount of memory needed by the proposed method is less than the others. Of special note, our method clearly outperforms ALL and TRIGGER with type1 and type3.

6.5 Parameters for query result caching

We measure the effects of parameters for query result caching. As described in Section 5, α (> 0) and $\beta(\in [0, 1])$ are thresholds that control production and reuse of cache data. In this experiment, the selectivity of operators is fixed at

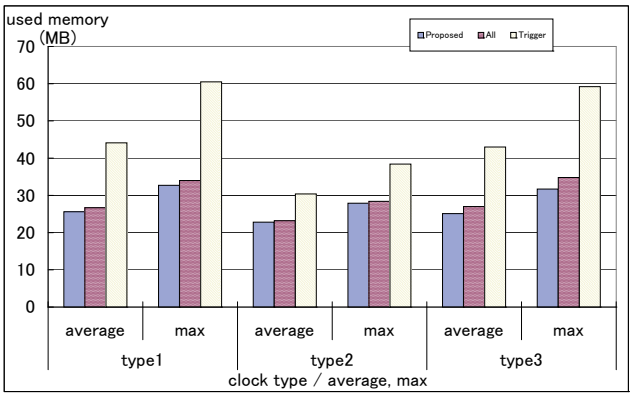

Fig. 24. Memory usage of selection queries

0.2. The arrival rate of join is 50 and that of selection is 200.

Figure 25 shows the results of different α values. The horizontal axis expresses α values and clock types. The vertical axis expresses the averages of processing times. β is fixed at 0. The left side of Figure 25 is the result of join queries, and the right side is the result of selection queries. Since we use periodic clock streams, the average ratio of window overlaps is almost fixed. When α is included in the range [0, 1.0], Production_Gain/Production_Loss is greater than α in most cases. Therefore, processing time does not change. Processing times of join queries increase when α equals 1.5. With a larger α value, the system becomes more reluctant to replace the cache area with new intermediate results. Since most cache data becomes obsolete, the efficiency of cache reuse is degraded.

Figure 26 shows the results of different β values. The left side of Figure 26 is the result of join and the right side is the result of selection. α is fixed at 0. With clock type2 and type3, varying the β value does not affect performance. With clock type1, processing times increase when the β value is set to 0.6 and 0.9. Two queries triggered by the adjacent two clocks have an overlap of 50% of their windows. Thus, the system does not reuse cache data when the

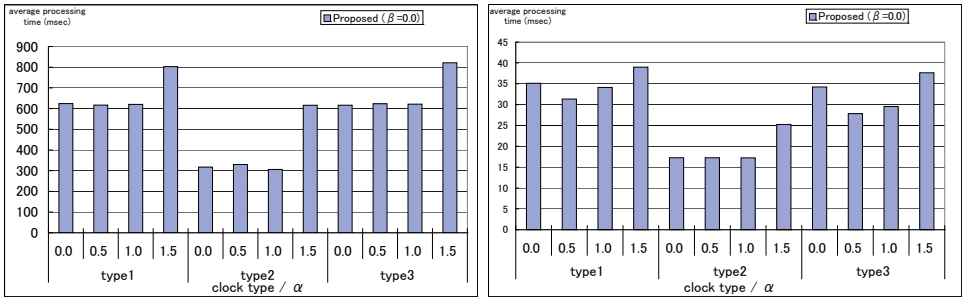

Fig. 25. Average processing time for several α (left) join queries, (right) selection queries

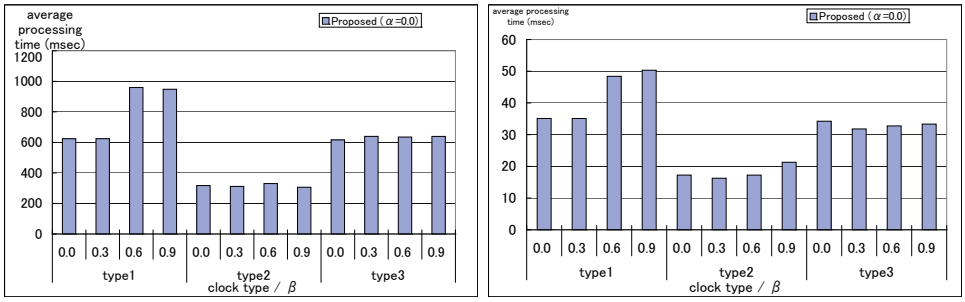

Fig. 26. Average processing time for several β (left) join queries, (right) selection queries

β value is greater than 0.5.

Setting small values to α and β increases the number of accesses to caches. On the other hand, setting larger values makes the system rarely access the cache area. In the latter case, system behavior becomes similar to TRIGGER. Judging from the results of our experiments, setting small values is better in our system environment. Deciding the optimal values for the parameters is not easy. We have to consider two factors: how much time the system takes to produce a new tuple from scratch and how much time it takes to read a tuple residing in the cache area. It is better to use small values if the former is more than the latter. We suppose that this assumption often holds, since the production of new tuples involves several steps such as memory allocation, substitution and so on.

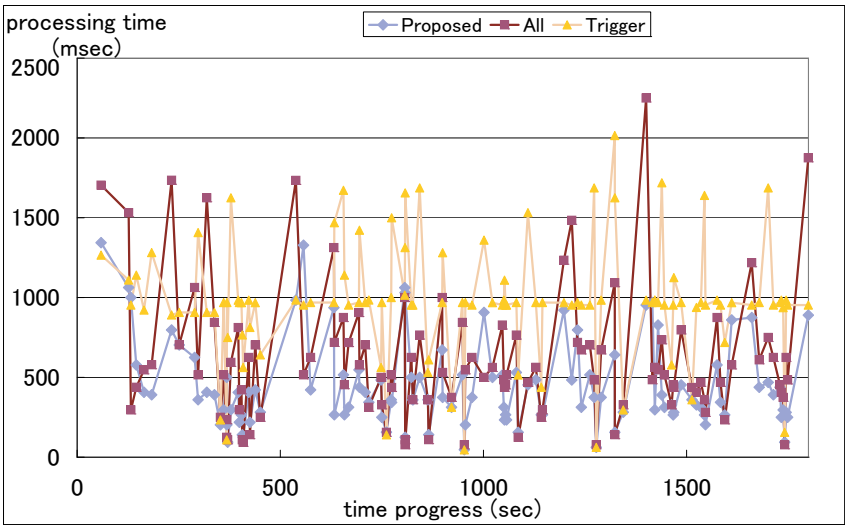

Fig. 27. Processing time of join queries with random execution patterns

6.6 Uncertain execution patterns

To evaluate performance in uncertain execution patterns, we use 10 master information sources that randomly change intervals of data arrival. Each master information source has a random number generator, and it chooses a sleep-interval from $[2sec, 300sec]$ each time. We give 10 different seeds to these random number generators. We use join queries and join selectivity is fixed at 0.2. Arrival rate is 50. α and β are fixed at 0.

Figure 27 is the result of executing queries for 30 minutes. The horizontal axis expresses time progress and the vertical axis expresses processing time for each activation. Queries are activated 113 times in this experiment. For each strategy, we count the number of fastest executions among the 3 strategies in 113 activations (Figure 28). The averages of processing times are shown in Figure 29. From these figures, our proposed method achieves efficient query execution even if execution patterns are not fixed and are unknown.

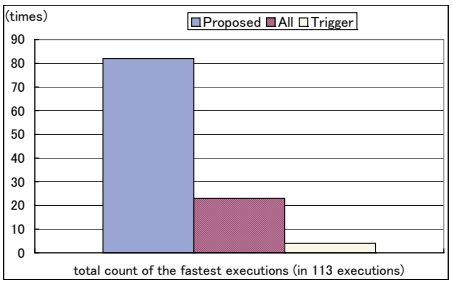

Fig. 28. Counts of fastest executions among 3 strategies (in 113 activations)

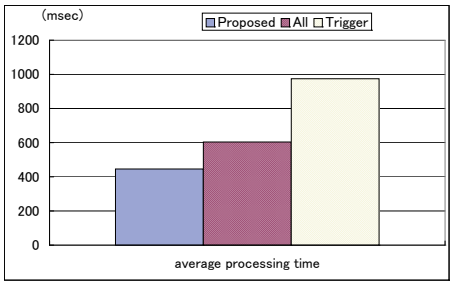

Fig. 29. Average processing time of join queries with random execution patterns

6.7 Window join of two streams

Next, we tried an experiment using queries with window joins of two streams. We generated 1000 queries from another query template: "MASTER Clock_i SELECT * FROM S1[1min], S2[1min] WHERE S1.A = S2.A". The arrival rates of S1 and S2 are 20 tuples per second. Join selectivity is fixed at 0.05. Master information sources are the same in Section 6.1 (type1, type2 and type3). For a window join of two streams, gain and loss are computed from overlaps of 2-dimensional windows (1 minute $\times$ 1 minute). The ratio of overlaps of 2-dimensional windows with type1 is 25% (30 second $\times$ 30 second). Similarly, one with type2 is 81%, and one with type3 is 1% or 81% in turn. α and β are fixed at 0.

Figure 30 shows average processing times. Our method outperforms the ALL approach and the TRIGGER approach. This result indicates our method is efficient for processing window joins of two streams.

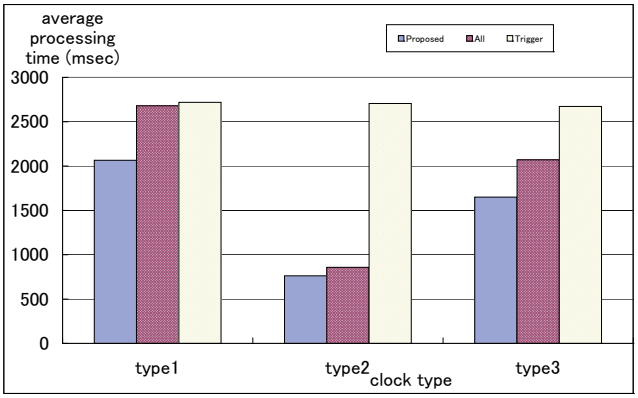

Fig. 30. Processing time of join queries of two streams

6.8 Experiments using real data

Finally, we show our experiment results using real data. We obtained network packet data [21], which contains 6 attributes: TIMESTAMP, SOURCE_IP, DEST_IP, SOURCE_PORT, DEST_PORT, and DATA_SIZE. Average arrival rate for this data is about 250 tuples/second. To generate queries, we used two types of templates: "MASTER Clock_i SELECT * FROM TCP[1min], PORT_INFO WHERE TCP.DEST_Port = PORT_INFO.PORT_NUM" and "MASTER Clock_i SELECT * FROM TCP[1min] WHERE TCP.DEST_Port < 1024". The first contains a join operator to integrate a packet data stream (TCP) and a port information table (PORT_INFO) including a pair of port number and detailed description of the port. The second contains a selection operator to monitor packets whose destination port number falls within 1–1024. These requirements are essential in network monitoring applications. Master information sources are the same in Section 6.1 (type1, type2 and type3). We generated 1000 queries from each template; we then measured processing times and the amount of memory used over 30 minutes.

Average processing times and amounts of memory used are presented in Fig-

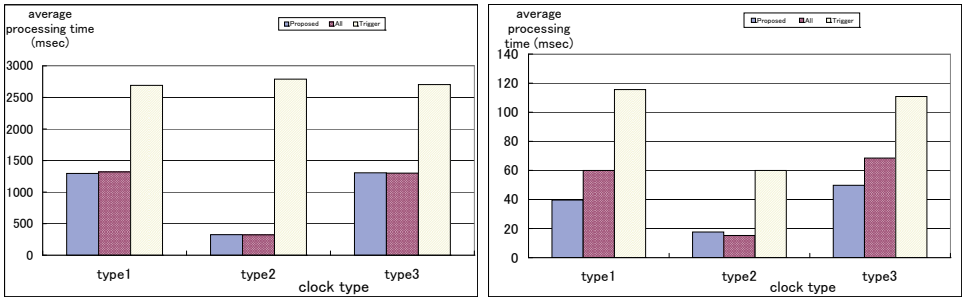

Fig. 31. Average processing time for join queries (left) and selection queries (right) with network packet data

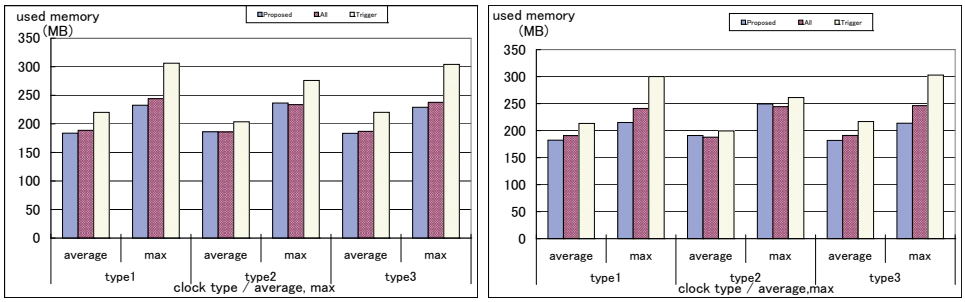

Fig. 32. Memory usage of join queries (left) and selection queries (right) with network packet data

ure 31 and Figure 32, respectively. The left side is the result of join queries; the right side is that of selection queries. Our method and the ALL approach outperform the TRIGGER approach in all cases, and our method is better than the ALL approach in type 1 and type 3. Because of low operator selectivities, the difference between our method and the ALL approach is not great; however, our method is good for real data.

7 Related work

7.1 Continuous query language

Generally, continuous queries are classified into two types: one is the arrival-based query, which is executed triggered by the arrival of new data units from the data stream; the other is the timer-based query, which is executed periodically, triggered by timer alarms.

Tapestry [19] supports continuous queries on append-only relational databases. Streams can be regarded as a kind of append-only relation. Continuous queries in Tapestry are timer-based queries that include temporal conditions in WHERE clauses.

OpenCQ [13,18] is a system integrating distributed heterogeneous information sources and supports continuous queries. Continuous queries in OpenCQ consist of three parts: query, trigger condition, and terminal condition. When the trigger condition is satisfied, the query is executed continuously until the terminal condition is satisfied. OpenCQ supports both arrival-based and timer-based queries.

STREAM [15] proposes CQL, a continuous query language [4] extending SQL. CQL supports three window types: time-based windows, tuple-based windows, and partitioned windows. In CQL, users cannot explicitly specify requirements for execution timings.

Aurora [1] proposes original algebraic operators and users specify requirements by drawing graphical flowcharts. Continuous queries in Aurora are only arrival-

based queries.

The scheme proposed in this paper considers both arrival-based and timer-based queries. We introduce the concept of a clock stream to regard the timer-based query as a special case of an arrival-based query. Thus, both types of continuous queries are uniformly treated within the scope of the proposed query optimization scheme. This paper considers only time-based windows, but our method can be applied to other window types as well with slight extension.

7.2 Multiple query optimization

Multiple query optimization schemes were originally proposed in the context of relational query processing [16,17]. They basically concentrate on extracting common subexpressions from among multiple queries to share intermediate query results. Targets of the methods are batch queries to be executed at the same time.

NiagaraCQ [9,8] proposes a multiple query optimization method for its continuous queries to handle incremental append and deletion of queries. When a new query is given, its query plan is constructed based on existing plans. In NiagaraCQ, continuous queries are triggered by tuple arrivals or timer events, and the multiple query optimization method is based on the ALL strategy. Continuous queries in NiagaraCQ, however, are simple and do not use window specifications to specify time intervals of interest. Thanks to its simplicity, the optimization method based on the ALL strategy can work well in the context of NiagaraCQ.

TelegraphCQ [6] is also a system for processing multiple continuous queries. CACQ [14] and PSoup [7] are prototype systems in TelegraphCQ. They can dynamically reorder operators to cope with changes in data characteristics and operator selectivities. They focus only on arrival-based queries and do not address timer-based queries. All queries using the same streams are evaluated at the same time, thus their multiple query optimization strategy is based on the ALL strategy.

OpenCQ [13,18] supports trigger grouping to achieve efficient evaluation of triggers. It shares evaluation of the same types of trigger conditions.

The novelty of the proposed method is multiple query optimization with query result caching and taking uncertain query execution patterns into account. We previously proposed a preliminary approach [20] to our method, which extracts execution patterns by simulating query execution using data arrival logs. The previous approach focuses only on static data streams, whose data arrival patterns are cyclic and predictable, and does not include query result caching.

7.3 Query result caching

Query result caching [3] was proposed in RDB. It improves performance by reusing results generated in the past. We have proposed continuous query processing with query result caching. In continuous query processing, the system must generate differential results from the previous activation, and take care of the problems mentioned in Section 3.1.

[5] proposes a caching scheme for multi-way join over data streams. It caches

intermediate results of MJoin to improve performance. [10] is a caching framework for expensive foreign functions over data streams. They do not consider multiple query optimization for event-driven continuous queries.

8 Conclusion

We proposed a framework for efficient event-driven continuous query processing with a mechanism for query result caching. The caching mechanism is used to perform adaptive multiple query optimization based on query execution patterns. A cache area is allocated for each operator; it stores intermediate results generated by the operator. The query optimizer dynamically determines which queries should produce/consume cache data. It achieves flexible optimization to share intermediate results. In experiments, we confirmed that the proposed system improves performance of query processing.

Future research issues remain. One is to combine our scheme with existing adaptive query processing schemes that do not take into account event-driven continuous queries. Another is consideration of memory limitations in resource-poor environments. We must carefully select operators to assign cache areas in such cases. Future research issues also include evaluation using real queries and data sets.

Acknowledgements

This research was supported in part by Core Research for Evolutional Science and Technology, Japan Science and Technology Agency and Grant-in-Aid for

Scientific Research (A) from Ministry of Education, Culture, Sports, Science and Technology.

References

[1] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. "Aurora: a New Model and Architecture for Data Stream Management," VLDB Journal Vol.12, No.2, pp. 120–139, 2003.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Centintemel, M. Cherniack, J. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. "The Design of the Borealis Stream Processing Engine," Proc. CIDR, pp.277–289, 2005.

[3] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. "Query Caching and Optimization in Distributed Mediator Systems," Proc. ACM SIGMOD, pp.137–148, 1996.

[4] A. Arasu, S. Babu and J. Widom. "The CQL Continuous Query Language: Semantic Foundations and Query Execution," VLDB Journal Vol.15, No.2, pp. 121–142, 2006.

[5] S. Babu, K. Munagala, J. Widom, and R. Motwani. "Adaptive Caching for Continuous Queries," Proc. ICDE, pp.118–129, 2005.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," Proc. CIDR 2003.

[7] S. Chandrasekaran, and M. J. Franklin. "PSoup: a System for Streaming Queries over Streaming Data," VLDB Journal Vol.12, No.2, pp. 140–156, 2003.

[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," Proc. ACM SIGMOD, pp. 379–390, 2000.

[9] J. Chen, D. J. DeWitt, and J. F. Naughton. "Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries," Proc. ICDE, pp. 345–356, 2002.

[10] M. Denny and M. J. Franklin. "Predicate Result Range Caching for Continuous Queries," Proc. ACM SIGMOD, pp. 646–657, 2005.

[11] J. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. "A Cooperative, Self-Configuring High-Availability Solution for Stream Processing," Proc. ICDE, pp.176–185, 2007.

[12] J. Kang, J. F. Naughton, and S. D. Viglas. "Evaluating Window Joins over Unbounded Streams," Proc. ICDE, pp. 341–352, 2003.

[13] L. Liu, C. Pu, and W. Tang. "Continual Queries for Internet Scale Event-Driven Information Delivery," TKDE, pp.610–628, 1999.

[14] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. "Continuously Adaptive Continuous Queries over Streams," Proc. ACM SIGMOD, pp. 49–60, 2002.

[15] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. "Query Processing, Resource Management, and Approximation in a Data Stream Management System," Proc. CIDR, 2003.

[16] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. "Efficient and Extensible Algorithms for Multi Query Optimization," Proc. ACM SIGMOD, pp. 249–260, 2000.

[17] T. K. Sellis. "Multiple-Query Optimization," ACM TODS, 13(1), pp. 23–52, 1988.

[18] W. Tang, L. Liu, and C. Pu. "Trigger Grouping: A Scalable Approach to Large Scale Information Monitoring," Proc. IEEE NCA, pp. 148-155, 2003.

[19] D. Terry, D. Goldberg, and D. Nichols. "Continuous Queries over Append-Only Databases," Proc. ACM SIGMOD, pp. 321–330, 1992.

[20] Y. Watanabe, and H. Kitagawa. "A Multiple Continuous Query Optimization Method Based on Query Execution Pattern Analysis," Proc. DASFAA, LNCS 2973, pp. 443–456, 2004.

[21] LBL-TCP-3, The Internet Traffic Archive.
http://ita.ee.lbl.gov/html/contrib/LBL-TCP-3.html