An Efficient Execution Scheme for Designated Event-based Stream Processing

Yan Wang † and Hiroyuki Kitagawa ‡

† Graduate School of Systems and Information Engineering, University of Tsukuba

‡ Faculty of Engineering, Information and Systems, University of Tsukuba

1-1-1 Tennodai, Tsukuba 305-8573, Japan

E-mail: † wangyan@kde.cs.tsukuba.ac.jp, ‡ kitagawa@ kde.cs.tsukuba.ac.jp

Abstract With the increase of streaming information sources, stream processing has been an important research issue. In this paper, we introduce the *designated event-based stream processing scheme* in the stream processing engine. Different from the traditional stream processing scheme, the query results are supposed to be generated only when tuples come from designated streams specified by the user. After introducing the proposed scheme, we consider its implementation. We discuss some important implementation internals of the stream processing engine and show an efficient *designated event-based execution scheme*. We also consider multi-queries which is a common situation in the stream processing engine. The experiment shows the advantages of the proposed execution scheme.

Keyword Stream processing engine, Designated event-based processing

1. Introduction

Recent years, the amount of unbounded generated data is growing rapidly like network packets, stock trades and sensor data. Such unbounded generated data is called streams. Many stream processing engines have been developed to deal with the streams like STREAM[1], S4[2], Borealis[3], D-stream[4] and Storm[5]. Users can register continuous queries on these systems and a continuous query is often involved with multiple stream sources. The traditional stream processing scheme is that whenever data comes from any stream sources, relevant continuous queries are supposed to be evaluated and the query results would be generated. However, such a processing scheme is not always what users want. There is another kind of requirement - sometimes, the result is supposed to be generated only when data comes from particular streams specified by the user. For example, suppose there are two stream sources: one is Twitter stream, and the other is News stream. Users may want to get the related Tweets in 30 minutes on the arrival of a piece of news. Then the users are able to analyze the query results to see want people are thinking about the latest news. In this case, the query results are to be triggered by the News stream only on the arrival of a piece of news. On the arrival of a Tweet, the query is not supposed to be evaluated and no query result would be generated. We call such a processing scheme as *designated event-based stream processing scheme* which implies the query results are triggered by the designated streams. The traditional stream processing engine dose not support the designated event-based processing. In this paper, we introduce the designated event-based processing scheme in the stream processing engine and propose an efficient execution scheme for it.

This paper is organized as follows. In section 2, we talk about related works. In section 3, we show the traditional stream processing model. Section 4 describes designated event-based processing scheme. Some basic implementation internals are explained in section 5. A naive execution scheme is explained in section 6 and a smart execution scheme is proposed in section 7. How to deal with multiple queries is explained in section 8. Section 9 shows the experimental evaluation and section 10 gives the future work and conclusion.

2. Related Works

STREAM[1] is a traditional stream processing engine. It accepts CQL[6] queries and translates a query into an execution plan containing leaf operators to get tuples from input streams. No matter which leaf operator accepts an input tuple, the whole execution plan tree is executed and a query result is generated. It does not support the designated event-based processing.

Discretized stream[4] is a distributed real time computation system. It processes streams in small time intervals. The input tuples in each interval are processed in a batch and then a query result is generated. However, it does not support the designated event-based processing.

Storm[5] is also a distributed real time computation system. Topology can be registered in the system. The topology contains spout nodes getting input tuples from

stream sources. Whenever a spout node gets input tuples, the whole topology is evaluated and a query result is generated.

3. Traditional Stream Processing Model

Before explaining designated event-based stream processing, we introduce the traditional stream processing model first.

How to perform queries on streams which contains unbounded data? We are familiar with the relational databases which contain a finite number of tuples in each table. Some relational operators, such as join and aggregation, must work on the finite number of tuples. When performing the relational operators on a stream, the stream must first be 'cut' into a relation containing a finite number of tuples, then the relational operators can work on the relation. In the stream processing engine, window operators are used to generate a relation from a stream.

Type	Name
Stream-to-relation	Tuple-based window
	Time-based window
Relation-to-relation	Selection
	Projection
	Join
	Groupby_aggregation
Relation-to-stream	Istream
	Rstream
	Dstream

Table 1 Operator

Three classes of operators are defined over streams and relations. They are stream-to-relation operators, relation-to-relation operators and relation-to-stream operators. A stream-to-relation operator takes a stream as an input and produces a relation as an output. A relation-to-relation operator takes a relation as an input and produces a relation as an output. A relation-to-stream operator takes a relation as an input and produces a stream as an output.

Some basic operators are shown in Table 1. A tuple-based window operator is a stream-to-relation operator and it always has a window size, then it generates a relation containing the number of tuples equal to the window size. For example, a tuple-based window operator with a window size of 20 always generates a relation containing the latest 20 tuples. After generating relations from streams, the relation-to-relation operators can take such relations as inputs and perform relational logics such as selection, projection, join and so on. The outputs of these operators are also relations.

In the traditional stream processing engine, a query is supposed to be executed whenever new tuples arrive at the system. It is easy to know that whenever a new tuple arrives at the system, the window operator responsible for accepting it would generate a new relation which possibly has a large overlapping with the previous one. It is the same with the relation-to-relation operators. It is not a wise way to always output the query results which have a large overlapping. Usually, we want to output the difference between the current output relation and the previous output one and that is what relation-to-stream operators do. An Istream operator always outputs the new tuples in the current output relation.

4. Designated Event-based Stream Processing

```
Master Stream1
SELECT *
FROM   Stream1[Rows 2],
       Stream2[Rows 2]
Where  Stream1.x = Stream2.x
```

Figure 1 Query Example

In this section, we explain the designated event-based stream processing scheme. We first define a discrete, ordered time domain T. A general time instant t is any value from T. In the traditional stream processing engine, a tuple may arrive at the system at any general time instant. The query result may be generated at any general time instant. As for our designated stream processing, the query is supposed to be evaluated only when tuples come from particular streams specified by the user. We call them *master streams*. The timestamps of tuples coming from master streams form a discrete, ordered time domain $T' \subseteq T$, and a master time instant t' is any value from T'. The query is supposed to be evaluated and generate query result only at t'.

Now we introduce the designated event-based stream processing scheme. In a query plan tree, the innermost operators are stream-to-relation operators and the outermost operators is a relation-to-stream operator producing a stream. The query result is supposed to be generated only when a new tuple comes from a master stream at the master time instant t'. We denote the result of the query at t' as S(t'). We emphasize that the traditional stream processing scheme is a special case of our scheme. Just specify all streams as master streams, then the results of our scheme are the same as the traditional one.

We give an example of the proposed processing scheme. Let us consider the query in Figure 1. This query intents for a join operation on the latest two tuples from Stream1

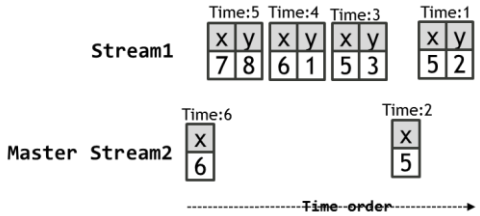

Figure 2 Example input

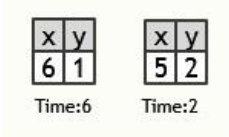

Figure 3 Query result

and the latest two tuples from Stream2 on the x attribute. Stream1 is a master stream. The example input tuples are shown in Figure 2. The tuples arrive at the stream processing engine in a timestamp order from stream1 and stream2. Each tuple coming from stream1 contains an x attribute and a y attribute. Each tuple coming from stream2 contains just an x attribute. Because stream2 is a master stream and tuples arrive at the stream processing engine at time instants 2 and 6, the output tuples are generated at time instants 2 and 6 as shown in the Figure 3.

5. Incremental Execution Scheme

Before discussing the implementation of the designated stream processing scheme, we look into some important implementation internals of the stream processing engine.

A window operator generates a relation from a stream. Whenever a new tuple comes to the window operator from a stream, the output relation of the window operator would be changed - the new coming tuple is inserted in the relation and some tuples may go outside of the window and are deleted from the relation. As for implementation, it is inefficient for the window operator or relation-to-relation operators to work on the whole relation each time. Since a relation is changing in a timestamp order including newly arriving tuples and omitted obsolete tuples, it is more efficient to do incremental computation. We append each tuple with a plus tag or a minus tag as well as a tuple identifier. A plus tag represents for a newly arriving tuple and a minus tag represents for an obsolete tuple. The tuple identifier is used to uniquely identify a tuple. Now an element is represented by <t,id,T,s> consisting of a timestamp, identifier, tag and tuple.

Now stream-to-relation and relation-to-relation

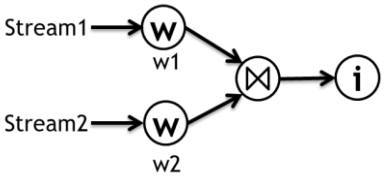

Figure 4 Query plan tree

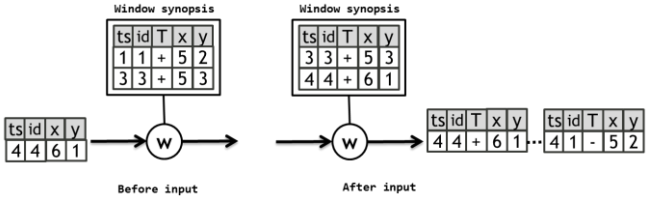

Figure 5 Window operator

operators output the update of the relation instead of the whole relation. Some operators like join, aggregation and window operators, need to know the whole relation for execution. These operators maintain synopses in themselves to save the current state. For example, a row-based window operator with a window size of 2 contains a synopsis always saving the latest 2 tuples.

In order to make the behaviour of operators and synopsis more clearly, we give an example. The query in Figure 1 is translated into a query plan tree in Figure 4. We use the input tuples in Figure 2. On the arrival of the tuple at time instant 4 from Stream1, the internal state of the window operator w1 changes as what we show in Figure 5. Its window size is 2 and the window synopsis always saves the latest 2 tuples. On the arrival of the tuple <4,4,6,1>, this tuple is inserted into the window synopsis, the number of tuples in the synopsis exceeds the window size, and the oldest one should be deleted from the synopsis. Then the output of the window operator are tuple <4,1,-,5,2> and tuple <4,4,+,6,1>. The status of the window operator after the procssing is shown in Figure 5 on the right.

6. Naive execution scheme

6.1. Execution model

We first present a naive execution scheme for implementing the designated event-based processing. The basic idea is very simple – everything works the same with the traditional stream processing engine except for the relation-to-stream operator which is the outer most operator in the query plan tree. Our relation-to-stream operators generate output only on the arrival of tuples originated by the master streams. The whole query plan tree is evaluated on the arrival of tuples coming from any stream source.

6.2. Master mark

In order to know which tuples are triggered by master streams, each tuple is given a *master mark* to tell whether this tuple is originated by a master stream or not. The value of master mark is true or false. If a tuple has a true master mark, it means this tuple is originated by master streams. Now an element is represented by <t,id,T,M,s> where M represents for a master mark.

Then we can know whether a tuple is originated by a master stream by looking into the master mark. But how to make sure all tuples originated by master streams have true master marks? The approach is very simple. At first, we give a true master mark to each tuple coming from master streams and false master marks to other tuples. Then when an operator accepts an input tuple, all output tuples generated based on the input tuple are given the same master mark and timestamp as the input tuple.

6.3. Istream Operator

Figure 6 Istream Operator Input Sequence

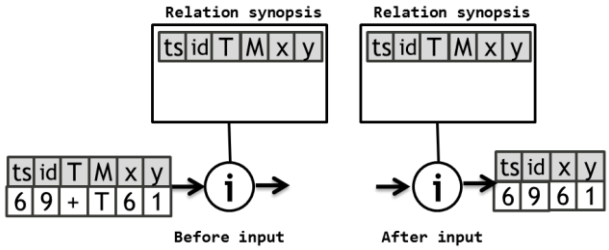

Figure 7 Istream Operator Execution Example

Now we introduce our Istream operator whose behavior is changed. Our Istream operator always outputs the new tuples triggered by arrivals of tuples with master marks. So it should maintain a synopsis saving the input relation. When an Istream operator accepts an input tuple, we first see whether it is a plus or minus tagged tuple. If it is a plus tagged tuple, the tuple would be inserted into the synopsis, and if it is a minus tagged tuple, the corresponding tuple is deleted from the synopsis. Next, it sees whether it is a true master mark tuple or a false master mark tuple. If it is a true master mark tuple, all of the tuples in the synopsis are output as the query results and the synopsis is cleaned up. If it is a false master mark tuple, it does nothing. With the input tuples shown in Figure 3, the input tuple sequence of the Istream operator is shown in Figure 6. The internal state of the Istream operator on the arrival of the tuple at time instant 6 is shown in Figure 7 on the left. The synopsis is empty and the input tuple is a plus tagged tuple with a true master mark. Then this tuple is inserted into the synopsis first and then all the tuples in the synopsis is output and the synopsis is cleaned up as shown in the Figure 7 on the right. We can see the tuples at time instant 2 and 6 have true master marks and the Istream operator generates output tuples when receiving them. The query result is exactly what we show in Figure 5.

6.4. Shortcoming

We can see that the whole query is evaluated on the arrival of tuples coming from any stream sources. On the arrival of a tuple coming from a master stream, some tuples from non-master streams may have gone outside the given window size and obsolete. They do not contribute to any query result, but they are processed by the whole query plan tree and many useless intermediate tuples may be generated.

For example, the tuple at time instant 3 from stream 1 in the Figure 3 does not contribute to any query result. However, such non-master stream tuples are processed by the whole query and generate many useless intermediate tuples because they are pushed to the following operators as soon as they comes to the window operator. Instead of that, we can change the behavior of the window operator to delete useless tuples from the beginning and reduce the processing cost. We call it smart approach.

7. Smart execution scheme

7.1. Execution model

One query has two execution modes, waiting mode and eager mode. At first, it is in the waiting mode and system gets input tuples from the information sources. If a tuple comes from the non-master stream, the query remains in the waiting mode. If a tuple comes from a master stream, the query's state changes to the eager mode and all operators are triggered. After the execution of all operators, the query's state comes back to the waiting mode.

The behavior of the window operator is changed. We introduce two kinds of window operators: one is *master window operator*, which is for master streams, and the other one is *smart window operator*, which is for non-master streams. The behavior of master window operators remains the same as we described in Section 4.3, while the behavior of smart window operators is different.

7.2. Smart Window Operator

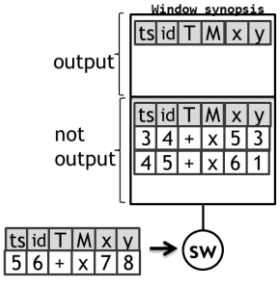

Figure 8 Smart Window Operator

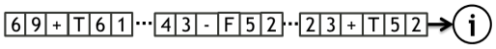

Figure 9 Istream Operator Input Sequence

When the query is in the waiting mode, a smart window operator accepts all input tuples. Then, it buffers the tuples in the window synopsis and does not output them. When the query is in the eager mode, a smart window operator outputs all buffered tuples.

The synopsis of the smart window operator is divided into two parts to specify which tuples are buffered. One is *output part* saving the tuples that have been output and the other is *non-output part* saving the tuples that have not been output. In the waiting mode, the incoming tuples are inserted in the non-output part. In the eager mode, the tuples in the non-output part are output and move to the output part.

When a tuple is inserted in the smart window operator in the waiting mode, the number of tuples in the synopsis may exceeds the window size and the oldest tuple should be deleted from the synopsis. If the oldest tuple is in the output part, it means this tuple has been processed by the whole query plan tree and the window operator should output the corresponding minus tuple. If the oldest tuple is in the non-output part, it means this tuple has not been processed by the whole plan tree and it can be deleted from the synopsis directly without generating any plus or minus tuples.

Again we look back to our example of input tuples in Figure 2. The tuple arrives at time instant 5 from stream 1. The state of the smart window operator is shown in the Figure 8. Two tuples have already saved in the non-output part of the window synopsis. When the coming tuple is inserted in the synopsis, the oldest tuple should be deleted. Because the oldest tuple which has a tuple identifier of 4 is in the non-output part, it can be deleted directly. Then this tuple is only processed by the smart window operator instead of the whole query plan tree in the naive approach. The input tuple sequence of the Istream operator is shown in Figure 9. Because less intermediate tuples are generated, we can see its length is shorter than the sequence in Figure 6. It generates the same results.

7.3. Analysis

On the arrival of tuples from non-master streams, the coming tuples are buffered in the smart window operators. If the number of the buffered tuples exceeds the window size, the overflowing tuples are just thrown away. Such tuples are not processed by the whole query plan tree, thus the processing cost will be reduced.

8. Multiple query

There are always more than one query registered in a stream processing engine. We think about how to support designated event-based processing dealing with multiple queries. If some common streams are involved in multiple queries, a direct thought is sharing the window operators for these common streams. However, though a stream may be involved in multiple queries, the corresponding window operator may behaves differently for different queries. For example, tuple-based or time-based, master or smart. Even for the tuple-based window operator, the window sizes may also be different. Furthermore, different queries have different execution modes. A smart window operator acts differently in the eager mode and waiting mode. How to deal with the situation that a shared window operator runs in the eager mode for one query and in the waiting mode for another query?

Because a shared-window operator works differently depending on the query, it is better to abstract a *virtual window operator* and let different queries register their own strategies in it. The strategy contains (window type, window size, output queue, query id). Window type's value may be tuple smart, tuple master, time smart or time master. The memory space of the synopsis belonging to the virtual window operator is shared among multiple queries. The overlapping tuples belonging to different queries are also shared in the synopsis. During the execution of a virtual window operator, it executes all registered strategies. It can get the execution mode by query id, execute its logic, and push output tuples to the corresponding output queue. Different strategies also should maintain their own pointers to save the position information of tuples in the synopsis.

```
Query1:                    Query2:
Master stream1             Master stream2
SELECT x                   SELECT *
FROM stream1[Rows 50]      FROM Stream1[Rows 100],
                                Stream2[Rows 20]
                           WHERE Stream1.x=Stream2.x
```

Figure 10 Multi-query example

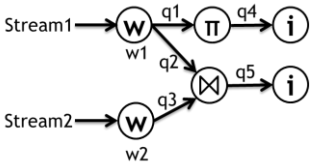

Figure 11 Query plan tree for multi-query

We give an example of multi-queries in Figure 10 which is translated into a query plan tree in Figure 11. As window operator w1 is shared by multiple queries, it is a virtual window operator and is registered with strategies (row master, 50, q1, Query1) and (row smart, 100, q2, Query2). On the arrival of a tuple from stream1, query1 is in the eager mode, while query2 is in the waiting mode. During the execution of the window operator w1, it processes its processing logic for each strategy and generates the output tuples. We can see that the memory space of the latest 50 tuples are shared between query1 and query2 thus the memory cost is reduced.

9. Experiment

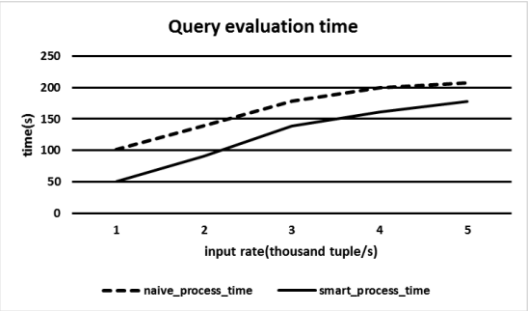

Figure 12 Query evaluation time

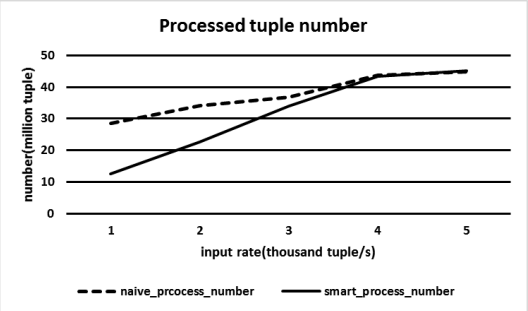

Figure 13 Processed tuple number

We have implemented both the naive and the smart execution schemes of the designated event-based processing.

We show experimental results using two streams: stream1 is not a master stream and stream2 is a master stream. The query is the one presented in Figure 1. The incoming rate of stream1 is 5 thousand tuple/s. We changed the incoming rate of stream2 from 1 thousand tuple/s to 5 thousand tuple/s. We executed the query for 5 minutes and observed the time spent on processing the query as well as the total number of tuples processed by the whole query plan tree for both the naive approach and the smart approach.

The time spent is shown in Figure 12 and the number of tuples processed is shown in Figure 13. We can see when the input rate of master stream2 is smaller than stream1, the time of the smart approach is less than the naive approach because many intermediate useless tuples are not generated. When we increase the input rate of master stream2, it approaches to the naive one because less buffered tuples are deleted from the window operator directly.

10. Conclusion and Future Work

We have proposed designated event-based stream processing and proposed its efficient execution scheme. We have developed a stream processing engine implementing the proposed execution scheme, and shown its advantages by experiments.

Future research issues include more elaborated analysis of the proposal and the parallel execution of multiple queries in the designated event-based stream processing.

Reference

[1] A. Arasu, et al. STREAM: The Stanford Data Stream Management System, 2004.

[2] L. Neumeyer, et al. S4: distributed stream computing platform. In KDCloud, 2010.

[3] D. J. Abadi, et al. The design of the Borealis stream processing engine. In Proceedings of the Conference on Innovative Data Systems Research, CIDR, 2005.

[4] M. Zaharia, et al. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In 9th USENIX NSDI, Apr. 2012.

[5] http://storm-project.net/

[6] A. Arasu, et al. CQL: A Language for Continuous Queries over Streams and Relations. In Proc. of the Ninth Intl. Conf. on Database Programming Languages, September 2003.