D-026

Design and Evaluation of Designated Event-based Stream Processing

Yan Wang†　　Hiroyuki Kitagawa‡

1. Introduction

Recently, the amount of stream data is growing rapidly like network packets, stock trades and sensor data. It becomes more and more important to deal with the stream data efficiently. Many stream processing engines are developed to process the stream data like STREAM[1], S4[2], Discretized Streams[3] and Storm[4]. The traditional stream processing scheme is that whenever data comes from any information sources, the relevant queries are supposed to be evaluated and the query results are generated. However, this is not what users always want. Sometimes, users want to get query results only when data comes from some particular streams. For example, considering there are two streams, one is a network connection stream, the other is a system failure stream. The administrator may want to know the network condition only when some failure occurs. So the stream processing engine would generate the query result only when data comes from the particular stream which is the system failure stream. We define such processing scheme as “designated event-based stream processing scheme” and we call the triggering streams “master streams”.

In this paper, we show the designated event-based stream processing scheme and its efficient execution scheme. We have implemented the execution scheme on our stream processing engine called JsSpinner and we describe the evaluation of it by experiments.

2. Data model

The data model of JsSpinner is JSON document. It is semi-structured and semi-structured data is an important part of the big data today. It is much more lightweight than XML so it is easy for machines to parse and generate. It has nested values and its expression power is more than just key-value pairs. Some stream processing engines like STREAM[1] have relational data models, and stream processing engines like Storm[4] support user defined types and do not support processing JSON data natively. JsSpinner is more friendly to query JSON stream data.

3. Query

```
stream1 = readFromWrapper ("stream1", true) ;
stream2 = readFromWrapper ("stream2", false) ;
tmp1 = stream1 -> window[rows 1];
tmp2 = stream2 -> window[rows 100];
j = join s in tmp1,
     d in tmp2
    where s.A == d.A
    into {s.A,d.B};
```

Figure 1 A query example

† Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba

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

Jaql[5] is a query language for querying JSON data but it is not designed for querying stream data. We make an extension of the Jaql query to support querying JSON stream data. JsSpinner allows users to register such queries.

A simple join query is shown in Figure 1. This query intents for a join operation on the latest document from Stream1 and the latest 100 documents from Stream2 on A attribute. Stream1 is a master stream and Stream2 is a non-master stream.

4. Traditional Stream Processing Model

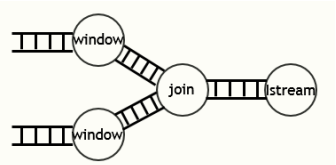

Figure 2 An example query plan tree

A query is usually translated into a query plan tree. For example, the query in Figure 1 is translated into a query plan tree in Figure 2. It contains three kinds of operators.

The window operators are responsible for ‘cutting’ the infinite input streams into a finite number of documents according to the window size. We call the finite number of documents collection. Then the algebraic operators like join and aggregation can work on the collections and generate collections.

In the traditional stream processing engine, a query is supposed to be executed whenever new documents arrive at the system. It is easy to know that whenever a new document arrives at the system, the window operator responsible for accepting it would generate a new collection which possibly has a large overlapping with the previous one. It is the same with other operators. It is not a wise way to always output the query results which have a large overlapping. Usually, we want to output the new documents in the current output and that is what an Istream operator does.

5. Designated Event-based Stream Processing

In our designated stream processing, the query is supposed to be evaluated only when documents come from master streams. 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.

6. Incremental Execution Scheme

Whenever a new document comes to the window operator, the output collection of the window operator would possibly have a large overlapping with the previous one. As for implementation, it is not wise to output the whole collection on each input document. Since a collection is changing in a timestamp order including newly arriving documents and excluding obsolete documents, it is more efficient to do incremental computation.

We borrowed the idea from CQL[6] and append each document with a plus tag or a minus tag. A plus tag represents for a newly arriving document and a minus tag represents for an obsolete document.

7. Naive Execution Scheme

We first present a naive execution scheme for implementing the designated event-based processing. The whole query plan tree is evaluated on the arrival of documents coming from any stream sources. We change the behavior of the outer-most operator which is the Istream operator.

In order to know which documents are triggered by master streams, each document is given a master mark to tell whether this document is originated by a master stream or not. The value of master mark is true or false. If a document has a true master mark, it means this document is originated by master streams. Our Istream operator always outputs the new documents triggered by the arrivals of documents with true master marks.

When a document comes from a master stream, some documents which have arrived from non-mater streams may have reached beyond the given window and are obsolete. However, they are processed in the naive execution scheme and generate useless intermediate documents, while they do not contribute to query results.

8. Smart Execution Scheme

The naive execution scheme may generate many useless intermediate query results. The smart execution scheme address this problem. We change the behavior of the window operator. We introduce the smart window operator for non-master streams. When a document comes from a non-master stream, a smart window operator accepts it. Then, it buffers the document and does not output it. When a document comes from a master stream, all buffered documents in the smart window operator should be output.

When a document is buffered in the smart window operator, the number of documents in the window operator may exceed the window size and the oldest document should be deleted. If the oldest document was already output, the window operator should output the corresponding minus document. If the oldest document is not yet output, it can be deleted from the buffer directly without generating any plus or minus documents. Thus the useless intermediate documents are not generated.

9. Experiment

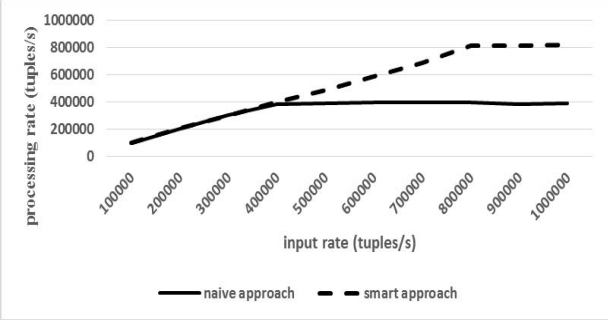

Figure 3 Processing rate

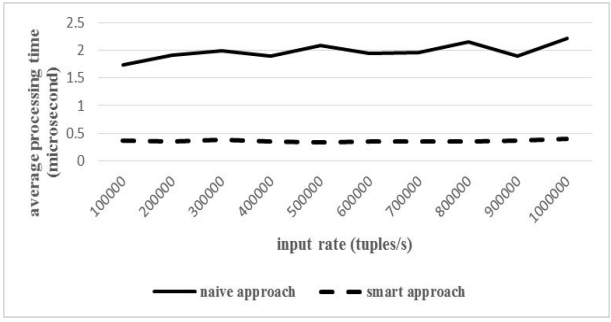

Figure 4 Average processing time

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 a master stream and stream2 is a non-master stream. The query is the one presented in Figure 1. The incoming rate of stream1 is 1/1000 of stream2. We changed the incoming rate of stream2 from 100,000 tuples/s to 1,000,000 tuples/s. We executed the query for 5 minutes and observed the processing rate as well as the average processing time of each document for both the naive approach and the smart approach.

The processing rate is shown in Figure 3 and the average processing time is shown in Figure 4. We can see the smart approach can deal with more input documents than the naive approach when the input rate is between 300,000 and 800,000 tuples/s. The average processing time of smart approach is less than the naive approach because many useless intermediate documents are not generated.

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 the parallel execution of multiple queries in the designated event-based stream processing.

Acknowledgement

This research was partly supported by the program "Research and Development on Real World Big Data Integration and Analysis" of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

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] M. Zaharia, et al. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. In HotCloud, 2012.

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

[5] https://code.google.com/p/jaql/

[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.