4L-01

Query language designed for processing JSON data in a complex event processing engine

Tao Wenlong† Satoru Fujita‡

Graduate School of Computer and Information Sciences, Hosei University†

Computer and Information Sciences, Hosei University‡

ABSTRACT

In recent years, with the popularity of the Internet and the high-speed flow of various information, dealing with this kind of streaming data has become a meaningful topic. Besides, this kind of research on complex event processing (CEP) can be applied to many fields, such as finance, cloud computing, and education. We designed a system that can be used to analyze JSON data from the stock market, with the help of container technologies. This paper presents the design of the query language, which can be used to query data from streams. In addition, our language has query, union, decouple, and other complex operations, which contains most of the operations in a CEP system.

Keywords

streaming data, CEP, JSON, query language

1. INTRODUCTION

For any CEP system, the design of the query language is an important part, and it is also important for users to learn and use the query language easily in the system.

In this paper, we describe the design of an efficient query language based on JSON data. First, we present a CEP system architecture and details of its components. Then, we show the syntax details of the query language and the data structure we used, and then, we give some examples to show how this query language is working.

2. SYSTEM DESIGN

The basic architecture of our system is shown in Figure1. It contains data resources, stream queue, user query, condition tree, query engine, and output stream[1]. These components are described in more detail below.

Data Resources: This system can get data from different resources, such as databases, the internet, inputting by users through their computers, and so on. For testing, we used stock data from the database.

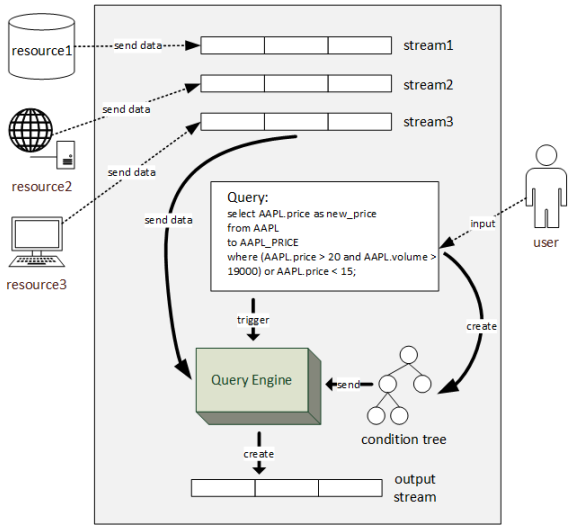

Figure1: The architecture of CEP system

Stream Queue: When data recourses send their data to the system, it creates priority queues to store data temporarily, and resend them to query engines.

User Query: Users need to present their query (similar to SQL) into the system before data come.

Condition Tree: When our system gets the query input by a user, the system compiles the query into a condition tree[2] used in the query engine later.

Query Engine: After the query is compiled, the system activates a query engine, which continuously check if data from stream queues are satisfied with the conditions in the query.

Output Stream: While the engine is running, it generates the data that users need into a new data stream, and stores it into a new queue.

3. LANGUAGE DESIGN

3.1 Data Model

We first describe the data model which is the base for the language. Our target data are stock data from the stock market, so we assume that it contains price, volume, and timestamp. In addition, data structure that we choose is JSON, because of its easiness to use, and it looks like the following:

```
{ "time":"2020/9/15/13:50",
 "price":"25.5",
"volume":"256568" }
```

Query language designed for processing JSON data in a complex event processing engine
†Tao Wenlong, Graduate School of CIS, Hosei University
‡Satoru Fujita, CIS, Hosei University

3.2 Query Operation

The basic grammar of the query operation is constructed as follows:

select < attributes >
from < input streams >
to < output stream >
where < conditions >
window <window condition>

The grammar of the query operation is similar to SQL. For example, the ***select*** statement consists of a part specifying attributes for output data and some clauses. The ***from*** clause is used to get input stream names and the ***to*** clause is to define the output stream name. The ***where*** clause represents conditions, which need to be satisfied with, in the input data, and it is transformed into condition trees in query engines, and the ***window*** clause is an optional constraint to the lifespan of data [3][4].

Example1:

```
select AAPL.price as new-price
  from AAPL
  to AAPL-PRICE
  where AAPL.price > 20 and AAPL.volume >19000;
```

This sample query checks data from the AAPL stream, and gets an attribute called “price” from the input stream. Moreover, it renames “price” to “new-price”. It also defines that “AAPL-PRICE” as the name of our output stream. This query only gets results when these two conditions are satisfied: price data > 20, and volume data > 19000.

Example2:

```
select AAPL[1].price as new-price
  from AAPL
  to AAPL-PRICE
  where AAPL[1].price > AAPL[2].price
  window size 5;
```

This example is similar to the example1, but it specifies a window size as 5. AAPL[1] determines the first data within the window. Similarly, AAPL[2] determines the second one.

3.3 Union Operation

union < stream expression >
to < output stream >

The ***union*** statement contains more than two input streams to be combined and the ***to*** clause used to define the name of the output stream. Furthermore, in the stream expression, we can use new streams created by the **Query Operation** that we mentioned before. In this situation, it will become a complex query.

Example3:

```
union AAPL and IBM
  to AAPL-IBM;
```

This union operation creates a new stream called AAPL-IBM as a result of combining AAPL and IBM streams.

3.4 Decouple Operation

decouple < stream expression >
to < output stream prefix>
by < decouple key >

The ***decouple*** statement divides an input stream to several output streams defined by the ***to*** clause according to a keyword defined by the ***by*** clause. The names of output streams are the specified prefix with colon and values corresponding to the keyword.

Example4:

```
decouple STOCK
  to company
  by STOCK.name;
```

This operation decouples a stream called “STOCK” to new streams according to the name attribute of the data. For example. If there are two kinds of name attributes in the data, “AAPL” and “IBM”. Then we will get two output streams, like “company: AAPL” and “company: IBM”. Each of the output streams only contains data whose name attribute is “AAPL” or “IBM” respectively.

4. CONCLUSION

We successfully designed an efficient query language for a CEP system, and explained the details of it. We are also designing and developing a query engine for the CEP system, but have not completed yet. There are many optimization points in the engine to increase the operating speed of the system and additional functions required for the query language.

REFERENCES

[1] Krishnamurthy, S., Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., ... & Shah, M. A. (2003). TelegraphCQ: An architectural status report. IEEE Data Eng. Bull., 26(1), 11-18.
[2] Liu, Hongying, Satoshi Goto, and Junhuai Li. "The study and application of tree-based RFID complex event detection algorithm." Proceedings. The 2009 International Symposium on Web Information Systems and Applications (WISA 2009). Academy Publisher, 2009.
[3] Gyllstrom, D., Wu, E., Chae, H. J., Diao, Y., Stahlberg, P., & Anderson, G. (2006). SASE: Complex event processing over streams. arXiv preprint cs/0612128.
[4] Demers, Alan J., et al. "Cayuga: A General Purpose Event Monitoring System." Cidr. Vol. 7. 2007.