International Journal of Control and Automation
Vol.8, No.11 (2015), pp.13-20
http://dx.doi.org/10.14257/ijca.2015.8.11.02

Exploiting Window Query Semantics in Scalable Data Stream Processing

Hyeon Gyu Kim

Department of Computer Engineering, Sahmyook University,
Seoul 139-742, Republic of Korea
hgkim@syu.ac.kr

Abstract

Recently, we have witnessed the emergence of new data stream management systems that can scale up to a large number of machines for the real-time processing of big data. These systems typically provide a procedural programming interface with their own APIs. However, to enable the rapid development of applications, it is desirable to support a declarative interface with clear processing semantics, such as window SQL. This paper examines the programming interfaces of the state-of-the-art data stream management systems and discusses the necessity of SQL support to help users write stream queries easily and in an integrated manner.

Keywords: *Data stream processing, Scalability, DSMS, SQL, Sliding windows*

1. Introduction

In recent years, high-speed real-time data has exploded in volume and availability [1]. Examples of such fast data include sensor data streams, real-time stock market data, and social-media feeds generated from Twitter and Facebook. Numerous applications must process these fast data, often with minimal latency and high scalability. Well-known applications include:

- Personalization of search results and advertising by analyzing user behavior logs (e.g., clicks and search keywords) [2, 3];
- Building and updating a search index from a stream of crawled web pages [4];
- Monitoring abnormal events or hot topics from social log data [5-7];
- Healthcare monitoring using mobile devices or wireless sensor network [8].

Google's MapReduce has emerged as a popular framework for parallel processing of Internet-scale big data using a cluster of low-end commodity machines [9]. However, it is not well suited for real-time processing of fast data because it is primarily designed for batch processing of queries on large-scale datasets. As an alternative solution, legacy data stream management systems (DSMSs) such as Aurora/Borealis [10] and STREAM [11] can be considered for processing the fast data. However, as discussed in [3, 5], these systems have limitations in their scalability because they cannot scale to hundreds or thousands of machines to manage big data.

To address this issue, a new class of data stream processing systems has been introduced, including Yahoo!'s S4 [2], Twitter's Storm [5], Walmart's Muppet [6], Google's MillWheel [7], LinkedIn's Samza [12], IBM's InfoSphere Streams [13], Microsoft's StreamInsight [3], and many others. These systems can be characterized by scalability and fault-tolerance, which distinguishes them from the legacy DSMSs.

The new DSMSs typically provide a procedural programming interface with their own APIs; however, the interface and APIs are different in each system. Therefore, when users want to develop an application with a specific system, they must first become familiar with its programming syntax and semantics. Debugging is also difficult in this circumstance because users may not fully understand how the system processes a given application when they begin programming with the new APIs.

As a solution for this issue, window SQL [14] can be used to support the rapid development of stream applications in the new DSMSs. Window SQL is an extension of standard SQL with syntax to define sliding windows (necessary to limit the scope of query processing over infinite, continuous data streams) and provides well-defined syntax with clear processing semantics. Thus, if it is supported in a system, users can prototype their applications easily and in an integrated manner.

Nevertheless, its support has been rarely discussed in the literature. Beginning in 2014, pilot projects were initiated to support window SQL on Storm and Samza [15, 16]. However, it is expected that the support of window SQL may not be a simple task. For example, owing to the differences of fault-tolerance and event processing mechanisms of each system, it may not be easy to fully satisfy the semantics of window SQL, which will be discussed below.

In this paper, we discuss the issues of supporting window SQL in the new DSMSs. We first describe the programming interfaces of several popular DSMSs including S4, Storm, and StreamInsight. We then discuss the necessity of SQL support and address the issues of its supporting in the current systems. For convenience, the new DSMSs are sometimes referred to as *scalable DSMSs* in this paper because their scalability is the most prominent characteristic distinguishing them from the legacy DSMSs.

2. Programming Interfaces of Scalable DSMSs

2.1. Yahoo!'s S4

For real-time processing of fast data, Yahoo! launched an open source stream processing engine called S4 [2], which is now a part of the Apache Incubator project. It has a decentralized and symmetric architecture where all worker nodes in a cluster are identical without centralized control. Apache ZooKeeper [17] is used to coordinate the nodes within the cluster. When a node fails, S4 does not recover the events lost from the failure. Instead, it attempts to detect a failed worker quickly and redirect new events to another worker to minimize the latency. This strategy is called *lossy failover* in literature.

The computations in S4 are performed by *Processing Elements* (PEs). The programming interface to define a PE is similar to MapReduce and consists of two primary functions: *processEvent*() as an input event handler and *output*() as an output mechanism. For each PE, a configuration file can be provided to interconnect the input and output channels. By describing this, a directed query graph can be constructed where a node represents a PE and an edge between two nodes corresponds to a communication channel connecting two PEs.

Figure 1 illustrates an example of the PE definition to count the number of query strings. The PE is organized to receive events of type *QueryEvent* and output its processing results to a persistent store called *externalPersister*. The output is written to the store every 10 minutes. Such input and output channel information is defined in the configuration file, which is presented in Figure 2.

```
private queryCount = 0;
public void processEvent(Event event) {
queryCount++;
}
public void output() {
String query = (String) this.getKeyValue().get(0);
persister.set(query, queryCount);
}
```

Figure 1. Example Of PE to Count the Number Of Query Strings In S4 (Querycounterpe.Java)

```
        <bean id="queryCounterPE" class="com.company.S4.QueryCounterPE">
        <property name="keys">
        <list><value>QueryEvent queryString</value></list>
        </property>
        <property name="persister" ref="externalPersister">
        <property name="outputFrequencyByTimeBoundary" value="600">
    </bean>
```

Figure 2. Configuration For Querycounterpe (Querycounterpe.Xml)

2.2. Twitter's Storm

Storm [5] was initially created by Nathan Marz at BackType and was acquired by Twitter in 2011. It is now also in Apache Incubator. More than 60 companies are either using or experimenting with Storm and it is currently considered the most popular scalable DSMS.

```
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("words", new TestWordSpout(), 10);
        builder.setBolt("exclaim1", new ExclamationBolt(), 3).shuffleGrouping("words");
        builder.setBolt("exclaim2", new ExclamationBolt(), 2).shuffleGrouping("exclaim1");
```

Figure 3. Example of The Topology Description In Storm

A query graph in Storm is called a topology and consists of two types of computation: *spouts* as stream sources and *bolts* as data translators or processors. A topology can be implemented in Java or other programming languages. Figure 3 is an example of a topology description in Java, which consists of one spout and two bolts. Each bolt has at least one input stream. For example, bolt *exclaim1* receives input events from spout words, whose outputs are then inputted to bolt *exclaim2*. Each spout or bolt is required to implement predefined event handlers such as *prepare*(), *execute*(), and *cleanup*().

During run-time, computations in a topology are distributed into worker nodes in a cloud, and the worker nodes are managed by a master node called *Nimbus*, which is similar to Hadoop's JobTracker. Apache ZooKeeper is also used for the coordination of worker nodes. Basically, Storm does not provide a data recovery mechanism; hence, data can be lost when a node fails. However, data recovery can be supported using Trident [18] with Storm.

2.3. Google's MillWheel

MillWheel [7] is a scalable DSMS developed in Google for building low-latency data processing applications such as *Zeitgeist*. Google's Zeitgeist is an application designed to analyze queries from Google searches to build a historical model for each query and then perform anomaly detection, such as spiking or dipping searches, as quickly as possible.

```
computation SpikeDetector {
  input_streams {
    stream model_updates {
      key_extractor = ‘SearchQuery’
    }
    stream window_counts {
      key_extractor = ‘SearchQuery’
    }
  }
  output_streams {
    stream anomalies {
      record_format = ‘AnomalyMessages’
    }
  }
}
```

Figure 4. Example of the Computation Definition In Meelwheel

The programming interface of MillWheel is similar to that of Storm, where a query is specified in the form of a directed computation graph. Each processing unit is called a *computation* in Storm. Figure 4 presents an example of the topology definition for computation *SpikeDetector*. It receives two input streams, *model_updates* and *window _counts*, and outputs its results to the stream *anomalies*. To define the computation, users must implement two event handlers, *ProcessRecord*() and *ProcessTimer*().

MillWheel provides fault-tolerance at the framework level, where any node in the query topology can fail at any time without affecting the correctness of the result. A persistent storage such as BigTable [19] or Spanner [20] is used for data recovery. To ensure idempotency, it provides an algorithm to ensure that each record is exactly once from the user’s perspective; this scheme is called an *exactly-once delivery* in literature.

2.4. MS’s StreamInsight

StreamInsight [3] is a scalable stream processing system developed by Microsoft. Distinguished from the majority of scalable DSMSs, it supports a declarative query language similar to SQL. Figure 5 presents an example of a query to obtain the word count, which is described in LINQ, the query language of StreamInsight. In the example, *WordStream* denotes a stream of records with schema (*word*, *ts*).

```
var WordCountQuery = from e in WordStream
group e by e.word into wordGroups
from window in wordGroups.HoppingWindow(TimeSpan.FromHours(1),
        TimeSpan.FromMinutes(15))
select new {
Word = wordGroups.word,
Count = window.Count(e)
 };
```

Figure 5. Example of the LINQ Query to Obtain the Word Count in Streaminsight

Although the query is specified in a declarative fashion, users may not be familiar with the LINQ syntax. The query shown in Figure 5 is considerably different from the standard SQL syntax, where the FROM clauses are presented twice and dedicated APIs must be used to define sliding windows.

3. Window SQL Support

As a solution for the issue discussed above, SQL can properly be used to describe continuous queries in scalable DSMSs. The benefits of using SQL as a programming interface are as follows:

- SQL is a standard with well-defined syntax, which enables users to write their applications easily;
- The processing semantic is clear; hence, users can simply focus on what results a query should provide, not on how the query is processed;
- Many optimization techniques that have been introduced in the literature can be borrowed for the query processing.

In data stream processing, sliding windows are essential. This is because data streams are excessively large or often inherently unbounded. Thus, queries on the streams cannot be answered if they involve *blocking operators* such as joins or aggregates because such operators cannot start processing until all the inputs are ready. A common solution for this issue is to restrict the range of stream queries using sliding windows that contain the most recent data of the stream [21].

For the syntax to specify sliding windows in a query, we use an approach proposed by Li. [22]. Their syntax includes two basic parameters for window specification: RANGE to denote a window size and SLIDE to denote a window update (or slide) interval. Using SQL with this window syntax, the above query to count the number of words in search keywords can be easily specified as follows.

```
SELECT word, COUNT(*)
FROM WordStream [RANGE 1 HOUR, SLIDE 1 MIN]
    GROUP BY word;
```

Figure 6. Example of the Window SQL Query to Obtain the Word Count

Using the syntax presented by Li , many other kinds of windows can also be specified. For instance, *landmark windows* can be defined, which are similar to sliding windows except that each window starts at the beginning of the stream. *Tumbling windows* can be specified where adjacent windows do not overlap, usually by setting the values of the RANGE and SLIDE parameters to be equal. Several attributes can be used to define the RANGE and SLIDE parameters of a window. For more details, refer to the paper [22].

A top-*k* query can easily be specified in a similar fashion. The query below is to obtain the top-10 frequent words presented in the search keywords.

```
SELECT word, COUNT(*) as cnt
FROM WordStream [RANGE 1 HOUR, SLIDE 1 MIN]
    GROUP BY word
    ORDER 10 BY cnt
```

Figure 6. Example of the Window SQL Query to Obtain the Top-10 Frequent Words

Despite the benefits of using window SQL, its support in the scalable DSMSs has rarely been discussed in literature. Beginning in 2014, two pilot projects were initiated to support window SQL on Storm and Samza [15, 16]. However, it is expected that the support of window SQL may not be simple. Owing to the difference of fault-tolerance and event processing mechanisms of each system, it may not be easy to fully satisfy the semantics of window SQL. For example, data recovery mechanisms of the scalable DSMSs (regarding fault-tolerance) can be classified into the following three groups:

- *No data recovery*
  Data lost owing to node failures is not recovered in this scheme. S4, Muppet, and Storm (without Trident) belong to this category
- *At-least once delivery*
  Although there is no data loss, an input record can be re-delivered (duplicated) when a node is recovered from a failure. Samza and StreamMapReduce belong to this category
- *Exactly-once delivery*
  This scheme guarantees that an input record is delivered only once even when a node fails. MillWheel, SparkStreaming, and Storm with Trident support this scheme.

From the above discussion, the exact semantics of window SQL can be preserved only in systems supporting “exactly-once delivery”. The support of window SQL can also be influenced by the event processing mechanisms, which can be categorized as follows:

- *Event-driven processing*
  In this scheme, user queries are evaluated whenever a new input event arrives in the system. The majority of the scalable DSMSs including Storm, Samza, and MillWheel are included in this category
- *Small-batch processing*
  User queries are evaluated whenever a predefined time interval elapses. SparkStreaming is included in this category.

In event-driven processing systems, it is not easy to support periodically updated windows that are defined with the SLIDE parameters. In small-batch processing systems, tuple-driven windows without SLIDE parameters may not be supported.

From the above discussion, we can see that it is difficult to satisfy the exact semantics of window SQL in the current scalable DSMSs owing to the differences of the fault-tolerance and event processing mechanisms. Consequently, further research is required to provide accurate query results in these systems.

4. Conclusion and Future Work

In this paper, we discussed programming interfaces for several popular scalable DSMSs and described the necessity of a declarative interface for rapid application development. As a solution, we presented window SQL and demonstrated that it can be used to describe a query easily and in an integrated manner. We then discussed the difficulty in satisfying the exact semantics of window SQL in the current systems because of the differences of their fault-tolerance and event processing mechanisms. In future, we plan to perform further research to overcome this issue.

References

[1] T. Heinze, “Cloud-based data stream processing”, *Proceedings of the DEBS,* **(2014)**, pp. 238-245.
[2] L. Neumeyer, “S4: Distributed stream computing platform”, *Proceedings of the ICDMW,* **(2010)**, pp. 170-177.

[3] B. Chandramoulin, J.Goldstein and S. Duan, "Temporal analytics on big data for web advertising", *Proceedings of the ICDE*, **(2012)**, pp. 90-101.
[4] D. Logothetis and K. Yocum., "Ad-hoc data processing in the cloud", *Proceedings of the VLDB Endowment,* **(2008)**, pp. 1472-1475.
[5] A. Toshniwal, "Storm @ Twitter", *Proceedings of the ACM SIGMOD*, **(2014)**, pp. 147-156.
[6] W. Lam, "Muppet: MapReduce-style processing of fast data", *Proceedings of the VLDB Endowment*, 5(12) **(2012)**, pp. 1814-1825.
[7] T. Akidau , "MillWheel: fault-tolerant stream processing at internet scale", *Proceedings of the VLDB Endowment,* 6(11) **(2013)**, pp. 1033-1044.
[8] C. Doulkeridis and K. Norvag, "A survey of large-scale analytical query processing in MapReduce", *VLDB J*., 23(3) **(2014)**, pp. 355-380.
[9] J. T. Kim, "Analyses of characteristics of u-healthcare system based on wireless communication", *J. lnf. Commun. Converg. Eng.*, 10(4) **(2012)**, pp. 337-342.
[10] D. J. Abadi, "Aurora: a new model and architecture for data stream management", *VLDB J*., 12(2) **(2003)**, 120-139.
[11] A. Arasu, "STREAM: The Stanford stream data manager", IEEE Data Eng. Bull., 26(1) **(2003)**, 19-26.
[12] C. Riccomini, "Apache Samza: LinkedIn's real-time stream processing framework", **(2013)**, available at https://engineering.linkedin.com/data-streams/apache-samza-linkedins-real-time-stream-processing-framework
[13] M. Hirzel, "IBM Streams Processing Language: analyzing big data in motion", *IBM J. Res. Develop*., 57(3/4) **(2013)**, pp. 7:1-7:11.
[14] H. G. Kim and M. H. Kim, "A review of window query processing for data streams". *J. Comput. Sci. Eng*., 7(4) **(2013)**, pp. 220-230.
[15] Squall, **(2014)**, available at https://github.com/epfldata/squall/wiki
[16] High-level language for samza, **(2015)**, available at https://issues.apache.org/jira/browse/SAMZA-390
[17] P. Hunt, M. Monar, F. P. Junqueira and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems", *Proceedings of the Usenix Annual Technical Conference*, **(2010)**, pp. 1-14.
[18] Storm Trident, available at https://storm.apache.org/documentation/Trident-tutorial.html
[19] F. Chang, "Bigtable: A distributed storage system for structured data", *ACM Trans. Comput. Syst*., 26(4) **(2008)**, 1-26.
[20] J. C. Corbett, "Spanner: Googles globally-distributed database", *Proceedings of the OSDI*, **(2012)**.
[21] H. G. Kim, C. Kim and M. H. Kim, "Adaptive disorder control in data stream processing", *Comput Inform*, 31(2) **(2012)**, 393-410.
[22] J. Li, "Semantics and evaluation techniques for window aggregates in data streams", *Proceedings of the ACM SIGMOD*, **(2005)**, pp. 311-322.

Author

Hyeon Gyu Kim, he received B. Sc. (1997) and M. Sc. (2000) degrees in Computer Science from the University of Ulsan and received his Ph.D. degree (2010) in Computer Science from the Korea Advanced Institute of Science and Technology (KAIST). He was a Chief Research Engineer at LG Electronics from 2001 to 2011, and a Senior Researcher at the Korea Atomic Energy Research Institute (KAERI) from 2011 to 2012. He joined the faculty of the Department of Computer Engineering at Sahmyook University, Seoul, Korea in 2012, where he is currently an assistant professor. His research interests include databases, data stream processing, mobile computing, and probabilistic safety assessment.