High Efficient Complex Event Processing Based on Storm

Shengjian Liu, Yongheng Wang, Shuguang Peng, and Xinlong Zhang

Abstract In recent years there is a huge increase in real-time data, which cannot be processed efficiently. Complex event processing has become a very important method to get meaningful information. However, supporting complex event detection in multiple sources environments is a challenging problem. To allow for inferring high level information from vast amounts of continuous arriving data. In this paper, we present a complex event processing system based on a novel distributed computing platform Storm, which goes further than distributing queries and achieves better scalability by parallelizing event detection, and also higher efficiency through the use of some optimizations. The experimental shows that the event processing system is effective and better scalability.

Keywords Complex event processing • Event detection • Distributed processing

1 Introduction

In recent years there has been a huge increase in the data produced on the Internet, effectively doubling in size every year. In many cases, the data is continuously produced by software applications in quantities that are not examinable manually. Complex event processing (CEP) is a set of techniques to address such classes of problems [1]. CEP targets applications that both require the processing of large amounts of data and low processing latencies. One application scenario for such systems is when too much data is generated, but only a fraction of the data can be stored.

Historically, the common approach for such applications was to store the generated data in databases or logs, and process it afterwards in batch processing

S. Liu (✉) • Y. Wang • S. Peng • X. Zhang
College of Information Science and Engineering, Hunan University, Changsha, China
e-mail: phosic@sina.cn

B. Zhang et al. (eds.), *The Proceedings of the Second International Conference on Communications, Signal Processing, and Systems*, Lecture Notes in Electrical Engineering 246, DOI 10.1007/978-3-319-00536-2_77,

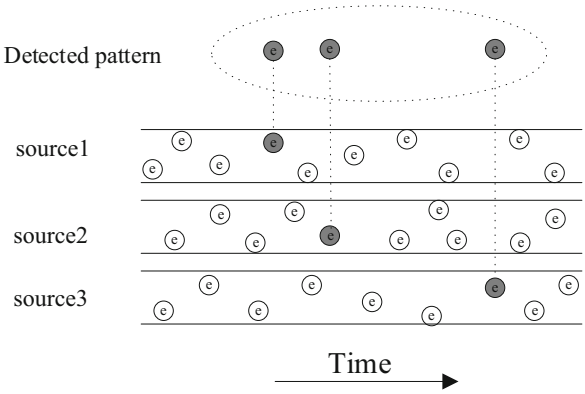

Fig. 1 Example of event pattern detection from three different streams

jobs, leveraging distributed frameworks such as Hadoop. However, it is becoming increasingly inefficient to store all the data in its whole entirety. Many businesses also require answers as soon as the data becomes available. Furthermore, the businesses are not primarily interested in raw data, but rather in the high-level intelligence that can be extracted from it. As a response, systems were developed that can filter, aggregate and correlate data, and notify interested parties about its results, abnormalities, or interesting facts.

The latest advance in such systems is the development of high performance complex event processing engines that are capable of detecting patterns of activity from continuously arriving data. As illustrated in Fig. 1, CEP systems receive continuous streams of events from multiple data sources over underlying network, they discover patterns of interest among the events, and notify the users.

In this paper, we focused on distributing complex event processing to achieve better scalability. We design a CEP system that explores the possibilities of parallelizing complex event detection, and also achieves higher throughput, while retaining low detection latency. Furthermore, the implementation would try to obtain a balance between CPU, memory and available network usage, and also explore some optimizations, which could be applied to event detection for higher performance.

The rest of the paper is organized as follows: Sect. 2 introduces the related work of this research; Sect. 3 introduce Event and temporal model; Architecture design and implementation issues will be described in Sects. 4 and 5 describes the evaluation of the system; Sect. 6 concludes this paper.

2 Related Work

CEP has been extensively studied in active database [2]. These works focused on evaluation of complex patterns in traditional databases which cannot accommodate real time and streaming requirements of current applications. In [3], a declarative

event language-SASE is proposed to define, filter, aggregate and correlate events from stream data. However, NFA (non-deterministic finite automata) implementation of SASE event queries can't work efficiently when domain size in event definition is large and event objects of the same type are considered, as it focuses mainly on RFID applications. CEDR [4] is an event streaming system with a declarative query language which cannot define multiple object-oriented event queries. Distributed detection of pattern has been first explored in [5], with a very simple language. An important contribution comes from [6], where the authors study how patterns can be rewritten for efficient distribution. Cayuga [7] is a centralized general purpose event processing system that allows event detection through a small number of well-defined operators. It is developed from a pub/sub system which is focused on common sub-expressions searching amongst multiple patterns. In Cayuga, a SQL-style event definition language is proposed. Objects involved in Cayuga event queries are single object-oriented. SAMOS [8] is a petri Nets based systems which are able to support concurrency but are very complex to express and evaluate. Plan based CEP across distributed sources has been studied by Mert [9]. The STREAM system [10] from Stanford University uses the CQL [11] query language for relational style queries. The STREAM system uses adaptive algorithms for ordering of pipelined operators, to minimize processing cost.

3 Event and Temporal Model

Event in our system are triples $< p, t_0, t_1 >$ that continuously arrive from external sources at some specified mean event arrival rates. Throughout the paper we refer to these events as to external or primitive events. Here p corresponds to event's payload, which contains values that correspond to event's attributes, as specified in an event schema. The values are populated by an input source and their types and order of occurrence must be the same as declared in the event schema.

Apart from the payload, each event carries a pair of timestamps t_0 and t_1. These are arbitrary integer values representing respectively the start timestamp and the end timestamp of the event, where timestamps follow logical time (i.e. they do not need to correspond to system time). We require that start timestamp is smaller or equal to the end timestamp, i.e. $t_0 \leq t_1$. Furthermore, we require all events on the same input stream to be ordered by their end timestamps. Since events include two timestamps, some care has to be taken when defining their ordering. To specify the semantics of various operators, we need to make this clear. First, let us specify the set of all possible external events as:

$$\mathrm{E} = \{< p, t_0, t_1 > \wedge\, t_0, t_1 \in \wedge\, t_0 \leq t_1\} \tag{1}$$

Then we can define an ordering on events $\prec$ as a tuple, where $<$ is the usual ordering on natural numbers and:

$$\forall e, e' \in E. e \prec e' \Leftrightarrow t_1 < t_0' \text{ where } e = < p, t_0, t_1 > \text{ and } e' = < p', t_0', t_1' > \quad (2)$$

This means that an event e precedes another event e' only if the end timestamp of e is smaller than the start timestamp of e'.

4 Event Processing Framework

In this section we will describe the Step (Storm Complex Event Processing) system. To implement a distributed complex event processing system that works over streams, we use the Storm stream processing framework. This section describes the Storm framework, event streams and event processing topology.

4.1 Storm Framework

Storm [11, 12] is a distributed real-time stream processing platform that can be used to assemble and execute stream processing elements. The applications that run on top of Storm cluster are called topologies.

A topology in Storm is a data flow graph of computation, which consists of elements called Bolts and Spouts, connected together with streams. Streams are unbounded sequences of tuples, and a tuple is a list of values of any type. The sources of streams are Spouts, which read data from an external source, for example stock exchange, sensors or program logs. The streams are consumed by Bolts, which do some processing and possibly emit new streams that can be consumed by further Bolts. The processing done at Bolt can be anything, from filtering or aggregation, to saving tuples to a database. An example of a topology can be seen in Fig. 2.

At runtime, each component of a topology will run within a number of tasks, which are specified by a parallelism of that component. Every task for the same component executes the same blueprint code, but is a different instance and runs in a separate thread of execution. A task receives a tuple on its input queue, processes it and may emit new tuples to its output streams. Streams are divided between multiple tasks depending on specified stream grouping. Available stream groupings include shuffling stream in round robin fashion, replicating stream to all tasks, or shuffling stream depending on tuple attributes. An illustration of component tasks communicating over streams can be seen in Fig. 3.

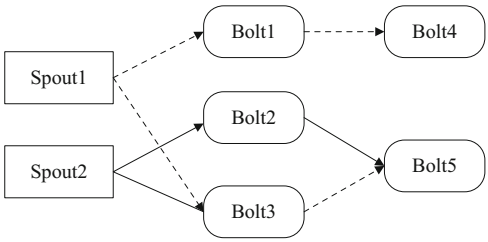

Fig. 2 A storm topology

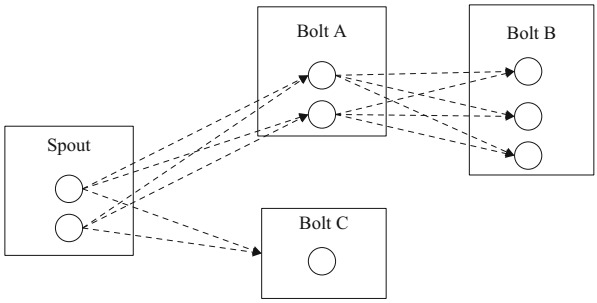

Fig. 3 Component tasks communicating over streams

4.2 Event Input Streams

Events to our CEP system arrive on continuous input streams from input adapters and detected events are consumed by output adapters. We will now briefly discuss the stream input, output streams.

The input to the system is handled by Storm components called Spouts. As seen in Fig. 4, Spouts receive events from external sources by means of input adapters, and distribute them to different Bolts corresponding to operators detecting events. A single Spout may emit input events to multiple Bolts.

Input adapters are user defined, enabling data to arrive from a variety of sources, for example sensors supplying measurements, databases providing historical data. Note that to handle high event rate, we require input adapters to be parallelizable. This is because for high input event rates more than one task for a Spout will be spawned, such that input can be consumed in parallel.

4.3 Event Output Streams

After a complex event was detected, it will be reported by a projection operator to the corresponding event sink adapter, as seen in Fig. 4. Similarly to input adapters, event sinks cannot be single objects, but should be designed to run in multiple instances. This is because for high event rates projection operators will be parallelized, and each projection task will instantiate its own sink adapter.

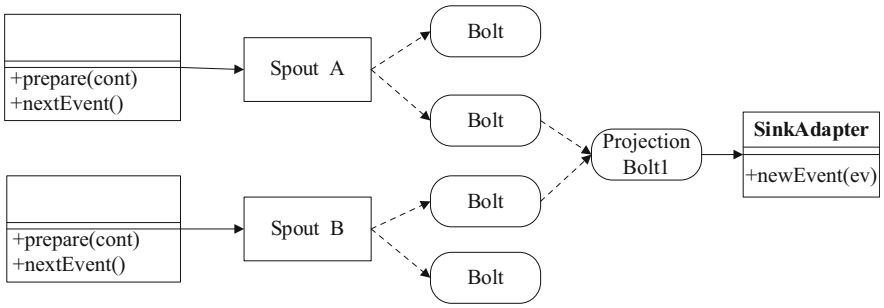

Fig. 4 The input and output of Step runtime framework

This will also be the case when different projection operators (queries) report to the same sink adapter class. Thus, it may happen that different adapter instances will run on different physical nodes. Since detected events consist of multiple external events, each having their own fields, the reporting format is different from external event format. An output complex event contains start and end timestamps, and a list of fields. A field is described by its name, value, and a name of an external event stream, from which the value originates.

4.4 Event Processing Method

The event detection happens on data-flow graphs, which is a method also used by some DSMS systems (e.g. Borealis). Data-flow graphs consist of boxes connected by streams, where each box performs some computation over its input streams and may emit some events to its output streams. It can be seen that data-flow graphs exactly resemble the Storm topology paradigm, and hence are the best design choice when using Storm as an underlying framework. From now on we will refer to event detection graphs as Step topologies.

Step topologies consist of Spouts and Bolts connected by streams with a specified grouping. We already know that one of the responsibilities of Spouts is to handle event input and projection Bolts handle event output. In fact, each topology component (whether Bolt or Spout) corresponds to some operator in the Step language and performs its function. For example, Spouts are implementations of the external operator and projection Bolts of the projection operator. An illustration of a general topology built from operators can be seen in Fig. 5.

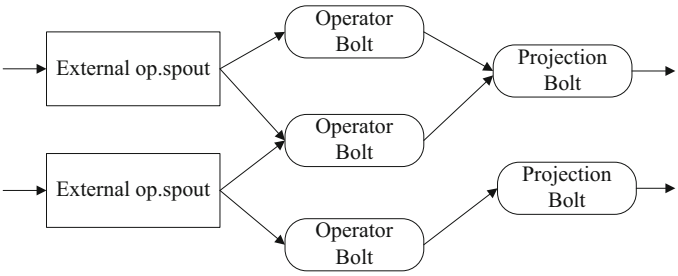

Fig. 5 A Step topology containing components that correspond to operators in the Step

5 Performance Evaluation

Our goal was to develop an efficient, scalable and fast system for complex event detection. In this section, we evaluate our Event processing framework. Our experiments were executed on PC with 4 GB memory and two Intel Core 2 Duo processors. The operating system is Ubuntu 9.1.

We will measure the number of events that the Bolt can receive per second and the network usage. The measurements of sending throughput and network utilization for different batch sizes can be seen in Fig. 6. Note that the experiment was performed over 1 Gbps LAN network. The throughput rises in a logarithmic curve, radically up to the event batch of size 100 and then increases at a slow steady rate. The corresponding network usage rises accordingly.

For small batch sizes the bottleneck is in high CPU overhead at the Spout. This is caused by generation of message IDs, determination of destination Bolt task. As we increase the batch size, the CPU usage gets smaller and the bottleneck starts shifting towards network.

We chose the batch size which yields a high sending event throughput. This allows CPU resources to be used for event detection. Figure 6 demonstrates that any size above 300 is good, as it maximizes the throughput.

6 Conclusions

In this paper, we study event detection problems over data streams which involve multiple sources. We put forward a novel framework based on Storm to support efficient complex event processing while remaining high scalability. The experimental results verify our framework has good performance and scalability in CEP query processing. Furthermore, we achieved high event detection throughput and high network usage by distributing event queries. We believe that this framework is usable and will become popular when dealing with continuous data streams and their computations.

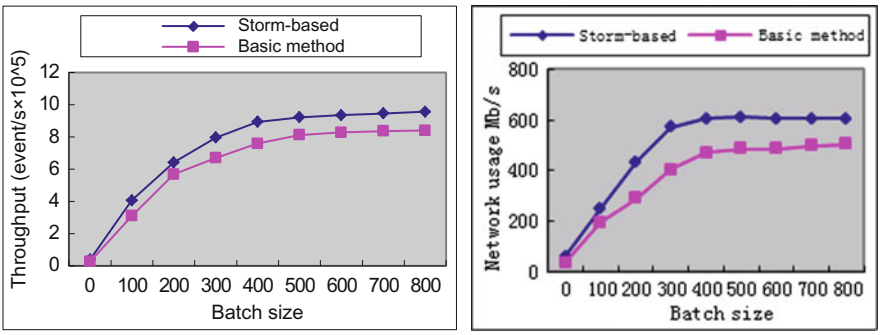

Fig. 6 Throughput and network usage experiments

Acknowledgments This project is sponsored by Hunan Provincial Natural Science Foundation of China "Context-aware and proactive complex event processing for large scale internet of things (13JJ3046)" and supported by the "complex event processing in large scale internet of things (K120326-11)" project of Changsha technological plan.

References

1. Luckham DC (2001) The power of events: an introduction to complex event processing in distributed enterprise systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
2. Zimmer D, Unland R (1999) On the semantics of complex events in active database management systems. In: ICDE, p 392–399
3. Wu E, Diao Y, Rizvi S (2006) High-performance complex event processing over streams. In: SIGMOD conference, p 407–418
4. Barga RS, Goldstein J, Ali MH, Hong M (2007) Consistent streaming through time: a vision for event stream processing. In: CIDR, p 363–374
5. Li G, Jacobsen HA (2005) Composite subscriptions in content-based publish/subscribe systems. In: Middleware'05. Springer, New York
6. Schultz-Moeller NP, Migliavacca M, Pietzuch P (2009) Distributed complex event processing with query optimization. In: DEBS'09. ACM, Nashville
7. Demers AJ, Gehrke J, Panda B, Riedewald M, Sharma V, White WM (2007) Cayuga: a general purpose event monitoring system. In: CIDR, p 412–422
8. Chen J, DeWitt DJ, Tian F, Wang Y (2000) Niagara CQ: a scalable continuous query system for internet databases. ACM SIGMOD Record 29(2):390
9. Mert Akdere, Ugur Çetintemel, Nesime Tatbul, Plan-based complex event detection across distributed sources, Proceedings of the VLDB Endowment, vol 1(1), Aug 2008
10. Arasu A, Babcock B, Babu S, Cieslewicz J, Datar M, Ito K, Motwani R, Srivastava U, Widom J (2004) STREAM: the Stanford data stream management system. In: Garofalakis, Gehrke, and Rastogi (eds) A book on data stream management
11. Arasu A, Babu S, Widom J. The CQL continuous query language: semantic foundations and query execution. Technical report, Stanford University
12. Marz N. Storm wiki. URL https://github.com/nathanmarz/storm/wiki