Smart Distributed Query Execution for Event-driven Stream Processing

Salman AHMED SHAIKH† and Hiroyuki KITAGAWA††

† Center of Computational Sciences
University of Tsukuba, Japan
†† Faculty of Engineering, Information and Systems
University of Tsukuba, Japan
E-mail: †salman@kde.cs.tsukuba.ac.jp, ††kitagawa@cs.tsukuba.ac.jp

Abstract Due to the rapid growth of stream data being generated by sensors, micro-blogs, e-businesses, etc., many organizations require on-line processing of their data for real time analysis and actionable alerts. It is not possible to process such voluminous and velocious data in real time using the traditional centralized stream processing engines. Hence distributed stream processing has emerged to facilitate such large scale real time processing. In this work we present a smart distributed event-driven stream processing approach. In contrast to the ordinary stream processing, event-driven stream processing generates query results on the occurrence of specified events only. In the basic event-driven stream processing, even when no event is raised input stream tuples are continuously processed by query operators, though they do not generate any query result. This results in increased system load and wastage of system resources. Whereas in the smart event-driven stream processing scheme, incoming tuples are processed in the presence of events only resulting in reduced system load. The proposed smart distributed event-driven stream processing utilizes the concept of smart query execution to distribute the data stream among the distributed worker nodes in the presence of events only; while in the absence of events no data is distributed as it can not generate query output. This smart data distribution can significantly reduce the network traffic in the absence of events and ultimately results in improved overall system throughput. Detailed experiments are performed to prove the effectiveness of the proposed framework.

Key words distributed data processing, data stream processing, smart query execution, event-driven stream processing

1. Introduction

With the passage of time, the number of devices connected to the internet is increasing exponentially. Gartner, Inc., forecasts that 8.4 billion connected devices will be in use worldwide by the end of 2017, up 31 percent from 2016, and will reach to 20.4 billion by 2020 [12]. These devices include sensors, GPS, smart phones, personal computers and servers. They generate continuous data, also known as data streams. Many organizations require timely processing of such data for on-line actionable alerts or for real-time analysis. For instance, 1) Car insurance companies install sensors on cars to track their position for monitoring purpose and receive location data as continuous data stream which must be processed in real time to avoid car thefts; 2) Mobile companies receive call records and data usage information from their subscribers continuously in the form of data stream which must be processed in real time for billing, etc. It is not possible to process such voluminous and velocious data in real time using traditional centralized Stream Processing Engines (SPEs). A new class of applications has thus emerged to facilitate such large scale real time data processing known as distributed stream processing systems. In this work we present a *smart distributed query execution framework* for event-driven stream processing. An event-driven query is activated (generates query output) with the occurrence of an event and is deactivated (stop generating query output) after executing for a fixed time duration or after the expiration of the event. For instance:

Endangered Species Monitoring: Africa is known to have more than 400 species of endangered animals and monitoring endangered species is an essential and critical step in their conservation [19]. Wearable sensors and surveillance cameras are usually employed to monitor the movement of these animals. Rather than recording the video of all the endangered animals continuously, which is memory costly, an event-based recording may be used. Hence, on the detection of abnormal movement of an animal from sensor stream

(which can be considered as an event in this work), wild-life experts may be interested in recording the video for some fixed time duration or as long as the abnormal movement continues.

The above mentioned event-driven processing can be achieved by join query utilizing the time-based window for the event stream, however use of the time-based window generates a lot of useless intermediate tuples which do not contribute to the query output. To address this problem, we proposed a *smart event-driven stream processing scheme* in [17]. The smart scheme uses smart windows to buffer the tuples arriving from ordinary (non-event) streams during the absence of event stream tuples. (We say the query is *inactive*.) The tuples in the smart windows buffers are flushed and processed by the downstream operators only on the arrival of tuples from the event streams. (We say the query is *active*.) In addition, the buffered tuples which get expired due to the window sizes are deleted directly from the window buffers without being processed by the downstream operators. This results in reduced system load and improved system throughput. The smart scheme is especially useful when the event stream input rate is lower than the ordinary streams which is the usual case.

In this work we propose a *smart distributed event-driven stream processing framework* to process the voluminous data which can not be handled by a centralized event-driven stream processing engine. The proposed framework employs a *task manager* to distribute the stream processing among the distributed worker nodes (dispatchers and executors) in a way to minimize the data movement among the worker nodes. Precisely, the dispatchers make use of smart windows to buffer or forward the non-event stream tuples among the executor nodes. Hence in the absence of an event, no data need to be sent from dispatchers to executors resulting in reduced system and network load. The main contributions of this work can be summarized as follows:

- A smart distributed event-driven stream processing framework capable of minimizing data movement among its worker nodes for an event-driven query.
- Detailed experimental evaluation to prove that the proposed framework can significantly reduce the network bandwidth usage and is more scalable than ordinary distributed event-driven stream processing frameworks.

The rest of the paper is organized as follows. Section 2. discusses the related work. Section 3. presents essential concepts. In section 4., the proposed smart distributed query execution framework is presented. Section 5. presents an extensive experimental study while Section 6. concludes this paper and discusses future directions.

2. Related Work

We now discuss previous related research, focusing primarily on continuous query execution and distributed processing.

There are two main execution strategies for continuous queries (CQs): timer-based [18] [21] and change-based [5] [4] [16] [3]. Timer-based (periodic) CQs are triggered only at the time specified by the user and are executed periodically for some constant time interval. Each time the timer-based CQ is triggered, it evaluates the newly arrived data. The timer-based CQs are analogous to the event-driven CQs, where the occurrence of an event or arrival of data from event stream triggers the query. On the other hand, change-based CQs are triggered as soon as new data become available. The queries in this paper are based on the CQL query language [7]. Although the main focus of CQL is change-based CQs, we can use time-based windows to achieve the functionality of event-driven CQs. For this sake, the interval between the arrivals of event stream tuples serves as the interval between consecutive query triggerings, and the size of the time-based window serves as the duration for which the query remains active. In this paper, a CQ which employs the time-based windows is called an *event-driven* query.

With the rise of the need to process voluminous data, many distributed stream processing frameworks have been proposed. Spark Streaming [2], Storm [3], Flink [9], Samza [1], Borealis [4], Heron [14] are some of the well known and commonly used distributed stream processing frameworks which also supports event-driven stream processing. However our proposed framework does not only support distributed event-driven processing but it also supports smart data distribution among the distributed worker nodes. The proposed framework is capable of minimizing the network bandwidth usage by limiting the amount of data that must be transferred among the distributed nodes for the distributed query processing. Load balancing in distributed stream processing engines is a known research issue and is handled via key grouping by many existing distributed frameworks [11] [20] [13] [8] [10] [15]. When a load imbalance situation is detected, the system starts a balancing procedure which moves part of the keys away from an overloaded worker node. The proposed smart distributed framework on the other hand primarily focus on reducing the network usage by transferring the data among the worker nodes smartly and can be used with the existing state-of-the art distributed frameworks to utilize the state-of-the-art load balancing and fault tolerance schemes.

3. Essential Concepts

3.1 Incremental Computation over Data Streams

The computation which updates its output incrementally

instead of re-computing everything from scratch for successive runs of a job with input changes is called incremental. It is important to employ incremental computation for data streams, where the data arrives continuously and at high speed and recomputing everything from scratch is expensive.

To achieve incremental computation in some of the protoype SPEs like STREAM [6] and JsSpinner [17], each stream tuple is additionally tagged as either an insertion "+" or deletion "-". These tagged tuples inform the query DAG's downstream operators to update their synopsis. E.g., when a new stream tuple s is read by the *JsSpinner* SPE, it is appended with a timestamp t and a "+" tag, thus forming an element e of the form $< s, t, + >$. It inserts element e in the window operator's synopsis. On the other hand, if an old element e' expires due to the window size it is removed from the synopsis. The window then outputs elements $< s, t, + >$ and $< s', t, - >$, which are sent to the query downstream operators' synopsis to reflect the addition and deletion of elements e and e' respectively.

3.2 Event-driven Stream Processing

Event-driven stream processing can be defined as the processing of a continuous query (CQ) activated by the occurrence of an event. It is implemented using the time-based window operator available in most of the existing SPEs (hereafter called *basic scheme*), however the use of time-based window results in the generation and processing of useless intermediate tuples which do not contribute to the query output. To solve this problem we proposed a smart query execution scheme (hereafter called *smart scheme*) in [17]. In the following we summarize the working of the basic and the smart schemes which are essential in understanding the proposed smart distributed query execution scheme.

3.2.1 Basic Query Execution

The basic query execution scheme processes the incoming ordinary stream tuples continuously however generates query output only on the occurrence of event(s) or in the presence of event stream tuples. Processing of ordinary stream tuples in the absence of event(s) keeps the system resources busy unnecessarily.

[Example 1] Query 1 shows a simple event-driven query written in CQL. In the query S1 is an event stream, with window size τ seconds, whose tuples activate the query whereas S2 is an ordinary stream with window size n. Let $\tau = 2$ seconds and $n = 500$ rows. Assume that at timestamp t_1, event stream window (W_e) is empty whereas ordinary stream window (W_o) contains 500 tuples. Since W_e is empty the query is inactive, i.e., do not generate any query result however in the basic scheme, "+" elements corresponding to the 500 ordinary stream tuples are sent to the downstream query operators (selection and join operators in case of Query 1).

```
Select S1.A, S1.B, S2.C
From S1[Range τ], S2[Rows n]
Where S1.A = S2.A
```

Query 1 Join query

Further assume that at timestamp t_2, W_e receives one tuple and W_o receives another set of 500 tuples. This causes the query to become active (generates result) and send 500 "-" elements corresponding to the expired ordinary stream elements which arrived at t_1. The query remains active for 2 seconds (i.e., for timestamps t_2 and t_3) which is the size of W_e and we call it active duration. If no event stream tuple arrive at timestamps t_3 and t_4, the query becomes inactive again.

3.2.2 Smart Query Execution

To avoid the generation of unnecessary tuples which do not contribute to the query output, we propose a smart event-driven stream processing in [17]. In the basic event-driven stream processing, ordinary stream tuples need to be processed to maintain the current status of row-based windows to guarantee correct query results on the arrival of an event stream tuple. However in [17] we identified that the corresponding "+" elements do not need to be sent to downstream operators when the query is inactive. The smart scheme uses a *smart window* to buffer the tuples arriving from ordinary stream during the absence of event stream tuples.

Smart window is a modified window operator of CQL specification [7]. The synopsis of the smart window operator is divided into two parts: *output* and *suspended*. Both the output and the suspended parts keep recent incoming tuples of the ordinary (non-event) stream. In the smart scheme, when a tuple e arrives from an ordinary stream, the system checks whether the query Q is active or inactive. If Q is active, the smart window keeps e in the *output* part and sends a corresponding "+" element to $Q's$ downstream operators. While if Q is inactive, the smart window buffers e in the *suspended* part and does not output any "+" element. On changing the state from inactive to active, the smart window of query Q generates "+" elements for all the buffered elements and sends them to the downstream operators. During $Q's$ inactive duration, the buffered elements which expire due to the window size are deleted directly from the smart window without the need to generate "-" tuples for them, resulting in reduced system load.

[Example 2] Once again consider Query 1 and the parameter values given in Example 1. At timestamp t_1, the query is inactive due to the absence of any tuple in W_e. In the smart query execution scheme, the 500 incoming ordinary stream tuples at t_1 are buffered in the suspended part of

the smart window and no "+" element is sent to the downstream operators. The query activates with the arrival of an event stream tuple at t_2 and the 500 tuples buffered in the suspended part of the smart window expires due to window size, however in contrast to the basic scheme, the expired tuples are deleted directly from the smart window without the need to send corresponding "-" tuples to downstream operators. Since the query is active at t_2 and remains active for $\tau = 2$ seconds, the ordinary stream tuples arriving at t_2 and t_3 contribute to the query output and deactivate at t_4 if there is no tuple in W_e.

The smart scheme is especially useful when the event stream has lower arrival rate than ordinary stream. Consider a simple join query (Query 1) and let I_e: event stream tuples arrival interval (sec), W_e: event stream window size (sec), R_o: ordinary stream arrival rate (tuples/sec), and W_o: ordinary stream window size (no. of rows). The main advantage of the smart scheme comes from the ordinary stream tuples deleted directly from the suspended part of smart window. The number of ordinary stream tuples arriving during the inactive duration can be given by $(I_e - W_e) * R_o$. Hence the smart scheme is advantageous if the number of ordinary stream tuples arriving during the inactive duration is greater than the ordinary stream window size W_o, which can be given by: $(I_e - W_e) * R_o > W_o$. For details, refer our work [17].

4. Smart Distributed Event-driven Stream Processing

This section presents our proposed smart distributed event-driven stream processing framework. The proposed framework is capable of reducing network load by minimizing data movement between distributed nodes and can achieve better scalability than the existing distributed event-driven stream processing frameworks. In the following we will talk about the main components of the proposed framework and the smart distributed query execution for event-driven stream processing.

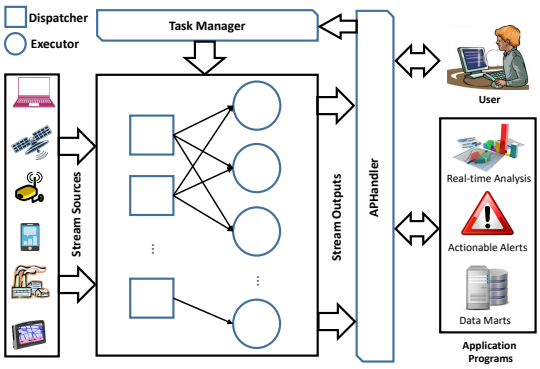


Figure 1 Smart Distributed Query Execution Framework

4.1 Distributed Framework

The main focus of the proposed framework is the reduction of network bandwidth usage and scalable data stream processing. Figure 1 presents our proposed framework comprising of the following four main components 1) APHandler, 2) Task manager, 3) Dispatcher, and 4) Executor.

4.1.1 APHandler

This is an application programmer interface (API) for accepting user queries and registering them to the task manager. The APHandler is also responsible for accepting query results from executors, integrate them and send them to user or application programs. Depending on the number of executors and the size of query output, the number of APHandler nodes for query output integration may vary.

4.1.2 Task Manager

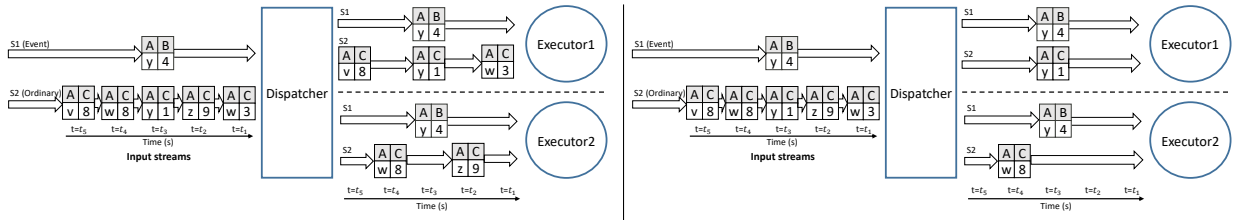
This component receives user queries from APHandler, parses them into query intermediate representations and converts them into directed acyclic graph (DAG) of operators. The task manager also keeps track of the active worker nodes (dispatchers and executors) and assigns them tasks. A worker node is considered active if it is ready to accept query. The task manager is provided with a configuration file with each query which is submitted by end-user while registering query, specifying the number of executors and dispatchers to be used for each query. A worker can be assigned the role of a dispatcher or an executor dynamically. One dispatcher can feed data to several executors. The amount of data dispatched by a dispatcher is limited by its network adapter and the category of ethernet cable, whereas the amount of data which can be processed by an executor is limited by the query complexity assigned to it. Hence, depending upon the available resources and query complexity, end-user can determine the number of dispatchers and executors.

4.1.3 Dispatcher

Dispatchers in the proposed framework behave like a stream source and can vary from one to several. Generally dispatchers will read tuples from an external source and emit them to the executor(s). Dispatcher can emit more than one stream for a query. Dispatchers employ different data distribution policies, which are discussed later. At the time of role assignment, the task manager specifies the stream source(s), distribution policies and the destination(s) of input stream(s) for each dispatcher. Dispatchers play main role in reducing the network bandwidth usage in the proposed framework which is discussed in Sec. 4.2.

4.1.4 Executor

Executors are the main processing units of our proposed framework. Generally, executors will execute the query DAGs and perform operations including selections, projections, joins, etc. Depending upon the complexity of the



(a) Ordinary Dispatcher Distribution

(b) Smart Dispatcher Distribution

Figure 2 Dispatcher data distribution

query, a query DAG may be divided among multiple executors. The executors output is available to end-user via APHandler.

4.2 Smart Distributed Query Execution for Event-driven Stream Processing

As discussed in section 3.2.2, the *smart query execution scheme* makes use of a smart window for ordinary (non-event) stream to buffer the stream tuples in the absence of event stream tuples. The smart window releases the buffered tuples to downstream operators only on the occurrence of an event.

We utilized the *smart query execution scheme* to reduce the network load which ultimately results in improved throughput and better scalability. In the proposed smart distributed framework, the dispatchers read the input data streams, however in the absence of event stream tuples, no input stream tuple is sent from dispatchers to executors. The ordinary (non-event) stream tuples during the query inactive duration are buffered in dispatchers and sent to executors only on the arrival of event stream tuples. The ordinary stream tuples are deleted directly from the smart window buffer (which is located in dispatchers) on their expiration without the need to send "-" tuples to executors, resulting in reduced network load.

[Example 3] Consider an event-driven join query (Query 1) which is to be processed by a distributed stream processing framework. Assume that the stream S1 is an event stream, with window size $\tau = 2$ seconds, whose arrival activates the query while S2 is an ordinary stream with window size $n = 1$. Furthermore assume that the dispatcher employs round-robin distribution scheme. In the distributed framework, the dispatcher must distribute incoming data streams among all the executors in a way to guarantee correct join output. One simple approach, which is used in this example, is to replicate stream S1 to all the executors and distribute stream S2 in a round-robin fashion among the executors. Hence in the basic dispatcher distribution in Fig. 2(a), the only tuple of S1 <A:y, B:4> is replicated to both the executors while the tuples of stream S2 are distributed in the round-robin fashion among the two executors.

However in the proposed smart distributed framework, dispatchers employ smart windows for either buffering or forwarding data streams to executors for join execution. The smart windows in dispatchers buffer the stream tuples in the absence of event stream tuples. Hence no tuple is forwarded to executors at timestamps t_1 and t_2 as shown in Fig. 2(b). At timestamp t_3, the query activates with the arrival of S1 tuple <A:y, B:4>, hence <A:y, B:4> is forwarded to all the executors while S2 tuple at timestamp t_3, i.e., <A:y, C:1>, is sent to the Executor1. Since the event stream window size is 2 seconds, the query remains active at timestamp t_4, causing the dispatcher to forward the stream S2 tuple at timestamp t_4, <A:w, C:8> to the Executor2. At timestamp t_5, the S1 tuple <A:y, B:4> expires, which deactivates the query, hence no tuple is sent to any executor. Comparing Figs. 2(a) and 2(b), one can easily find that the proposed smart distributed event-driven stream processing can significantly reduce the data that need to be sent over network, resulting in reduced network load and improved throughput and scalability.

In the proposed distributed framework, the main query processing is done by executor nodes. On the other hand, the worker nodes which are assigned the role of dispatcher reads the data stream tuples from one or more external source(s), manages their window operators and distributes them to its assigned executor node(s). The user can specify the data distribution policy to distribute the stream tuples among the executor nodes including 1)replicate: each stream tuple is forwarded to all the executor nodes, 2)roundrobin: stream tuples are evenly distributed among the executor nodes in round robin fashion, and 3)hashing: stream tuples are hashed among executor nodes with respect to primary key or other suitable attribute. For the case where one input stream need to be dispatched by multiple dispatchers, hashing is employed to distribute input stream among dispatchers in addition to one of the above distribution policies for the distribution of input stream among executors.

The worker nodes which are assigned the role of executor need to process incoming data streams from the dispatchers using the smart query execution scheme discussed in Section 3.2.2 only during the query active duration. Since the smart query execution scheme is capable of reducing the system load by limiting the movement of useless intermediate tuples

```
Select S1.A, S1.B
From S1[Rows n]
Where S1.B > 100
```

Query 2 Selection query

```
Select S1.A, S1.B, S2.C
From S1[Rows n], S2[Range τ]
Where S1.A = S2.A
```

Query 3 Join query

among query operators, it results in increased throughput. Furthermore, in case of complex queries where a query DAG is divided among multiple executors, the smart query execution need to transfer minimum amount of data to the next executor saving significant amount of network bandwidth.

5. Experiments

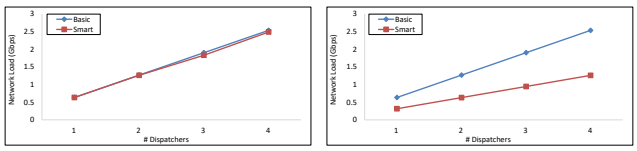
5.1 Experimental Setup

For the sake of experiments we used our locally developed prototype SPE, JsSpinner [17], which enables users to register CQL queries. JsSpinner supports both the basic and the smart event-driven stream processing schemes. The experiments are performed on HP BladeSystem c7000 server with 10 nodes, where each node is equipped with Intel Xeon 20 core processor (ES-2650 v3 @ 2.3GHz), 6 GB RAM and 10 Gbps Ethernet networking card. Each node is operated by Ubuntu 14.10 OS. For the experiments with higher number of executors than the number of nodes, multiple executors are deployed as separate processes on separate cores of the same node.

Stream Data Generation and Join Processing: For the experiments, two synthetic data streams (S1 and S2) are used with schemas S1(A, B) and S2(A, C), respectively. Here A is a common string attribute of both the streams, while B and C are the integer attributes. The data streams are generated using random strings for the string attributes and random integer values for the integer attributes. For the evaluation we used Queries 2 and 3. Query 2 is a simple selection query while Query 3 is a join query with S1 as an ordinary stream and S2 as an event stream. In order to guarantee correct join processing and efficient load balancing in Query 3, event stream S2, which has low arrival rate, is forwarded to all the executor nodes where as ordinary stream S1 is distributed using round-robin policy among the executor nodes. In the Query 3, arrival of data from S2 activates the query. The query remains active for the duration of the time-based window size. Unless otherwise stated, the following default parameter values are used in the experiments: I_e = 5 seconds, W_e = 1 second, R_o = 500k tuples/second and W_o = 1000 rows.

5.2 Experimental Evaluation

The experimental evaluation is divided into network load comparison of the proposed framework (Smart) with the basic distributed event-driven stream processing (Basic) and measurement of the scalability of the proposed framework.



(a) I_e = 1s (b) I_e = 5s

Figure 3 Network Load - Basic Vs. Smart Distributed Schemes

5.2.1 Network Load

This section shows that the use of the proposed framework can significantly reduce the network load which results in improved throughput. Here the *network load* can be defined as the amount of data that need to be sent by dispatchers to executors to process the query. Figures 3(a) and 3(b) compare the network load in gigabit per second (Gbps) for I_e values 1 second and 5 seconds respectively for the Query 3. In both the figures, the network load increases with the increase in number of dispatchers which is due to the increase in the number of connected dispatchers and executors. Here one dispatcher dispatches data to four executors.

In Fig. 3(a), the network load for the basic and the smart schemes is same. This is due to the fact that at I_e = 1 second and W_e = 1 second, the query remains active all the time and the dispatchers need to send data continuously to the executors, which is the case of basic distributed scheme. On the other hand in Fig. 3(b) with I_e = 5 seconds, the smart scheme results in reduced network load for all values of number of dispatchers. This is due to the fact that with I_e = 5 seconds and W_e = 1 second, the query remains active for 1 second while inactive for 4 seconds during which dispatchers do not send any data to the executors, hence resulting in reduced network load for smart scheme in Fig. 3(b).

5.2.2 Scalability

Next we perform experiments to evaluate the scalability of the basic and the smart event-driven query execution schemes in the distributed environment in terms of system load and maximum system throughput. The *system load* can be defined as the total number of tuples processed by all the query operators, whereas the *maximum system throughput* can be defined as the total number of input streams (including event and ordinary) tuples processed by system per unit time. One of the obvious benefit of distributed processing is the scalability as can be observed from Fig. 4. Since Query 2 is a simple selection query, we can achieve better throughput for this query compared to Query 3 which is a join query. Furthermore, our distributed framework scales quite linearly with the increase in the number of executors.

Next we perform experiments to compare the maximum system throughput and the system load of the basic and the

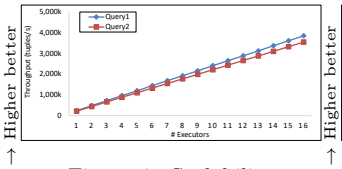


Figure 4 Scalability

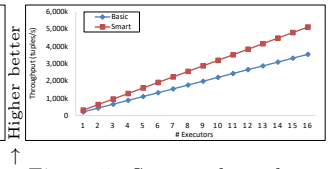


Figure 5 System throughput

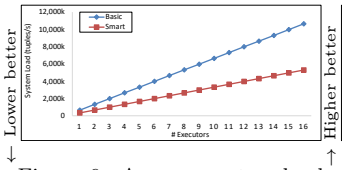


Figure 6 Average system load

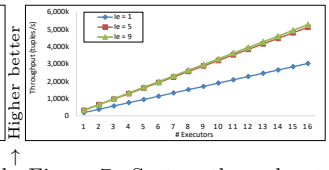


Figure 7 System throughput

smart distributed schemes for the Query 3. Figure 5 shows that the smart distributed query execution clearly outperforms the basic distributed query execution. This difference in throughput can be explained through Figure 6 which compares the system load of the basic and the smart schemes. Lower system load enables the system to process higher number of incoming tuples, resulting in improved system throughput. Here again we can observe that the proposed smart distributed query execution framework scales linearly. Finally experiments are performed for different values of I_e. From Figure 7, it is clear that the system throughput is lower for the lower values of I_e and vice versa. It is because for the higher I_e values the query remains inactive for long duration resulting in reduced network traffic and the direct deletion of a majority of tuples from the smart window. This let the system process higher number of stream tuples resulting in improved system throughput.

6. Conclusion and Future Work

This work proposes a smart distributed query execution framework for data streams which is an event-driven stream processing approach. The proposed framework employs a task manager to distribute stream processing among the distributed worker nodes (dispatchers and executors) in a way to minimize the data movement among worker nodes. Dispatchers make use of smart windows to buffer or forward the non-event stream tuples among executor nodes. Hence in the absence of an event, no data need to be sent from dispatchers to executors resulting in reduced system and network load. Experiments prove that the proposed framework can significantly reduce the network bandwidth usage and is more scalable than basic distributed event-driven stream processing. In the future we plan to integrate our proposed framework with the existing state-of-the-art distributed stream processing frameworks in order to combine the strengths of both the frameworks, i.e., smart distributed processing of the proposed framework and the load balancing and fault tolerance of the existing frameworks.

Acknowledgments

This research was partly supported by the program "Research and Development on Real World Big Data Integration and Analysis" of the RIKEN, Japan.

References

[1] The Apache Software Foundation., Apache Samza. `http://samza.apache.org/`, 2017. [Online; accessed 27-April-2017].

[2] The Apache Software Foundation, Apache Spark. `http://spark.apache.org/`, 2017. [Online; accessed 27-April-2017].

[3] The Apache Software Foundation, Apache Storm. `http://storm.apache.org/`, 2017. [Online; accessed 27-April-2017].

[4] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[5] D. J. Abadi, D. Carney, et al. Aurora: A new model and architecture for data stream mgmt. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *STREAM: The Stanford Data Stream Management System*, pages 317–336. 2016.

[7] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. Technical report, Stanford InfoLab, 2003.

[8] C. Balkesen, N. Tatbul, and M. T. Özsu. Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 15–26, 2013.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[10] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proc. of the 2013 ACM SIGMOD*, SIGMOD '13, pages 725–736, 2013.

[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[12] I. Gartner. Technology research, gartner, inc. `http://www.gartner.com/newsroom/id/3598917`, 2017. [Online; accessed 27-April-2017].

[13] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539, 2014.

[14] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, et al. Twitter heron: Stream processing at scale. In *Proc. of the ACM SIGMOD*, pages 239–250, 2015.

[15] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. *CoRR*, abs/1504.00788, 2015.

[16] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Dec 2010.

[17] S. A. Shaikh, Y. Watanabe, Y. Wang, and H. Kitagawa. Smart query execution for event-driven stream processing. In *2016 IEEE BigMM*, pages 97–104, April 2016.

[18] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD*, SIGMOD '92, pages 321–330, 1992.

[19] W. Tracking and Monitoring. Wildlife Act. `http://wildlifeact.com/`, 2017. [Online; accessed 17-July-2017].

[20] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 791–802, 2005.

[21] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proc. 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012.