Siddhi: A Comprehensive Architecture for Stream Processing

S. Suhothayan, I.L. Narangoda, K. Gajasinghe, S. Chaturanga and V. Nanayakkara
Department of Computer Science and Engineering
University of Moratuwa
Moratuwa, Sri Lanka

***Abstract*—Complex Event Processing (CEP) is one of the most rapidly emerging fields in data processing. Processing of high volume of events to derive higher level events is a vital part of several applications. The use-cases found in Business applications, financial trading applications, operational analytics applications and business activity monitoring applications are directly related to complex event processing.**

This paper discusses different design decisions associated with CEP Engines, and proposes to improve CEP performance by using a stream-processing-style pipelines. Furthermore, the paper discusses Siddhi, a CEP Engine that implements pipelines for stream processing and presents a performance study that exhibits that Siddhi CEP Engine has significantly improved performance. Primary contributions of this paper are, a critical analysis of the CEP and Event Stream Processing engine architecture and identifying areas for improvements, implementing those improvements through Siddhi, and demonstrating the soundness of those suggestions through empirical evidence.

***Index Terms*—Event Stream Processing, Complex Event Processing, Events, Data Processing**

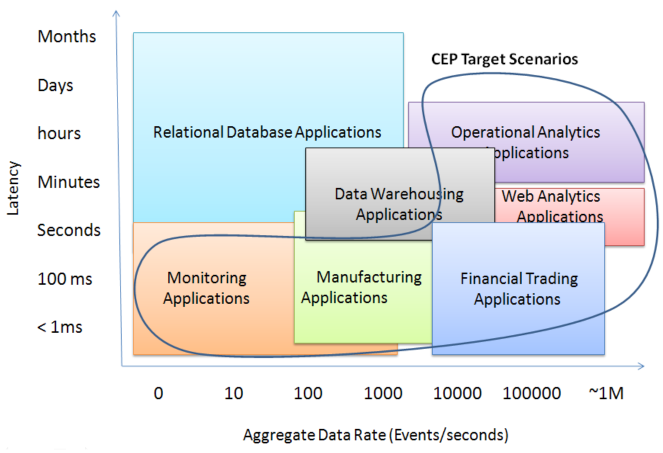


Figure 1. Use cases of Complex Event Processing

I. Introduction

During the last half a decade, Complex Event Processing (CEP) [1] has become one of the most rapidly emerging field in data processing. The major task of the CEP is to identify meaningful patterns, relationships and data abstractions among unrelated events and fire an immediate response. Due to massive amount of business transactions and numerous new technologies, like RFID (Radio Frequency Identification), it has now become a real challenge to provide real time Event driven systems that can process data and handle high input data rates with near zero latency. In this paper, we are proposing a high performance Complex Event Processing Engine which performs Event processing in an efficient manner.

As shown in Figure 1, Complex Event Processing is used in applications such as Monitoring, Manufacturing and Financial Trading that requires low latency, while applications such as data warehousing, web analytics and operational analytics that need to handle higher data rates. Both these categories of applications can utilize CEP Engines to efficiently perform their tasks. Due to these real time requirements, producing a real time Event-driven system that can handle high input data rates and process data with near zero latency has posed major challenges.

Siddhi addresses some main concerns of Event processing world where there is an absolute need of processing huge flood of Events that may go well over one hundred thousand Events per second with a near-zero latency.

Goals of this paper are to critically evaluate decisions taken at CEP and Event Stream Processing Engine design and to present Siddhi CEP Engine that incorporates several improvements covered at the discussions. To that end, next section presents a survey of CEP Stream processing Engine designs, and the section 3 describes the design and the implementation of Siddhi. Thereafter, section 4 presents a performance comparison of Siddhi and Esper [2], where latter is an open source, established CEP Engine.

Primary contributions of this paper are, a critical analysis of the CEP and Stream Processing Engine architectures and identifying areas for improvements, implementing those improvements through Siddhi, and demonstrating the soundness of those suggestions through empirical evidence.

II. Related Work

In relation to previous work we can find many papers and projects on Complex Event Processing and Stream Processing systems. Here we discuss a representative subset of those projects and their architectures to come up with an optimal design for a high performance Complex Event Processing Engine. When it comes to processing large number of data, the obvious decision is to use databases and perform data-mining on the collected data. But in the context of processing incoming continuous data streams, using databases will not be effective [3] as these database systems have an architecture which stores all incoming events in a database and then query them whereby significantly affecting the performance.

Therefore to overcome this issue, while leveraging the power of databases and data-mining the field of stream databases [4]–[6] emerged. Systems like these have very powerful query languages, typically subsuming SQL with provisions for sliding windows and stream grouping features. TelegraphCQ [4] is a such system that was designed to provide Event processing capabilities alongside the relational database management capabilities. In TelegraphCQ they have utilized PostgreSQ [7], an open source database, modifying its existing architecture to allow continuous queries over streaming data [4]. Since TelegraphCQ was designed with a storage subsystem to exploit sequential write workload and with a broadcast-disk style read behaviour, queries accessing data spans memory and disk raising significant Quality of Service issues. This eventually leads to deciding what work to be dropped when the system is in danger of falling behind the incoming data stream. Further, though these query languages are powerful, they are not optimized for processing sequential patterns that occur frequently in the target applications, and all its expressiveness only comes with the trade-off of their performance. TelegraphCQ has also being improved and commercialized into Truviso [8] where enabling storage of historical data by integrating them with relational databases. Still these systems have failed to perform well when it comes to real complex event processing tasks in real time.

On the other hand there are also Event Stream Processing engines that follow the model of publish-subscribe systems [9], [10]. They support query languages that has a very limited expression power and allow only simple selection predicates applicable to individual Events in a data stream. These systems trade expressiveness for performance.

The basic functionality of the Complex Event Processor is to match queries with Events and trigger a response. Queries describe the details about the Events that the system needs to search within the input data streams. The Event Stream Processing engines are enough to handle all the primitive CEP tasks. In contrast to traditional databases, Event Stream Processing engines operate with stored queries running constantly against extremely dynamic data streams. In fact, these Event processing systems are an upside down view of a database.

One such notable project is Aurora [11] where their developers describe it as a general-purpose data stream management system. Its primary goal is to provide a single infrastructure that can handle real-time streaming in applications such as monitoring. As described in Figure 2, the overall system architecture of Aurora is based on the "boxes-and-arrows" process and work-flow systems [12]. Here the data flows through the system as Tuples, along pathways, which are arrows in the model. The data gets processed at operators, which are the boxes and after the last processing component they are delivered to an application for protestation [13].

Systems like these can optimize themselves by combining

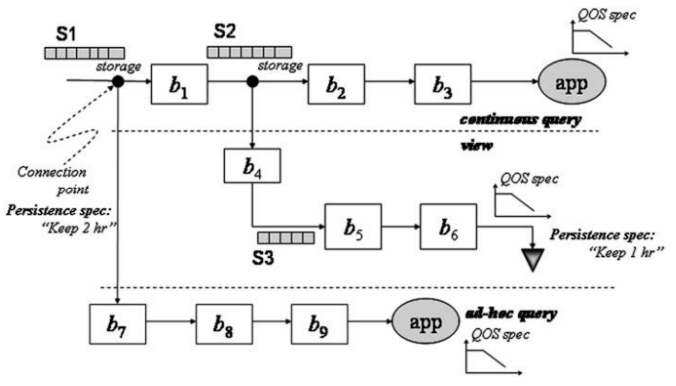


Figure 2. Aurora Architecture

two boxes into a single larger box, reordering boxes, and load shedding [14]. STREAM CEP [15], [16] also follows a similar structure to Aurora [11] where the incoming Data Streams are processed using a pipeline architecture. Here the intermediate states of the queries are stored in queues and passed on to the next processor for further processing. The continuous queries of STREAM remain active until they are explicitly deregistered. The results of these continuous queries are transmitted as output data streams. Similar to Aurora, though STREAM has many advantages in the stream processing, currently it has several limitations. The most important limitations are merging sub expressions and handling intermediate queues. As illustrated in Figure 3, the number of Tuples in a shared queue at a particular moment of the STREAM Query Plan depends on the rate at which Tuples are added to the queue, and the rate on which the slowest subscribed operator consumes Tuples. Here, monitoring which subscribed operator has not yet processed a Event Tuple in the shared queue, which Event Tuples has been processed by all the subscribed operators of the shared queue and ready for deletion, are some major monitoring processes that needs to be done related to stream processing. This drastically reduces the performance of these systems [6]. Though STREAM CEP has now officially wound down [15], it is being used as base for Coral8 CEP [17] which is now a part of Event Zero CEP [18].

Further the CEP Engines like Esper [2], Cayuga [19] and SASE [20] use a different approach to handle Complex Event Processing. Since these systems mainly focus on the state machine implementations to identify patterns in the incoming streams, they have not optimized themselves in stream processing. Where in SASE, since it outputs all the Events that match a pattern query, its architecture does not allow those events to be streamed into a new query. On the other hand in systems like Cayuga, since they are only having one core processing thread, it is not efficient enough for processing many thousand of Events in a higher speed. Therefore stream based systems; due to the nature of their architecture, they have the ability to provide much better performance than a typical CEP Engine.

Some CEP implementations such as TelegraphCQ [4] and Cayuga [19] uses priority queues to handle 'out of order

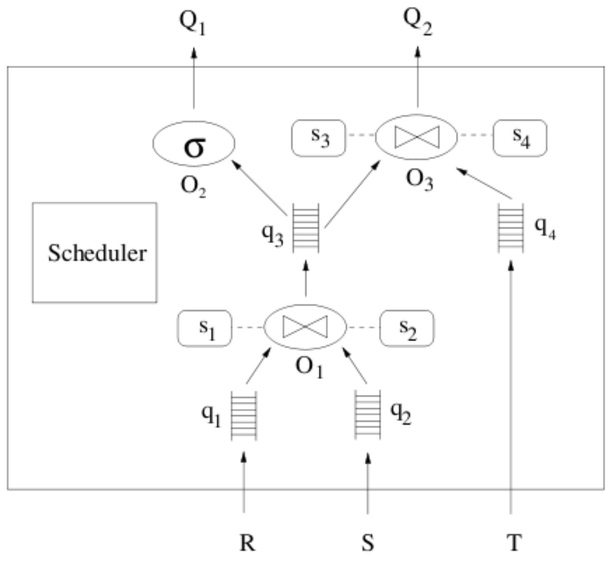


Figure 3. STREAM Architecture

Event arrivals', and in stream based systems this issue has been seamlessly tackled by the use of existing queues in the systems. The use of queues also helps to manage the memory requirements of the stream based systems just by adjusting the queue size [21] over time. Load-shedding techniques can be used to handle failures of the systems. Here load shedding is one of the most important optimization technique used in stream based systems [13], where the number of Events presented for processing are reduced to handle the overloaded states.

As discussed above, though stream based systems have several advantages, their performance depends on many decisions such as;

- Threads per Query or thread per task of execution,
- Query optimizations when leveraging multi core processors
- Efficiently streaming Events
- Temporal window implementation

In the next section we will describe Siddhi, which was developed after critically evaluation of each design decisions in order to improve the performance.

III. Methodology

A. System Overview

The basic functionality of a complex event processing engine is to match queries with events and trigger a response. These queries describe the details about the events that the system needs to search within the input data streams. Unlike traditional systems like typical Relational database Systems which are operating with hundreds of queries running for short durations against stored static data, event driven systems operate with stored queries running constantly against extremely dynamic data streams. In fact, an event processing system is an upside down view of a database. The tasks of CEP Engine is to identify meaningful patterns, relationships and data abstractions among unrelated events and fire an immediate response such as JMS message or a SOAP response or simply an alert message. Here we discuss the design and implementation of such a system–Siddhi CEP–which provides high throughput and efficiency. The high level architecture of

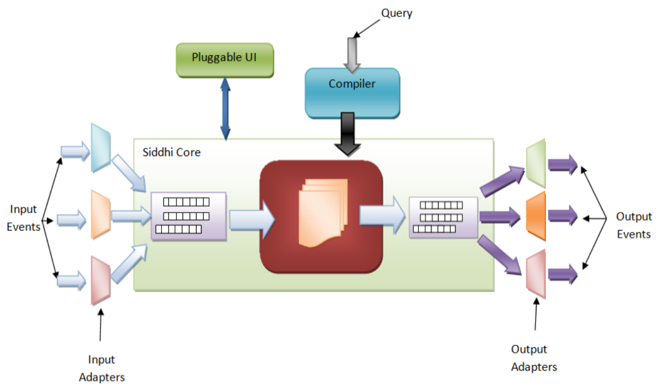


Figure 4. Siddhi System Architecture

Siddhi CEP is presented in Figure 4. In a very high level Siddhi can be viewed as follows. The events come in to the system, and goes through several predefined customer rules (queries). If a match occurs on any one of the rules, system triggers the corresponding complex event. In other words, data go through a set of queries and the matching data will be triggered as complex events. So as explained above the system functionality can be split in to several sub-categories as follows. Capture incoming events from several event sources (i.e. stock exchange, Enterprise Service Bus etc.) through input adapters. Input adapter will pass the events to a thread safe blocking queue. There are several threads which does the dispatching events from event queue to the event processor. Event processor/processors will process events and when a match occurs it will pass to a callback method where customer can do whatever he wishes with that event.

B. System Design

When designing Siddhi we have taken some critical design considerations which have eventually made Siddhi much better.

Siddhi receives events through its input adapters. The major task of input adapters is to work as an interface for event sources to send events to Siddhi. These input adapters can be of different types where each accepts different types of events such as XML events, POJO (Plain Old Java Object) Events, etc. and convert all of those different types of events in to a particular data structure, a Tuple, for internal processing. We decided to use Tuple as the data structure to represent an Event because Tuple is very simple and since it closely resembles a row in relational database tables, by using Tuples, Events in the data stream can be simply mapped into a relational table. This will allow us to use SQL like queries and introduce

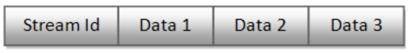


Figure 5. Siddhi Event Tuple

relational database optimization techniques in the system. Further retrieving data from a Tuple is also quite simple compared to other alternatives (like XML) which help Siddhi to process events much faster by minimizing the overheads in accessing data in each event. Siddhi-core is the heart of the

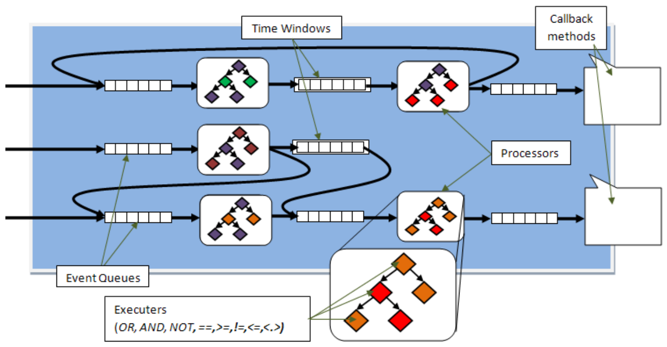


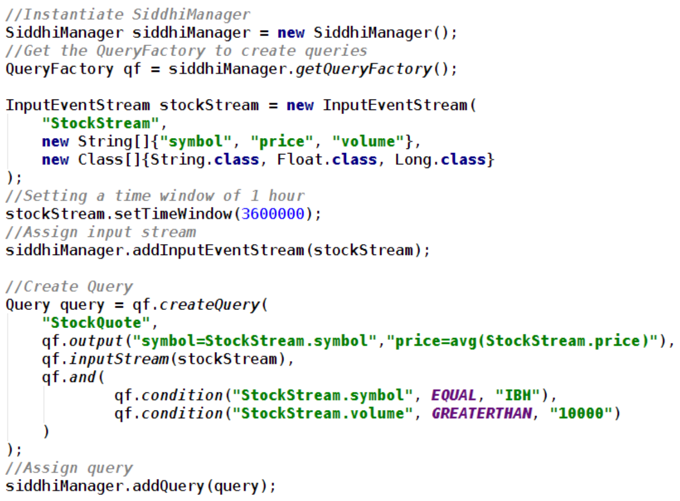
Figure 6. Siddhi Core

Siddhi complex event processing engine. All the processing is done by this component. The Siddhi-core consists of many sub components as shown in Figure 6. The input events to the core are placed in queues before processing, and the processors use the producer-consumer architecture to fetch events from the queues and process them to find matching occurrences. When a matching event was found the processors will create the appropriate output events according to the user query and place them in the queues for dispatching or further processing. As we can see from the literature, some complex processing engines such as Cayuga, uses more of a single processor architecture where just a single thread is used for core processing. Though this may produce some advantage over the complexity of the query we can still achieve the same complexity by processing the complex queries like pattern and sequence marching using a single processor within the pipeline architecture. The efficient implementations of the pattern and sequence processors are out of scope of this paper.

Leveraging multi-core processors is essential for CEP engines to have high throughput. Therefore, when comparing stream based systems to Esper and Cayuga, Stream based systems seems to be more attractive. Its quite obvious that multi-threading achieves higher performance, but it also needs a careful division of tasks between threads because processing happens in real-time. There could be some erroneous results due to arrival of events out of order, etc. So, we have studied various implementations, and found that allocating each query a thread produces much better performance with 100% accuracy.

Though thread-per-query model produces high performance it also has some drawbacks when it comes to resource utilization. This is because in most cases many complex queries have common sub queries, and Stream based systems usually ends up by running many duplicate sub queries at the same time. This issue was appropriately handled in Siddhi using pipeline architecture model and using transparent query object models. The Query object model is the internal representation of the Siddhi query, which the Siddhi-core can understand. Due to the transparent nature of the Siddhi Query object module, it facilitate the users to easily understand what internally happened with in Siddhi whereby enabling them to write much better queries.

Figure 7 provides a Siddhi Query snippet which is closely related to SQL. This decision was taken in order to make the object model to fall in-line with the relational algebraic expressions by following SQL standards. Through this, Siddhi queries can also be optimized using SQL optimization techniques that are used in relational databases. This method is acceptable because SQL queries mostly express the same set of functions the CEP engines have. In Siddhi, users can create the queries via its Java API which consists of simpler methods for adding SQL-like statements such as SELECT, WHERE, FROM etc. It is planned to provide the usual way of input SQL queries, i.e. entering the query in direct SQL-like language, in near future. One of the Significant feature



```java
//Instantiate SiddhiManager
SiddhiManager siddhiManager = new SiddhiManager();
//Get the QueryFactory to create queries
QueryFactory qf = siddhiManager.getQueryFactory();

InputEventStream stockStream = new InputEventStream(
    "StockStream",
    new String[]{"symbol", "price", "volume"},
    new Class[]{String.class, Float.class, Long.class}
);
//Setting a time window of 1 hour
stockStream.setTimeWindow(3600000);
//Assign input stream
siddhiManager.addInputEventStream(stockStream);

//Create Query
Query query = qf.createQuery(
    "StockQuote",
    qf.output("symbol=StockStream.symbol","price=avg(StockStream.price)"),
    qf.inputStream(stockStream),
    qf.and(
            qf.condition("StockStream.symbol", EQUAL, "IBM"),
            qf.condition("StockStream.volume", GREATERTHAN, "10000")
    )
);
//Assign query
siddhiManager.addQuery(query);
```

Figure 7. Siddhi API

of the Siddhi Query model is where each query will produce a stream on its name and these streams can be then passed to other queries whereby creating complex queries similar to Nested SQL queries. These loosely-coupled query objects not only facilitate easy query construction but also enable queries to be written in order to eliminate common sub-queries.

Query optimization is also done at Executors which are critical components in Siddhi processors. Executors are the principle processing element in Siddhi processor. These Executors are generated by the Query parser by parsing the transparent Siddhi Query Object Model which is defined by the user. The formed Executors will have a tree like structure. When an Event is passed to the Executor tree, it will process and return true if the Event matches the query or return false if the event does not match the Query. At the same time, there can be many Executor trees present in the processor; but only one get executed at any given time. These Executor trees are processed in a Depth First Search (DFS) order. If a mismatch is occurred at any of the nodes while processing, the process gets terminated and the nodes in Executor tree will recursively returns false until the root and notifies the failure to the processor, making that event obsolete. If the tree achieved a success state either by travelling through all or part of the nodes, the root will notify true to the processor which then appropriately handles that success event.

These Executors could also have been processed sequentially but it will result in rejecting non-matching Events at an early stage, due to the duplicate Execution nodes in sequences. Therefore the Tree executor model was used whereby the Executors can not only be arranged in an optimal order but they also enable SQL query optimization techniques by the use of Siddhis transparent Query Object Model. It also allows the users to construct the Executor tree in an optimal manner where the nested condition object passed to the WHERE clause are simply converted into a Executor tree with same ordering. This greatly improves early detection of non-matching events.

Hence, we rectify the issues occurred in the Stream Based systems using the transparent query object model by enabling the construction of running only one sub-query at any given time. Through this we ensure that the underline pipeline architecture only runs one sub query in the system facilitating higher performance. Along with this since the processors are running in separate threads this also facilitates higher performance through the parallelisation of execution.

When it comes to streaming events, Siddhi has a much different implementation in contrast to STREAM. Through this Siddhi over comes the issue in handling intermediate queue outputs. Siddhis logical event stream representation and the actual physical streams implementation in here are quite the opposite. In Siddhi, when defining a query, we can assign one or more input streams to each query where the query will then produce one output stream in its name. But in the physical implementation since we are using queues to store intermediate events, we use only a single input event queue for each query. This is to overcome the issues of managing many event streams where when a query is assigned two input streams checking both input queues and keeping track of which event is yet to be processed by queries, which events have already being processed by all queries and which should be discarded makes this process much complex. Further the use of a single queue to represent all incoming streams will not be an issue because each processor only process one event at a time. To facilitate the above architecture the implementation of output event streams is also changed. Here the use of single output queue was eliminated and whenever an output is produced by the processor the reference of the output events are simply adds to all the subscribed queries input queues. Through this approach of using single input queue model we can easily eliminate monitoring of intermediate events and accelerated streaming of events whereby improving the overall performance.

Siddhi also has the ability to handle aggregation of events by the means of averaging, summing, counting and finding minimum & maximum within a given time-window/length-window frame. Here, when an event is arrived to the time/length window it will be treated as a new event and the expired events which are falling out from the window are treated as old events. Siddhi also uses an effective architecture in processing the temporal windows (time and length windows). In contrast to many other CEP Engines, Siddhi handles the processing of temporal windows as part of its queues then processing them with in its processors. Through this it also achieves much better performance because it provides the ability to utilize the same temporal window by many queries facilitating in high performance

Siddhi also supports dynamic adding and removing of queries at run-time which most of the previous works [22] did not handle.

IV. Results and Discussion

As our major objective is producing a high performing CEP engine we continuously did testing and profiling in order to come up with an effective architecture that facilitates high performance. As part of performance analysis we compared Siddhi against one of the existing CEP Engine, Esper, which is currently the most popular and the best performing open source CEP implementation in the market [23].

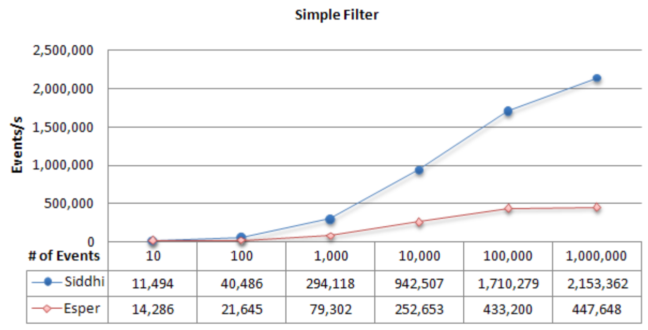
The results obtained through testing are depicted in Figure 8 & 9. These results are obtained by testing both CEP Engines in exactly the same conditions and we compared the performance with two different types of queries which cover most of the important CEP functionalities that are related to streaming of data. These are simple filtering of Events and filtering Events within a given time windows. In-order to test the performance we selected a machine having a configuration of Intel- Core 2 Duo 2.10 GHz processor, and 1.9GB memory running Ubuntu 9.10 with Linux Kernel 2.6.31-14-generic and GNOME - 2.28.1.

Following graphs illustrate throughput while processing various number of Events which are programmatically sent to both Siddhi and Esper. With each graph, the table below the graph shows actual numbers.

Performance of simple filter for the Event Processing Language (EPL) query;

```
select symbol, price
from StockTick(price>6)
```

This shows the performance comparison of a bare filter without any specific temporal Event processing. This analysis shows how both CEP Engine frameworks perform without any additional work load. Here though both CEP Engines were performing very well when number of Events increases, Siddhi outperforms Esper more then three times.

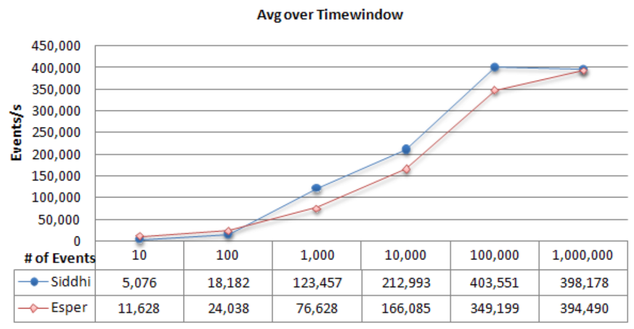


# of Events	10	100	1,000	10,000	100,000	1,000,000
● Siddhi	11,494	40,486	294,118	942,507	1,710,279	2,153,362
◇ Esper	14,286	21,645	79,302	252,653	433,200	447,648

Figure 8. Siddhi Vs Esper Simple Filter Query Comparison

The aggregation of Events on temporal windows are also tested, here we are presenting the performance Comparison graph for average (the most costly aggregation function) with a given time-window.Figure 9 depicts the performance comparison of average over time window using EPL query;

```
select irstream symbol, price, avg(price)
from StockTick(symbol='IBM').win:time(.005)
```



# of Events	10	100	1,000	10,000	100,000	1,000,000
● Siddhi	5,076	18,182	123,457	212,993	403,551	398,178
◇ Esper	11,628	24,038	76,628	166,085	349,199	394,490

Figure 9. Siddhi Vs Esper Average over Time Window Query Comparison

This shows the performance for a query that gets Events from the StockTick stream with a time-window of five milliseconds. The output Events of this query consist of attributes symbol, price and average price. Here, both new and expired Events are taken in to consideration when conducting performance testing and calculating the performance gain. As we can see here Siddhi only have a marginal gain over Esper, performance of Siddhi will be further improved when more complex queries are defined, this is because in Esper it handles the temporal window processing in the query itself but in Siddhi, it manages the temporal Events in its Event queues. So in situations where many queries needs to use a temporal window, in Siddhi they can share the same temporal window queue whereby achieving much better performance. Overall, Siddhi shows considerable improvement in performance compared to Esper. Further, considerable time has also been spent on the Siddhi design to make its architecture to be extensible, such that adding features later, can be done in a minimum cost.

As we discussed in the literature, Event Stream Processing engines have demonstrated better performance and here using SIddhi we have further enhanced the streaming capabilities using a better architecture whereby leveraging multi-threading and pipelining in order to achieve higher performance.

Consequently, Siddhi architecture also provides the facilitates to query optimizations whereby eliminating the bottlenecks in the stream processing systems such as eliminating multiple common sub queries, managing pipelines and streams to deliver Events to more than one Processor in an efficient manner, optimizing execution patterns and seamlessly using the pipeline architecture reusing the temporal windows over multiple queries. Through Siddhi architecture users can also achieve high performance by efficiently constructing the queries were the proper use of the simple processes such as filters can be used to massively reduce the number of Events. Placing these processors at the front end of the streams and more complex time consuming tasks coming later can facilitate the throughput of the systems to be massively increased. Here the query optimizer can still be based on the semantics of the traditional transformation rules.

Siddhi results demonstrate that proposed approach can provide better performance, and use of the pipeline architecture enables future CEP Engines to take advantage of ample computing power available (e.g. through multi-core). An important future work to Siddhi is to design a textual query language which would make it easier for users to define queries rather than using the current Java API.

References

[1] D. Luckham and R. Schulte, "Event processing glossary-version 1.1," *Event Processing Technical Society (July 2008).*

[2] "EsperTech - event stream intelligence," `http://www.espertech.com/`, [Online; accessed 20-Sept-2011]. [Online]. Available: http://www.espertech.com/

[3] S. Rizvi, "Complex event processing beyond active databases: Streams and uncertainties," 2005.

[4] S. Sirish, S. Krishnamurthy, O. Cooper, M. Franklin, J. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. Shah, "TelegraphCQ: an architectural status report," in *IEEE Data Engineering Bulletin*, 2003.

[5] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: A new class of data management applications," in *Proceedings of the 28th international conference on Very Large Data Bases*, 2002, pp. 215–226.

[6] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, resource management, and approximation in a data stream management system," in *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. Shah, "TelegraphCQ: continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 668–668.

[8] "Truviso web analytics software | scalable, real-time, Multi-Source web analytics tools," `http://www.truviso.com/`, [Online; accessed 21-Sept-2011]. [Online]. Available: http://www.truviso.com/

[9] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra, "Matching events in a content-based subscription system," in *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, 1999, pp. 53–61.

[10] F. Fabret, H. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *ACM SIGMOD Record*, vol. 30, 2001, pp. 115–126.

[11] "Aurora project page," `http://www.cs.brown.edu/research/aurora/`, [Online; accessed 21-Sept-2011]. [Online]. Available: http://www.cs.brown.edu/research/aurora/

[12] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul *et al.*, "Retrospective on aurora," *The VLDB Journal*, vol. 13, no. 4, pp. 370–383, 2004.

[13] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin *et al.*, "Aurora: a data stream management system," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 666–666.

[14] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine," in *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA*, 2005, pp. 277–289.

[15] "Stanford stream data manager," http://infolab.stanford.edu/stream/, [Online; accessed 21-Sept-2011]. [Online]. Available: http://infolab.stanford.edu/stream/

[16] D. Arvind, A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "STREAM: the stanford stream data manager," in *IEEE Data Engineering Bulletin*, 2003.

[17] "Complex event processing (CEP) technology & Real-Time business process management - sybase inc," http://www.sybase.com/products/financialservicessolutions/complex-event-processing, [Online; accessed 21-Sept-2011]. [Online]. Available: http://www.sybase.com/products/financialservicessolutions/complex-event-processing

[18] L. Flp, G. Tth, R. Rcz, J. Pnczl, T. Gergely, A. Beszdes, and L. Farkas, "Survey on complex event processing and predictive analytics," 2010.

[19] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. White *et al.*, "Cayuga: A general purpose event monitoring system," in *Proc. CIDR*, 2007.

[20] "SASE - SASE home," http://sase.cs.umass.edu/, [Online; accessed 21-Sept-2011]. [Online]. Available: http://sase.cs.umass.edu/

[21] M. Cammert, C. Heinz, J. Krmer, A. Markowetz, and B. Seeger, "Pipes: A multi-threaded publish-subscribe architecture for continuous queries over streaming data sources," Citeseer, Tech. Rep., 2003.

[22] T. Owens, "Survey of event processing," DTIC Document, Tech. Rep., 2007.

[23] "EsperTech - Partners," http://www.espertech.com/partners/partners.php, [Online; accessed 20-Sept-2011]. [Online]. Available: http://www.espertech.com/partners/partners.php