Siddhi: A Second Look at Complex Event Processing Architectures

Sriskandarajah Suhothayan
University of Moratuwa, Sri Lanka
suho@apache.org

Isuru Loku Narangoda
University of Moratuwa, Sri Lanka
isudana@gmail.com

Subash Chaturanga
University of Moratuwa, Sri Lanka
subashsdm@gmail.com

Kasun Gajasinghe
University of Moratuwa, Sri Lanka
kasunbg@gmail.com

Srinath Perera
WSO2 Inc. Mountain View, CA, USA
srinath@wso2.com

Vishaka Nanayakkara
University of Moratuwa, Sri Lanka
vishaka@uom.lk

ABSTRACT

Today there are so much data being available from sources like sensors (RFIDs, Near Field Communication), web activities, transactions, social networks, etc. Making sense of this avalanche of data requires efficient and fast processing. Processing of high volume of events to derive higher-level information is a vital part of taking critical decisions, and Complex Event Processing (CEP) has become one of the most rapidly emerging fields in data processing. e-Science use-cases, business applications, financial trading applications, operational analytics applications and business activity monitoring applications are some use-cases that directly use CEP. This paper discusses different design decisions associated with CEP Engines, and proposes to improve CEP performance by using more stream processing style pipelines. Furthermore, the paper will discuss Siddhi, a CEP Engine that implements those suggestions. We present a performance study that exhibits that the resulting CEP Engine—Siddhi—has significantly improved performance. Primary contributions of this paper are performing a critical analysis of the CEP Engine design and identifying suggestions for improvements, implementing those improvements through Siddhi, and demonstrating the soundness of those suggestions through empirical evidence.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; I.5 [**Pattern Recognition**]: Implementation

General Terms

Performance, Design

Keywords

Complex Event Processing, Events, Data Processing

1. INTRODUCTION

We are facing an avalanche of data. If we look around, there are so much data being available from sources like sensors (RFIDs, Near Field Communication), web activities, transactions, social networks, etc. Those data can be broadly categorized as data at rest vs. data in the move. Ubiquity of sensors has increased amount of continuous data streams significantly, and many use-cases require processing them in near real time.

E-Science is a field of study that applies computer science in the context of sciences, and many usecases in E-science tries to make sense of data collected in national or global scale. For example, let us consider Los Angeles Smart Grid Demonstration project [28], which is an effort to monitor and react to data collected from millions of sensors in the power grid to optimize the power usage and operations. This is only one example that requires processing and reacting to moving data streams. Among other examples are reacting to data collected from weather sensors, battle field control, traffic control, etc. Since e-Science often deals with extreme use-cases of national and global scale, e-Science middleware frameworks needs support for processing large-scale data streams in real time [18]. For example, Apache Airavata is an effort to build a middleware framework to support e-Science usecases, and in the face of aforementioned streaming e-Science usecases, CEP runtime could be a great addition to middleware stacks like Airavata.

Complex Event Processing (CEP) [19] is one of the most rapidly emerging fields in data processing, and it is a principal technology solutions for processing moving data streams in real time. A Complex Event Processor identifies meaningful patterns, relationships and data abstractions among apparently unrelated events, and fire an immediate response.

Applications such as monitoring (e.g. e-Science usecases), surveillance, manufacturing and financial trading requires low latency, while applications such as data warehousing, web analytics and pperational analytics need to handle higher data rates. Both these categories of applications can utilize CEPs to efficiently perform their tasks.

CEP Engines provide the runtime to perform Complex Event Processing where they accept queries provided by the user, match those queries against continuous event streams, and trigger an event or an execution when the conditions specified in the queries have been satisfied.

Goals of this paper are to critically evaluate decisions taken at CEP design and to present Siddhi CEP Engine that incorporates several improvements uncovered at the

GCE'11, November 18, 2011, Seattle, Washington, USA.
ACM 978-1-4503-1123-6/11/11.

discussions. To that end, next section will present a survey of CEP Engine design, and Section 3 will describe Siddhi. Thereafter Section 4 will present a performance comparison of Siddhi and Esper [14], where latter is an open source, established CEP Engine. The primary contributions of this paper are critical analysis of the CEP Engine design and identifying area for improvements, implementing those improvements, and demonstrating the soundness of those suggestions through empirical evidence. The paper describes how those improvements are achieved through a new CEP called Siddhi.

2. RELATED WORK

Cugola et al. [11] provides a detail discussions of event processing/CEP implementations. Let us look at some of major work in the area. Early on, it was apparent that processing continuous data streams by storing events in a database and querying them yields very poor performance (e.g. Rizvi et al. [24]). Consequently, stream databases [10, 8, 20, 27] immerged. Those systems still used a storage centred approach with expressive query languages, yet were not optimized for processing sequential patterns. For example, TelegraphCQ [29] was built by modifying the existing architecture of open source PostgreSQL database to support continuous queries, and later Truviso [32] provided closer integration with databases supporting historical queries as well. However, each of those approaches failed to deliver the expected speed.

Stream Processing follows the model of publish-subscribe systems such as [3, 15] which are characterized by query languages having very limited expression power and only allowing simple selection predicates applicable to individual events in a data stream. Systems like these trade expressiveness for performance. With these stream based systems, data flows through them (often as an acyclic graph) where data streams are filtered, combined, or transformed on the way. Those systems are often distributed, highly parallel, and support sharding. Among examples of these systems are Aurora [5, 2], PIPES [23, 7], STREAM [30], Borealis [6, 1] and S4 [25].

For example, S4 developed by Yahoo! supports massive scale processing of data streams using actor model [21]. Though it seems to perform well in stream processing, it still can not be categorised as an effective CEP Engine as it does not even have the basic temporal event processing capabilities over time or length windows, and it cannot handle complex events. Although well engineered stream based systems exhibit very high scalability in both number of queries and stream rates, their inability to express queries that span multiple input events makes them unsuitable for Complex Event Processing [13].

Therefore when considering full Complex Event Processing systems, ODE [17], SASE [26], Esper [14] and Cayuga [9] much similar in their full CEP support. These systems use some variants of a Nondeterministic Finite Automata (NFA) model. While many of them support some form of parameterised composite events such as Cayuga [13]. There are also several commercial CEP engines in the market but most of them had been built on top of some open source CEP engine such as Truviso on TelegraphCQ, StreamBase on Aurora, Coral8 on STREAM and Oracle CEP 10g using Esper as its engine [16]. Detail about most of them have limited availability due to their commercial nature, and Cugola et al. [11] discuss most of the available information.

Among CEP engines, SASE [26] uses a dataflow paradigm with native sequence operators and pipelining query-defined sequences subsequent to the relational style operators. Diao et al. [33] provides a comparison between SASE and a relational stream processor, TelegraphCQ [31] developed at the University of California, Berkeley. In this study, SASE performs much better than TelegraphCQ because it uses NFA to naturally capture the sequencing events, and the Partitioned Active Instance Stack (PAIS) algorithm to handle the equivalence test during execution of NFA, finally yielding in much better scalability [33]. Unlike many previous works SASE not only reports that the query got satisfied but also explicitly report what events are used to match the query, this significantly increases the complexity of query processing. SASE having all these advantages it also have some major limitations such as it not being able to handle hierarchy of complex event types, where the output of one query cannot be used as an input to another (chaining).

The drawback of SASE where the output of one query cannot be used as an input to another was appropriately handled in Cayuga which was developed by Cornell. Here they describe Cayuga as a general-purpose Complex Event Processing system [13] which can be used to detect event patterns in multiple unrelated event streams. Since Cayuga is designed to leverage the traditional publication-subscription techniques it allows high scalability [12], and its system architecture also supports large number of concurrent subscriptions. One of the most novel components of Cayuga is the implementation of its Processing Engine that utilizes a variation of NFA [12]. This single threaded Processing Engine reads the relational streams and processes the automata. The use of automata allows storing of input data enabling new inputs to be compared against previously encountered events. Not every Cayuga query can be implemented by a single automaton and in order to process arbitrary queries, Cayuga supports re-subscription, which is similar to pipelining. This re-subscription model based architecture of Cayuga requires each query output to be produced in real time for the next process to utilize that. Here each output tuple of a query will have the same detection time as the last input event that contributed to it, hence processing (by re-subscription) of output event must take place in the same epoch in which that event has arrived. Therefore when processing sequence of queries for the same event there might be other queries starving which could produce more useful final output in no time. To handle this issue Cayuga uses priority queues and it also tries to perform multi query optimization techniques to merge manifestly equivalent states events having the same time stamps to be processed together [13]. Nevertheless since the core is single threaded, the performance improvements obtained through these optimizations were not major.

Similar to Cayuga, Esper [14] also has the ability to express complex matching conditions that includes temporal windows, joining of different event streams, as well as filtering and sorting them. Further it also has the ability to detect sequences and patterns of unrelated events. The internals of Esper are made up primarily relying on state machines and delta networks in which only changes to data are communicated across object boundaries when required. Esper has also being used as the core for many other CEP so-

lutions [16] and one of such is BEA WebLogic [22] which is used in Oracle CEP [22]. Though Esper has multi-threading, its architecture predominantly depends on the observer pattern.

To qualitatively compare Siddhi with other CEP engines, we have used Cugola et al. [11] that compares several CEP engines including Esper. To save space, we will describe Siddhi in contrast to Esper using the comparisons defined in Cugola et al. In terms of Functional and Processing Models and Data, Time, and Rule Models (see Table I and III of Cugola et al.), Esper and Siddhi behave the same. In terms of Supported language model (table IV), Esper supports Pane and Tumble windows, User Defined operators and Parameterization while Siddhi does not support them, and Siddhi supports removal of duplicates that is not supported by Siddhi. Both support all other language constructs.

Above systems, mainly stream based systems and the complex systems like Cayuga, Esper and SASE, presents following useful design patterns, and in the next section we will describe Siddhi, which combines following design decisions to improve the performance.

1. Multi-threading
2. Queues and use of pipelining
3. Nested queries and chaining streams
4. Query optimization and common sub query elimination

3. SIDDHI ARCHITECTURE

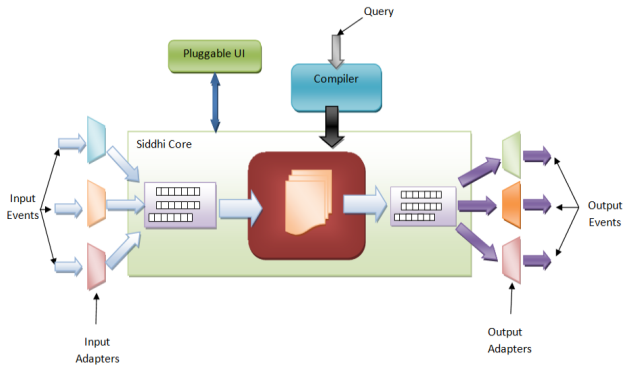


Figure 1: Siddhi System Architecture

Figure 1 depicts the high-level architecture of Siddhi. As shown in the figure, Siddhi receives events from event sources through Input Adapters and converts them to a common data model: tuple. For instance, if a Plain Old Java Object (POJO) or an XML arrives at Siddhi, its Input Adapter converts the POJO or XML into a tuple for internal processing. Following sub section discusses this internal data model. When user submits a query to Siddhi, the Query Compiler coverts the query to a runtime representation (Processors) and deploys that in the Siddhi core. The core is the heart of Siddhi where it does all the processing. It consists of Processors and event queues, which will be discussed later. As indicated in the diagram, input events are placed on input queues and the Processors then fetch the events for processing. When those events match the query of the Processor, corresponding output events are produced and placed in the output queues. Those events are either sent to external subscribers through Output Adapters or consumed by other Processors for further processing. Siddhi architecture also allows manipulating queries on the fly, thus allowing users to add or remove queries while Siddhi Engine is running.

3.1 Event Tuples

Figure 2: Event Tuple

As shown by the Figure 2, Siddhi represents events using a tuple data structure, which resembles a row in relational database tables. This not only enables the use of SQL like queries but also incorporates relational database optimization techniques to the system. Through tuple data structure, frequently retrieving data from a tuple will get much simpler and efficient compared to other alternatives (like XML) which indeed help Siddhi to process events faster by minimizing the overheads incurred by accessing data from events.

3.2 Query Object Model

Code snippet given in Figure 3 shows a sample Siddhi query. Currently, Siddhi uses an API that builds a query object model to represent queries. Siddhi query object model follows an SQL like query structure, and hence the Query object module too resembles SQL by facilitating the users to easily understand and write queries in Siddhi. SQL like queries fall in-line with the relational algebraic expressions, and through this, Siddhi queries can also utilize the optimization techniques that are used in SQL and relational databases. Other CEP solutions also follow this approach. In Siddhi, users can create the queries via the Java API provided by Siddhi, which consists of simpler methods for adding SQL-like statements such as SELECT, FROM, WHERE etc.

Following listing 1 uses a BNF like notation to provide an approximation of the Siddhi grammer. We are working on defining a query language on top of the query model using XText DSL modelling framework. In the listing, capital letters represent variables while simple letters represent literal text.

Each Siddhi query produces a stream, which can be passed into another query as an input stream thus creating complex queries. In other words, queries are recursively composable. 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 whereby improving the overall performance. When writing multiple nested complex queries, it is common for similar sub queries to occur throughout. In Siddhi since each query's output can be fed into many queries, the same query being repeated will be eliminated, thus eliminating common sub queries from the query plan.

```
//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 3: Siddhi API Usage

Listing 1: BNF Style Grammer

```
QUERY :=STM_NAME( select OUT from STM
 { [where COND] [  [having COND]
 [groupby STM_NAME.NAME]]|[PATTERN]
 | [sequence(SEQ)]
})
STM :=STM join STM|QUERY|
  NAME(IN_NAMES, IN_TYPES [, (STM_PROP)])
STM_PROP:=[time_window|batch_window
  |window_length|unique|firstUnique]*
PATTERN :=every(COND)|any(COND)|
  COND|nonOccurrence (COND)
SEQ :=COND SEQ|star(COND) COND
COND := COND N_OP COND | COND L_OP COND
  | STM_NAME.NAME N_OP [VAL|STM_NAME.NAME]
  | STM_NAME.NAME L_OP STM_NAME.NAME
OUT :='*'|NAME=EXP, OUT| NAME=EXP
EXP :=[avg|sum|max|min|count]+(EXP)|NAME| STM_NAME.NAME
STM_NAME:=NAME
VAL :=[0_9.]*|true|false
NAME:=[A_z0_9]*
N_OP:=eq|noeq|lt|gt|lteq|gteq
L_OP:=and|or
IN_NAMES:=NAME [,NAME]*, IN_TYPES:=list of types
```

3.3 Pipeline Architecture

As depicted by Figure 4, the Siddhi architecture consists of Processors connected through event queues. Incoming events are placed in event queues, and Processors listening to those event queues process those events and place any matching events into output queues of that Processor, which will then be processed by other Processors or send to end users as event notifications. Each Processor is composed of several Executors that expresses the query conditions, where each Executor processes the incoming events and produces a Boolean output conveying whether the event has matched or not. The non-matching events are simply discarded, and matching events are processed by logical Executors downstream. Siddhi uses a pipeline model where it breaks the execution into different stages (through Processors), and moves the data through the pipeline using a publication-subscription model.

In contrast to Siddhi, CEP Engines like Cayuga uses a single processor architecture where a single thread performs all events to query matching. Though this may reduce the complexity of query processing, it is not suitable for actual

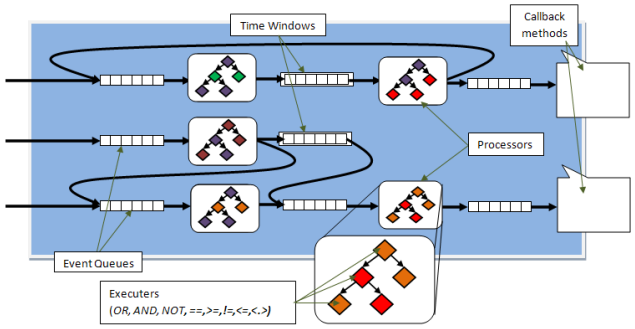


Figure 4: Siddhi core

streaming of data like in Aurora and STREAM. Furthermore, this model is also not good with utilizing multiple CPU cores in the system. On the other hand, the stream-based systems allocate each query to a thread. Although this model produces higher performance, it too has drawbacks in resource utilization. This is because in most cases, many complex queries do have common sub queries, and stream based systems will end up running many duplicate sub queries at the same time.

As described in the architecture section, Siddhi addresses above issues by using the pipeline architecture and a transparent query object model that clearly illustrate the internal data structure of the system. Siddhi manages similar sub queries by constructing a single sub query that can be connected with other queries (through queues) while constructing the complex query, and this ensures that the underlying pipeline architecture runs only one common sub query in the system. Also, threads assigned to different stages of the pipeline parallelly process events achieving faster execution and higher throughput. All processing done by multiple threads, and to minimize blocking, we use non-blocking data structure when implementing the queues between pipeline stages.

To avoid the complexities of handling intermediate queues, Siddhi uses a different implementation of intermediate queues in contrast to STREAM [4]. With this approach, Siddhi's logical event stream representation and the actual physical implementation of the streams are quite the opposite. In Siddhi, when defining a query more than one input streams can be assigned to each query and the query will then produce one output stream in its name. However, in the physical implementation, when more than one input streams are assigned to a Processor (the runtime representation of the query), checking all input queues and keeping track of which events are yet to be processed, being processed, and to be discarded makes the processor implementation very complex. To overcome this issue, Siddhi implements multiple streams using a single input event queue by multiplexing them together. Implementing all incoming streams using a single blocking queue will not be a problem because each event contains the id of its stream, and hence the Processor could easily identify to which stream the event belongs to. To facilitate this change, the implementation of the output event streams is also changed such that whenever an output is produced by the Processor, a reference of that output event is add to all the subscribed queries' input queues. This

approach of using single input queue model eliminated the need for monitoring intermediate events and improved Siddhi's performance.

3.4 Processor Topology

Siddhi Processor has two major components. Those are Executors and Event Generator. As explained, Processor accepts one event at a time and produces an output event if the input event matches the conditions evaluated by that Processor (temporal or conditional). This evaluation is done by the Executors which are generated by the query parser by parsing the query object model constructed by the user. Siddhi arrange these Executors in a tree like structure, and when an event is passed to the root of the tree. The tree processes and return true if the event matches the query represented by the tree, and else returns false if the event does not match the query. Though at the same time there can be many Executor trees present in the same Processor, only one gets executed at a time. Siddhi evaluates the tree in depth first search (DFS) order, and Siddhi optimizes this process by stopping the execution whenever there are enough conditions to know that tree will not evaluate to true. Siddhi could have processed Executors sequentially but this will have issues in rejecting non-matching events at an early stage, due to the duplicate execution nodes in the sequence. But through the use of Tree Executor model, the Executors can be arranged in an optimal order to leverage the query optimization techniques used in relational databases.

Siddhi supports simple condition queries, sliding and batch windows, sequence detection, and pattern detection in addition to stream handling (e.g. joining, filtering, aggregation, having and group-by), etc. Following two sections describe how state machine processors and window operations are implemented.

3.5 State Machine

Pattern Queries: Siddhi uses state machines to support pattern and sequence queries. Pattern queries fire an event when the given series of conditions get satisfied one after the other. The Pattern Processor handles the processing of these pattern queries. Consider the example where we define the sequence of conditions as, A → B → C. Let's consider the incoming event sequence as follows,

A1, B1, A2, A3, C1, A4, B2, B3, C2

Here an event will be fired when Siddhi receives C1 where the captured event pattern will be A1, B1 and C1. In the above mentioned incoming event sequence, after firing the first event, Siddhi stops looking for new events. If we want Siddhi to continuously monitor the pattern occurrences, we have to use the 'Every' operator.

The Pattern Processors process the incoming events and generate an output event when necessary. In Siddhi since the basic unit of condition matching is the Executor, the Pattern Processor contains series of Followed-by Executors which corresponds to each state in the NFA. In Siddhi all the currently active Followed-By Executors are stored in a Map, and within the Map, Siddhi uses linked lists to hold the Followed-By Executors categorized based on their streams.

When an input event reaches the Pattern Processor, it sends the event to the currently active Executors. Executors check whether the input event can be matched with the defined conditions. If so they return true. When a successful match occurred, the input event is stored in the Executor which has returned true, and the Processor spawns a new Executor belonging to the next state from the current Executor and adds that to the active event listener Map. In this process, the previously stored events are also passed to the newly created Executor for future references and the matched Executor (the one returned true) gets removed from the active event listener Map. If there is no next state exists, which means we have achieved the final state, the Processor will generate the output event based on the output definition. When generating the output event, the data is extracted from the previously stored input events.

When processing, Siddhi only sends the events to the Executors which correspond to the input event stream id rather than sending to all the available Executors. That is one strategy used to improve the performance of the state machine. Further, Siddhi also provides facilities to eliminate duplicate states in order to make the processing much faster.

Sequence Queries: In sequence queries we can define Siddhi to fire an event when series of conditions satisfied one after the other in a consecutive manner. Consider the example where we define the sequence of conditions as, A → B → C. Let's consider the incoming event sequence as follows,

B1, A1, B2, C1, A2, B3, B4, C2

Here an event will be fired when Siddhi receives C1 where the captured event sequence will be A1, B2, and C1. Unlike in pattern queries here the sequence of conditions has to be satisfied consecutively where no event will get fired for the following event sequence,

B1, A1, A2, C1, A3, B3, B4, C2

Sequence matching starts when Siddhi receives the A1 event and then it will listen to a B event. Since Siddhi receives A2 event, the sequence fails. For the sequence queries we can also use kleene star operator to define an infinite number of intermediate conditions.

Sequence Processor handles the processing of sequence queries where its implementation is very much similar to the Pattern Processor. In Pattern Processor, when a condition match is occurred, the currently matched Executor (the one returned true) will get removed from the active Executor Listener Map, but in Sequence Processor all the active Executors will be removed irrespective of success or failure of the condition. The reason for this is, in sequence queries, series of conditions should be satisfied one after the other in a consecutive manner.

In Sequence Processor, we have used a Linked List to store currently active Executors. Compared to Pattern Processor, the number of active Executors in the Sequence Processor is very low. Hence there is no need to use a map to filter out Executors. This makes the implementation much simpler and prevents Siddhi from unnecessary overheads during execution.

3.6 Sliding Windows and Batch Windows

Siddhi supports sliding-window and batch-window based queries where they let users reason about collection of events. These can further be divided into time-based, and length-based. The time-sliding-windows keep track of the events arrived within a given amount of time to the past from the

current time. This window is useful for analyzing events that arrived within a limited amount of time (e.g. statistical analysis of the arrived events such as average, sum of a particular attribute). The length-sliding-windows are similar to the time-sliding-windows except it keeps track of a certain number of events arrived recently while the window keeps sliding for each new event that arrived to the stream.

In contrast, batch-windows perform processing in event batches. In time-batch-window when the time elapsed and in the length-batch-window when the maximum number of events has arrived, it throws the batch away, generating the appropriate output events and starting a new batch. The operations of batch-windows are similar to sliding-windows except they process in batches.

Siddhi implements windows within event queues rather than within Processors because assigning a time window for stream is more appropriate than assigning one to queries. This also allows multiple queries to utilize the same time window whereby improving memory consumption and performance of Siddhi. Siddhi supports windows through a special stream implementations where it delivers events placed inside a stream to Processors attached to that stream (event queue) only when the conditions of the window have been fulfilled.

Systems like Esper uses two different streams to output events. The new incoming events from time window are sent through one and the expired events from the time window are passed through the other. But in Siddhi though it uses one stream it sets a flag in the event to differentiate the new and expired ones. This design not only makes the process less complex, but it also improves the performance.

3.7 Duplicated Event Detection

Siddhi has the capability of detecting duplicated events. The criteria on determining the duplicity can be specified by the user by specifying a set of event attributes that needs to match. Usual context of this is specifying a related id for comparing. There are two ways to deal when a duplicated event is found. These are called UNIQUE, and FIRSTUNIQUE, where UNIQUE only take into account the last arrived event, while in FIRSTUNIQUE only the first duplicated event will be processed, and the newly arriving duplicate events will be dropped. For the time window and length window queries, the existing event queues are used to minimize the overhead. In some cases, duplicate detection is needed for non-windowed queries as well. It'll be expensive since all the arrived events needed to be tracked. The cost has been minimized using Java Sets, but this needs to be used with precaution since this is an iterative process.

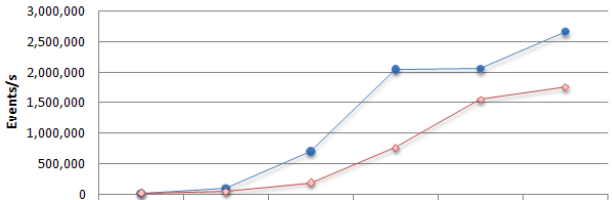
4. RESULTS

To empirically evaluate the effectiveness of the proposed approach, we have compared Siddhi with Esper, the most widely used open source CEP Engine. Esper has many customers and it has also being used as the core for many other CEP Engines like ORACLE. Both CEP Engines were provided exactly the same conditions and we compared the performance with three different types of queries which cover most of the important CEP functionalities such as simple filtering, filtering with time windows and pattern matching. In-order to test the performance we selected a typical server machine that has an Intel(R) Xeon(R) X3440 processor 2.53GHz, 4 cores, 8M cache, 8GB of RAM and running Debian operating system on a Linux Kernel 2.6.32-5-amd64. For each case, 10 to 1000,000 events were sent through the system and the throughput of processing those messages were measured. To be fair by both CEP Engines, to eliminate event type conversions, the event on their native type are selected where POJO was used for Esper and Tuple was used for Siddhi. Here both types of events are small and contained equal amount of data.

Following three 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)
```

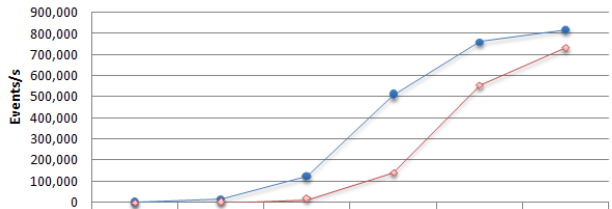


# of Events	10	100	1,000	10,000	100,000	1,000,000
Siddhi	21,739	92,593	704,225	2,053,388	2,061,856	2,662,336
Esper	16,667	36,900	185,185	760,456	1,560,062	1,752,265

Figure 5: Siddhi Vs Esper Simple Filter Query Comparison

Figure 6 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)
```

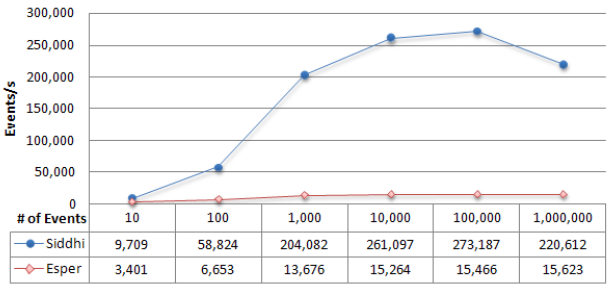


# of Events	10	100	1,000	10,000	100,000	1,000,000
Siddhi	1,613	15,060	122,249	509,424	763,359	820,392
Esper	201	1,838	17,734	142,837	556,824	732,982

Figure 6: Siddhi Vs Esper Average over Time Window Query Comparison

Figure 7 depicts the performance comparison of the state machine using a pattern matching EPL query

```
select f.symbol, p.accountNumber, f.accountNumber
from pattern [every f=FraudWarningEvent2 ->
              p=PINChangeEvent2(accountNumber
              = f.accountNumber)]
```



# of Events	10	100	1,000	10,000	100,000	1,000,000
Siddhi	9,709	58,824	204,082	261,097	273,187	220,612
Esper	3,401	6,653	13,676	15,264	15,466	15,623

Figure 7: Siddhi Vs Esper State Machine Query Comparison

As three Figures depict, both the CEP Engines behave in a stable manner throughout the workload. Here, all messages are sent to the CEP Engines using a for-loop, without any delay, for them to process and trigger events.

In first two cases, Siddhi does about 20-30% better, and we believe this is achieved through the design decisions discussed in the architecture section. For the state machine case, Siddhi did significantly better by performing 10-15 times faster. Siddhi achieves this level of improvement by grouping its active Executers, which are the basic processing elements, by stream ids of the events on which they are waiting for. This strategy has made the Pattern Processor to only process the Executors that are relevant to the incoming events, whereby increasing the success rate of the event being matched at the Executor.

When considering throughput, we believe both CEP Engines did very well where they processed about 0.3 million messages per second for complex cases, to 2 million messages per second for simple cases. However, overall Siddhi has performed better in all cases.

5. CONCLUSIONS AND FUTURE WORK

As we discussed in the literature, Stream Processing Engines have demonstrated better performance over the actual Complex Event Processing Engines, but Complex Event Processing Engines have rich sets of queries where they were able to find meaningful patterns and sequences from unrelated event streams. Also often, CEP Engines tended to use a single processing thread. As we discussed, Siddhi tries to bring in stream processing aspects like multi-threading and pipelining into a CEP Engine in an effort to make it faster.

Consequently, the high level architecture of Siddhi resembles stream processing systems where it utilizes the current multi core processors through multithreading and producer consumer architecture. It is also efficiently designed to make use of its queues and processors to eliminate the bottlenecks in the stream processing systems such as eliminating multiple common sub queries and managing pipelines and streams to deliver events to more than one Processor in an efficient manner. Furthermore, Siddhi has seamlessly used the pipelining architecture to handle temporal events where it keeps the runtime state of time and length windows within event streams rather than implementing them outside (e.g. within query representations). By this Siddhi enables many queries to use the same window improving the overall performance. In addition to streaming and temporal event processing capabilities, Siddhi supports processing of event sequences and patterns. Here it uses an NFA based processing approach, whereby managing the state transitions in an optimal manner.

Siddhi results demonstrate that proposed approach can provide better performance, and use of pipeline architecture should enable future CEP Engines to take advantage of ample computing power available (e.g. through multi-core). Here we have not only demonstrated how efficiently queues can be used to get higher performance but also showed how complex operations could be efficiently done through effective state-machine implementation. For example, pattern matching and sequence processing are the major features of a CEP Engine, and our NFA based implementation only keeping the active state in memory and processing the states which are relevant to the event, enabled to achieve high performance.

An important future work to Siddhi is to define a textual query language. Currently it let users define queries using an API, and functionally it is equivalent to a query language. However, a textual query language would make it easier for users to define queries. Furthermore, Siddhi does not support negation and handling out of order events, which we will handle in the future. In the current implementation of Siddhi, kleene closure sequence processing only provides the first and the last events out of matching events in order to simplify the query construction. We plan to extend Siddhi to output all arrived events like SASE [33].

Finally, the most crucial challenge is to scaling up Siddhi to handle large event rates. There are several potential solutions to scale Siddhi.

1. For simple queries, we could partition events between several CEP Engines each matching events against the query. However, this is not possible with queries that require complex correlations.

2. We could place different queries in different CEP Engines and duplicate events to every CEP Engine. Each CEP Engine has to handle all events; therefore, the approach has scalability limits.

3. We could break down queries into several stages and place them as a pipeline, where the initial queries in first stages are simple and handled by multiple CEP Engines while later stages have complex queries, but event rate would have been significantly reduced when they reach latter stages. But decomposing queries in this fashion need expertise and also often need human intervention.

However, each of these approaches has its own shortcomings, and a scaling architecture would need further research. We are exploring the possibilities of automatically breaking down queries and processing them using a pipeline.

6. REFERENCES

[1] D. Abadi, Y. Ahmad, et al. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA*, pages 277–289, 2005.

[2] D. Abadi, D. Carney, et al. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 666–666, 2003.

[3] M. Aguilera, R. Strom, et al. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61, 1999.

[4] 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.

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

[6] The borealis project. `http://www.cs.brown.edu/research/borealis/public/`. [Online; accessed 25-Sept-2011].

[7] M. Cammert, C. Heinz, et al. Pipes: A multi-threaded publish-subscribe architecture for continuous queries over streaming data sources. Technical report, Citeseer, 2003.

[8] D. Carney, U. Çetintemel, et al. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226, 2002.

[9] Cornell database group - cayuga. `http://www.cs.cornell.edu/bigreddata/cayuga/`. [Online; accessed 21-Sept-2011].

[10] S. Chandrasekaran, O. Cooper, et al. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, 2003.

[11] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys, 2011.*

[12] A. Demers, J. Gehrke, et al. A general algebra and implementation for monitoring event streams. 2005.

[13] A. Demers, J. Gehrke, et al. Cayuga: A general purpose event monitoring system. In *Proc. CIDR*, 2007.

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

[15] 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*, volume 30, pages 115–126, 2001.

[16] L. FÃijlÃűp, G. TÃşth, et al. Survey on complex event processing and predictive analytics. 2010.

[17] N. Gehani, H. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the International Conference on Very Large Data Bases*, pages 327–327, 1992.

[18] T. Hey and A. Trefethen. The UK e-science core programme and the grid. *Future Generation Computer Systems*, 18(8):1017–1031, 2002.

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

[20] R. Motwani, J. Widom, et al. 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.

[21] L. Neumeyer, B. Robbins, et al. S4: distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, page 170âĂŞ177, 2010.

[22] Oracle and BEA systems. `http://www.oracle.com/us/corporate/Acquisitions/bea/index.html`. [Online; accessed 21-Sept-2011].

[23] Homepage der AG datenbanksysteme, fachbereich mathematik und informatik, Philipps-UniversitÃd't marburg. `http://dbs.mathematik.uni-marburg.de/Home/Research/Projects/PIPES/`. [Online; accessed 21-Sept-2011].

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

[25] Home - s4 documentation (v0.3.0). `http://docs.s4.io/`. [Online; accessed 19-Sept-2011].

[26] SASE - SASE home. `http://sase.cs.umass.edu/`. [Online; accessed 21-Sept-2011].

[27] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *ACM SIGMOD Record*, volume 23, pages 430–441, 1994.

[28] Y. Simmhan, V. Prasanna, et al. Towards data-driven demand-response optimization in a campus microgrid. *ACM Embedded Sensing Systems For Energy-Efficiency In Buildings*, 2011.

[29] S. Sirish, S. Krishnamurthy, et al. TelegraphCQ: an architectural status report. In *IEEE Data Engineering Bulletin*, 2003.

[30] Stanford stream data manager. `http://infolab.stanford.edu/stream/`. [Online; accessed 21-Sept-2011].

[31] The telegraph project at UC berkeley. `http://telegraph.cs.berkeley.edu/`. [Online; accessed 21-Sept-2011].

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

[33] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, 2006.