J. Parallel Distrib. Comput. 79–80 (2015) 42–51

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Wihidum: Distributed complex event processing

Sachini Jayasekara [b], Sameera Kannangara [b], Tishan Dahanayakage [b,*], Isuru Ranawaka [b], Srinath Perera [a], Vishaka Nanayakkara [b]

[a] WSO2 Lanka (Pvt.) Ltd., Colombo, Sri Lanka
[b] Department of Computer Science & Engineering, University of Moratuwa, Sri Lanka

HIGHLIGHTS

- Proposes a mechanism to scale up CEP systems.
- Make use of processing pipelines, data partitioning, distributed CEP operators.
- Provide load balancer and distributed operators to support scaling methods.
- We demonstrate the proposed approach using empirical results.

ARTICLE INFO

Article history:
Received 16 November 2013
Received in revised form
14 February 2015
Accepted 11 March 2015
Available online 2 April 2015

Keywords:
Complex event processing
Distributed systems

ABSTRACT

In the last few years, we have seen much interest in data processing technologies. Although initial interests focused on batch processing technologies like MapReduce, people have realized the need for more responsive technologies such as stream processing and complex event processing. Complex event processing has been historically used within a single node or a cluster of tightly interconnected nodes. However, to be effective with Big Data use-cases, CEP technologies need to be able to scale up to handle large use-cases. This paper presents several approaches to scale complex event processing by distributing it across several nodes. Wihidum discusses how to balance the workload among nodes efficiently, how complex event processing queries can be broken up into simple sub queries, and how queries can be efficiently deployed in the cluster. The paper focuses on three techniques used for scaling queries: pipelining, partitioning and distributed operators. Then it discusses in detail the distribution of few CEP operators: filters, joins, pattern matching, and partitions. Empirical results show that the techniques followed in Wihidum have improved the performance of the CEP solution.

1. Introduction

With the advent of Big Data, processing large data sets and deriving insights from that data have received much attention. With some use cases, users can afford to wait hours for processing to take place. However, in other use-cases like traffic, health, stock market analysis, fraud etc. it is needed to generate results as soon as possible, often within milliseconds. We use the term real-time analytics to refer to use-cases that need to process data fast.

Complex event processing (CEP) is one of the key technologies for supporting real-time analytics. It processes data as they are flowing, without writing them to the disk and supports complex temporal queries [8].

As an example, in a scenario of a credit card transaction, a common heuristic used is that if a transaction of large sum comes right after a transaction of a small sum then there is a possibility of a credit card fraud. Complex event processing systems are capable of identifying such scenarios and trigger within few milliseconds of its occurrence.

A CEP use-case may include large number of events, queries, or complex queries and a single CEP node will not be sufficient to handle those scenarios. This paper proposes an architecture to scale CEP query execution so that they can run across several machines and handle much larger event rates. Wihidum uses an open source CEP engine called "Siddhi" as the underlying complex event processing engine.

We define an event as a triple (s, t, d), where s is the name of the stream, t is a timestamp, and d is some data included in the event, which is a set of key value pairs.

Event stream s is an ordered set of events of the form $(s, *, *)$ that have increasing timestamps. We also assume that events in a

* Corresponding author.
E-mail address: dtishanpubudu@gmail.com (T. Dahanayakage).

http://dx.doi.org/10.1016/j.jpdc.2015.03.002
0743-7315/© 2015 Elsevier Inc. All rights reserved.

stream s have special events of the form $(s, t, \{\})$ that are timing events. We assume such events are always there by some out of bound means.

Operator is a function that receives an event e, and produces an optional output event. We denote the output for operator op for event e as $\text{op}(e)$. Operator applied to a stream $S1$ generates another stream $S2$, and we denote this by $\text{op}(S1) \Longrightarrow S2$.

Following are several common event operators supported by Siddhi.

1. Filters—filter is a function that accepts an event and produces the same event if the first event matches a given criteria.
2. Windows—keeps updating a subset of events of a stream as new events arrive. Siddhi supports both batch windows and sliding windows. Batch window processes events in batches, while sliding window keeps the last N number of events. In sliding windows, an output is triggered whenever a new event is added and in batch windows an output is triggered when all the events of the batch are collected. Common window types in Siddhi are time windows, time batch windows, length windows and length batch windows.
3. Patterns—Pattern is a NFA (Non Deterministic Finite Automata) that transitions state when events arrive. A NFA instance is initialized when a starting event is received. It generates a new event when some NFA reaches an accepting state. Often NFA is described in a language that is similar to regular expressions.
4. Joins—Given a condition of two streams $s1$ and $s2$, with a window $w1$ being defined over $s1$, and a window $w2$ being defined over $s2$, when an event e is received over a stream $s1$, the output event is defined by $\{(e, e')|e' \in w2 \cap \text{condition}(e, e')\}$ and vice versa when an event $e1$ is received over a stream $s2$.

The next section gives a brief introduction to the semantics of Siddhi query language.

The following query is an example filter query written in Siddhi.

from StockExchangeStream[price >= 20]
select symbol, volume
insert into StockQuote.

This will produce an output with the events of the StockExchangeStream stream, fulfilling the condition, price >= 20 to the StockQuote stream.

A window query can be written as follows,

from StockExchangeStream#window.length(50)
select symbol, avg(price) as avgPrice
insert into StockQuote.

This will process batches of 50 events and produce an output symbol and per symbol average price to the StockQuote stream.

Siddhi pattern query can be written in the following format.

frome1 = Stream1[price = 20]− > e2 = Stream2[price >= e1.price]
select e1.symbol as symbol, e2.price as price
insert into StockQuote.

For an example, the above query will produce an output of events to the StockQuote stream, when an event arrival at Stream1 with price >= 20 (e.g. 23), followed by an event arrival at Stream2 having price >= 1st event's price (e.g. 25).

The following query is an example join query written in Siddhi.

from TickEvent[symbol == '*IBM*']*#window.length*(2000)*ast*
join NewsEvent#window.time(500) as n
on t.symbol = = n.company
insert into JoinStream.

When an event arrives at the TickEvent, that will be matched with all NewsEvents that have arrived within 500 ms, and if the TickEvent's symbol = = NewsEvent's company, the output event will be generated and sent via JoinStream.

Each operator transforms a stream and creates a new stream, and a query is an acyclic graph having operators as nodes and streams as edges. We call any stream that is only consumed is an *input stream* and any stream that only produces events (not consumed) is an *output stream*.

If a stream receives an event from multiple operators, then it is assumed those events are reordered before they are being emitted into the stream. There are other operators used for joining events against data in a database, which are not discussed within the scope of this paper.

This paper makes the following contributions.

1. We discuss the feasibility of using processing pipelines, data partitioning, and distributed CEP operators to scale up CEP systems.
2. We provide a load balancer and distributed operators to support scaling methods explained above.
3. We demonstrate the proposed approach using empirical results.

2. Related work

CEP have had a vast amount of related work in literature as discussed below.

Stream processing is another term closely related to complex event processing. A stream may contain a sequence of event tuples. In stream processing, input events are processed in parallel. Hence, scaling can be easily achieved by increasing the number of processors. Some of the advantages of stream processing are fault tolerance, scalability and composability. Although stream processing is faster and useful in many occasions, it cannot be used for all complex event processing scenarios. Some complex operators supported by CEP are not supported by stream processing. Stream processing is more focused on simple, parallel, and large scale, queries, while CEP is focused on analyzing, identifying complex patterns and extracting information. According to the definition of EPTS event processing glossary [5], stream processing is computing on inputs that are event streams. But CEP analyzes not only event streams but also event clouds. According to EPTS event processing glossary [5], event cloud is a partially ordered set of events (poset), either bounded or unbounded, where the partial orderings are imposed by the causal, timing and other relationships between the events. Event cloud may contain many event types, event streams and event channels. There is no event relationship that totally orders the events in a cloud as in a stream.

MapReduce is the batch processing counterpart for CEP. This model allows users to specify computation in terms of map and reduce functions. Apache Hadoop is a well-known MapReduce implementation. The paper "MARISSA: MapReduce Implementation for Streaming Science Applications" [1] provides an alternative framework for Hadoop streaming to gain better performance and addressing the lesser flexibility of Hadoop streaming.

By considering the way CEP solution utilizes available processing power, the available solutions can be divided into 4 categories.

2.1. Single node solution

These systems operate on a single server. Therefore any given CEP query is deployed and executed within a single server. Since the solutions run only on a single server, maximum performance value attainable by these solutions are bounded by the server's capabilities. Wihidum does not face the same limitation since it can be run in multiple nodes.

Following are two examples of such systems.

Esper [3] is an open source complex event processor available for Java as Esper, and for.NET as NEsper, which is an example for this category. It supports a wide variety of event representations, such as Java beans, XML documents, legacy classes, or simple name value pairs. It supports sliding windows, tumbling windows, combine windows with intersection and union semantics, partitioned windows, dynamically shrinking or expanding windows, expiry-expression-driven windows etc. According to its architecture, it runs on a single server executing CEP queries utilizing the server's capabilities. Therefore, this solution cannot exceed the capability of the server.

Drools Fusion [2] is a common platform to model and govern the business logic of the enterprise. It was also not created as a distributed solution. Fusion handles CEP part of the module and event detection can be done using an event cloud or set of streams and it can select all the meaningful events and only them. Drools has the capability to correlate events considering temporal and non-temporal constraints between them. It has the ability to reason over event aggregation.

2.2. *Shared memory*

Solutions in this category operate in a set of CEP servers by sharing information about processing state of queries in each server with other CEP servers as necessary. Even though this approach has the ability to distribute stateful CEP operators over a set of CEP servers, attainable results are limited by the increased communication to sync the state of processing over the CEP network. Since processing has to be continued with the syncing state, some CEP nodes have to pause and resume processing while syncing the state. Hence the limitations here is that the shared memory methodology scales only for few nodes.

2.3. *Network of small operators*

These solutions rewrite the given CEP query and deploy it over a CEP server network so that each CEP operator fits within a single CEP server. And the event communication is set between servers so that the aggregate outcome of all deployed operators will create the expected output. Similar to the first category, a single operator executes in a single server, and the processing capability of the operator is bounded by the server's capabilities. Wihidum focuses on how each operator can be distributed to gain improved performance. Unlike these systems, Wihidum can distribute one task and place it in different machines as we will discuss later. The next section explains some of the examples that follow a similar approach.

The paper "High-Performance Complex Event Processing over Streams" [19] explains mechanisms to deal with large sliding windows and large intermediate result sizes. It proposes a complex event language and a query-plan based approach to implement the language. Query planning and optimization techniques used in the paper ensures that operators used in a given complex query are allocated to a single server and event streams are set to generate the expected output.

"Placement Strategies for Internet-Scale Data Stream Systems" [10] discusses several strategies suggested by various researchers in order to place stream-processing tasks (operators) in a networked system to maximize performance of the overall system. This paper provides an evaluation of existing placement strategies for a given problem based on distributed vs. centralized nature of the processing system, the node locations relative to each other, the administrative domain, the rate of topology changes and the placement requests, the incoming data rates and the nature of queries to be executed.

The paper "Complex Event Processing in Distributed Systems" [11] outlines how complex event processing concepts can be applied to a fabrication process management system.

Many of the commercially available distributed CEP systems take an approach of a mix of strategies two and three. Furthermore, they provide centralized control and easy to use graphical user interfaces to configure CEP system.

Integrasoft [8,9] CEP cloud service is one of the major distributed CEP solutions build upon Esper that exists commercially. Integrasoft CEP solution provides a framework to network set of CEP engines together. These CEP engines are running together for virtualized services. This solution offers transparency, scalability, holistic and intelligent service and inter-CEP engine communication. Transparency ensures the fact that applications does not need to know about the underlying architecture. Scalability is achieved by having sub clouds of CEP engines. In addition, the solution is capable of intelligently processing all events against a defined rule set and providing a cloud-wide holistic view. Inter-CEP engine communication is added to facilitate needed communication for transparency, scalability and manageability.

Fujitsu Interstage Big Data Complex Event Processing Server [7,6,18] is another distributed complex event processing solution available commercially. In Fujitsu Interstage Big Data Complex Event Processing Server, Distributed and parallel CEP execution environment manager has been introduced to dynamically apply parallel and distributed techniques for a variable load. Manager component enhances the granularity of processing which enables optimal distribution of processing across multiple nodes. Also the environment manager component has the ability to choose the optimum candidate to transfer processing. This presents an event processing that can be dynamically scaled in and scaled out. This solution considers each query and its data parallelization and then refines it to small pieces. Since both data parallel distribution and query distribution is taken into consideration, Fujitsu Interstage Big Data Complex Event Processing Server can adjust in real time with limited number of resources.

The StreamBase [16,17] complex event processing platform is a commercial high-performance system for rapidly building applications that analyze and act on real-time streaming data. StreamBase's CEP platform combines a visual application development environment, an ultra low-latency high-throughput event server, and over 150 pre-built connectivity and visualization options to real-time and historical data, including exchanges and liquidity providers, middleware, and numerous third-party applications.

Cayuga [14,12] is an expressive and scalable Complex Event Processing (CEP) system developed at the Cornell Database Group. Cayuga supports on-line detection of a large number of complex patterns in event streams. Event patterns are written in a query language based on operators. This enables Cayuga to perform query-rewrite optimizations.

TelegraphCQ was designed to provide event-processing capabilities alongside the relational database management capabilities by utilizing the PostgreSQL. Since PostgreSQL is an open source database, its modified existing architecture allows continuous queries over streaming data [13]. TelegraphCQ focuses on issues such as scheduling and resource management for groups of queries, support for out-of-core data, allow variable adaptively, dynamic QoS support, and parallel cluster-based processing and distribution.

Stanford Stream Data Manager [15] is a system designed for processing continuous queries over multiple continuous data streams and stored relations. It has the capability to process high volume data streams and large number of complex continuous queries.

Amazon Kinesis, a fully managed service for real-time processing of streaming data at massive scale, IBM Infostreams, a stream computing solution to enable real-time analytic processing of data, and Oracle CEP are some other key players in the field of complex event processing.

2.4. Distributed operators

With the distributed operator approach a CEP operator is run parallely in a set of CEP servers and results from each processing node is aggregated to create the final result. Unlike the above approaches, this solution is not limited by the capabilities of a single server as this is a distributed solution. As the distribution happens in operator level, single operator's processing power will not be limited by server's capabilities like in category 3 approach. The challenge if implementing this approach is finding ways to avoid usage of shared state used in the second category.

Wihidum introduces a set of distributed operators avoiding implementation of a shared state, utilizing data partitioning and pipelining.

In a broader perspective, Wihidum focuses on delivering a reusable architecture for scaling a complex event processing system. Unlike commercial solutions such as Integrasoft and Interstage explained earlier, Wihidum does not address dynamically scaling the processing power. Rather it depends on redeploying the given query as a set of sub queries in a distributed setting utilizing techniques (pipelining, partitioning and distributed operators) as we will explain in the later sections.

In order to cater internode communication regarding deployment of sub queries over the setting we are using Hazelcast. And Wihidum uses a custom transport based on byte streams over TCP connections that uses fixed sized messages for passing the events between processing nodes which are running Siddhi CEP engines. Commercial systems described earlier make nodes communicate for control information while complex event processing takes place. By sharing control information such as load balancing method, sub query to run on a particular node, the number of nodes which are running the same sub query, and the event pipelining information at the startup of the cluster over the HazelCast, Wihidum reduces the chance of internode communication congesting the network while data related to complex event processing is transferred.

Developed techniques and distributed operators make sure that the minimum amount of data duplication and retransmission happens in the running cluster to gain the expected efficiency. Similar to the way query deployment is planned in "High-Performance Complex Event Processing over Streams" [19], Wihidum suggests a method to redeploy stateful queries as a set of sub queries that can be deployed over a set of CEP nodes in order to process increased event loads.

In the following sections we explain the techniques we are using to distribute complex event processing and analyze the performance of the resulting system.

3. Wihidum design and scaling CEP operators

A query may include one or more operators. For example, the following includes a window operator and a filter operator.

from MeterReadings[load $>$ 5]#window.time(30 s)
select avg(load).

Wihidum runs queries using multiple nodes, and resulting system needs to yield the same result in the same order as if the query was executed in a single node. Furthermore, the system needs to evenly balance the load across the processing nodes for efficiency.

One potential approach to scale CEP queries is to use a shared state, where all nodes can refer to a shared state like a Distributed Cache while processing. We have earlier implemented a distributed CEP solution using a Hazelcast Distributed Cache, but it was proved to be too slow in our benchmarks where the distributed CEP only achieved 10,000 events per second while a standalone CEP node achieves as 10 times as much.

Wihidum uses three techniques for scaling CEP queries.

1. Partitions—partition events into multiple nodes such a way that the nodes can process event independently from each other without any communication. Given stream of events $E = e1..en$ that occurred one after the other and an operator op, we define an event partition (op, p_rule) where $E1$, Ek are partition of the set E according to the partition rule p_rule such that

   $\{\mathrm{op}(e)|e \in E\} = \cup_{i=1\,\mathrm{to}\,k}\{\mathrm{op}(e)|e \in Ei\}$,

   and order is preserved. We define two partition rules $p\_\mathrm{rand}(n)$, which randomly partition events to n partitions and $p\_\mathrm{attr}(n$, attr_name) which partition events to n based on an attribute attr_name.
2. Pipeline—break the query processing into multiple stages that are executed one after the other using multiple nodes. Given a list of operators op1, op2..op$N$, we define a pipeline $(p1, p2, ..pn)$ as a operator where pipeline $(p1, p2, ..pn)(e) = pn(..p2(p1(e)))$. We achieve distribution by placing each operator in a different machine and sending output of processing from one node to the node having the next operator.
3. Distributed operators—operators designed to run in multiple nodes. Given an operator op and an event stream S that occurred one after the other, the distributed operator for operator op is set of operators (op1..op$n$, op_master) and Event partition $E1..En$ according to the partition rule p_rule of E such that

   $\{\mathrm{op}\ (e)\ |e \in E\} = \{\mathrm{op}_{\mathrm{master}}(e')|e' \in (\cup_{i=1\,\mathrm{to}\,k}\{\mathrm{op}i(e)|e \in Ei\})\}$

   and event order is preserved.

Given a complex query composed using filters, joins, windows, patterns, partitions, pipelines, and distributed operators, Wihidum can execute it using multiple machines as described below. The following Fig. 1 depicts the setup.

1. Wihidum will place each filter, join, window, and pattern operators in its own node.
2. Wihidum places operators that are in a pipeline in series of machines that will send output from each operator to the following operator.
3. Wihidum takes a partition definition and runs the given operator using a partition of data. Each partition can run separately, and Wihidum makes a separate deployment of operators for each partition. Finally, Wihidum joins and reorders the results from all the partitions.
4. Given a distributed operator, Wihidum runs each operator against the defined event partition and joins results and reorders them as defined in the query.

CEP Queries may be stateful or stateless. Stateful queries remember some state when it matches events after processing the event (e.g. windows). In contrast, the stateless queries like filters do not remember any state after processing an event. We can use stateful queries with partitions only if events can be partitioned such that values calculated within partitions give same results as non-partitioned case. This correctness is included within the definition and it is up to the programmer to find a valid partition.

The next section explores how the above implementations are carried out. It is worth noting that this work does not distribute queries automatically into multiple nodes, rather describes how a user can use partitions, pipelines, and distributed operators to

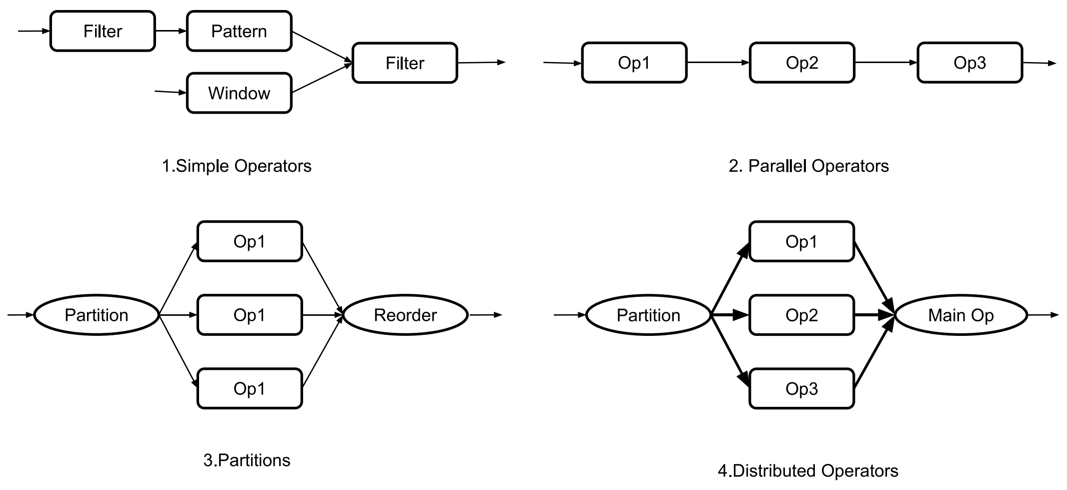

Fig. 1. Wihidum setup.

build a scalable solution. Composition of the solution rests with the programmer. As demonstrated by success of MapReduce, finding natural event partitions is something programmer can do well, although automatic partition is a useful problem as well.

Wihidum provides a set of distributed CEP operators (join and pattern matching) and a load balancer to balance event load among distributed CEP operators, which are useful to build such queries. However, this work does not provide distributed operators for windows or joining data in the disk, which we plan to cover as future work.

4. Implementation of pipeline, partition, and distributed operators

We will explain how to use pipelines, partitions, and distributed operators using the well-known "Fast flower delivery" use case introduced by "Event Processing in Action" [4] book.

Fast flower delivery scenario describes a flower shop. When the store receives an order for flowers, it chooses a driver by considering the driver's location and ranking and notifies the driver. Then the driver comes and picks up the flowers. If the driver does not pick up the flowers within a certain time (e.g. 5 min), the system generates a Flower pick up alerts. Delivery alert will also be generated if delivery does not occur within a certain period. Finally, the drivers are evaluated monthly based on the number of assignments completed each day and the system sends a notification if the driver has improved. Please note that the process of report generation based on the driver's performance has been excluded in from this discussion, for the ease of explaining.

Simple distributed event processing

For example, let us consider the situation where the flower stores asks to select drivers who are having ranking above a specified level. Since filter operators do not have any state, we can use data partitions to scale the system.

However, setting up such a system is not trivial, and the following picture shows how we can use Wihidum load balancer to distribute filter processing (see Fig. 2).

As per our definition, when f_rl is the filter (rank $>=$ 3), the above distributed query can be written as partition (f_rl, $p\_\text{rand}(n)$). This breaks the event to n random partitions and applies the filter operator. Those filters can run independently.

However, the actual implementation needs to handle distributed concerns. System may receive events via one or several load balancers, which will distribute the load among several filter processes. Wihidum handles those transparently from the user.

However, our definition requires ordering of events. In order to ensure ordering when joining results from different streams, the incoming events can be buffered and ordered according to their timestamp before the join operator. However, the ordering operator cannot detect when all events have been received, which information it needs to decide when to generate an output. To handle this problem, each load balancer sends a periodic timing event to each CEP node as described in Section 3, which will replay that timing event with the results. When the reorder operator receives the timing event t from all the event streams it listens for, it can reorder and emit all events that are less than time t.

Distributed partitions

Let us assume, in the flower delivery scenario, we need to count the number of assignments per day for each driver for each day on which the driver has been active. Furthermore, when a driver has improved his delivery count from the previous day, we should send an event.

We can implement the above scenario using following queries.

1. *define partition DriverPartition DeliveryEvents.driverID, DeliveriesPerDay.driverID;*
2. *from DeliveryEvents.windowTimeBatch(1day) insert into DeliveriesPerDay count(delivery), driverID;*
3. *from DeliveriesPerDay as e1,DeliveriesPerDay[e1.count $<$ count] as e2 insert into ImprovingDriver.*

The time batch window query will get the number of assignments per day for each driver. The output of time batch windows is directed to a set of pattern queries to get the drivers' performance. Then the patterns are used to categorize the drivers according to deliveries completed per day. A driver is considered as an improving driver if his number of assignments increases or stays the same day by day.

Let us assume f_win1 denotes window defined in line 2 and $f\_pattern1$ denotes the pattern defined in line 3. Then, we can write the above query as following.

partition(pipeline(f_win1, f_pattern1), p_attr(driverID,n)).

Using the model described in earlier section, we will place each partition in a different node and use a load balancer to distribute events to the appropriate partition. Wihidum will partition by looking at the driver name in the event. If an event with an already encountered driver name comes to the system, it will be directed to the same node to which the earlier events with same driver name were directed. If the driver name is a new one it can be directed to any node.

Fig. 3 shows the deployment. If required we can run batch query and pattern query in two nodes.

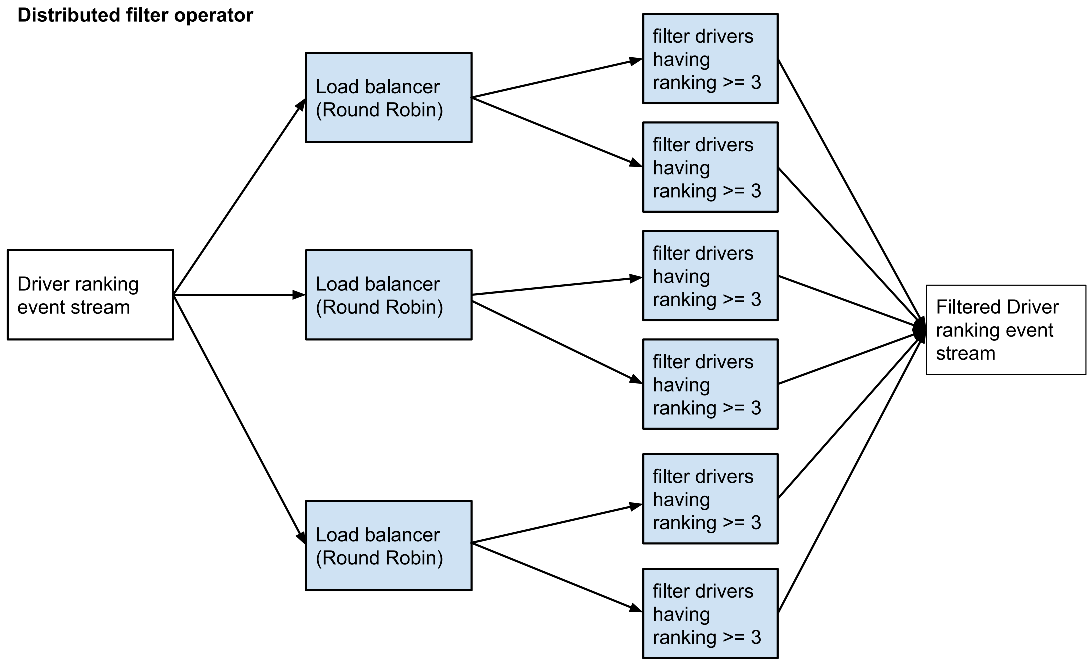
Distributed filter operator
Driver ranking event stream
Load balancer (Round Robin)
Load balancer (Round Robin)
Load balancer (Round Robin)
filter drivers having ranking >= 3
filter drivers having ranking >= 3
filter drivers having ranking >= 3
filter drivers having ranking >= 3
filter drivers having ranking >= 3
filter drivers having ranking >= 3
Filtered Driver ranking event stream

Fig. 2. Distributed filter operator.

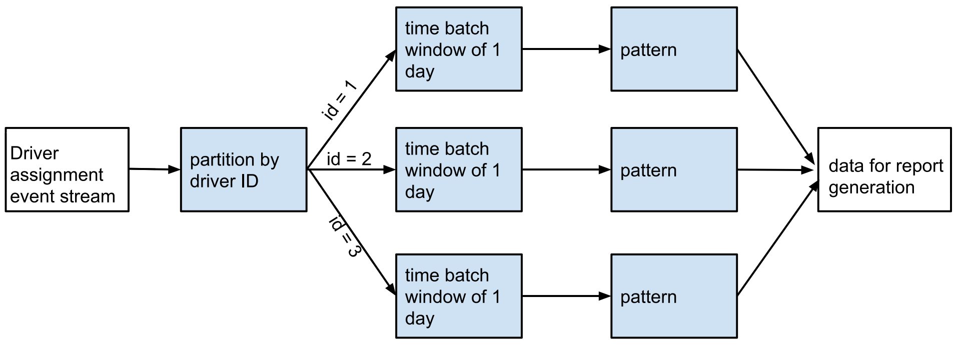
Driver assignment event stream
partition by driver ID
id = 1
id = 2
id = 3
time batch window of 1 day
time batch window of 1 day
time batch window of 1 day
pattern
pattern
pattern
data for report generation

Fig. 3. Event partitioning.

Using distributed operators: distributed join

Let us assume that the flower delivery system needs to find suitable drivers in close proximity given an order. To do this, it needs to join the delivery order stream and driver location streams. Often, event rate reduces after the filter operator. If the reduced rate can be handled by a join operator deployed in a single node, the problem is solved. Let us consider the case where the event rate is high even after filtering.

For example, let us assume we need to find drivers who are close to the pickup location and who have a rank above 3. In order to do so, we need to join two streams; stream containing driver ranking information and stream containing driver locations. Events that have same driver ID will be joined. Then the drivers can be chosen.

Wihidum implements distributed join operator by deploying the query in several nodes and partitioning the events among those nodes such that the events that have the same value for the join attribute will be send to the same node (in the above scenario join attribute is driver ID). Event partitioning is done using a hash function deployed in the load balancer to reroute incoming events based on value used for join condition. That is we define partitions $p\_attr(n, \text{driverID})$ and apply join operator to each partition.

When an event comes, a hash function is used on the value of the join attribute to get the node to which the event should be sent. The same hash function is used for both event streams mentioned in the query. Hence, each node will get events limited amount of incoming events with matching values in joining attributes, which will lead to improve performance compared to a query deployed in single node (see Fig. 4).

Distributed operators: distributed patterns

In the delivery process when the driver picks up flowers from the store, a pick up confirmation event is triggered. Then when the driver delivers flowers to customer, his/her confirmation triggers a delivery confirmation event. Then the system generates a delivery confirmation report. If the system does not receive a pick up confirmation event within five minutes of the committed delivery time, then the system will generate a pick up alert. In addition, if the system fails to receive a delivery confirmation within ten minutes of committed delivery time, the system will also generate a delivery alert.

Following query implements the scenario using a pattern. Here we detect the event pattern where CommittmentStream happens followed by timer event or confirmation event.

```
for every(
        s1 = CommittmentStream[s1.type = = "pickUp"]
          ->
        (s2 = pickUpConfirmationStream[s2.orderID = s1.orderID]
        or 10sTimer[s1.time + 5m > time])
)
insert into pickUpAlertStream s1.driverID, s1.orderID
```

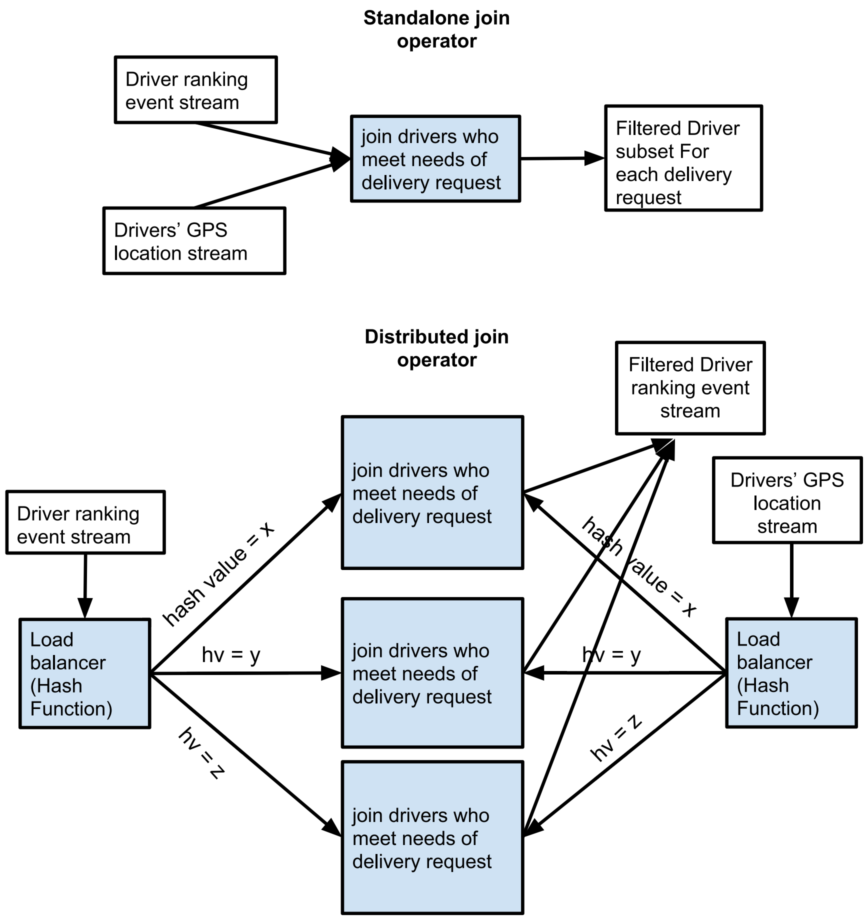
Standalone join operator
Driver ranking event stream
Drivers' GPS location stream
join drivers who meet needs of delivery request
Filtered Driver subset For each delivery request
Distributed join operator
Driver ranking event stream
Load balancer (Hash Function)
hash value = x
hv = y
hv = z
join drivers who meet needs of delivery request
join drivers who meet needs of delivery request
join drivers who meet needs of delivery request
Filtered Driver ranking event stream
Drivers' GPS location stream
Load balancer (Hash Function)
hash value = x
hv = y
hv = z

Fig. 4. Standalone vs. distributed join operator.

This query also scales the patterns by extracting filters as before. To make this query scalable, Wihidum takes the pattern query and extracts the filter operations first. Extracted filters are deployed in different nodes. Events that satisfy the filter conditions are sent to the pattern query. This will reduce the load coming to the pattern query because filters will discard all the events which do not satisfy the condition.

When distributing the above mentioned query, we extract out the filter query, which is [s1.type == "pickUp"]. Then this filter query is deployed in a separate node, which will direct only relevant events to the node carrying the updated window query (which does not have the filter condition). If necessary, Wihidum can deploy each filter query in more than one node, reducing the load of the nodes. Fig. 5 shows the distributed deployment.

5. Performance analysis

For the performance analysis, the partitioned filter and distributed join scenarios were implemented as explained below. Implementation was tested under four different deployment scenarios and event clients were used to generate required event streams. Below are the used deployment configurations.

1. Single standalone CEP server, which processes a given query. Results of standalone setting were used to compare the performance improvements between standalone and distributed setting.
2. Single load balancer with two CEP nodes. Distributed query was deployed in two CEP nodes and load balancer was configured to balance the event load among the CEP nodes.
3. Single load balancer with three CEP nodes. Distributed query was deployed in three CEP nodes and load balancer was configured to balance the event load among the CEP nodes.
4. Two load balancers with six CEP nodes, in which each load balancer divides events among three of the CEP nodes.

Tests were conducted with one(Machine 1) 32 core Intel(R) Xeon(R) CPU E5-2470 0 @ 2.30 GHz processor with 64 GB RAM running Debian wheezy (kernel 3.2.0-4-amd64) and three 8(Machine 2,3,4) core Intel(R) Xeon(R) CPU E5-2470 0 @ 2.30 GHz processor with 24 GB RAM running Debian wheezy (kernel 3.2.0-4-amd64). All these machines are connected over a 10G network.

Machine (1) was used to deploy CEP servers since they require most processing power. Machine (2) was used to run a load-balancer which will evenly distribute load among CEP nodes. Machine (3), (4) was used to run event clients which were used to produce events for performance testing.

First experiment tested the filter operation and used the following query, and the following graph depicts the results (see Fig. 6).

from driverRankingStream[rank > 30]
insert into elegibleDriversStream driverID, rank, location, availability;

As the graph depicts, distributed deployments achieved higher performance figures than the standalone deployment. The deployment consists of three servers has achieved less performance improvement per increased CEP server than the deployment consists of two servers. This effect of diminishing performance improvement when comparing single server setup against two server setup and two server setup against three server setup is caused by the single load balancer used in both servers settings. Load balancer

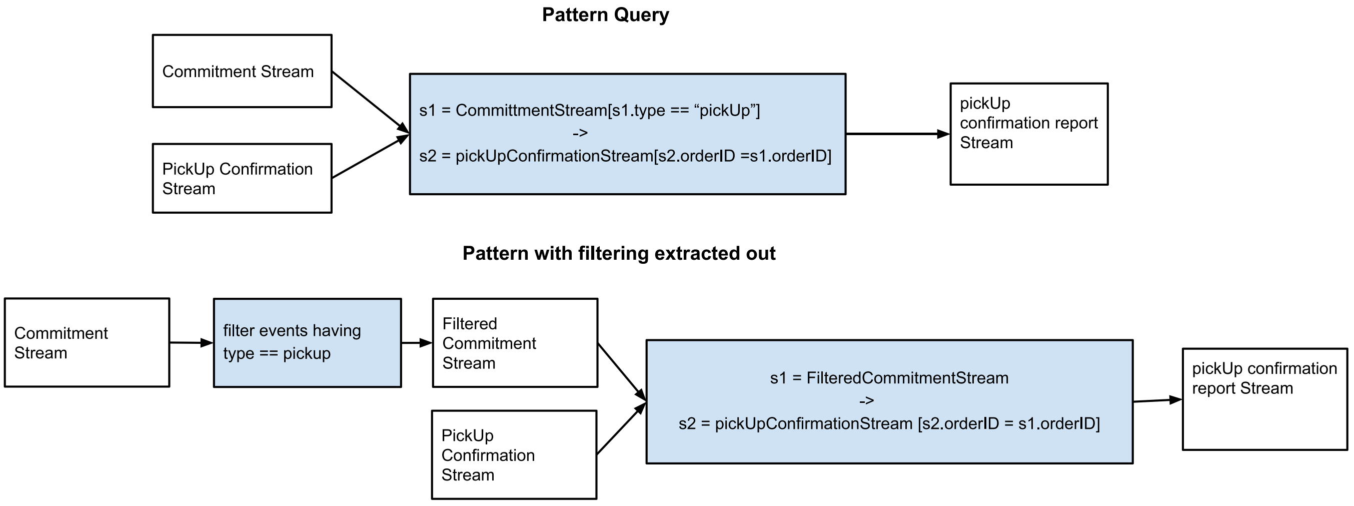
Pattern Query
Commitment Stream
PickUp Confirmation Stream
s1 = CommittmentStream[s1.type == "pickUp"]
->
s2 = pickUpConfirmationStream[s2.orderID =s1.orderID]
pickUp confirmation report Stream
Pattern with filtering extracted out
Commitment Stream
filter events having type == pickup
Filtered Commitment Stream
PickUp Confirmation Stream
s1 = FilteredCommitmentStream
->
s2 = pickUpConfirmationStream [s2.orderID = s1.orderID]
pickUp confirmation report Stream

Fig. 5. Standalone vs. distributed pattern.

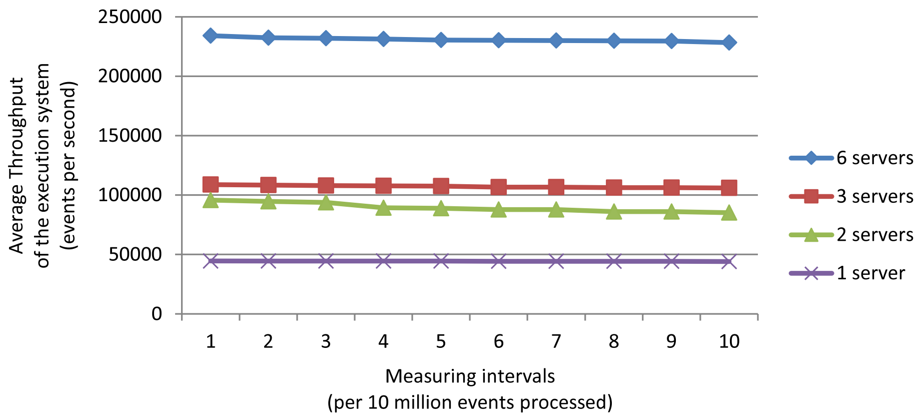

Fig. 6. Filter operation results.

has to balance the same load as in the two servers setting for the three servers setting which makes load balancer a bottle neck in the process. But as six servers setting has two load balancers balancing event loads with three CEP servers to be handled by a single load balancer, six servers setting achieves a better performance increase.

The following speedup graph shows that adding more servers improves the performance. Hence this shows that proposed approach is scalable. The minor performance degradation at 3 nodes is caused by load balancer overhead (see Fig. 7).

Second experiment tested the join operation and used the following query, and the following graph depicts the results (see Fig. 8).

from elegibleDriversStream #window.length(2000) as p
join pickupOrderStream#window.time(500) as c
on p.rank $==$ *c.rank into JoinStream*
p.driverID as driverID, c.orderID as orderID;

When observing the graphs of different join operator deployments, we can see a clear increase of throughput when number of CEP servers increase in distributed join deployment. In this join operator scenario, individual processing nodes may become bottle necks because each event stream is sent to a load balancer which balances load based on a hash value calculated based on a given attribute. As some nodes might receive increased load than other nodes, they will be more congested than nodes handling less traffic. Therefore we can see that the three node deployment has a higher performance increase compared to the two server setting unlike in distributed filter operator. As six server deployment has twice the resources for each stream when compared with three server deployment, it achieves almost as twice as performance achieved in the three node deployment.

6. Discussion

The paper discussed three techniques used for scaling queries: pipelining, partitioning and distributed operators. Then we discussed implementation for those techniques and empirical results shows that those techniques have improved the performance of the CEP solution.

To use Wihidum, users have to provide queries, details about which node each query should run, and how those queries are connected. Wihidum has implemented a load balancer to distribute events across multiple nodes and distributed operator implementations. When events come in, load balancers will route the traffic to the relevant node where event processing should take place. Sensors or event generators can directly publish to one of the load balancers. Wihidum can scale upstream transport by adding more load balancers to handle increased load as needed.

However, Wihidum does not handle automatic partition of queries, and users need to use basic constructs pipeline, partitions, and distributed operators to scale their queries. We believe the above approach draws a parallel between ideas like MapReduce that depends on programmers to find the best composition. In spite of the fact, MapReduce have proven to be a very effective tool for distributed executions.

Following are some of the approaches for query design, although they are not exhaustive.

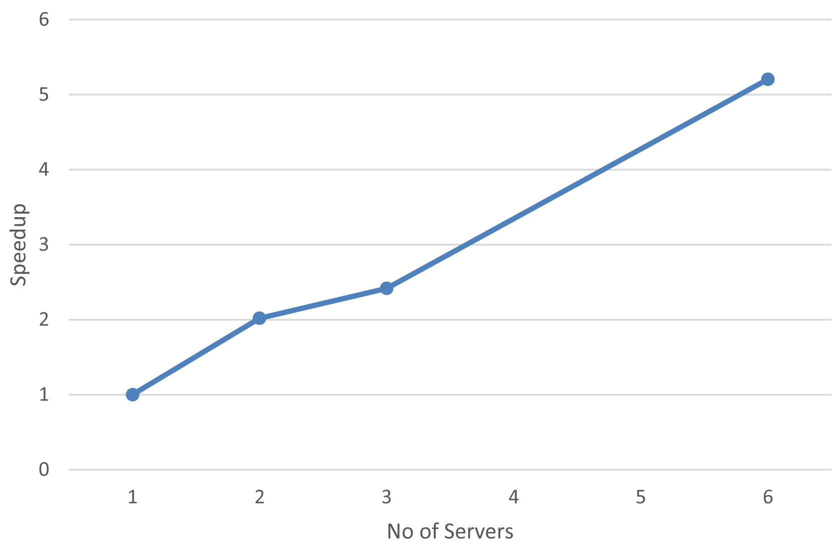

Fig. 7. Speedup relative to single server.

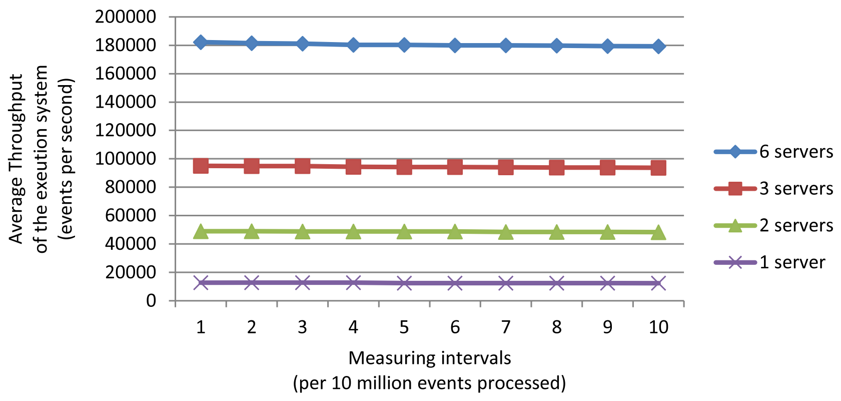

Fig. 8. Join operation results.

1. If the event streams can be partitioned, partition events and run all queries within a partition. This is equivalent to embarrassingly parallel problems in parallel programming.
2. Set up the pipeline of executions. Queries with longer pipeline can use more computers to solve the problem with this method.
3. Try to partition some parts of the pipeline into several machines. Since most queries will have lesser and lesser event rate received by succeeding operators in the pipeline, then partitioning operators in front of the pipeline can significantly improve the performance.
4. If 1, 2, 3 does not work, identify bottlenecks in the pipeline and use distributed operators.

It is worth noting that our CEP processing definition mandates the order. However, in some cases, that order may not be required (e.g. final output is a count) and the reordering option can be omitted.

A notable omission is the lack of support for windows as a distributed operator, which is a key factor when distributing a complex event processing engine. As a future work of this project, we are going to work on distributing window operators. There are different kinds of windows such as time windows, time batch windows, length windows and length windows. In order to distribute these operators, nodes should communicate with each other. For an example, in a length window, each node should notify other nodes about the number of events it has received. So the mechanism to communicate between nodes is required to distribute windows operators. However, time batch windows can be distributed without much effort. Assume that there is time batch window of one minute. We can deploy the query in several nodes and each node will generate an output every minute. Then, Wihidum can combine the output of each node and create the actual output event. For example, if the time batch window query is used to get a maximum value of a particular attribute, then each node where the query is deployed will generate the output that includes the maximum value among all the events it received. Wihidum can then take results of all window batch queries and select the maximum among those results to calculate the final output.

By inspecting the results of the distributed complex event processing operations, we observed that it is possible to gain significant performance increase. In addition, distributed setup can withstand high event loads such as 10 million and operate in a stable manner. However, we are distributing only some of the operators such as filters, joins, operators and performance gain is obtained by disassembling the given query and by processing partial queries in the distributed system and by combining partial results.

In summary, Wihidum meets expected goals of a distributed Complex Event Processing system by improving performance of CEP to meet increasing event incoming rates and by using pipelining, partitioning and distributed operators.

References

[1] E. Dede, Z. Fadika, J. Hartog, M. Govindaraju, L. Ramakrishnan, D. Gunter, R. Canon, MARISSA: MApReduce implementation for streaming science applications, in: Proceedings of the 2012 IEEE 8th International Conference on E-Science, e-Science, 2012, pp. 1–8.

[2] "Drools Fusion—JBoss Community". [Online]. Available: http://www.jboss.org/drools/drools-fusion.html (accessed: 12.04.13).
[3] "Esper—Complex Event Processing". [Online]. Available: http://esper.codehaus.org/ (accessed: 12.04.13).
[4] Opher Etzion, Peter Niblett, Event Processing in Action, Manning, Greenwich, CT, 2011, pp. 21–24.
[5] "Event Processing Glossary; Version 2.0". [Online]. Available: http://www.complexevents.com/2011/08/23/event-processing-glossary-version-2/ (accessed: 30.05.14).
[6] Fujitsu Develops Distributed and Parallel Complex Event Processing Technology that Rapidly Adjusts Big Data Load Fluctuations [Online]. Available: http://www.fujitsu.com/global/news/pr/archives/month/2011/20111216-02.html (accessed: 18.09.13).
[7] Fujitsu Interstage Big Data Complex Event Processing Server home page [Online]. Available: http://www.fujitsu.com/global/services/software/interstage/solutions/big-data/bdcep/ (accessed: 18.09.13).
[8] Integrasoft's home page [Online]. Available: http://www.integrasoftware.com/news.html (accessed: 18.09.13).
[9] Interview with Michael Di Stefano from Integrasoft on their CEP Cloud Services using EsperGigaSpaces | GigaSpaces on Application Scalability | Open Source PaaS and More. [Online]. Available: http://blog.gigaspaces.com/interview-with-michael-di-stefano-from-integrasoft-on-their-cep-cloud-services-using-esper-gigaspaces/ (accessed: 12.09.13).
[10] Geetika T. Lakshmanan, Ying Li, Rob Strom, Placement strategies for Internet-scale data stream systems, IEEE Internet Comput. (2008) 50–60.
[11] David C. Luckham, Brian Frasca, Complex event processing in distributed systems, August 1998.
[12] Nicholas Poul Schultz-Møller, Distributed Detection of Event Patterns, ImperialCollege, London, 2008, www3.imperial.ac.uk/pls/portallive/docs/1/55087696.PDF.
[13] ShariqRizvi, Complex Event Processing Beyond Active Databases: Streams andUncertainties, University of California at Berkeley, Report UCB/EECS-2005-26,December, 2005, http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-26.pdf.
[14] "SourceForge.net: Cayuga Complex Event Processing System—Project Web Hosting -Open Source Software." [Online]. Available: http://cayuga.sourceforge.net/ (accessed: 18.04.13).
[15] "Stanford Stream Data Manager." [Online]. Available: http://infolab.stanford.edu/stream/ (accessed: 30.05.14).
[16] StreamBase | Complex Event Processing, Event Stream Processing, StreamBase Streaming Platform. [Online]. Available: http://www.streambase.com/ (accessed: 18.04.13).
[17] G. Tóth, R. Rácz, J. Pánczél, T. Gergely, A. Beszédes, L. Farkas, L.J. Fülöp, "Survey on Complex Event Processing and Predictive Analytics,", July 2010. http://www.inf.u-eged.hu/~beszedes/research/cep_pa_tech2010.pdf.
[18] Satoshi Tsuchiya, Yoshinori Sakamoto, Yuichi Ysuchimotoand, Vivian Lee, Big data processing in cloud environment, April. 2012, http://www.fujitsu.com/downloads/MAG/vol48-2/paper09.pdf.
[19] Eugene Wu, Yanlei Diao, Shariq Rizvi, High-performance complex event processing over streams, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of data, 2006, pp. 407–418.

Sachini Jayasekara is final year undergraduate at Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka.

Sameera Kannangara is final year undergraduate at Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka.

Tishan Dahanayakage is final year undergraduate at Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka. His main interests are big data and distributed systems.

Isuru Ranawaka is final year undergraduate at Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka.

Srinath Perera is currently employed as Director of Research at WSO2 Lanka (Pvt) Ltd. His primary research interest is distributed systems, and his primary focus has been distributed management systems. In addition, he have done lot of work on Web services, E-science, and Grid in last few years. In general, he is interested in problems related to distributed architectures, distributed communication models (e.g. Pub/Sub system, P2P systems, Group communications, gossip), and their scalability characteristics.

Vishaka Nanayakkara is currently serving as Deputy Project Director/HETC Project and Senior Lecturer at University of Moratuwa, Sri Lanka.