STHITHIKA

COST-BASED QUERY DISTRIBUTION WITH QUERY RE-WRITING IN DISTRIBUTED COMPLEX EVENT PROCESSING SYSTEMS

Imiya Mohottige Thakshila Dilrukshi

(158211H)

Thesis submitted in partial fulfillment of the requirements for the degree Master of Science

Department of Computer Science & Engineering

University of Moratuwa

Sri Lanka

January 2019

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date:

Name: I.M. Thakshila Dilrukshi

The above candidate has carried out research for the Masters of Science thesis under my supervision.

Signature of the supervisor: Date:

Name of the supervisor: Dr. Surangika Ranathunga

Abstract

Complex event processing (CEP) is very useful in analyzing event streams and identifying useful patterns from them. Due to the distributed nature of existing applications, high volume of event generation and complex queries, using a single node CEP became problematic. One way to overcome this problem is to introduce multiple complex event processing nodes and distribute the queries among them for load balancing. However, due to the stateful nature of events, distributing queries among CEP nodes is not an easy task. Query distribution across CEP nodes is an NP hard problem.

This research is focused on the problem of optimally processing a large number of different event streams using a large number of CEP queries in a distributed manner. Optimization of query processing and distribution is done in two aspects: optimizing the individual query by introducing query rewriting, and optimizing query distribution across multiple nodes by introducing new factors to the query distribution algorithm. Cost of individual queries, number of event streams common to queries, CPU and memory utilization of nodes that run CEP queries, type of queries, and the number of queries in each node are the factors considered for query distribution. Usability improvement is done in two ways: adding standard communication by introducing JSON messages for communication, and integrating firebase messaging service to standardize the event source.

Experiments show that with these optimizations, compared to existing systems, STHITHIKA is capable of providing a higher system throughput, without making an adverse impact on event duplication or process load variance across processing nodes. It has the ability to handle higher number of queries compared to existing system. It is also more robust to event bursts. Due to the changes in query distribution and re-writing, time taken for initial query distribution has increased. Usability improvement enabled the easy integration with other technology and decoupling event source from the system.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Dr. Surangika Ranathunga, my research supervisor, all the lectures and MSc course assistants of Department of Computer Science and Engineering, University of Moratuwa for the continuous support given for the success of the research.

I would like to thank section head and the team members of web technologies team at Mubasher Technologies (Pvt) Ltd, for giving me laptops for testing and providing leaves when required.

Finally, I would like to thank my family and all the friends for being the motivation and encouraging me to complete the research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

CEP - Complex Event Processing

CN - Client Notifier

CPU - Central Processing Unit

DISP - Dispatcher

EP - Event Processor

EP-CTL - Event Processing Controller

HA - High Availability

NFA - Non-deterministic Finite Automata

NP - Non-deterministic Polynomial

PE - Processing Element

PEC - Processing Element Container

QP - Query Processor

QT – Query Type

RTL - Register Transfer Level

S4 - Simple Scalable Streaming System

SCTXPT - Scalable Context Delivery Platform

SOA - Service Oriented Architecture

SQL - Structured Query Language

1 INTRODUCTION

Availability of data in digital format increases exponentially with the advancement of technology. Real-time analytics of big data created the need for fast data processing. In highly competitive business environments, the ability to analyze real-time events and giving a quick response is highly advantageous rather than storing data in a database to be analyzed later. Real-time data analysis is helpful for decision making in areas such as the stock market, fraud detection in online transactions, health care, traffic analysis, etc. Existing applications achieve this objective [1, 2] with the real-time input stream analyzing capability using Complex Event Processing (CEP). Using CEP, high volume of event streams can be handled to identify complex events. Achieving high performance in a high volume of event streams is a major challenge.

There are several problems associated with single node CEP systems. Single point of failure is possible in an erroneous situation. The CEP node overloading is another possible problem when there is a high load of input event streams. A single node CEP system might not be able to handle complex and large scale event streams. There are difficulties in getting the expected performance when using a central server [3, 4] as well as scalability issues.

When using a single node CEP system, all the low level events generated by the geometrically distributed sources need to be sent to a single location, increasing the communication overhead. As a result of a high volume of events being transferred, network latency can increase. There are security concerns when low level events are being transferred across the network as low level data can be manipulated by third parties while being transferred across the network.

As a solution, distributed complex event processing systems were introduced, where queries used to identify useful patterns in event streams are distributed across the processing nodes. It is a Non- deterministic Polynomial-time (NP) hard problem [2, 4] because the parameters that affect for query distribution is much higher. There is no specific optimal way to distribute queries efficiently across processing nodes.

Parameters such as the number of shared events between nodes, the processing power of the CEP node and the underlying CEP engine can affect the throughput of the system. With Complex Event Processing (CEP), it is possible to analyze these data streams based on a given set of queries. A pre-defined rule set is available in each of the CEP nodes. Based on these set of rules, input event streams are analyzed and the output is produced. The rules in the CEP nodes are written using a simple query language similar to SQL. There are two techniques in distributed CEP; operator distribution and query distribution. In operator distribution, operators in a query are distributed across multiple processing nodes and input events need to follow a sequence of operators. In query distribution, queries are distributed across multiple nodes. Operator distribution is used to handle complex queries whereas query distribution is used to handle higher amount of queries. In this research, the main focus is query distribution.

1.1 Research Problem

There are multiple problems that need to be addressed in distributing CEP queries. Implementing a global algorithm is problematic when the number of queries and event sources are keep on increasing. The query distribution algorithm needs to scale up with respect to the number of queries and the number of events. The real problem in scaling up is query distribution. Some queries may not execute in high frequencies whereas some queries may execute frequently with respect to the incoming event stream. If the few queries that execute frequently belong to the same node, there is a higher probability of overloading that node while other nodes remain idle. There are many parameters that need to be considered for query distribution. Amount of queries, input/output streams, network latency, communication overhead, processing power of the nodes, allocated queries in a particular node, and statistics of the event stream are some of the parameters. Therefore, optimal distribution of queries among CEP nodes is an NP hard problem [2, 4] because the number of parameters affects for optimal query distribution is much high.

Due to the above-mentioned facts, distribution of queries among processing nodes is not an easy task. Queries should be distributed among processing nodes in such a way that each CEP node receives events without overloading. In such a system, heterogeneous event processing can achieve high performance by dividing processing into separate nodes [6].

The query distribution algorithm should be capable enough to distribute the queries among processing nodes without overloading and with minimum communication overhead.
The SCTXPT system for query distribution [3] maintains a threshold level when allocating queries to a node. However, correlated queries grouped into the same node can lead to overloading when there is a high volume of such events. Therefore, there should be a mechanism to predict the load in a particular node before allocating a query. Cost based query allocation is an existing solution for this. The VISIRI system is an improvement of SCTXPT which uses cost based query allocation [7]. In the VISIRI system, three main factors have been considered for query distribution; the number of attributes in a query, the count of input/output streams and the window length. However, the VISIRI system does not focus on parameters such as event type, or processing power of a node.

Most of the existing distributed CEP systems including VISIRI system focus on improving only the query distribution. It is possible to have alternatives that can improve the overall system performance other than the query distribution. Those factors are not considered in most of the CEP systems. But we can improve the overall efficiency by integrating new features such as query optimization. At the end, the actual execution is based on the underline query. Time taken to produce the output depends on the execution plan of the query. Therefore having optimized query execution plan will provide the fast responses. Since there can be thousands of queries in a long run, optimizing each single query will cause for high throughput of the system. If a node can produce fast responses, then the possibility of node overloading will decrease. Then the overall efficiency will eventually increase even though the query distribution cannot be further optimized. Therefore, query optimization is another aspect of improving the performance of distributed CEP systems.

Another aspect of CEP systems is the usability. Most of the existing distributed CEP systems including VISIRI system are mainly focusing on performance and efficiency. But in actual usage, usability of the system and ease of integration is matters. As an example, existing VISIRI system is tightly coupled with Java and it cannot integrate with any other technology. Therefore improving the usability and decoupling the technology based restrictions is required with the help of standard communication mechanisms. It will reduce the developer work when it needs to integrate with different environments.

1.2 Research Objectives

Main objective of this research is to improve the overall performance and usability of distributed CEP systems. To achieve the main objective followings needs to be done.

Maximize the throughput in a distributed CEP system, while handling a high query count. There should have a balanced resource utilization across all the CEP nodes without overloading. In order to achieve this, it is essential to build query distribution algorithms using the parameters mentioned above in section 1.2.

Improve the query execution plans of distributed CEP systems which will help to minimize the execution time of each single query. The lower execution time will contribute to achieve high throughput of distributed CEP system.

Finally, standardization of query distribution of CEP system since most of the distributed CEP systems are having technology based restrictions which have reduced the usability of the systems. Decoupling these technology based restrictions in order to improve the usability and introducing standard event source to the distributed CEP system as an enhancement.

1.3 Research Contributions

There are multiple optimizations and enhancements done as research contributions. First contribution is static query distribution algorithm optimization by introducing resource utilization and event type. Second contribution is individual query optimization within the distributed CEP system. The queries are optimized before the distribution between nodes. The queries are re-writing and checking for duplicates under query optimization.

There are two system enhancements. The Json messages are introduced to standardize the communication and reduce the technology based restrictions. The firebase messaging service is introduced to decouple the system from inbuilt event source and standardize the communication.

1.4 Structure of Thesis

The chapters of the thesis are organized as: Chapter 2 of the thesis contains the literature review, and Chapter 3 contains the research methodology. Chapter 4 is the evaluation, chapter 5 is the future work, chapter 6 is conclusion and 7 is references.

2 LITERATURE REVIEW

This chapter comprises of the background related to complex event processing. The chapter begins with an introduction to complex event processing and details about complex event processing engines such as Siddhi, Esper and Cayuga CEP engines.

Under the distributed complex event processing section, the need of distributed CEP system, the problems in a distributed CEP system and distributed CEP techniques are discussed. There are two ways to do the event distribution; operator distribution and query distribution. These two techniques and the research carried out in these two areas are discussed under the distributed CEP section. It includes detailed descriptions on Wihidum, COSMO, S4, Next CEP, Borealis stream processing engine, SCTXPF, WSO2 distributed CEP system and VISIRI distributed CEP system. The latter part of this chapter contains details on the query cost calculation.

2.1 Complex Event Processing

With technology advancements, most manual processes are replaced by computer systems. The availability of data in digital formats leads to the introduction of novel technologies. One such area is data processing. The ability to analyze data generated from different types of applications leads to real-time identification of unusual behaviors when events occur. Some examples of application domains are social networks, financial services, sensor networks (Weather, Spacecraft's etc.), web activities, health care, and business applications.

In the early stages, data processing was done by storing data in Database Management Systems (DBMS). It required data to be stored and indexed in persistent storage. Then when a user request comes through, data can be analyzed using a set of given queries. However, this does not achieve the real need as timing matters. Using persistent storage is asynchronous with respect to information arrival. For example, consider the data stream of credit card transactions. Identification of fraudulent behavior in real-time is important in order to prevent losses as opposed to analyzing the data later. Another example is data streams received from weather sensors. If there is an unusual weather pattern, it is better to identify it on the fly to prevent any losses. So the actual need is to be able to process data flows in real-time. This requirement introduces new concepts in processing data flows by

using a predefined set of rules. After several years of research, two models emerged. Those are complex event and data stream processing [8].

Information flow processing belongs to the data stream processing model. It processes the data streams from multiple sources and produce new output data stream. In contrast, complex event processing identifies an incoming stream as a notification of the events occurring, which need to be filtered in order to identify useful patterns of events occurring. This concept originated from the publisher-subscriber systems. Based on the interested content, the subscriber identifies required events. Complex event processing extends this concept to identify event patterns [9]. The architecture of CEP system is shown below in figure 2.1. The processing and routing events from sources to interested sinks is done by CEP system.

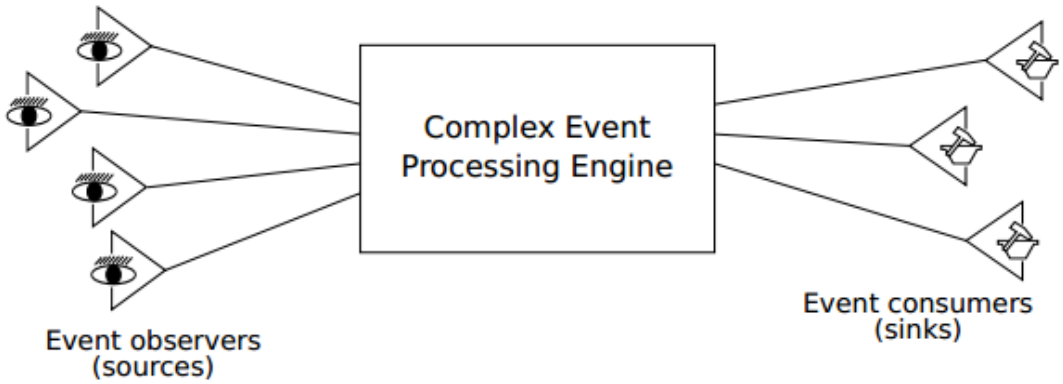

Figure 2.1- Complex event processing architecture [8]

2.1.1 CEP Engines

As shown in Figure 2.1 above, processing the event streams is done by CEP engine. The CEP engine provides the runtime to complex event processing. It executes the predefined queries on top of the incoming event stream. CEP engines accept queries given by the user, match the event stream generated by multiple sources against the queries given by the user and produce output when there's a match between the query and the input stream [10]. The performance of the CEP engine defines the performance of the query processing. Examples for event processing engines are Siddhi [10], Esper [11], Cayuga [12] and S4 [13]. These heterogeneous CEP engines are comprised of different capabilities that are suitable for their operation domain. Therefore, the CEP engine itself affects the query distribution.

2.1.1.1 Siddhi

Siddhi is an open source CEP engine and the architecture is shown in below Figure 2.2.

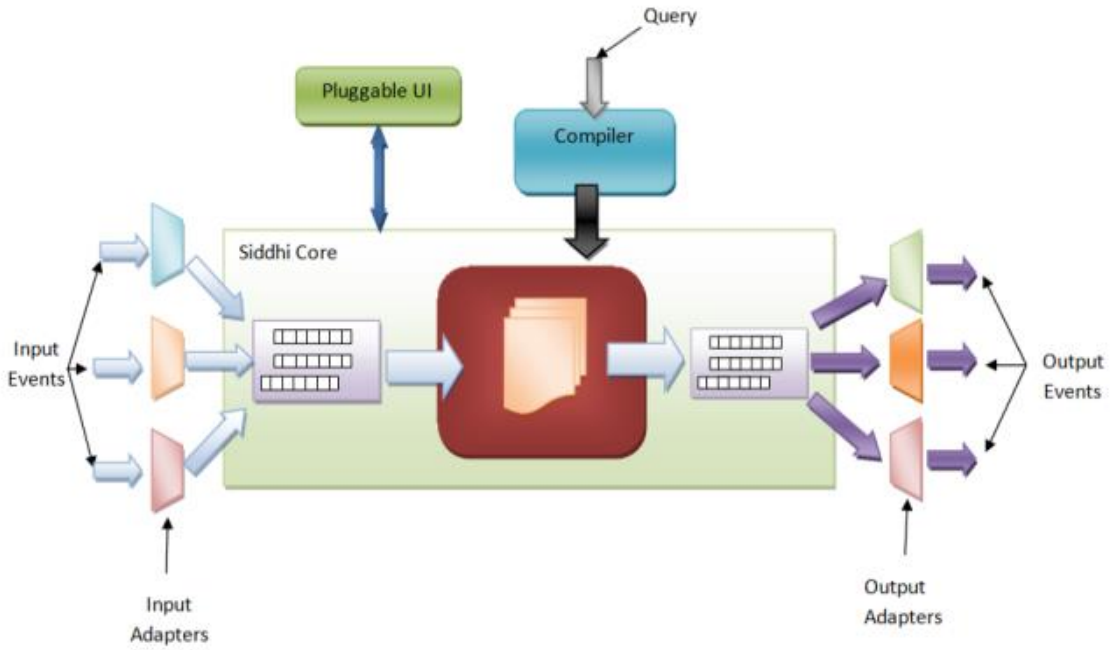

Figure 2.2 - Siddhi CEP architecture [10]

As shown in Figure 2.2 above, Siddhi CEP engine has input adapters, siddhi core and output adapters. Input adapters accept input events. Siddhi has its own internal representation of events. After an event is received, it converts the event to the internal representation. Siddhi

core handles all the operations performed on events. Siddhi core consists of queues and the processor. Input queue store the incoming events until the processor starts to process them. If an event matches with a condition defined in the queries, the processor selects that event for processing. Output queue is collecting the result events. These results can be delivered to third party subscriptions or can be used for further processing by another processor. Output adapters are used to send output events. The query compiler is used to compile the user given queries. Real time query processing is supported by Siddhi CEP. Adding and removing queries in run time is supported by CEP engine.

The Siddhi CEP engine comprises of an architectural design that enables high performance. One such design is event tuples. Inside the Siddhi core, all the incoming events are converted to its internal representation as tuples. These tuples are similar to a database row. There are a couple of advantages of this tuple data structure. It enables the use of an SQL-like query language, relational database optimization techniques are applicable and it gives higher efficiency than when using XML.

Siddhi uses own structured query language similar with relational algebraic expressions. Those queries contain simple SQL syntaxes such as SELECT, FROM, and WHERE. It enables the ease of use since everyone is familiar with SQL queries. Siddhi comprises of Java API to create queries.

Another important feature of Siddhi is that it enables complex queries. The output stream of one query can be an input to another query. The combination of these queries creates complex queries. This feature enables the reuse of common queries and eliminates common sub queries, which will help to improve the overall performance.

Siddhi has pipeline architecture. Multiple processors can connect together via event queues. The output of a CEP engine can feed as input to another and consequently send the event response to users. Other CEP engines such as Cayuga [12] uses single processor architecture. It is less complex compared to the Siddhi architecture and inefficient in resource utilization. The main disadvantage of this design is sub query duplication. Due to the use of complex queries, common sub queries can be available in complex queries. This is a performance factor that enables parallel processing. Nondependent queries can be processed in parallel at different stages of query processing leading to fast execution as well as higher throughput.

The Siddhi processor consists of two components; Executor and Event Generator. The processor processes an event taken from the input queue. If the output event matches with another processor, then that processor picks that output event for further processing. The Executor component handles this event selection.

Siddhi is comprised of a state machine. The state machine manages pattern and sequence queries. In pattern queries, if a particular event satisfies a series of conditions in a particular order, an alarm is fired, whereas in sequence queries an alarm is fired when a sequence of conditions is matched repeatedly. Siddhi also uses window queries. Both time based and length based sliding window and batch window queries are supported by Siddhi. The time sliding window observes the input events for a predefined duration with the purpose to analyze event arrival within a particular time interval. The length sliding window observes the number of events. Event processing is done by batches in the batch window. After the duration expires, time batch window provides the appropriate output and after completing the event count, length batch window provides the relevant output by removing the batch away.

Duplicate event detection is also supported by the Siddhi CEP. The user can define the criteria of duplicate events by specifying a particular set of attributes. Once the selected attributes are satisfied by an input event, it will be detected as event duplication and the relevant output will be produced based on the given criteria.

2.1.1.2 Esper

Esper is another CEP engine designed to process a high volume of events. Similar to Siddhi, Esper has its own event processing language with filtering, aggregation and joins. Esper supports sliding window based queries and pattern semantics for complex queries.

Esper is compatible with a multiple types of input event streams such as Java beans, XML, key value pairs etc. It is a Java based application, therefore integrating with existing java applications or using it as a middleware, has minimal overhead. Queries need to be registered with the Esper CEP container before use. After that, events feeding into the system will be

analyzed based on the queries and relevant output is produced as plain old java objects (POJO). This ensures easy integration with existing SOA architecture [11].

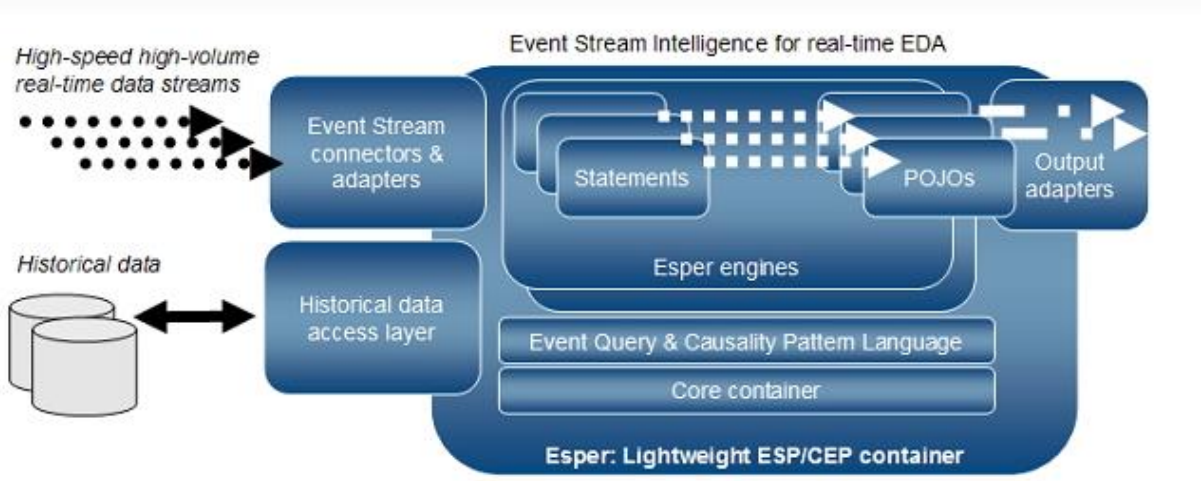

Figure 2.3 - Esper CEP architecture [11]

The architecture of Esper CEP is shown in Figure 2.3 above. Similar to Siddhi CEP, Esper also has input stream adapters, output adapters and an engine. Esper has a historical data access layer, which is not available in Siddhi CEP.

2.1.1.3 Cayuga

Cayuga [12] is another CEP engine similar to Siddhi and Esper. The architecture of Cayuga CEP is shown below in Figure 2.4.

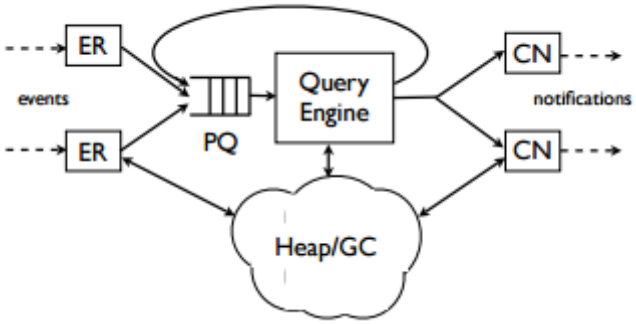

Figure 2.4 - Cayuga CEP architecture [12]

As shown in Figure 2.4 above, the Cayuga system is comprised of an Event Receiver (ER), Priority Queue, Query Engine and Client Notifiers (CN). Event Receivers take in external events and feed them into the system. Each ER has a separate thread, which receives events from a particular source. It helps to distribute the load among ERs.

The query engine is a single threaded application. It supports sequential processing. Heap/GC is a customized memory manager with a Garbage Collector. It helps to run the application with a small amount of memory. Efficient object sharing and high performance are also achieved via this memory manager.

Similar to Siddhi, Cayuga has its own internal representation of events since input streams can be encoded in different ways. After converting events to the internal representation, events are added into the priority queue. Dequeuing events from the priority queue happens based on the detected time of the event. Since the priority depends on the detected time, Cayuga has a correction for clock skew to overcome the timing errors that occur due to network delay, time of data source and event reordering.

The Client Notifier (CN) is used to send notifications based on the input events to the subscribed clients. One particular CN per connected client is used. CNs can subscribe to output events. When the query engine detects that a particular event has an output, the query engine sends an output matching that input to all the CNs subscribed to that output event.

The Garbage Collector is a special feature in Cayuga CEP that is not available in other CEPs. While event processing is happening, strings and objects are generated frequently. Space and time overheads can happen due to the large number of generated strings and objects. The Garbage Collector encourages object sharing.

2.1.1.4 Simple Scalable Streaming System

Simple Scalable Streaming System (S4) [13] is a general purpose distributed streaming platform with high scalability. It can operate with high input data rate and with high volume of data. The design goals are to provide a simple programming interface, cluster with high availability and scalability, maintaining local memory in each processing node to reduce latency, avoid using shared state, use a decentralized symmetric architecture, adopt to a pluggable architecture for customization, easy to program and flexibility. S4 has two main

assumptions. The first assumption is that failover loss is acceptable. The state of a process is lost when a server fails since the state is maintained in a local memory. The second assumption is that no runtime adjustments such as node addition or deletion in clusters.

S4 has a decentralized symmetric architecture. It has achieved a high level of simplicity due to the symmetric architecture. There is no centralized control over processing nodes. All the nodes are identical. S4 consumes event streams, compute intermediates values and emits streams. The input stream is considered as a sequence of events with a tuple of keys and attributes.

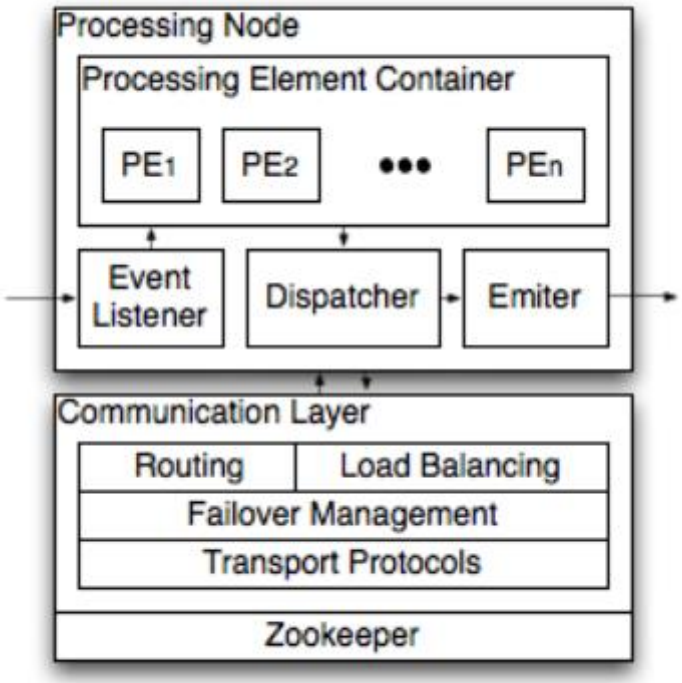

Figure 2.5 - S4 processing node architecture [13]

The architecture of a processing node of S4 is shown above in Figure 2.5. The basic computational unit of S4 is the Processing Element (PE). Each PE has four main components. These are the functionality of the PE, types of events the PE consumes, keyed attribute of the events and value of the keyed attribute. These four components are used to uniquely identify processing elements. Each PE only consumes key values that are an exact match. A new instance of PE is created if there is no corresponding key value. Keyless PEs, which assign keys to inputs events, are also available. These PEs can be removed for memory management. If a particular PE is in use, it can be deleted and the state of the PE will be lost.

Processing elements are located in the Processing Node (PN) component. PNs are the logical host of PEs. As shown in Figure 2.5 above, PNs do the following tasks, monitoring incoming events, query execution on input events, event dispatching with the use of the communication channels and emitting outputs. In a S4 cluster, multiple PNs exist. Routing event streams to these PNs happen based on a hash function. The hash function calculation uses the known keys in event. The multiple PNs can receive a single event. The S4 cluster configuration contains all possible keying attributes. The event listener of the PN receives the events and routes it to the Processing Element Container (PEC). Then the PEC will invoke the appropriate PEs in the required order.

The communication layer handles cluster management and does the mapping of physical nodes to logical nodes and automatic failure detection using failover management. In the communication layer, there is a pluggable architecture for network layer protocols. Coordination between nodes in the S4 cluster is handled using ZooKeeper. The ZooKeeper is a centralized coordination service for distributed applications.

2.2 Distributed Complex Event Processing

Details about complex event processing and complex event processing engines are given in the above section. However, we cannot use a single CEP in practice. There are some practical problems associated with single CEPs leading to the introduction of distributed CEPs. The problems associated with single node CEP are discussed below.

2.2.1 Need of distributed CEP

We can use a CEP node to process event streams from multiple sources and produce outputs. However, using a single node CEP can be an overhead in practice. Complex event processing is applicable in different domains including the banking and financial sector, network monitoring, sensor network, social network, web activities, health care etc. Most of these applications are distributed in nature, containing multiple components dispersed across several countries.

For example, consider a sensor network that analyzes weather conditions. There may be multiple server instances in different countries to collect weather changes and a data center

stationed in a different country. Analyzing data can happen from all over the world. If this system has a single CEP system located in the UK, collects data from USA, Canada, and Japan etc. and then needs to send the collected data to the central CEP in the UK, it can increase network traffic. It will increase the communication overhead and the network latency due to the high volume of events transferred across the network. Instead, if we can locate multiple CEP instances, then the network overhead will reduce.

In a single node CEP system, low-level events are transferred across the network. This causes security threats as low-level data can be manipulated by third-parties while being transferred across the network. If the transferring data comprises of sensitive information, there is a high security threat.

Another problem with single node CEP is the single point of failure. Node failure is possible due to a hardware issue, network issue etc. In such an adversary situation, the entire event processing will stop.

Scalability is another problem associated with a single node CEP. When the system needs to scale up, a single node CEP can be a bottleneck. As a solution to these problems, distributed CEP was introduced.

2.2.2 Problems with distributed CEP

Though there are problems with single node CEPs, it's easy to create and manage. Distributed CEP has practical problems that need to be overcome. In a distributed CEP, the load is distributed among a set of CEP nodes, which is not an easy task. It is not just balancing the number of queries in each node; we need to consider the probability of executing a particular query in an event stream and the correlation between queries before allocating a query to a particular node. Most of the time, a small subset of the operators is responsible for the performance bottlenecks. The easiest way to improve performance in such a system is to distribute the bottleneck queries among several processing nodes. If the bottleneck queries are stateful, dividing them among processing nodes becomes complicated [5].

When distributing the queries among the nodes, there are multiple factors that need to be considered apart from balanced load distribution. This depends on the query distribution and

the input events being transferred across the network. If the same event stream needs to be transferred across multiple nodes while processing, it causes an increase in the network overhead and increases bandwidth utilization as well as event duplication. Minimum event duplication, less communication overhead, low bandwidth utilization, using minimum resources, maximum throughput etc., are the factors that should be taken into account when distributing the load among processing nodes. Thus, optimal distribution of queries among CEP nodes while preserving balanced load is an NP hard problem [4].

2.2.3 Distributed CEP techniques

CEP queries can mainly be categorized into two types; stateful and stateless [5]. Stateless operators do not depend on the prior executed events. Filtering is an example of a stateless operator. Filtering does not require previously executed events and directly filters out the events based on the given conditions. In stateful operators, the current event being executed depends on the previously executed events, for example, window operation. Both time based and length based window operations require keeping track of previously executed events. While stateless operators can be easily distributed among CEP nodes, stateful operator distribution is more complex. For example, Figure 2.6 below shows some of the load balancing mechanisms in use [3]. According to Figure 2.6-a, if event A and event B are stateful operators and event B should execute after event A, if event A and event B are routed to two different processing nodes in a round robin manner, we cannot guarantee the execution of event B after event A.

We cannot use traditional load balancing mechanisms with stateful operators. There are existing techniques that can be used with stateful operators. One such mechanism is using a shared state [3] as shown in Figure 2.6-b. Hashing can be used to assign event streams to event processors [3,16]. It requires maintaining a shared state across the processing nodes. Since the shared state has a specific event allocated to a specific processing node, after event A is executed, a search can be made for event B using the shared state. However, this approach introduces an additional overhead of accessing shared states. To overcome shared memory access, the states need to be stored within the processing nodes.

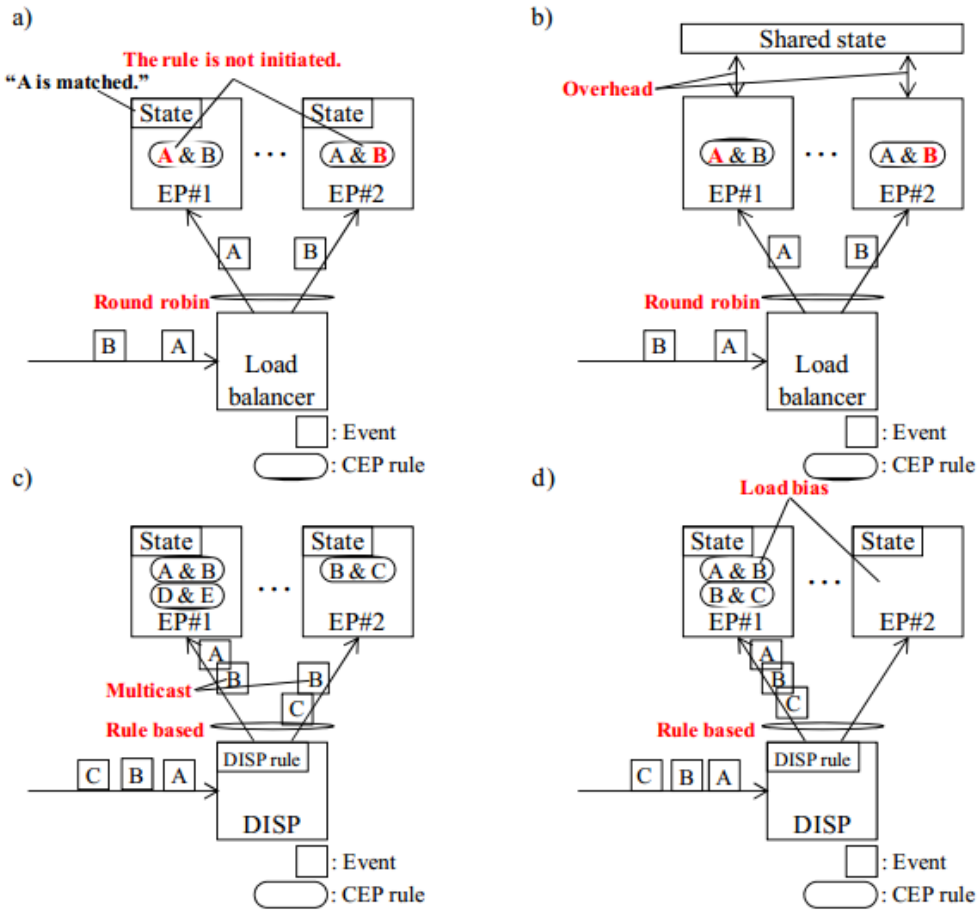

Figure 2.6 - Load balancing strategies in CEP [3]

Figure 2.6-c shows the use of the event dispatcher to dispatch events to processing nodes. The dispatcher (DISP) can dispatch the incoming event stream based on the rule set available in DISP. DISP rules contain the event condition and destination event processor pair. If event B is required for both processor 1 and 2, then it will be multicast to both event processor 1 and 2. This mechanism is introduced to overcome the overhead of too much multicasting. To avoid that, CEP rules related to the same event need to be deployed on one processing node. Figure 2.6-d above demonstrates that scenario, but it has unbalanced load distribution across the processing nodes.

There are two ways to do the query distribution; operator distribution and query distribution. Details of how operator distribution and query distribution can be used with existing mechanisms are discussed below.

2.2.3.1 Operator distribution

In operator distribution, the sequence of queries is distributed among nodes. There is a specific order of execution between distributed nodes. Operator distribution is achieved by using pipelining and partitioning. Wihidum [14], Next CEP [16] and COSMOS [4] are examples of operator distributed complex event processing applications. An example of operator distribution is given below:

Consider the following query:

From account [balance < 25000 and balance > 10000]#window.length(2000) select account No, account balance insert into bonus category

In operator distribution, operators are distributed into separate nodes. In the above query, the filter operation and window operation can be separated into two nodes. Then node 1 and node 2 will have the following queries:

Node 1:
from account [balance < 25000 and balance > 10000] select account No, account balance insert into insert into filtered accounts;

Node 2:
from filtered accounts#window.length(2000) select account No, account balance insert into bonus category

When the event stream contains events matching the above query, it is necessary to first route it to node 1 and then node 2. There is an order of execution between nodes in operator distribution. Wihidum, S4, COSMOS and Next CEP have used operator distribution, which is discussed further below.

2.2.3.1.1 Wihidum

Wihidum is a distributed complex event processor that uses operator distribution. Operators are distributed across the nodes and the final result is calculated by aggregating results from all the CEP nodes. Wihidum uses the Hazelcast caching framework for inter-node communication. By using this caching framework, Wihidum is maintaining shared state information including load balancing mechanism, the node sub query count, and event pipelining information at startup. It helps to reduce the inter-node communication between CEP nodes, which can cause network congestion.

Data partitioning, data pipelining and distributed operators are used in Wihidum to avoid the use of shared states across processing nodes. Partitioning is used to distribute queries among the processing nodes. Independent queries are partitioned across these nodes in order to prevent inter-node communication. In pipelining, the queries are divided into multiple stages and distributed across multiple nodes. There is a sequential order of execution in these stages. The main objectives of using this approach are to reduce data duplication and retrain missions to achieve high efficiency. Wihidum presents a methodology redeploy stateful queries. Stateful queries are distributed across the CEP nodes along with the sub queries required to deploy in the nodes.

The query can contains complex operations such as filters, sequence operators, joins, pipelines etc. When Wihidum receives a query with these complex operators, it distributes across multiple nodes. The setup of Wihidum is shown in Figure 2.7 below.

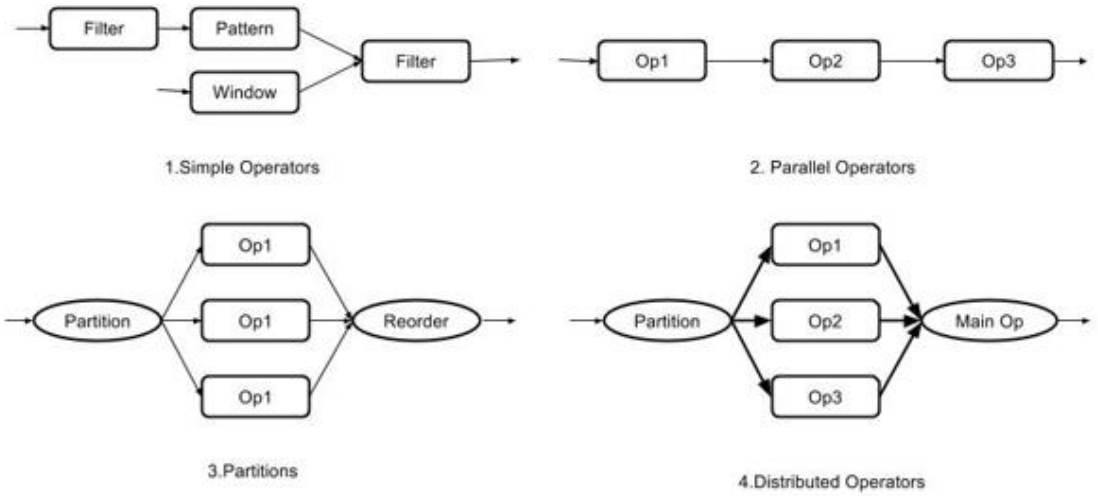

Figure 2.7 - Wihidum setup [14]

As shown in Figure 2.7 above, simple operators, parallel operators, partitions and distributed operators are the deployment strategies used. In simple operators, filtering, window and patterns are placed on the same node**.** Pipeline operators are placed in a node series when executing parallel operators. Partitioning executes the required operator on top of the partitioned data while distributed operator executes required operator on top of the events [14].

2.2.3.1.2 Next CEP

Next CEP is designed for query distribution and rewriting. The most important feature of Next CEP is automata based event detection. Similar to other CEPs, Next CEP has a high level query language and internal representation of events. Scalability is achieved by distributing automata among the cluster of nodes [16]. A cost model is used for query rewriting. Next and Union operators are the most commonly used operators for query rewriting. By using this cost model, greedily selects the deployment plan is selected to distribute queries for low resource usage in each node.

The main design goals of Next CEP are to receive TCP connections from clients, connect to sources, sinks and queries, multiple remote sources and sinks, multiple concurrent running queries and event automata models that can easily change operator semantics and an addition of new operators [24].

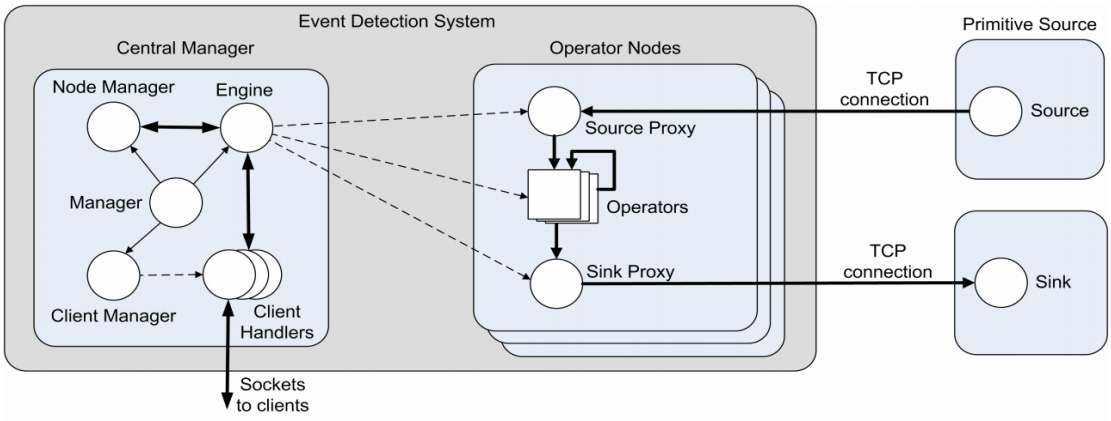

Figure 2.8 - Architecture of Next CEP system [24]

Erlang is the language used for Next CEP. The importance of Erlang, it has specifically designed for fault tolerant computing. Erlang is a language where processes are known as actors. The interaction between processes happens via asynchronous message passing.

The architecture of Next CEP is shown above in Figure 2.8. The Central Manager handles the receiving, processing, optimizing and instantiating of queries on an available node. The node manager is responsible for monitoring nodes, which are selected for operator distribution. The query receiving, transferring it to the engine and response returning are handled by the client manger.

The Operator consists of four different types of processes. The Guaranteed Detection Policy determines the stability of an event before consuming it. The Input process guarantees the stability of event-by-event buffering by other operators or until the next event becomes stable. Then the event is sent to the operator simulation process, which defines the operator types event emulating. The simulator processes the event and finally sends to other operators or to sink by the output process.

2.2.3.1.3 COSMOS

In distributed CEP systems, communication between event sources and consumers are tightly coupled. Typically in operator distribution, all the queries are collected and an operator graph is generated. Using an optimization algorithm, the distribution of the operators is identified in order to have a low communication cost. The main assumption is that, there is optimized

global operator graph. However, having optimal operator graph is not possible. Maintaining optimal operator graph is a challenging with frequent adding and removing of queries. As a solution, a publisher-subscriber model can be used. For better optimization of operator distribution, duplicate data transfer must be avoided, early data filtering must be performed and the query operators must be placed in the proper places. The placement of the operator must be determined by considering the data rate and the common data components as well.

The publisher/subscriber concept can be used to overcome the above problems. It allows content based filtering and has the inherited advantages of multicast. In multicast, there is at most one distribution of a message across a link. The important aspect is that we can avoid the tight coupling without maintaining the records source and consumer by using the pub/sub concept.

In COSMOS [4], Pub/Sub communication is used to achieve loosely-coupled communication as well as query optimization. For the query optimization, fine-tuning and re-arranging the operators from source to destination has done after analyzing the common sub queries among the queries. A new query distribution algorithm is proposed in COSMOS for placing operators.

In order to reduce the complexity and achieve fast adoption, the query load is distributed in the unit of queries instead of operators. By allocating operators across multiple nodes, synchronization is required during query processing, insertion, and removals. It affects the scalability as well as the loose coupling of the system. These problems were addressed in the proposed query distribution algorithm.

The proposed query distribution algorithm is more scalable and having an operator graph is not mandatory. Scalability improvements are achieved via hierarchical techniques. It uses the characteristics of pub/sub communication with the target of load balancing and communication cost reduction.

In this system, an N number of processors are interconnected via an overlay network. Data from sources will feed into the system by using these processors. Users can be connected to these processors and place the queries. User queries are similar to SQL. User queries are fed into the COSMOS middleware. It will handle the placement of query to the processor and

achieve the system performance. Not having prior knowledge about the network topology and publisher/subscriber components are the main assumption in query distribution. This is the basis to obtain loose coupling in the system. This query distribution algorithm is constructed with two objectives; balancing the load across the processing nodes and reduce total communication cost.

For load balancing among the processors, CPU speed is used as a parameter. The CPU time in an indication of the given query. The maximum load of the processor is given below. L is the total query load and C is the total capability of CPU.

$$(1 + \alpha) \cdot c_i \cdot \frac{L}{C}.$$

Here, alpha is added to the load imbalance and the practical value is 0.1, c_i is the CPU time used to record time in the processor and has a value of 1.

The total communication cost is the combination of two components. Those are transferrin cost from source to the processor and transferring cost from processor to the user. Weighted unit time communication cost is used.

$$\sum_{\forall i,j} r(n_i, n_j) \cdot d(n_i, n_j)$$

Where, $r(n_i,n_j)$ is per-unit time traffic (bit/s) on the link between n_i and n_j and $d(n_i,n_j)$ is the transfer latency of the link.

In order to obtain the objectives of query distribution, it is necessary to distribute the queries with lowest communication cost while maintaining the balanced load constraints. The query graph and the network graph are created to obtain those objectives.

All the processing nodes are included to the network graph. Data sources also belong to these processors. Processing capabilities and communication latencies are included as weights of the edges. The query graph includes the query vertex and network vertex. The query vertex represents the query and the network vertex represents the node in the network. Edges in the query graph represent either data requested from the data source by queries or the results being sent back. By using the network graph and query graph, a mapping that obeys the minimum communication cost must be found along with load constraints and minimum weight of the edges.

The runtime algorithm is also required since constructing query graph and network graph is difficult when the nodes of the network and queries are scaling up. Finding an optimal path in the network graph is NP hard and the statistics of queries and the network can be changed at runtime. Therefore, dynamically adjusting the algorithm is possible. However, static query distribution is the main focus over dynamic query distribution in this research. Therefore, dynamic adjustment is not considered in this research.

2.2.3.1.4 Borealis stream processing engine

The operation goals of the Borealis distributed stream-processing engine are dynamic resource management, query optimization and high availability [20]. Borealis is comprised of the inherited stream processing functionality from Aurora [21] and the inter node communication from Medusa [22]. Incremental scalability and high availability are the key reasons motivated for distributed stream processing.

The Borealis stream-processing engine accepts multiple queries and these queries will be distributed among the nodes for processing. The main components of the Borealis site are shown below in Figure 2.9.

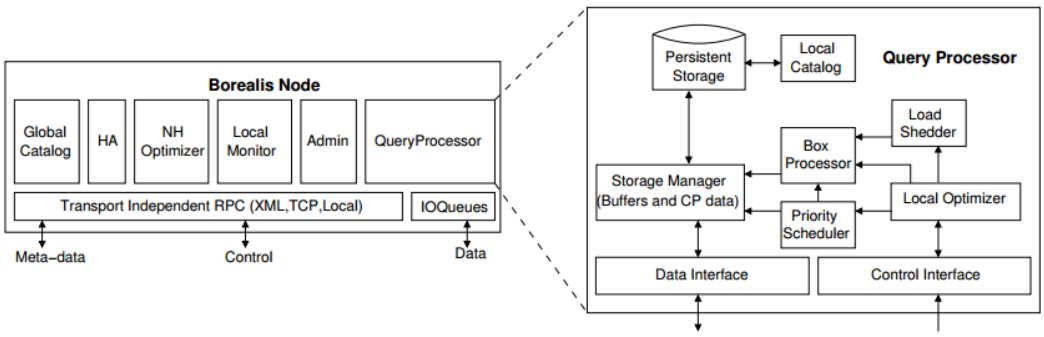

Figure 2.9 - Borealis Architecture [20]

The query processor is used for local query executions. The I/O queues are used to supply the incoming data into the query processor. Routing data among clients and nodes are also managed by I/O queues.

There is an admin module next to the query processor as shown above in Figure 2.9. It is responsible for controlling the query processor and coordinating with the local optimizer to perform local enhancements. The local optimizer is responsible for scheduling, updating runtime information and the load shedder. The load shedder discards low priority data in overloading scenarios. The role of Neighborhood Optimizer (NH optimizer) is the communication with other NH optimizers as well as maintaining balanced load among the processing nodes. Monitoring and handling failures are handled by High Availability (HA) module. The local Monitor is used to collect statistics.

The load distribution is based on correlation and has a dynamic load-balancing scheme. The purpose of load-balancing algorithm is to reduce the processing and latency among the nodes. The load-balancing algorithm in Borealis is used to balance the average load as well as minimize the variation among the nodes. It helps to minimize data queuing latencies and maximize the correlation between nodes. It will cause to the amount of dynamic load migrations.

Load shedding is responsible for handling overloading scenarios. The drop operator is added during load shedding. The drop operator is used to filter out messages. Load shedding helps to reduce the load on a downstream node. Borealis uses the distributed load shedding algorithm for load shedding. It will collect the statistics of the nodes. These statistics are used to pre determine the drop plan in compile time.

Recovery is required to achieve high availability. In Borealis, they focus on three recovery types. Gap recovery, which will discard the tuples in erroneous situations. The rollback recovery, which will restart the query processing from the last checkpoint, and precise recovery, which will recover from the correct failure point. Primary and secondary nodes are used for recovery. The primary node is responsible for sending periodic checkpoint messages to the secondary nodes and when the primary fails, the secondary can continue from the last checkpoint.

Borealis has initial static query distribution as well as dynamic query distribution, which will happen at runtime. The Borealis uses the CPU utilization as the fact for system loading [23]. It operates with the main assumption of load correlations between the node vary among

operators. In initial query distribution, minimizing the end-to- end time is the target, whereas in dynamic distribution, the goal is to achieve balanced load distribution.

By minimizing the average load variance or maximize the average load correlation [23] gives the average end-to-end time. The dynamic load distribution is achieved by using the pair wise algorithm. A centralized server will observe the load in each node and then order by average load. These nodes are paired by selecting the highest load with the lowest load, and so on. The operators are then moved between the nodes and the load is balanced between two nodes. There are two approaches in this algorithm; correlation based one-way distribution and two way redistribution. In the one-way correlation based algorithm, the most loaded node will move the load to the least loaded node. This will reduce the load movement. In the two-way correlation based redistribution, operators between the pair will be redistributed without considering prior location.

Global operator distribution is used for the initial query distribution. This algorithm has two stages. Obtaining balanced load across the nodes by reducing the average load is the first step. By selecting a node with a minimum load and assigning operators to that node is done in the first stage. In the second step, the average load correlation will be minimized. This will be achieved by further load balancing by a single round of pair wise one-way correlation.

2.2.3.1.5 WSO2 distributed CEP System

WSO2 CEP is a distributed CEP engine that supports real-time event detection, correlation and notification of alerts. It has a powerful GUI tool for monitoring [26]. Distribution is achieved by using the Apache Storm cluster. It’s a lightweight and easy-to-use CEP and is based on the Siddhi CEP engine. It supports various types of input and output protocols [25].

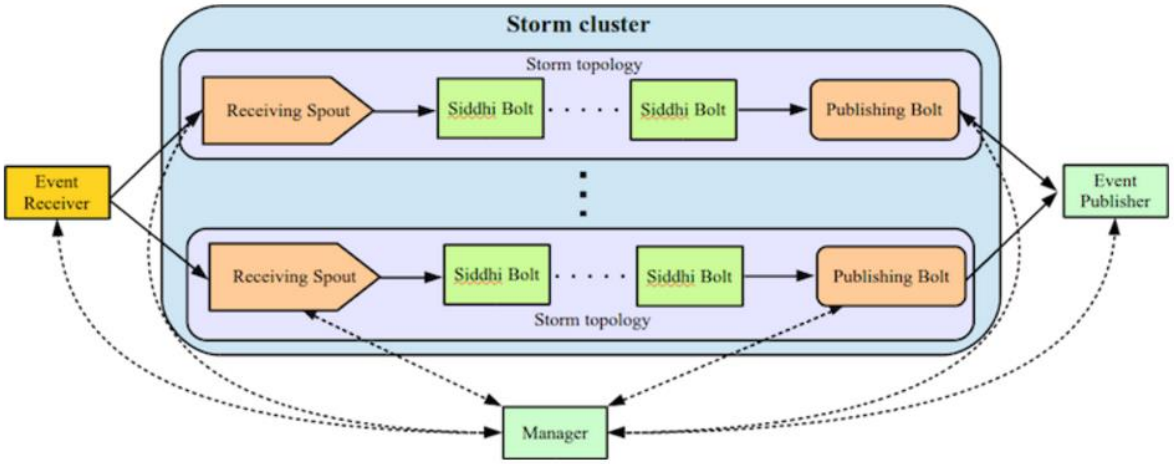

Figure 2.10 - System architecture of WSO2 CEP on Apache Storm [26]

As shown above in Figure 2.10, storm cluster topology is comprised of receiving spouts, Siddhi bolts and publishing bolts. Event receiver, manager, and event publisher are external to the storm cluster range. The same query language used by Siddhi CEP is used for the WSO2 CEP as well. Queries specified in the Management Console in the Siddhi query language are compiled into storm topology and will be deployed in the storm cluster. Event streams are received by an event receiver and will be fed into the storm cluster via receiving spouts. Then, the events will be processed in Siddhi bolts. The processed event outputs will be published into the event publisher via publishing bolts.

The WSO2 CEP comprises of a lot of features. It has the ability to handle the massive amount of event streams. This is due to the low latency engine used in the CEP system. Out of order event detection, data partitioning for distributed deployments, event stream filtering and transforming, temporal, logical and event sequence detection, supporting historical data stored in file system, external data sources and databases, dynamic editing and deployments of queries [26] are the features of WSO2 CEP.

It supports standalone fault tolerant deployment and distributed deployment with Apache Storm. The WSO2 CEP supports multiple data formats. Text, XML, JSON and map formats are supported as incoming event streams. For the data receiving and publishing, HTTP(s), JMS, SOAP, REST, Web sockets can be used. Another important feature is that it creates an interactive dashboard and configure the queries via interactive tool along with monitoring the system status.

2.2.3.1.6 ZStream

ZStream is a cost-based query processor for adaptively detecting composite events. It has the ability to process sequential patterns and detect other patterns such as conjunction, disjunction, negation, and Kleene closure. The tree based execution plan of the query is constructed. A cost model is introduced to calculate the computational cost of the query plan. Since this research is based on query cost calculation, the ZStream query cost calculation mechanism will be helpful to design the query cost calculation model. Usually, in order to evaluate CEP queries, non-deterministic finite automata (NFA) is used. However, NFA has a fixed evaluation order. The drawbacks of the NFA based query evaluation [27] includes inability to represent negation and difficulty to support concurrent event executions.

ZStream evaluates the events using a language tree. It constructs a physical tree plan and does the cost calculation. There can be primitive events with a single occurrence that cannot be split, composite events with a collection of primitive events, single class and multi class predicates for one and multiple event classes respectively. CEP queries have the following format:

PATTERN Composite Event Expressions
WHERE Value Constraints
WITHIN Time Constraints
RETURN Output Expression

The operators supported are sequence, negation, conjunction and disjunction, and Kleene operator. The connection between primitive and/or composite events are created by these operators.

As the first step in query processing, ZStream generates an internal tree representation of the query. A sample tree representation is given in below Figure 2.11. ZStream buffers incoming events according to the temporal order of leaf nodes and intermediates results in internal nodes. Time range sequence matching can be done with this buffer design. ZStream uses a batch iterator model, which periodically observes the nodes and takes actions based on the state. Events are discarded in leaf nodes and intermediate nodes in case of not reaching to the final state within pre-defined time duration. In idle rounds, the events are accumulated in leaf

nodes and populated in internal buffers. In the Assembly round, internal buffers are populated and an output is produced if the event has been received to the final events buffer. The Earliest Allowed Timestamp (EAT) is calculated in an assembly process. Any event with a time stamp before the EAT is discarded from processing.

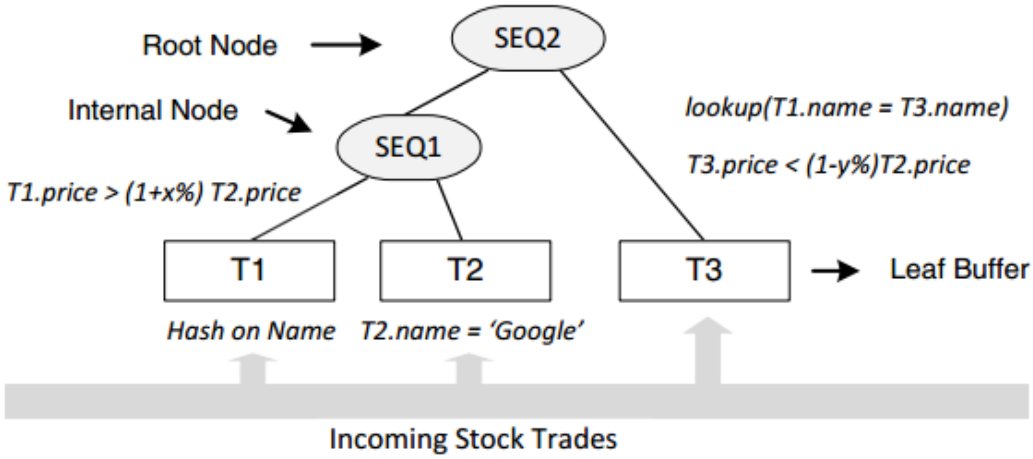

Figure 2.11 - ZStream tree representation [27]

Based on the operators that are used for the query, the query plan can be constructed. A query tree can be constructed for all the operators such as sequence, negation, conjunction and disjunction and Kleene operation. Based on the tree, the query cost can be constructed.

The optimal execution plan is identified by using the cost model in ZStream. A typical database estimates the cost of a query by using inputs, outputs, and the CPU cost. ZStream does not consider the I/O cost since the primitive events it uses are memory resident. It computes the CPU cost by three factors: cost of accessing input data, predicate evaluation as well as output generation cost.

$$C=C_i+(nk)C_i+pC_o$$

C - Total cost

C_i - Input data access cost

pC_o - Output data generation cost

$(nk)C_i$ – Predicate evaluation cost. ‘n’ is the amount of multi-class predicates

k - Weights

C_i, C_o – accessing input and assembling output data cost

2.2.3.2 Query distribution

Query distribution is allocating a set of queries among different nodes in order to minimize the processing load on each node and/or to minimize the communication overhead between nodes. In existing query distribution projects, network cost, resource utilization and event duplication are considered to distribute queries.

For example, consider a scenario with 1000 queries and 5 processing nodes. In query distribution, these 1000 queries are distributed among 5 processing nodes. It does not necessarily have to be 200 queries in each node. It depends on the frequency of events occurring, query complexity, query correlation, the processing power of the nodes, the CEP engine, window length, network latency, bandwidth utilization, the number of input events and the output events, etc. These queries need to be distributed among the 5 nodes in such a way that all the nodes have a balanced load with minimum communication overhead.

This research focuses on query distribution. There are existing projects with query distribution. SCTXPF [3] and VISIRI [7] are examples for query distributed complex event processing mechanisms. Details about those mechanisms and the pros and cons of those mechanisms are discussed below:

2.2.3.2.1 SCTXPF

The Scalable Context Delivery Platform (SCTXPF) is a distributed complex event processing [3] system designed to achieve load distribution of a CEP system by distributing queries among event processors. Effective allocation of queries to the CEP nodes in order to achieve scalability and high performance is another major concern. Scalability is based on a number of CEP rules and the volume of incoming traffic. The architecture of this system is shown below in Figure 2.12.

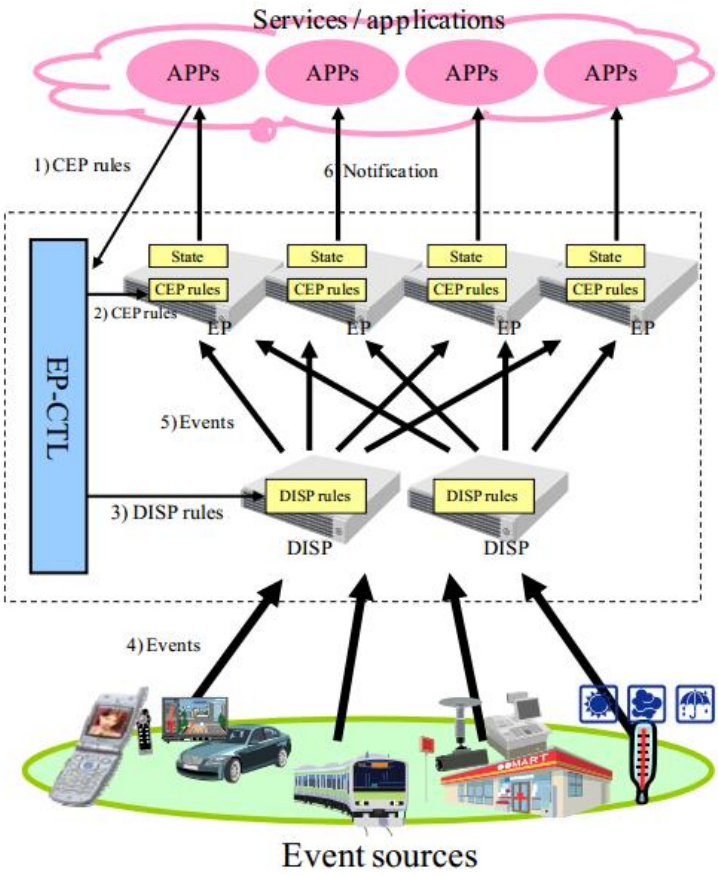

Figure 2.12 – SCTXPF architecture [3]

As shown above in Figure 2.12, SCTXPF has an Event Processing Controller (EP-CTL), Event Processor (EP) and a Dispatcher (DISP). The EP-CTL is responsible for managing the mapping of CEP rules to EPs. It generates DISP rules, which contain events that need to be dispatched to EPs. The DISPs will discard other events. EP handles the processing of events based on the given queries and produces outputs. DISPs are responsible for dispatching events to the relevant EP based on the dispatching rule [3]. The process of allocating a new query is given below:

New CEP rule will be added to EP-CTL. The query will be deployed in an EP and the dispatcher will be updated about the EP where the query was deployed. Once DISP receives an event from event sources, it will look up and find the matching EP for that query and dispatch the event to that EP. The EP will then process it and generate the output and places it on the relevant event sink.

The main objective of the query distribution algorithm used in SCTXPF is to minimize the number of CEP nodes that requires the same event stream and maintain equality between the numbers of queries deployed in each CEP node [3]. SCXTPF uses multicasting for event distribution if the same event is used by multiple EPs, as shown in Figure 2.6-c. Therefore, minimizing the number of multicasts and minimizing the EPs that use the same event are the responsibilities of the query distribution algorithm.

The query distribution algorithm used in SCXTPF tries to maintain a number of queries in EPs under the certain threshold. As the first step, it removes the EPs that have more queries than the $N_{threshold,}$ which will be defined compared to the EP with the least number of queries. Then, the EP with the most common values compared to the new query is selected. This selection helps to reduce the number of multicasts. If there are multiple EPs, the EP with the fewest conditions for CEP rules is selected. If multiple EPs are still present, one EP is randomly selected [3].

One of the main assumptions of this algorithm is that the processing load of each CEP rule is equal, and the load will not be summarized even if common CEP rules are put into the same EP [3]. This assumption is the main drawback of this query distribution algorithm. It does not consider the correlation between queries. Having an equal number of queries in each CEP node does not guarantee that the CEP node will not be overloaded. If two queries with high event frequency belong to the same node, then there is a higher possibility for that node to be overloaded.

2.2.3.2.2 VISIRI

Visiri presents a distributed complex event processing system with query distribution. It supports both static query distribution as well as dynamic query distribution. It focuses on reducing event duplication in order to minimize network traffic, reduce event duplication and achieve an even distribution of load among the processing nodes. The architecture of the Visiri system comprises of techniques that support these expected objectives. One of the special features of Visiri is that it supports both static query distribution as well as dynamic query distribution. Initial query distribution and query assigning to CEP node belongs to static query distribution. Query distribution at runtime belongs to dynamic query distribution. High-level architecture of the Visiri system is shown below in Figure 2.13.

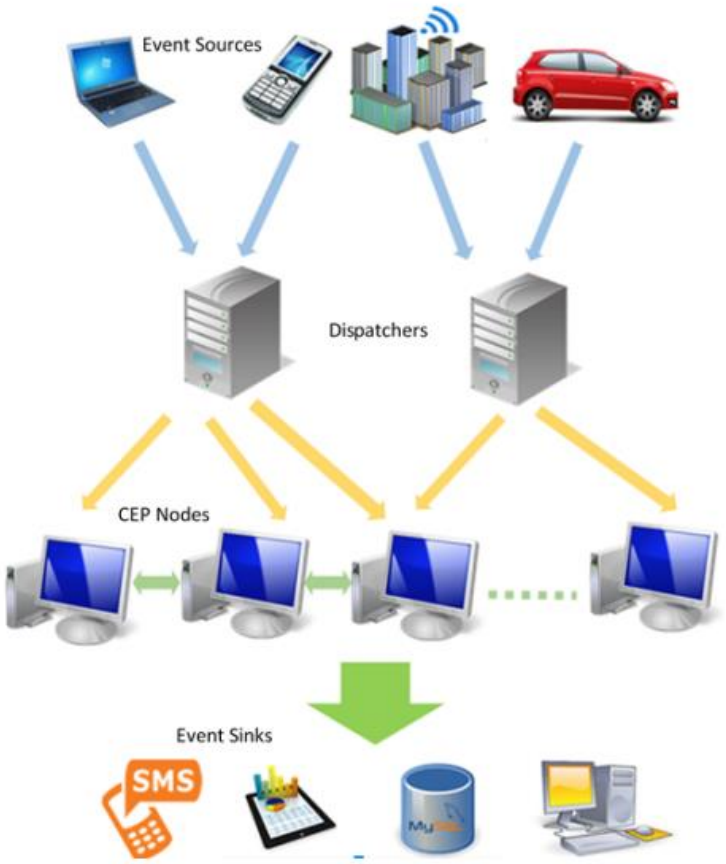

Figure 2.13 - High-level architecture of Visiri System [7]

The events are generated from event sources. The dispatcher controls the event assignments among the CEP nodes, which attempt to balance the load. Dispatchers need to have a mechanism to route events to CEP nodes in such a way that none of the nodes get overloaded. Then, after the processing of events, the output will be added to the event sinks.

CEP queries are deployed on the CEP nodes. When a new query is available, deploying the node of the new query depends on the query distribution algorithm. The dispatcher will be updated about the newly added query and the allocated CEP node. The dispatcher contains the forwarding table. The forwarding table contains the assignment of each query to the CEP nodes. The dispatcher controls the routing of events to CEP nodes based on query allocation. It is responsible only for sending the event to the relevant CEP node.

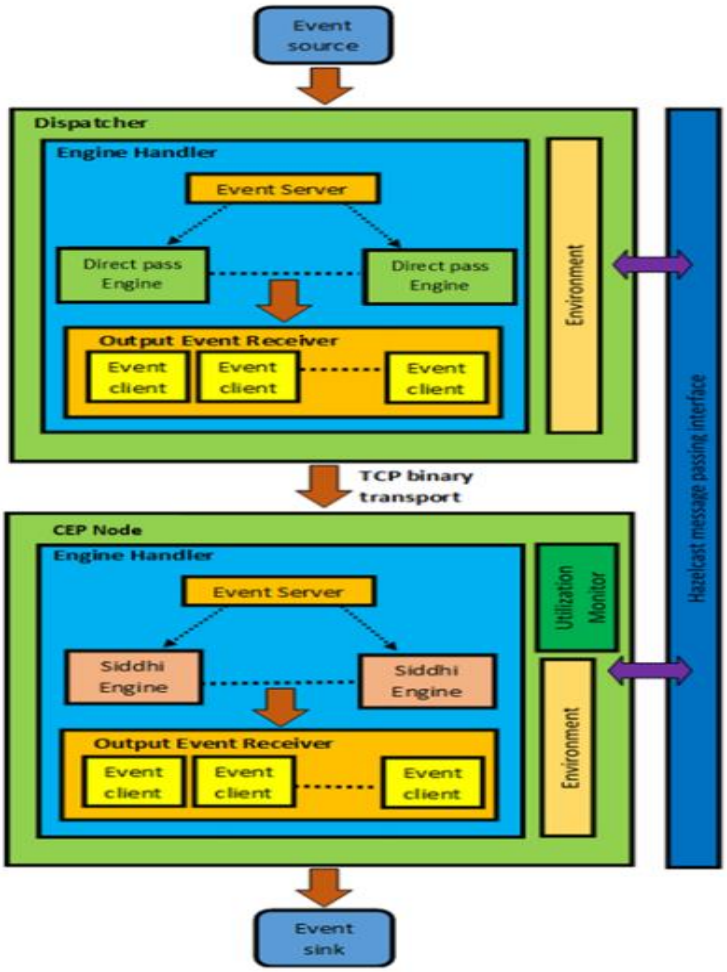

Figure 2.14 - Low-level architecture of VISIRI system [7]

System integration of the VISIRI system is shown above in Figure 2.14. Once the event stream comes into the dispatcher, it looks up the forwarding table and identifies which CEP node that particular event belongs to. Then, the event is placed in that particular CEP node. The TCP binary communication protocol is used to transport the event to a particular CEP node.

VISIRI uses Siddhi as the CEP engine. There is a Siddhi CEP engine per each node. Siddhi CEP engines are responsible for processing the input and producing the output. The Hazelcast caching framework is used for message parsing query allocation messages between the CEP nodes and the dispatcher.

Visiri uses the SXTPF query distribution algorithm as the starting point. The main difference between the SXTPF query distribution and the Visiri query distribution is that Visiri uses a cost model for query allocation. Queries with higher cost are not deployed in the same CEP node. This prevents node overloading. Having an equal number of queries across CEP nodes does not guarantee that there will be a balanced event distribution across the CEP nodes.

The Visiri system considers three main factors in query distribution. Those are:

- Cost of the query
- Number of queries in each node
- Number of common events required

The most important factor is the cost of the query. It depends on the complexity of the query. The VISIRI system has its own cost model. VISIRI Query distribution algorithm is given below Algorithm 2.1:

Algorithm 2.1: VISIRI Query distribution algorithm [7]

```
01 Distribute-Query(Query q,Node[] nodes)
02      candidates = nodes;
03      //find minimum queries
04      min-queries = min(nodes[0].queryCount,nodes[1].queryCount,...)
05      //filter nodes with too many queries
06      for node in candidates:
07      if node.queryCount> min-queries + QUERY_VARIABILITY:
08              candidates.remove(node)
09      // lowest queries
10      min-cost = infinity;
11      for node in candidates:
12      cost = sum(node.queries[0].cost,node.queries[1].cost, ...)
13      node.cost = cost
14      if min-cost > cost:
15              min-cost = cost;
16      // highest query filtering
```

```
17      for node in candidates:
18      if node.cost> min-cost + COST_VARIABILITY:
19              candidates.remove(node)
20      //find maximum common event types
21      q-inputs = q.inputStreams
22      max-common-nodes =[]
23      max-common-inputs = 0
24      for node in candidates:
25      commons = count(intersect(q-inputs,node.allInputs))
26      if max-common-inputs == commons:
27              max-common-nodes.add(node)
28      else if max-common-inputs > commons:
29              max-common-nodes.clear()
30              max-common-nodes.add(node)
31              max-common-inputs = commons
32      candidates = max-common-nodes
33      //select one randomly from the candidates
34      target = random.select(candidates)
35      return target
```

In the VISIRI query distribution algorithm, all the existing nodes are first taken as the candidate list along with the number of queries assigned to each node. The nodes are then removed from the candidate list, which have more queries than a certain threshold value. This helps to avoid the overhead of having a higher amount of queries in the same node. The next step is to calculate the minimum cost from all the nodes. The cost of a node is calculated using the sum of the costs of all the queries deployed on that particular node. If the cost of a node is less than the minimum cost, it will be assigned as a minimum cost. After calculating the minimum cost of existing nodes, the nodes with a cost higher than the minimum cost plus the variability are removed from the candidate list. By doing this, the cost will have a balanced distribution across the nodes.

After filtering out the maximum queries and higher cost nodes from the candidate list, the nodes that have a maximum number of event streams in common with the query need to be

deployed is selected. This will help to reduce event duplication since most of the events are common with the newly deploying query, and there is no need to send an event to a couple of nodes. Then, the node with the highest amount of events in common will be selected as the node in which to deploy the new query.

The arrival rate of the queries is not considered in this query distribution algorithm. Queries in common need to have higher priority since it directly deals with event duplication and network bandwidth. There is a possibility of removing nodes that have more common event streams than the selected one.

The main function of this query distribution is the cost of a query. The method of calculating the cost of the query directly affects the query distribution. The cost model used in the Visiri system is based on empirical studies done by Marcelo et al. [15] and Schilling et al. [6]. This cost model will give a numeric value for the cost of the query. These numerical values are assigned based on the importance of each factor. Higher the numerical value, higher the impact over the others.

The first factor to be considered for the cost of the query is the number of attributes in the query, such as filtering parts. A number of attributes in a query will increase the resource requirement of that query [6]. Therefore, the higher the number of attributes, the higher the cost of the query. The second factor to be considered is the count of input and output streams. This factor represents the impact of having a large number of streams in a query. The third and most affected factor is the window length, which increases exponentially with the window size. The cost model of VISIRI supports both sliding window and batch window with time or length. Finally, the logarithm of this numerical value, which represents the total cost of the query, is taken.

When comparing SXTPF and VISIRI query distribution algorithms, the VISIRI system has more options compared to SXTPF. The main difference between SXTPF query distribution and VISIRI query distribution is that VISIRI uses a cost model for query allocation. Queries with a higher cost do not deploy in the same CEP node. This prevents nodes from overloading. In SXTPF, all queries are assumed to have the same cost. Having an equal number of queries across CEP nodes does not guarantee that there will be a balanced, even distribution of load across CEP nodes. We cannot guarantee that all queries have the same

cost because it depends on the number of operators that the particular query has, as well as the frequency of events occurring in the input event stream.

When considering the VISIRI system, the existing algorithm of the cost model can be improved. The cost model has a relationship with the query engine. However, it is not considered in the VISIRI system. This cost model does not support join queries and pattern queries. The processing power of nodes is not considered. Query distribution has a relationship with the processing power of the node. Such attributes need to be identified via comprehensive performance analysis.

2.2.4 Query Cost Calculation

Query distribution is based on query cost. Therefore, effective ways of query cost estimation is required for better load distribution in a distributed CEP. In order to develop a cost model, the parameters affecting the query cost need to be identified. In order to develop a parametric model for the query cost, scope determination, data collection, data normalization, data analysis, data application, testing and documentation need to be done [17].

In cost model scope defining, defining the end use of the model, the physical characteristics of the model, the cost basis of the model, and its critical components and cost drivers need to be identified. In order to develop an appropriate cost model for query distribution, we need to consider the factors mentioned above. Based on the input event, the output will differ. However, there will be some outputs that are more critical than others. Without considering the number of input and output streams, a measure for criticality of the output needs to be introduced, which will introduce more cost to query. Other critical components affecting the query distribution such as operators need to be considered as well.

Pattern matching techniques are also important for the cost model and can either be temporal (time), dimension or direct filters. Temporal windows and dimension windows could be started and stopped based on set conditions [18]. Different costs depending on the window operator should be included for the query cost.

Queries can either be single query or multi query or operator placement [19]. Single queries have the performance of execution based on CPU usage, network consumption or processing

latency. Multi queries consist of overlaps between queries. Other than the CPU usage and network latency, the shared cost of query execution needs to be considered. The allocation of shared queries should have a minimum cost for the total query cost. Operator placement is the task of mapping each query execution into the set of available computational resources [19]. Operator placement should be done in such a way that query cost is minimized.

2.3) Query Optimization

Inside the CEP engine, queries execute and produce the output. Queries can be executed in different ways according to the query execution plan. Resource consumption performance characteristics will depends on the execution plan of the query. The goal of query optimization is to find the deployment plan or the execution plan with best possible behavior. In query distribution, this can achieve by re writing queries where as in operator distribution this can achieve by rewriting event patterns to better equivalent patterns [16].

Memory, CPU and network bandwidth utilization are the three main resources that cause for the performance of distributed CEP system. Bottleneck due to computation overhead in queries cause for high CPU and memory consumption even in low event rates. This can cause to reduce the throughput and scalability of the whole CEP system as well as overloading the system. Low detection frequency can be the result of such situation [16].

2.3.1 Query re-writing

Query re-writing comes as a part of query optimization. There is some research that has been followed in query re-writing but mainly those are focusing on operator distribution [16] which focus on distribution of query within multiple nodes [28].

Query re-writing mechansms in operator distribution aim to reuse existing operators to minimize CPU usage and latency. Receiving, processing and sending network packets consume CPU resources and traversing network links increases latency. By reusing existing operators, network packet process and traversing can be reduced. Patterns with union operator, patterns with next operator and patterns with exception operator are selected for query re-writing in operator distributed CEP systems [16]. Commutative and associative

behavior of union operator is the main focus in query re-writing. Example for commutative and associative property is given below [16]:
Ex: E1|E2 ≡ E2|E1 and E1| (E2|E3) ≡ (E1|E2) |E3

Cost of the execution of above queries depends on cost of E1, E2 and E3. Therefore arranging E1, E2 and E3 with minimum cost gives the best execution plan. The next operator is associative and thus E1; (E2; E3) ≡ (E1; E2); E3 holds. The cost of each of the two event patterns depends on the properties of E1, E2 and E3.

In both union and next operators, the lowest cost pattern can be found by enumerating all equivalent patterns and computing each cost. The exception pattern E1; (E2\E3) is equivalent to (E1; E2)\E3 because the terminating pattern E3 only influences the composite event detection after the next operator has detected E1, which does not depend on E2. Therefore this can process same as next operator [16]. This is the way of handling query re-writing in operator distribution where it will create multiple operator distributed query execution plan and get the execution plan with minimum cost for actual deployment. For this algorithm, there are four main assumptions: the deployment nodes are dedicated servers in a data center with only the CEP system running, network links between nodes are not congested, the source streams are Poisson distributed and the average event rate is available or can be measured and operators only consume CPU cycles when processing events, yielding the CPU when idle [16]. But in actual deployment environment, above conditions might not be satisfied.

Query re-writing in stream processing uses semantic analysis [29] as the method for query re-writing. First it transforms the query based on its ontology and creates a new query equivalent to the actual query. Transformed query can be unfolded and expressed as a union of conjunctive queries, as long as some restrictions are imposed on the expressivity of the ontology language [29] that allows converting them to languages like SQL without using advanced features like recursion

Another way of query re-writing is using a syntax parser and syntax tree. Event patterns and context are included in the syntax tree. Both pattern and context can be composite. Query analysis is divided into four parts: context resolving and event stream filtering, data partitioning, query plan generating and executing. Some context contains parameters and we

need to resolve the parameters first. The parameters can be resolved in three ways: event database, ontology and sub-queries.

Event partition can be done in three ways: spatial context where space is partitioned into N areas first, event sequences portioned in to these N areas, and segmentation context and temporal context. In query plan generation and execution, a sub query is generated for each partition and all the sub queries are parallel executed. Sometimes results of these sub queries need to share with other nodes. As an optimization, send the temporary result only to the selected proper local node instead of the main global node [28].

Even though the existing query re-writing mechanism support operator distribution and stream processing, extending these concepts for query distribution is still possible. This is not complicated as operator distribution, since optimizations should be done to the whole query instead of operators separately.

3 METHODOLOGY

As mentioned above, there are two types of distribution; query distribution and operator distribution. This project only focuses on query distribution. Since the VISIRI system has a proven architectural solution for distributed complex event processing, it has been used for this project.

Before moving into query distribution, individual queries can be optimized. Query rewriting is one such method of optimizing the query and it gives a low cost execution plan for the query. Query rewriting is the first optimization of the system. There can be duplicates in queries. Removing these duplicates will be done after rewriting the queries.

The next optimization is making improvements to the existing query distribution algorithm. The VISIRI system already has a query distribution algorithm based on the query cost. In the existing system, the number of attributes in the query such as filtering parts, the count of input and output streams, and the window length, which increases the cost exponentially with the window size, are considered for query cost calculation. Number of queries in a node, cost of the query and maximum common inputs are considered for query distribution. Other than the above considered parameters, there are other factors such as the correlation between queries, query type, processing power of nodes, and memory consumption of the node and CEP engine. Enhancing the VISIRI system query distribution algorithm by considering query type and resource utilization and analyzing the impact of those factors on the overall performance has been done in this project.

Standardizing the event source is an enhancement done in the system. For the existing VISIRI system, there's no standard event source. It has its in-built event source, which is tightly coupled with the application. Google firebase for real time event sending has been added to the application in order to standardize the event source.

Standardizing communication is another enhancement done in the system by adding standard JSON messages to communicate within the application. One drawback of the VISIRI system is that it support only Java applications. To avoid that drawback, standardized JSON messages are introduced. This will provide easy integration with any other technology, since standard message parsing is used instead of tightly coupled Java string message parsing.

3.1 Changes to the VISIRI system

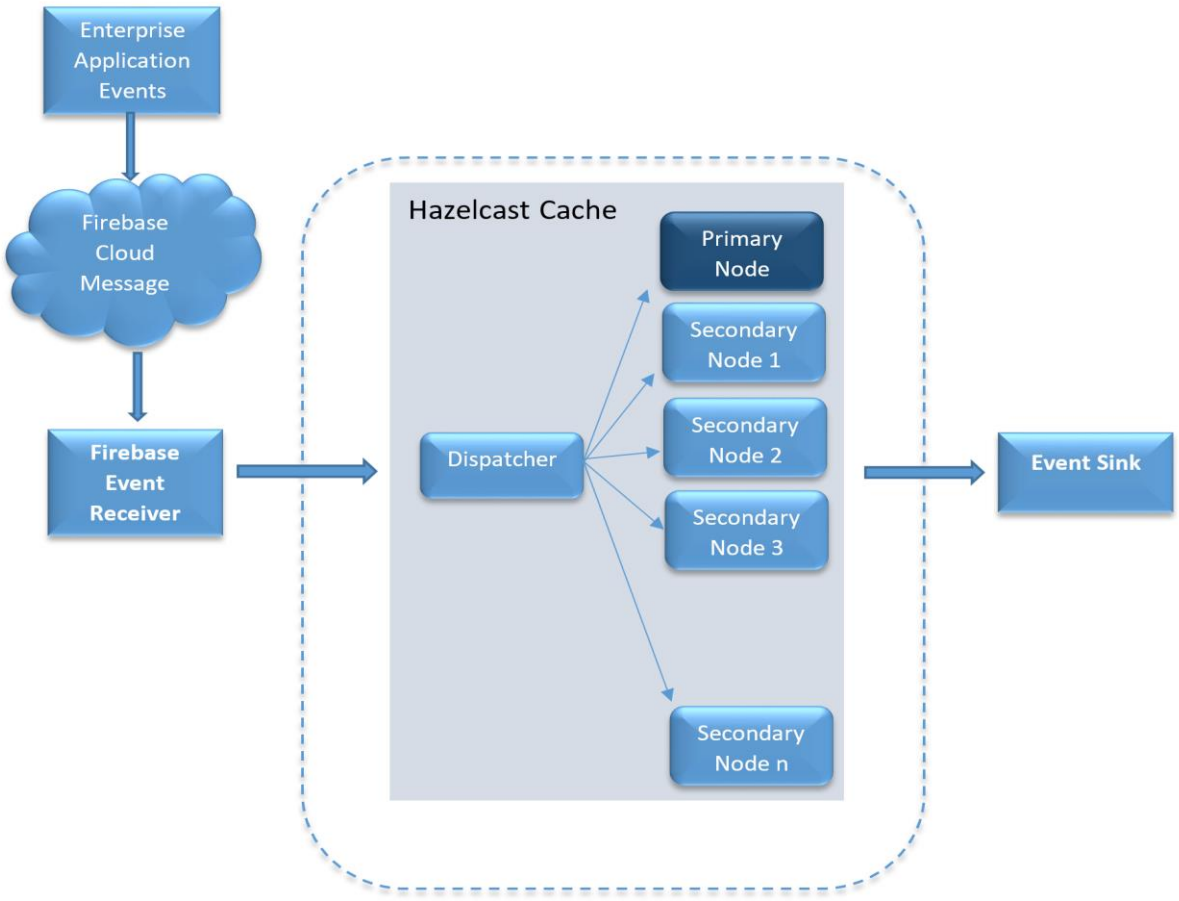

Figure 3.1 - Modifications to VISIRI Architecture

Figure 2.13 and Figure 2.14 show the VISIRI High Level and Low Level architecture respectively. Figure 3.1 and 3.2 show the changes in the existing VISIRI system due to the optimizations.

The existing VISIRI system is not compatible with external sources. Therefore, tight coupling with event source is removed from the VISIRI system and standard event source is introduced to the system by integrating with the Google Firebase push notification service.

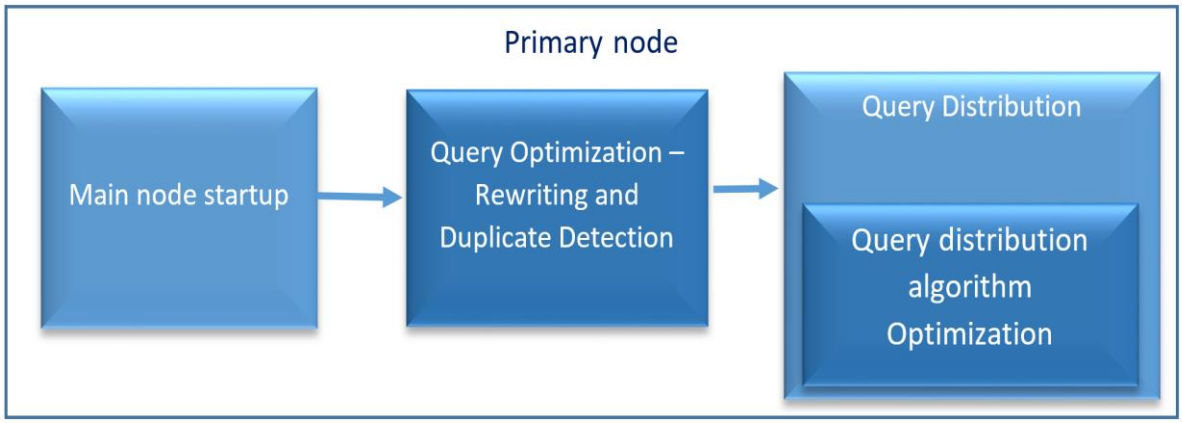

Figure 3.2 - Modifications to VISIRI Processing Node

In the existing VISIRI system, query rewriting is not available. Instead of focusing more on improving the query distribution algorithm, execution of single query optimization can give better performance. Therefore, query optimization is added in between the main node starting up and query distribution, as shown in Figure 3.2 above.

There were few factors that were missing in the VISIRI system, such as processing power of each node, event type, correlation between queries and the underlying CEP engine. By adding to the processing power of each node and event type, the query distribution algorithm was improved in this research.

3.2 Query Optimization

Each query has a cost of execution that decides how much memory the CPU needs in order to execute the query. The lesser the cost, the lesser the resource utilization. When we add a query to VISIRI System, there can be multiple query execution plans that give the same results. The cost of each query execution plan will be different. Low cost query execution will guarantee better overall performance in query execution.

ZStream [27] uses a tree-based query plan with operators in leaf nodes and intermediate results in internal nodes in runtime as given in section 2.2. Opposite to that, tree structure is created in query deployment time and recognizing re-writable nodes and optimizing has done under query optimization.

The queries are distributed in to the processing nodes based on the query distribution algorithm and the dispatcher is updated with query allocation. In query rewriting, queries are evaluated and rewritten to produce the lowest cost query execution plan. The query re-writing has done on filtering operators. Commutative and associative rules are used for query re-writing with highest priority to equal and 'AND' operator. Equal operator has the highest priority. The query rewriting algorithm is given below in Algorithm 3.1.

Query optimization logic is given in Algorithm 3.1:

Algorithm 3.1: Query Optimisation algorithm

```
1)   query re-writing algorithm
2)   Input : originalQuery
3)   Output : rewrittenQuery
4)    extractedQuery = Pattern.matcher(originalQuery)
5)    operatorList = comparingTokens = extractedQuery..createQueryTree();
6)    newQueryList = new ArrayList<String>(0);
7)   case QueryType of
8)    filterQuery:
9)     For each int i in tokens step by 3
10)      For each int j start from 3 sterp by 3
11)       if token match with comparingToken then
12)        if tokens.next === comparingTokens.next and contains "<" then
13)         if tokens.next < comparingToken.next then
14)          value = comparingTokens.next;
15)          operator=comparingTokens.current;
16)         else
17)          value = tokens.next;
18)          operator=tokens.current;
19)         end if
20)        else if tokens.next && comparingTokens.next and contains(">") then
21)         if tokens.next > comparingTokens.next then
22)          value = comparingTokens.next;
23)          operator=comparingTokens.current;
```

```
24)              else
25)               value = tokens.next;
26)               operator=tokens.current;
27)              end if
28)             end if
29)            end if
30)            newQueryList.add(tokens.previous);
31)            newQueryList.add(operator);
32)            newQueryList.add(value);
33)            needToAdd = false;
34)            tokens.previous = "";
35)            tokens.current= "";
36)            tokens.next = "";
37)          end for
38)          if needToAdd then
39)           newQueryList.add(tokens.previous);
40)           newQueryList.add(tokens.current);
41)           newQueryList.add(tokens.next);
42)          end if
43)        end for
44)       WindowQuery :
45)        For each int i in tokens step by 3
46)          For each int j start from 3 sterp by 3
47)            if token match with comparingToken then
48)             if tokens.next === comparingTokens.next and contains "<" then
49)              if tokens.next < comparingToken.next then
50)               value = comparingTokens.next;
51)               operator=comparingTokens.current;
52)              else
53)               value = tokens.next;
54)               operator=tokens.current;
55)              end if
56)             else if tokens.next && comparingTokens.next and contains(">") then
57)              if tokens.next > comparingTokens.next then
```

```
58)                value = comparingTokens.next;
59)                operator=comparingTokens.current;
60)               else
61)                value = tokens.next;
62)                operator=tokens.current;
63)               end if
64)              end if
65)             end if
66)             newQueryList.add(tokens.previous);
67)             newQueryList.add(operator);
68)             newQueryList.add(value);
69)             needToAdd = false;
70)             tokens.previous = "";
71)             tokens.current= "";
72)             tokens.next = "";
73)                   end for
74)             newQueryList.add(tokens[i]);
75)             newQueryList.add(operator);
76)             newQueryList.add(value);
77)             if tokens.length> i+3 then
78)              newQueryList.add(tokens[i+3]);
79)             end if
80)             needToAdd = false;
81)             tokens[j] = "";
82)             tokens[j+1] = "";
83)             tokens[j+2] = "";
84)          end for
85)          if needToAdd then
86)           newQueryList.add(tokens.previous);
87)           newQueryList.add(tokens.current);
88)           newQueryList.add(tokens.next);
89)          end if
90)        end for
91)     EndCase
```

```
92)     if newQueryList.size()%4 == 0 then
93)             newQueryList.remove(newQueryList.size()-1);
94)     end if
95)     rewrittenQuery = StringUtils.join(newQueryList, " ");
```

In the query rewriting algorithm, there are four main steps. Those are; extraction, syntax parsing, rearranging and re-construction of the query. As the first step, extract the executing section from the query (line 4). Then the syntax tree is created by parsing the extracted query (line 5). Each node of the syntax tree contains 3 nodes with attribute, operator and value. Then, the query is rewritten based on the operators. Filtering parts have the highest priority. AND operators are given the second priority. Commutative and associative rules are followed when rewriting the query. Based on these rules, rearranging is done (line 9-42). Finally, the query is reconstructed as given in line 95. Based on the query type, the algorithm will change. Filter queries are rewritten as given above. For window queries, extraction of the query is different from the filter query. A segment is extracted from the query and the rewriting is handled separately (line 45-90).

For window queries, two sub-trees are created separately and the query is reconstructed. The query rewriting algorithm is given above in Algorithm 3.1. Query re-writing logic and examples are given below in Figure 3.3 and Figure 3.4. A simple query has 3 parts in each segment; two parameters and a value.
For example : In Volume > 5, Volume and 5 are parameters, '>' sign is the operator

This creates Volume and 5 as child nodes and '>' sign as the parent node in the operator tree. All the available conditions are drawn in the node according to this convention. The parent node is always an operator. Figure 3.3 below shows the structure of an operator graph.

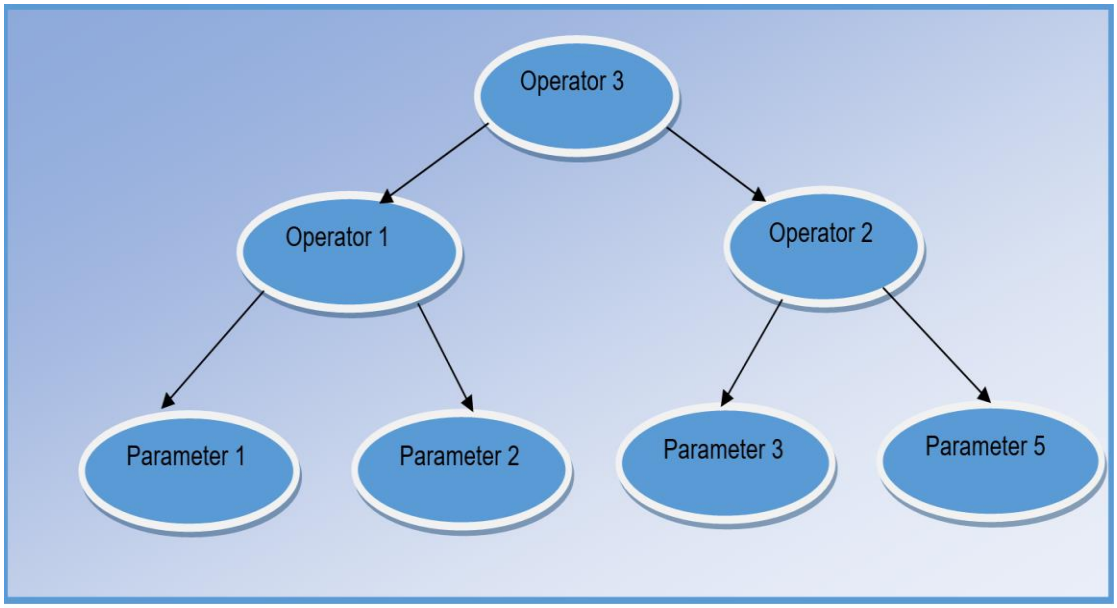

Figure 3.3 - Query re-writing logic

Sample query for query re-writing is given below:

"From stockPrice[Open > 2 and Close > 3 and LastTrade == 2] select Symbol, LastTrade insert into stockPriceOut";

In this query, the equal operator has the least priority. For an event, this query validates all 3 conditions if the first and second condition match. Since the first two operators are open-ended, most events can go through them, however, events are filtered out from the last conditions. But if the filtering part has the first priority, most of the events will skip the execution of open-ended operators and it will help to increase the throughput.

Query after extraction: Open > 2 and Close > 3 and LastTrade == 2
Extracted query is shown in the operator graph Figure 3.4 below:

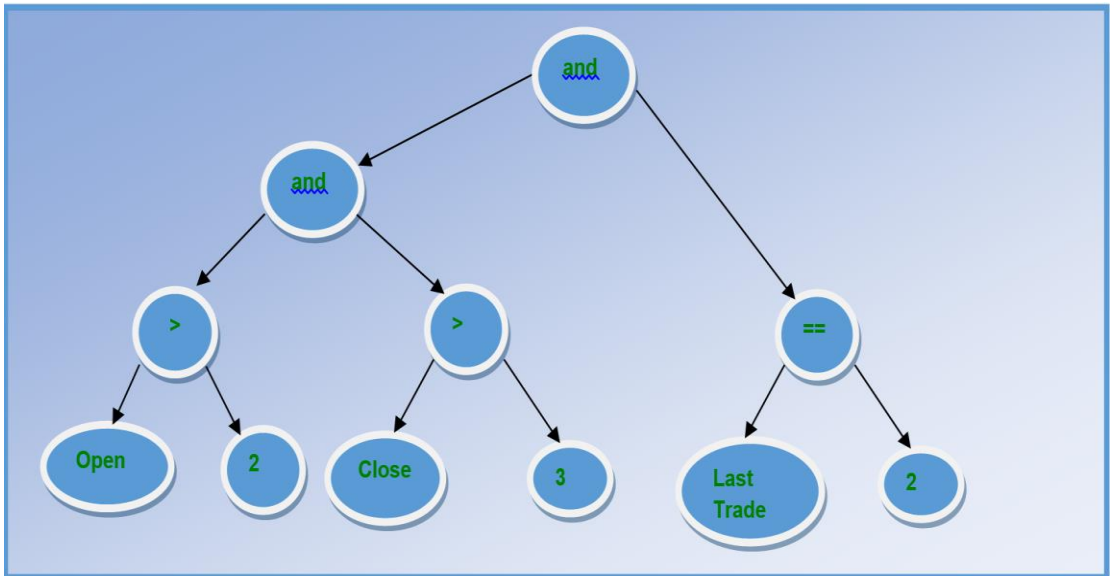

Figure 3.4 - Sample query re-writing logic step 1

Query after rearranging the operator tree is shown below in Figure 3.5.

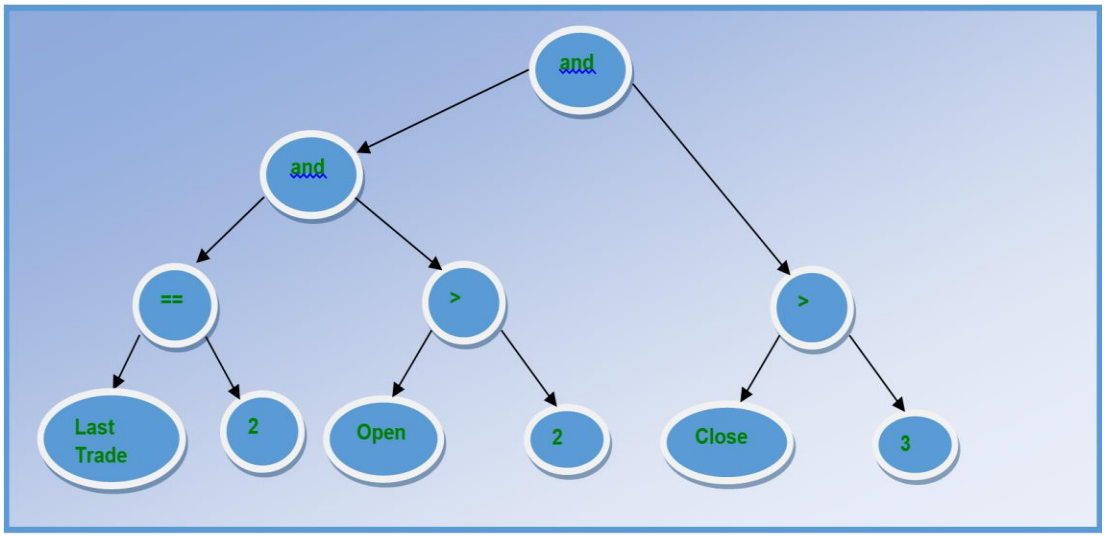

Figure 3.5 - Sample query re-writing logic step 2

The re-written query is taken by doing in-order traversal of the operator graph.

"From stockPrice[LastTrade == 2 and Open > 2 and Close > 3] select Symbol, LastTrade insert into stockPriceOut"

Duplicate query detection can be categorized into two sections; within query duplicate detection and between queries duplicate detection. If there are duplicates in queries, events will go through the similar operators repeatedly, which will reduce the overall throughput. Instead, if there's a single operator for each matching attribute, response generation will be fast and system throughput increases. Duplicate query detection is performed on top of the re-

written queries. Within query duplicate detection is tightly coupled with the query re-writing algorithm. After creating the syntax tree for query re-writing, if there's any duplicates, it's removed from the query. An example is given below:

Sample query for query re-writing is given below. Here Open > 2 and Open > 3 are duplicates which can reduce to Open > 2.

"From stockPrice[Open > 2 and Open > 3 and LastTrade == 2] select Symbol, LastTrade insert into stockPriceOut";

Query after extraction: Open > 2 and Open > 3 and LastTrade == 2

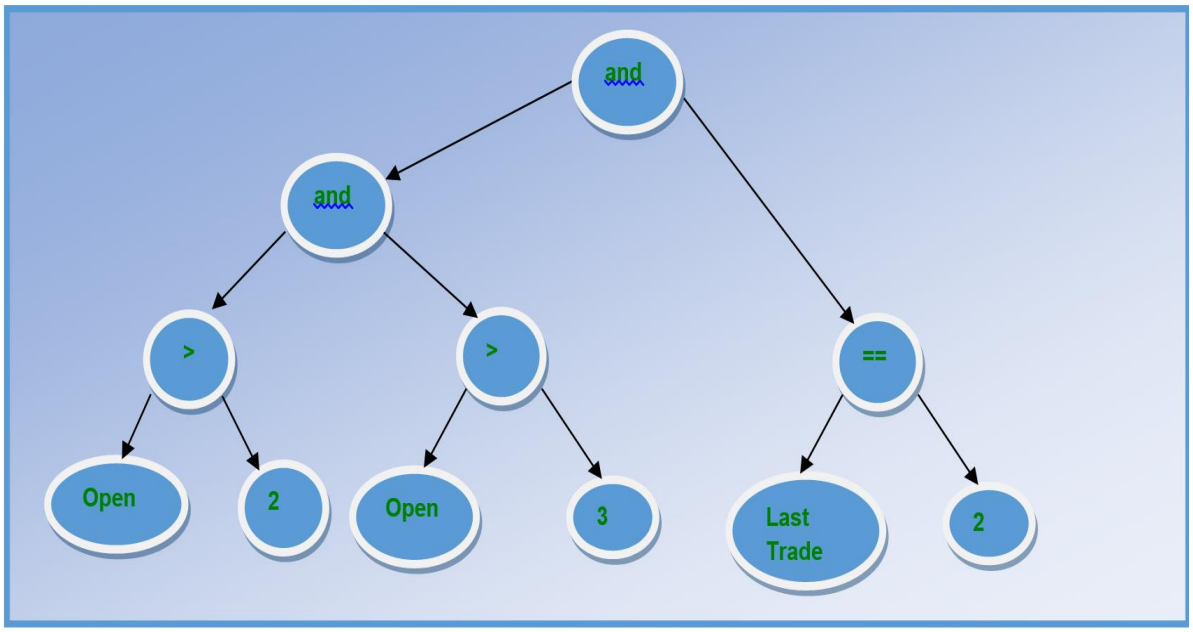

Figure 3.6 - Extracted query from original query

Query after rearranging

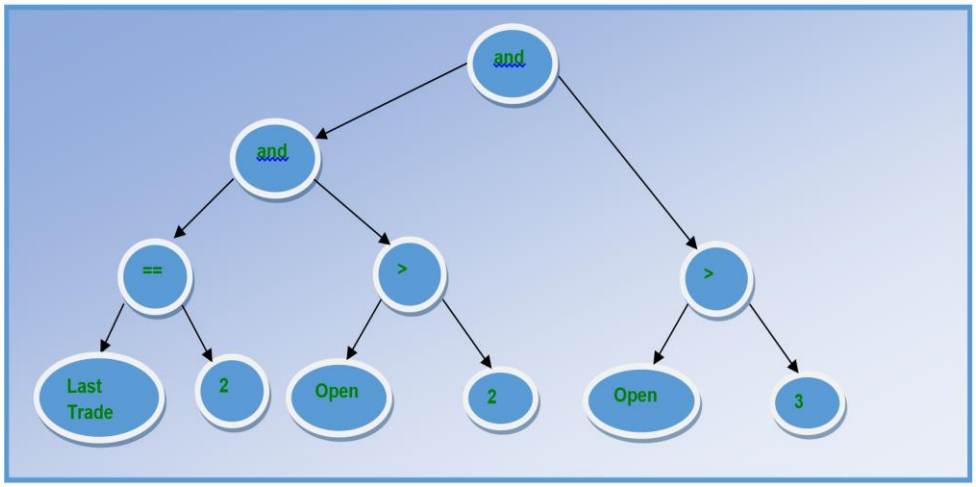

Figure 3.7 - Query after re-arrange

After duplicate removing

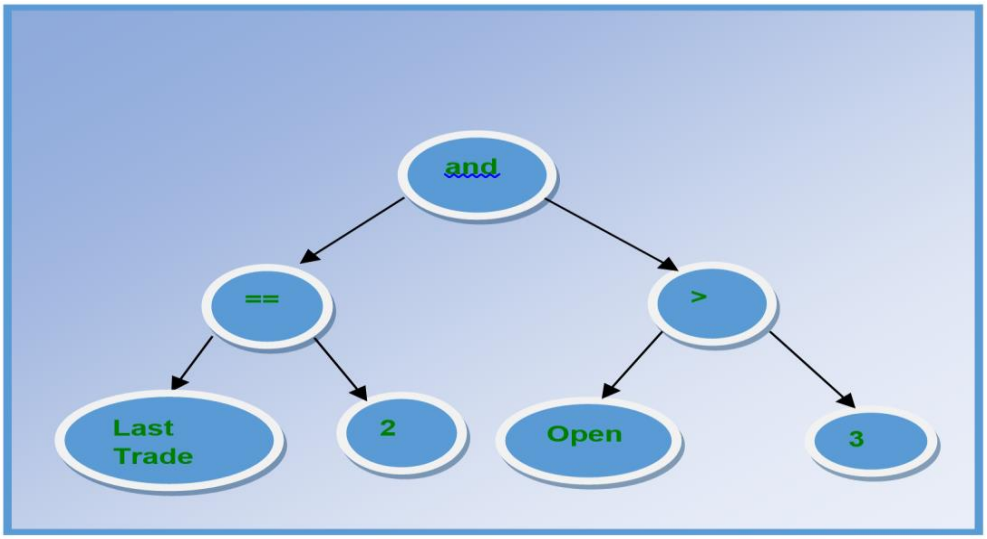

Figure 3.8 - Duplication removed query

Duplicate removed and re-written Query

"From stockPrice[LastTrade == 2 and Open > 3] select Symbol, LastTrade insert into stockPriceOut";

Between queries duplicate detection is executed after executing the query rewriting algorithm. The re-written query will be compared against the already re-written queries and duplicate queries will be removed. When detecting duplicated queries, the first step is

validating inputs and outputs. If inputs and outputs don't match, duplicate query detection continues to the next query. If the inputs and outputs match, then the execution segment of the query will be evaluated. Inheriting operators such as >, <, >= & <= will be evaluated and duplicate queries are detected.

3.3 Query Distribution Algorithm Optimization

Query distribution algorithm in VISIRI is one of the most efficient algorithms available since it calculates the query cost. But still, there are some areas where further optimization of the algorithm is possible. In practice, the processing power of a node and current memory consumption matters for performance. Dedicated server allocation is not always possible. Therefore, multiple applications can be executed in a particular node at a time. Therefore, considering processing power and memory consumption of a node is important.

The event type [30] is another important factor that wasn't considered in the VISIRI query distribution algorithm. Query type is identified based on the complexity (no. of conditions) of the query. A query type is assigned to each query in the system. Therefore, a processing node contains similar types of queries. The rationale behind selecting the query type is to distribute the queries to minimize the number of nodes required for event processing. Ultimately, this reduces the latency of distributing queries. In the example given below, No. of conditions =2, then Query type = 2
Ex: from stock[Volume >= 10000 and Bid < 2] select Symbol,Volume,Ask insert into volumes

Therefore, processing power and memory utilization of a node and event type, increases the performance of the query distribution algorithm. Algorithm 3.2 given below shows the steps of the modified query distribution algorithm. Before starting the query distribution, first we calculate the total cost of each node. Then, the query allocation is started.

Algorithm 3.2: Query distribution algorithm

1. **Require**: Query q, Node[] nodes, nodeAttributeMap
2. **Output**: targetNode
3. candidates = nodes;

4. //find minimum total cost
5. minCost = infinity
6. **for** node in candidates do
7. cost = sum(node.queries[0].cost,node.queries[1].cost, ...)
8. node.cost = cost
9. **if** minCost >cost **then**
10. minCost = cost
11. **end if**
12. **end for**
13. //filter nodes with too much cost
14. **for** node in candidates do
15. **if** node.cost >minCost + CostVariability **then**
16. candidates.remove(node)
17. **end if**
18. **end for**
19. //find maximum common event types
20. qInputs = q.inputStreams
21. maxCommonNodes =[]
22. maxCommonInputs = 0
23. **for** node in candidates do
24. node.allInputs=union(node.queries[0].inputStreams,node.queries[1].inputStreams.)
25. commons = count(intersect(qInputs,node.allInputs))
26. **if** maxCommonInputs == commons **then**
27. maxCommonNodes.add(node)
28. **else if** maxCommonInputs >commons **then**
29. maxCommonNodes.clear()
30. maxCommonNodes.add(node)
31. maxCommonInputs = commons
32. **end if**
33. **end for**
34. // remove nodes with high memory utilization
35. maxMemory = 0;
36. **for** (node in candidateNodes) do
37. freeMemory=nodeAttributeMap(node).getFreeMemoryPercentage();

38. **if** freeMemory > maxMemory **then**
39. maxMemory = freeMemory;
40. end if
41. end for
42. **for** (node in candidateNodes) **do**
43. freeMemory = nodeAttributeMap(node).getFreeMemoryPercentage();
44. **if** freeMemory < maxMemory - memoryUtilizationVariablility then
45. candidateNodes.remove(node)
46. end if
47. end for
48. // remove nodes with high cpu utilization
49. minCPU = infinity;
50. **for** (node in candidateNodes) **do**
51. cpuUtilization=nodeAttributeMap(node).getJvmCpuUtilization();
52. **if** cpuUtilization < minCPU **then**
53. minCPU = cpuUtilization;
54. end if
55. end for
56. **for** (node in candidateNodes) **do**
57. cpuUtilization=nodeAttributeMap(node).getJvmCpuUtilization();
58. **if** cpuUtilization > minCPU + CPUUtilizationVariablility then
59. candidateNodes.remove(node)
60. end if
61. end for
62. //find minimum queries
63. min-queries = min(nodes[0].queryCount,nodes[1].queryCount,...)
64. //filter nodes with too many queries
65. **for** node in candidates **do**
66. **if** node.queryCount >minQueries + QueryVariability **then**
67. candidates.remove(node)
68. end if
69. end for
70. // check query type
71. newQueryType = newQuery.getQueryType();

72. **for** (node in candidateNodes) **do**
73. queryList = nodeQueryTable.get(candidateNode);
74. count = 0;
75. **for** (query in queryList) **do**
76. **if** query.queryType === newQueryType **then**
77. count ++;
78. **end if**
79. end for
80. similaryMap.put(candidateNode, count)
81. end for
82. similarityMap.sort();
83. **if** (!similarityMap.isEmpty()) **then**
84. entry = similarityMap.entrySet().iterator().next();
85. targetNode = entry.getKey();
86. else
87. targetNode = random.select(candidates)
88. **end if**

The query distribution algorithm given above operates as follows. When there's a query to distribute, the first query distribution algorithm calculates the total cost of each node line 5-12. It is calculated by summing up the cost of queries in each node. Then the node with the minimum total cost is selected. Nodes having costs greater than the threshold, which is minimum total cost plus cost variability, are removed. A set of nodes are selected as candidate nodes after the cost-based filtering.

As the second step in line 20-31, finding events in common is done. This is based on common input streams. Input for a node is the union of all input streams of deployed queries. Nodes with maximum number of common input streams are selected, which will reduce the events needed to be sent over the network as inputs are minimized. This is to reduce event duplication and bandwidth utilization.

In the third step, lines 35-61 filter out the nodes with high memory and CPU utilization. While the system is starting up, each node will publish its CPU and memory utilization to the

hazelcast cache. The main node collects these details and uses in query distribution. This will reduce the need of dynamic query distribution. In case of an event burst, over utilizing nodes can easily get overloaded, which will trigger the costly dynamic distribution. Therefore, the resource utilization filter helps to reduce system overload.

Fourth step is to find the nodes with minimum number of queries from the remaining nodes, in lines 63-69. Nodes having queries more than the threshold will be removed from the candidate nodes. This is again to reduce the system overload in case of an event burst.

Finally, as the last step in lines 71-79, query type will be evaluated in each node. The nodes with maximum number of common query types are selected. If there are multiple candidate nodes, a node is randomly selected to deploy the new query.

3.4 Standardized Event Source & Communication

In VISIRI, there's an event source that is in-built with the application. But it is not useful in practice since it's tightly coupled with the application. VISIRI is tightly coupled with Java programming language and therefore can't be used with any other technology. Decoupling event source and integrating standard event source will improve the usability of the application. Introducing a standard communication mechanism will help to reduce the integration overhead with any other technology. Therefore, standardizing event source and communication helps to increase the practical use of the application.

As the standard event source, Google events will be used. Google provides firebase (https://firebase.google.com/) real-time database and push notifications. The main restriction to accommodate Google firebase cloud messaging service is that the application should either be an Anroid/iOS mobile app or web application. Therefore, HTML based web component is integrated with the system, which provides the connectivity to Google notifications. Having a real-time database has an additional benefit. Data can be disseminated and the real-time output can be received, which is one advantage. The second advantage is the storing and analyzing capability. VISIRI will provide both real time analysis and storing analysis by integrating with the Google notification service.

Standardized communication is achieved by introducing standard messages to the system. JSON is the widely used message format in the IT industry. Google provides Gson library for JSON message parsing. Therefore, JSON messages are introduced to the system as the communication mechanism. By doing so, the VISIRI system can easily integrate with any other JSON message supporting components instead of supporting only Java based applications. Firebase database with stock data given below in Figure 3.9.

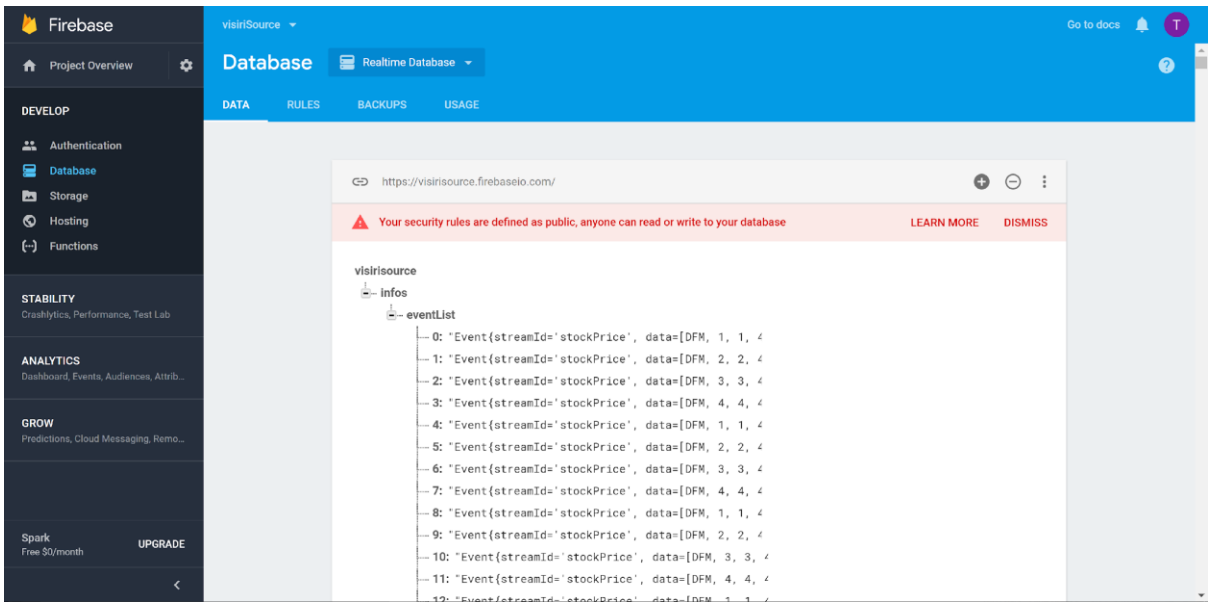

Figure 3.9 - Firebase real-time database with stored events

Security is a main concern in data-intensive applications. Data security is provided by the Google firebase itself. Only applications with apiKey, which is unique to the Google firebase server and the VISIRI application, can share the data. By providing the unique apiKey, Google preserves the data security. Firebase configurations of the project is given below in Figure 3.10:

```
firebaseConfig: {
  apiKey: "AIzaSyCvlzc62xNY6E-9f647QkInquVE9XrACdo",
  authDomain: "visirisource.firebaseapp.com",
  databaseURL: "https://visirisource.firebaseio.com",
  projectId: "visirisource",
  storageBucket: "",
  messagingSenderId: "994657826356"
}
```

Figure 3.10 - Firebase real-time database with stored events

4 EVALUATION

For the measurements, computers with different capacities are used. Machine configurations are given below:

- Intel Core i7 CPU @ 2.60GHz - 16 GB RAM
- Intel Core i7 CPU @ 1.80 GHz - 8 GB RAM
- Intel Core i3 CPU @ 2.0 GHz - 8 GB RAM
- Intel Core i3 CPU @ 2.4 GHz - 8 GB RAM

Common system configurations are given below:

- System type: 64-bit
- Operating System: Windows 8
- Wifi connection : 46.1 Mbps

Performance evaluation of the system is done between the VISIRI system and the STHITHIKA system. Firebase real-time database is pre-stored with actual stock market data taken from the Gulf region stock markets (mfg-uat-phoenix.mubashertrade.com). Event rate needs to be controlled in each test scenario when testing the VISIRI and STHITHIKA systems. It was achieved using threads and controlling the event sending rate from the event source. A random query generation algorithm was used to generate large query sets, which are used by VISIRI [7]. Query attributes are taken from the stock market event feed. Sample queries are given below:

- from stock[LastTrade <= 100 and LastTrade > 80] select Symbol,Date,Volume insert into stocks
- from stock#window.lengthBatch(10) select Symbol, max(Close) as close insert into stock

Evaluation of VISIRI and STHITHIKA was done in two aspects. Effectiveness of VISIRI initial query distribution algorithm vs the STHITHIKA query distribution algorithm was measured first and then the effectiveness of query re-writing was measured by enabling both improved static query distribution and query re-writing in the STHITHIKA system.

A. Comparison of VISIRI Vs STHITHIKA

Performance evaluation of the VISIRI system vs the STHITHIKA system was done in this scenario. Fixed event rate (75000 events/sec) and number of queries (1000 queries) are used in this analysis. The VISIRI system, STHIHIKA with query distribution algorithm

enhancements, and STHITHIKA with both query distribution algorithm enhancements and query re-writing, was tested individually in this scenario.

Figure 4.1 below shows the comparison of query distribution algorithms of VISIRI and STHITHIKA. According to the results, the query distribution algorithm of STHITHIKA gives better performance. Both resource utilization and event type provided better performance. Query re-writing has the highest impact on throughput compared to resource utilization and event type. In STHITHIKA, using resource utilization, event type and query rewriting, achieved better performance compared to the VISIRI system.

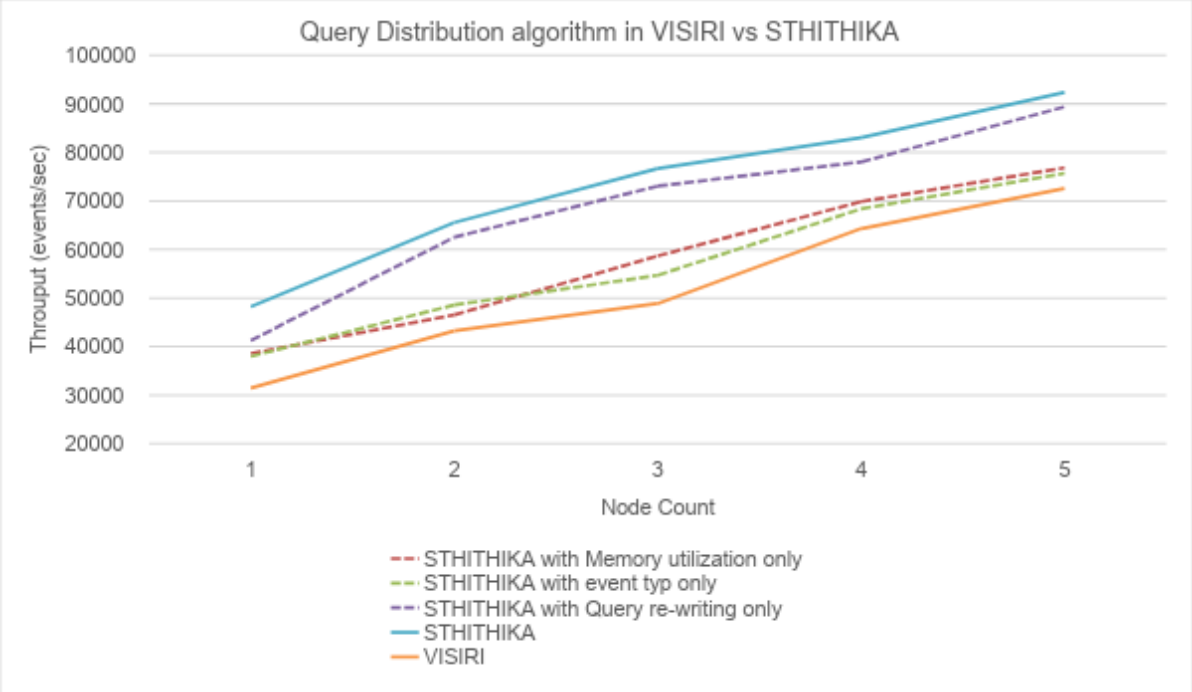

Figure 4.1 - Comparison of Query distribution VISIRI vs STHITHIKA

B. *Performance analysis for increasing query count*

The next evaluation scenario is to analyze the support for a large number of queries in both the VISIRI and the STHITHIKA system. In this scenario, event rate is kept constant (75000 events/sec) and query count increases. Throughput is analyzed in both the VISIRI and STHITHIKA systems. Since the STHITHIKA system is already performing better in query distribution according to Figure 4.1, only the VISIRI and STHITHIKA systems are compared in this scenario.

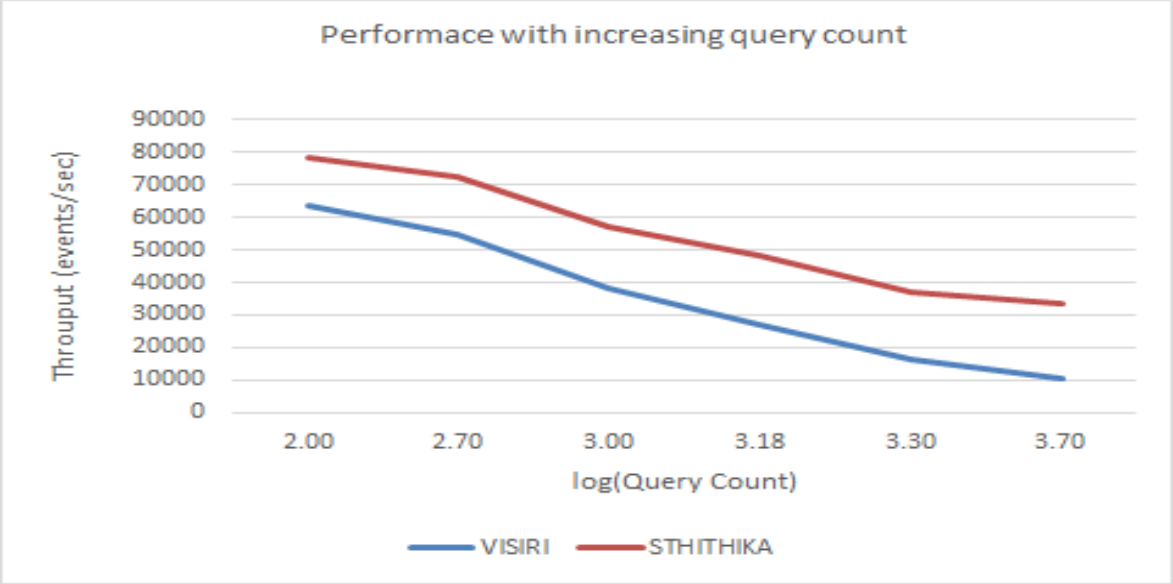

Figure 4.2 - Performance with increasing query count

As shown in Figure 4.2 above, the STHITHIKA system gives better performance compared to the VISIRI system, with increasing query count.

C. *Handling event bursts*

Next evaluation scenario is handling event bursts. This analysis is important because this will reduce the need of dynamic query distribution. In this scenario, query count is kept constant (1000) and the input event rate is increased. As shown in Figure 4.3 below, the STHITHIKA system gives better performance in event bursts compared to the VISIRI system.

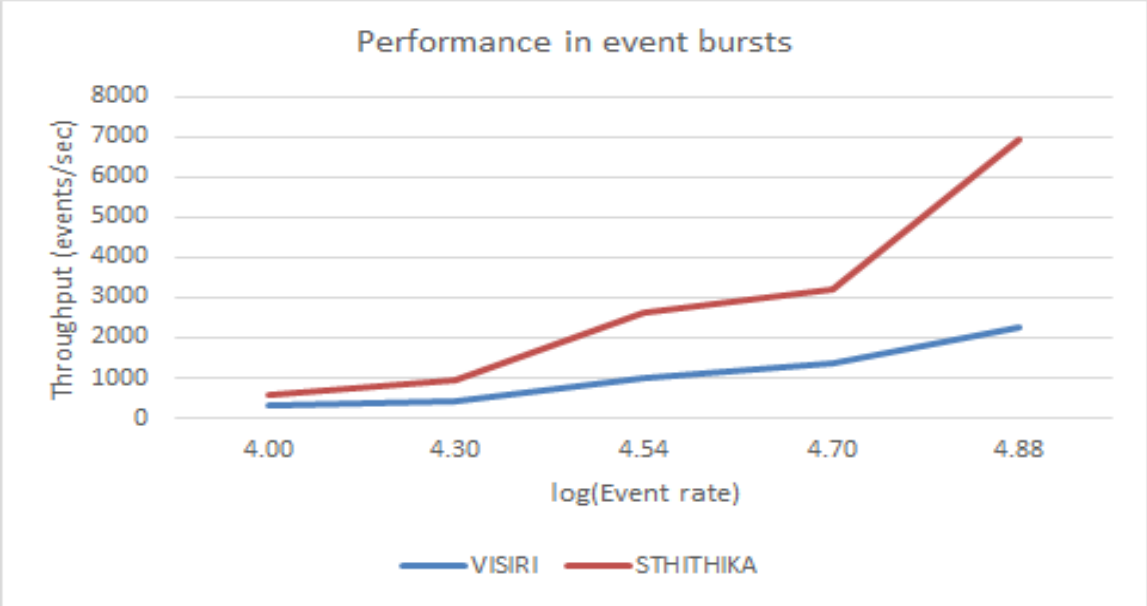

Figure 4.3 - Performance evaluation in event bursts

D. Analysis of time taken to query distribution.

In the STHITHIKA system, the query distribution algorithm has changed and query rewriting was introduced. Therefore, the time taken for initial query distribution can vary. This is the analysis done to identify the variations in time taken for query distribution due to the changes done to the STHITHIKA system. By increasing the query count, the time taken for the initial query distribution algorithm is measured. As shown in Figure 4.4 below, the time taken for query distribution has increased in the STHITHIKA system.

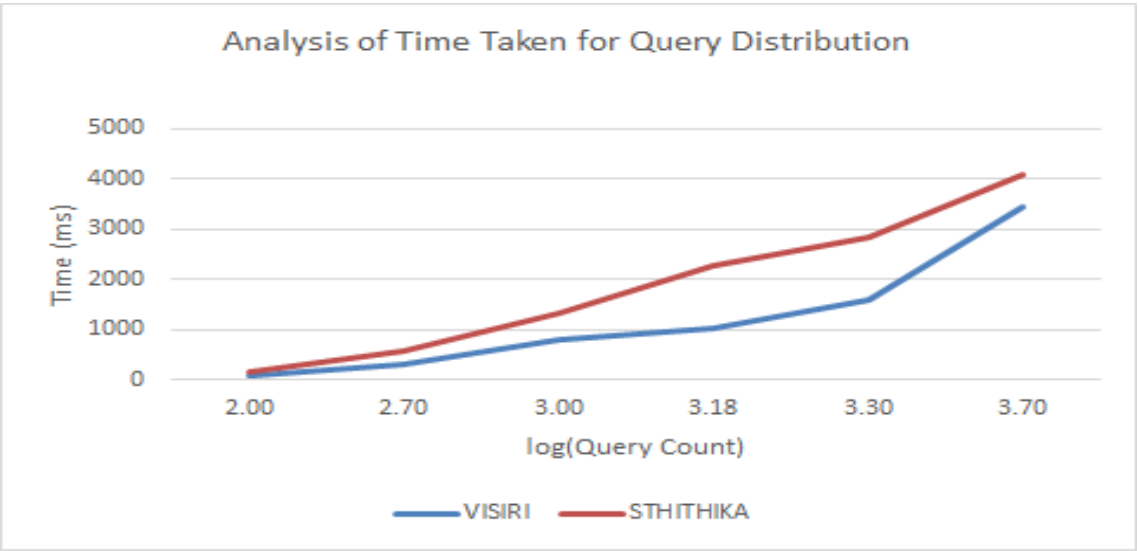

Figure 4.4 - Analysis of time taken for query distribution

E. Handling event duplication

Reduced multicast is a major enhancement in the VISIRI system over the other existing query distribution algorithm. The same scenario is tested with the STHITHIKA system to make sure that there won't be network overhead due to the changes done in the STHITHIKA system. Figure 4.5 below, shows how event duplication changes in the system for the VISIRI and STHITHIKA algorithms for 1000, 5000, and 10000 queries.

According to these observations, both VISIRI and STHITHIKA have the exact same results with respect to event duplication.

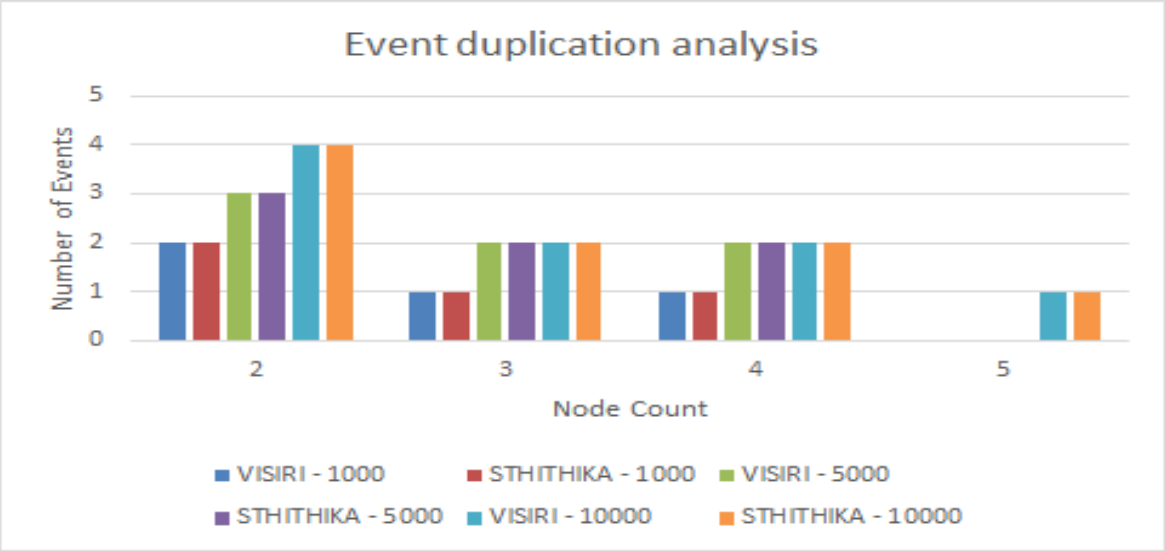

Figure 4.5 - Event duplication analysis

F. Cost Variance analysis

Another important analysis done in the VISIRI system is cost variance analysis. As the execution cost variance, VISIRI measured how much variance the processing nodes have when the queries are distributed with respect to the estimated cost values [7]. In a balanced system, nodes should have low cost variance. The cost threshold value (highest total cost of the queries deployed in a CEP node) for algorithms was kept at 400 while keeping the query count threshold (highest number of queries deployed in a CEP node) at 80 for both VISIRI and STHITHIKA algorithms. Figure 4.6 shows the analysis of the cost variance.

According to Figure 4.6, the cost variance of the STHIHIKA system is lesser than the VISIRI system. This is an indication that the STHITHIKA system has a balanced cost distribution, better than in the VISIRI system.

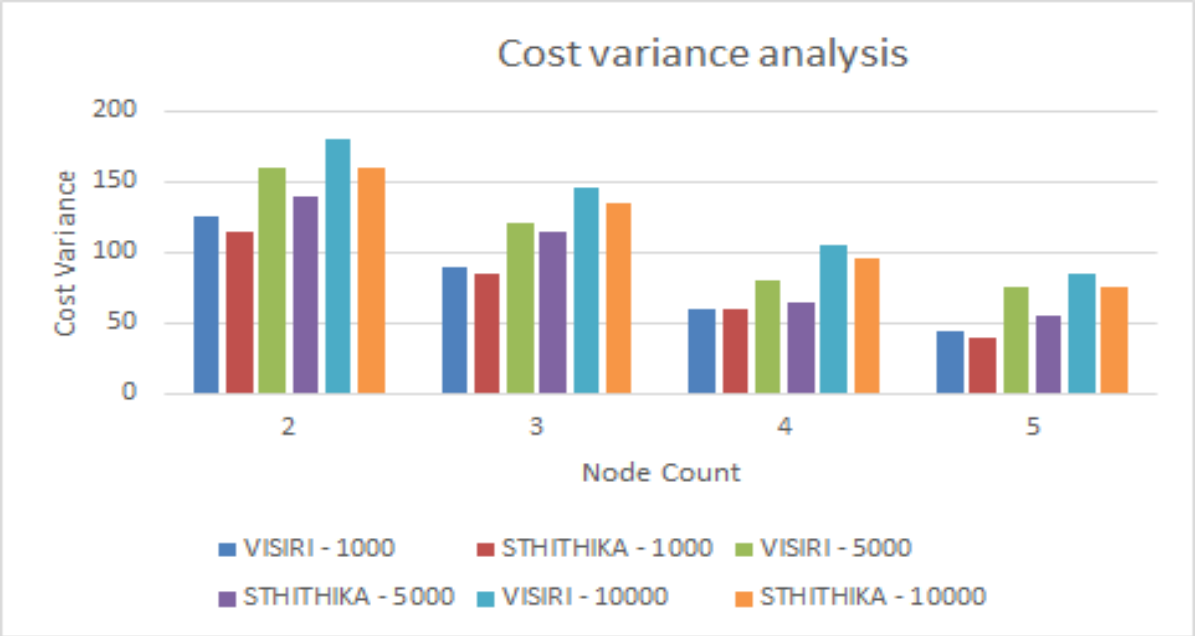

Figure 4.6 - Cost variance analysis

G. *Query rewriting comparison*

Another important factor is making sure that the output with query rewriting, and without query rewriting produce the same result. This is analyzed using a single node CEP system because query distribution on multiple nodes has no impact for this analysis. As shown in Figure 4.7 below, the same set of queries were deployed in two scenarios. The triggered query count and output were analyzed and the exact same results were produced in both scenarios.

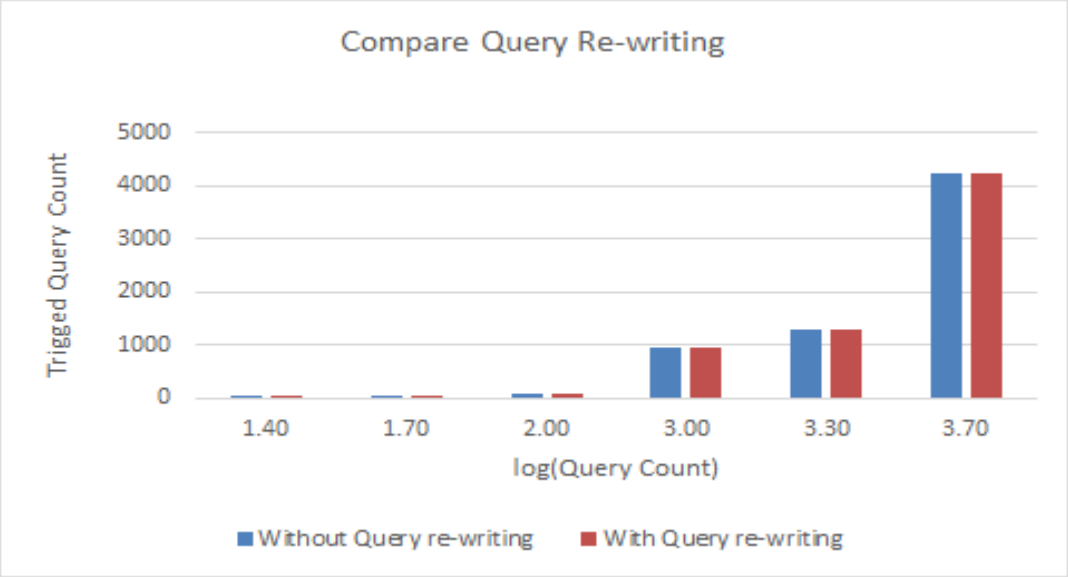

Figure 4.7 - Compare query re-writing

4.1 Result Analysis

The following observations were made after the above experiments:

A. *Query rewriting, together with resource utilization and the event type in query distribution, give better performance*

According to the experimental results, by introducing query rewriting and introducing resource utilization and event type to query distribution, better performance was achieved. Supporting a large number of queries and handling event bursts is possible with these enhancements. Therefore, the possibility for costly dynamic distribution decreases with these improvements.

B. *Time taken for query distribution can increase with query rewriting and query distribution enhancements*

The drawback of adding more features to query distribution and introducing query rewriting is that the time taken for initial query distribution increases when the algorithm becomes more complicated. This will increase the time required for initial system start up. Since its one-time investment, compromising initial start-up time won't be an issue.

C. *Handling event duplication*

Event duplication is a key factor considered in the VISIRI system. The lesser the event duplication, the lesser the multicasts. It helps to reduce the bandwidth utilization. According to the observations, STHITHIKA performs exactly the same as VISIRI when handling event duplications. Therefore, the STHITHIKA system is effective in bandwidth utilization.

D. *Cost Variance analysis*

The system should have a balanced cost distribution in order to perform better, since query cost is an indication of the complexity of the query. Balanced cost distribution reduces the possibility of node overloading. When comparing VISIRI and STHITHIKA, STHITHIKA has better cost distribution with less cost variance compared to the VISIRI system. This is due to giving priority to cost variance over query count and using CPU utilization and free memory percentage for query distribution.

It's clear that the STHITHIKA system performs better compared to the VISIRI system according to the above observations.

5 FUTURE WORK

Even though the event source is independent from the system, the events used are not actual real-time data. As future work, integrating the system with real event source can be done. An actual event source should connect to the firebase cloud messaging service in order to support the cloud messaging service. Adding a firebase supported HTML component inside VISIRI is the next enhancement. This HTML component acts as a separate component now. But integrating this as a part of the VISIRI system will reduce the overhead of system initialization.

Supporting heterogeneous event processing engines is another system enhancement that needs to be done. The existing system is tightly coupled with the Siddhi CEP engine. By providing multiple CEP engines support ex: Esper, it will increase the usability of the system since Siddhi focuses on a large number of query support, whereas support for complex queries is high in the Esper system. Different CEP engines have slightly different query

languages. Therefore, the query language needs to be uniform while supporting multiple CEP engines. Uniform meta language should be introduced, which will be converted to the respective CEP engine query language, since writing queries with different syntaxes is not helpful for the user.

6 CONCLUSION

In a distributed CEP, queries used to identify useful patterns in event streams are distributed across the processing nodes. It is a Non-deterministic Polynomial-time (NP) hard problem because the number of parameters that affect query distribution is much higher. There is no specific optimal way to distribute queries efficiently across processing nodes.

In query distribution, the proven solution is cost-based query distribution, which is used by the VISIRI system. STHITHIKA is an improvement to the cost-based distributed CEP systems. By optimizing the query distribution algorithm and reducing the execution time of queries by introducing query rewriting, the efficiency of the system was increased. Technology based restriction removing and usability enhancements were done by removing the tight coupling with Java language and the inbuilt event source. Usability has improved by introducing JSON messages for communication and Google firebase messaging service integration as the event source. With these changes, the overall efficiency and usability has improved in cost-based distributed CEP systems.

With the mentioned changes, better performance is achieved in the STHITHIKA system over the cost-based query distributed CEP systems. The STHITHIKA system has better throughput and performance in event bursts and performance in large number of queries is higher in the STHITHIKA system. Event duplication, which causes higher bandwidth utilization, remains the same while maintaining a lesser cost variance across the CEP nodes.

The other aspect of improving the system is improving the usability. Easy integration with event sources is required in order to increase the usability of the application. As an enhancement, a standard communication mechanism was developed using JSON messages for communication. By doing so, the STHITHIKA system has the ability to be used with any other technology that supports standard JSON message protocol. The system is integrated with the Google firebase messaging service in order to standardize the event source. In-built

event sources restricts system integration with third-party event providers. Therefore, standard Google messaging was introduced to the system.

With the above improvements, performance improvements and usability improvements were achieved in the STHITHIKA system. Further improvements can be done to this system, such as supporting heterogeneous event processing engines. By providing multiple CEP engine support, further improvements in usability can be achieved.

REFERENCES

[1] O.Etzion and P.Niblett. *Event processing in action*. Manning Publications Co., 2010.

[2] S. Perera, "How to scale Complex Event Processing (CEP) Systems", *Srinath'sBlog :My views of the World*, 2012. [Online]. Available: http://srinathsview.blogspot.com/2012/05/how-to-scale-complex-event-processing.html. [Accessed: 01- March- 2016]

[3] K.Isoyama, K.Yuji, S.Tadashi, K.Koji, Y.Makiko, and T.Hiroki "A scalable complex event processing system and evaluations of its performance." *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*.ACM, 2012.

[4] Y.Zhou, A.Karl, and Kian-Lee Tan. "Toward massive query optimization in large-scale distributed stream systems."*Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., 2008.

[5] S. Wu, V. Kumar, K. Wu and B. Chin Ooi, "Parallelizing stateful operators in a distributed stream processing system: how, should you and how much?", in *DEBS '12 Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, New York, NY, USA, 2012, pp. 278-289.

[6] B. Schilling, B. Koldehofe, U. Pletat and K. Rothermel, "Distributed heterogeneous event processing: enhancing scalability and interoperability of CEP in an industrial context", in *Proceeding DEBS '10 Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, New York, NY, USA, 2012, pp. 150-159.

[7] K.Kumarasinghe, G.Tharanga, L.Weerasinghe, U.Wickramarathna, and S.Ranathunga. (2015). VISIRI - Distributed Complex Event Processing System for Handling Large Number of Queries. In Tom Holvoet, MirkoViroli (Eds.), Coordination Models and Languages, Volume 9037 of Lecture Notes in Computer Science, 230-245. Springer

[8] G.Cugola and A.Margara. Processing flows of information: From data stream to complex event processing. ACM Computing Surveys (CSUR). 2012 Jun 1;44(3):15.

[9] G.Cugola, and A.Margara. "Deployment strategies for distributed complex event processing."Computing 95.2 (2013): 129-156.

[10] S. Suhothayan, K. Gajasinghe, I. LokuNarangoda, S. Chaturanga, S. Perera and V. Nanayakkara, "Siddhi: a second look at complex event processing architectures", in *GCE '11 Proceedings of the 2011 ACM workshop on Gateway computing environments*, New York, NY, USA, 2011, pp. 43-50.

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

[12] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte and W. White, "Cayuga: a high-performance event processing engine", in *SIGMOD '07 Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2007, pp. 1100-1102.

[13] L. Neumeyer, B. Robbins, A. Nair and A. Kesari, "S4: Distributed Stream Computing Platform", in *ICDMW '10 Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, IEEE Computer Society Washington, DC, USA, 2010, pp. 170-177.

[14] T.P.K. Dahanayakage, et al. "Wihidum - Distributed Complex Event Processing", Final Year Project-2013 , Department of Computer Science and Engineering, University of Moratuwa. December 5, 2013.

[15] M. RN Mendes, P.Bizarro, and P.Marques. A performance study of event processing systems. In Performance Evaluation and Benchmarking, pp. 221-236. Springer, 2009.

[16] N. Schultz-Moller, M. Migliavacca and P. Pietzuch, "Distributed complex event processing with query rewriting", in *DEBS '09 Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, ACM New York, NY, USA, 2009.

[17] L. Dysert, "Developing a Parametric Model for Estimating Process Control Costs", *http://www.costengineering.eu/*, 2001. [Online]. Available: http://www.costengineering.eu/images/papers/Developing_a_Parametric_Model_for_Estimating_Process_Control_Costs.pdf. [Accessed: 01- Mar- 2016].

[18] M. Liu, E. Rundensteiner, D. Dougherty, C. Gupta, S. Wang, I. Ari and A. Mehta, "High-performance nested CEP query processing over event streams", in *Data Engineering (ICDE), 2011 IEEE 27th International Conference*, 2011, pp. 123 - 134.

[19] W. A. Higashino, C. Eichler, M. A. M. Capretz, T. Monteil, M. B. F. D. Toledo and P. Stolf, "Query Analyzer and Manager for Complex Event Processing as a Service," *WETICE Conference (WETICE), 2014 IEEE 23rd International*, Parma, 2014, pp. 107-109. doi: 10.1109/WETICE.2014.53

[20] Y. Ahmad, B. Berg, U. Cetintemel, M. Humphrey, J. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing and S. Zdonik, "Distributed operation in the Borealis stream processing engine", in *SIGMOD '05 Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ACM New York, NY, USA, 2005, pp. 882-884.

[21] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul and S. Zdonik, "Aurora: a new model and architecture for data stream management", in *The VLDB Journal — The International Journal on Very Large Data Bases archive Volume 12 Issue 2, August 2003*, Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2003, pp. 120-139.

[22] S. Zdonik, M. Stonebraker, M. Cherniack, U. C¸ etintemel, M. Balazinska, and H. Balakrishnan. ”The Aurora and Medusa Projects”. IEEE Data Engineering Bulletin, 26(1), March 2003.

[23] Y.Xing,S.Zdonik, and J.Hwang. "Dynamic load distribution in the borealis stream processor."*Data Engineering, 2005.ICDE 2005.Proceedings.21st International Conference on*.IEEE, 2005.

[24] N. Schultz-Moller, *Distributed Detection of Event Patterns*, Advanced Computing of Imperial College London, 2008.

[25] S. Ravindra and M. Dayarathna, "[Article] Distributed Scaling of WSO2 Complex Event Processor | WSO2 Inc", *Wso2.com*, 2016. [Online]. Available: http://wso2.com/library/articles/2015/12/article-distributed-scaling-of-wso2-complex-event-processor/. [Accessed: 01- Mar- 2016].

[26] "Complex Event Processor | WSO2 Inc", *Wso2.com*, 2016. [Online]. Available: http://wso2.com/products/complex-event-processor/. [Accessed: 01- Mar- 2016].

[27] Y. Mei and S. Madden, *ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events*, 1st ed. Providence, Rhode Island, USA: ACM 978-1-60558-551, 2009.

[28] K. Cao, R. LI, and F. *Wang*. "The Research on CEP Based on Query Rewriting for CPS.” *International Journal of Modeling and Optimization. pp.Vol. 3, No. 6*, 2013.

[29] J. Calbimonte, J. Mora, and O. Corcho. "Query Rewriting in RDF Stream Processing.”

[30] Hettige Randika, Surangika Ranathunga, GATHIKA: A Dynamic Query Distribution Mechanism For Complex Event Processing Systems, Proceedings of International Conference on Advances in ICT for Emerging Regions, Colombo, LK, 2018