GATHIKA
DYNAMIC QUERY DISTRIBUTION MECHANISM FOR COMPLEX EVENT PROCESSING SYSTEMS

Hettige Chathura Randika

(158243G)

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

Department of Computer Science & Engineering

University of Moratuwa
Sri Lanka

May 2018

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: H. Chathura Randika

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 heavily used in real time systems where people are interested in extracting valuable information from event streams. Scalability and fault tolerance are major requirements for such systems that do complex event processing. It is very hard to rely on a single machine to do the processing of all events. Therefore, requiring distributed systems for processing event streams is an obvious choice. Such a system should be able to cater to the requirement of processing a large number of events. Event queries are deployed in event processing nodes to extract useful information from event streams. In real time, event processing nodes get overloaded due to event bursts. In addition, there are situations where a large set of queries need to be deployed to extract useful information from the events. Due to all these conditions, the overall throughput of the whole system degrades. Distribution of queries is therefore essential in a complex event processing system.

Distributing complex queries statically within the event processing nodes (at system initialization) is not a trivial task. Dynamic query distribution (during system operation time) is even harder due to factors such as fault tolerance, availability, scalability, predictable performance, and security requirements of the distributed CEP system. Network connectivity and the status of the processing nodes are some of the essential factors that need to be considered when doing query distribution.

This research focuses on developing dynamic query distribution mechanisms for a distributed complex event processing system. A dynamic query distribution algorithm capable of deploying the queries dynamically across the nodes of the distributed CEP system is designed. Query distribution is done considering the resource utilization levels of the event processing nodes, the complexity of the query to be deployed, and the type of queries deployed in the processing nodes.

Through our experiments, it was evident that the performance of the system is proportional to the number of processing nodes in the system. When dynamic query distribution is properly executed, the overall system performance can be improved by balancing the load among the processing nodes. Two important rules were defined to guarantee this proper execution: minimum time between two successive dynamic query distributions and minimum number of queries to trigger dynamic query distribution in the system. Having low latency when distributing queries dynamically and high throughput after dynamic query distribution are the key success of this dynamic query distribution mechanism. Therefore, it is beneficial to have a dynamic query distribution mechanism in CEP systems that experience frequent event bursts and query/node deployments.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Dr. Surangika Ranathunga, my research supervisor, for the continuous support given for the success of the research. Further, advice given by Prof. Gihan Dias and Dr. Malaka Walpola was a great help to the success of this research.

I would also like to thank all the staff at the Department of Computer Science and Engineering, University of Moratuwa for their kindness expressed in all occasions. Mainly I would like to thank Mr. Sujith Fernando and Mr. J.C. Rajapakse for helping me to configure the computers at the laboratories in order to conduct my experiments.

I wish to express my gratitude to my team lead at Mubasher Technologies (Pvt) Ltd, for the support given to me to manage my MSc research work.

I wish to thank my family for their support and encouragement throughout my research.
Finally, I would also like to thank all my friends who encouraged me to complete the research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ADX – Abu Dhabi Securities Exchange

ATM - Asynchronous Transfer Method

BSE – Bahrain Stock Exchange

CEP - Complex Event Processing

CPU - Central Processing Unit

DFM – Dubai Financial Market

DQD – Dynamic Query Distribution

EGX – Egyptian Exchange

EP - Event Processing

KSE – Kuwait Stock Exchange

MSM – Muscat Securities Market

POC – Proof of Concept

QP - Query Processor

QT – Query Type

RTL - Register Transfer Level

S4 - Simple Scalable Streaming System

SQL - Structured Query Language

TDWL – Saudi Stock Market

XML - Extensible Markup Language

1 INTRODUCTION

Main focus of this GATHIKA project is to come up with an efficient dynamic query distribution mechanism for distributed Complex Event Processing (CEP) systems. This chapter gives an introduction to the background of the problem, the research problem, and the motivation to do this research.

1.1 Background

CEP is an emerging field in data processing. CEP is the process of analyzing the stream of events and identifying useful patterns within them. The events are not related to a single type but are a combination of multiple types, hence the name complex events [1]. The idea behind CEP is to analyze the streams of events against a set of queries defined for the given context and to identify complex events. The queries can be predefined or provided at runtime. Typically, event driven systems operate with stored queries constantly running against the dynamic data input. This operation can be considered an upside down version of a search engine where stored data are matched against incoming queries [1]. The queries used for CEP systems are similar to normal SQL queries and depend on the environment in which the CEP system operates.

Usually, event processing systems require high processing power, I/O, and memory to operate, and require supercomputers to work efficiently. However, using a mainframe or supercomputer is costly. Another important factor is the event generation rate. With the development of technology and electronic devices (sensors and CCTV cameras), the number of events generated by the sources increases rapidly. Therefore, a single CEP node cannot handle the heavy load of events and event sources that are geographically distributed. This leads to the use of distributed CEP systems. High reliability, fault tolerance, confidentiality of low-level data and avoiding a single point of failure are major motivations to use a distributed CEP system.

1.2 Problem Description

Operator distribution and query distribution [2, 3] are the two types of available CEP distribution mechanisms. In this project, the focus is on query distribution. The deployment of CEP queries in a distributed CEP system is a complex task. There should be an efficient way of distributing the load of events and queries across the processing nodes. There are many challenges in developing distributed systems for complex event processing. The main problem is the requirement of handling a large number of complex events and queries in real time [2]. Other than that, achieving fault tolerance, availability, scalability, predictable performance and security in the system is also a challenge [1].

At runtime, if a particular node has failed or joined the cluster of nodes, there should be an efficient mechanism to detect and adjust the system behavior. Handling event bursts is also important in present-day CEP systems. The number of events generated by the event sources can dynamically increase, thus overloading some nodes in the distributed CEP system.

In this research, we consider the scenario of a large number of queries being deployed in the CEP system. A primary assumption we make here is that a query is deployed in only one node (no query duplication).

Such scenarios are common in stock trading platforms. Today, large institutional investors in stock trading are interested and working on implementing automated trading functionality in their trading platforms. Other than that, they provide real-time alerts to their customers about the changes of the stocks. These trading applications are getting data from stock exchanges all around the world. Hence, these trading applications receive millions of events per second. Therefore, the requirement is to process large volumes of events and extract events that are important to the business. To extract useful events a large set of queries are also deployed in this type of systems.

The requirement of changing the rules is necessary for an aforementioned trading application to execute trading instructions to obtain stocks in the best possible price. In the case of generating trading alerts to the customers, dynamically changing of rules during the market time provides great agility to such trading systems. Low latency and high throughput are most required features.

Dynamic distribution of queries for CEP is an NP-hard problem [4, 5, 6]. Parameters such as the current system load, the current load of a given node, the availability of the nodes, among others, need to be considered when distributing queries among the CEP nodes. The mentioned parameters have to be globally available, and there should be a mechanism to periodically update the required values at runtime with a greater accuracy. Therefore, an introduction of a dynamic query distribution mechanism is rather challenging.

Most of the existing distributed open source CEP solutions have not paid attention to the aspect of fault tolerance [2]. At the same time, there are other solutions [1, 7] that provide fault tolerance, but they do not have the dynamic event and query distribution features. Project VISIRI [8] is an example of a distributed CEP system that supports query distribution. FUGU [9] is an example of dynamic operator distribution, and Borealis [10, 11] is an example of dynamic stream processing. The main drawback as mentioned in VISIRI, is that there is a possibility of losing the queries during the query distribution period. Another drawback in VISIRI is that it only supports static query distribution. FUGU has performance problems due to the possibility of Central Processing Unit (CPU) spikes that occur during frequent operator movement. Another drawback in FUGU is that it only supports dynamic operator distribution. The major limitation in the Borealis system is that it does not support query distribution since it has been designed for operator distribution. According to the research mentioned above, it is important to note that dynamic query distribution is not fully supported in those.

1.3 Objectives

The objectives of this research project are as follows:

1. Designing and implementing a dynamic query distribution mechanism for CEP systems by optimizing the throughput and latency of the system. And answer the following three major questions:
    a) When should queries be distributed?
    b) What queries should be distributed?
    c) How can queries be distributed?

2. Evaluate the implemented dynamic query distribution algorithm and provide recommendations on how to execute dynamic query distribution efficiently in a given distributed CEP system.
3. Benchmark the solution with some existing solutions such as VISIRI. [13].

1.4 Contributions

The major contribution of GATHIKA is to introduce dynamic query distribution mechanism for CEP systems that improves overall system throughput. In addition, the initial query distribution algorithm of VISIRI is also improved to distribute the initial query set with low latency.

1.5 Organization of the thesis

Chapter 2 discusses related work found in the literature mainly on dynamic query distribution. Other than that, there are subsections on static query distribution and operator distribution. Chapter 3 presents the methodology in which the mentioned objectives of this research project are accomplished. This section describes in detail the design of the dynamic query distribution system, and the algorithms used. Chapter 4 gives a detailed description of all the experiments carried out, measurements taken, and the critical evaluation of each experiment. Chapter 5 discusses the results obtained from the experiments. Chapter 6 provides the conclusion and possible future work in this research area.

2 LITERATURE REVIEW

This chapter discusses research areas related to distributed complex event processing. The first part is a general introduction to complex event processing and the major terms used in the domain, followed by a section covering the available complex event processing engines. Next, query distribution is discussed, which is the main focus of this research. This part consists of query distribution strategies, existing research projects, and their drawbacks, as well as the positive aspects that can be taken from those projects.

2.1. Complex Event Processing (CEP)

2.1.1 Event

An event is an object that represents or records an activity that happens or is thought of as happening [1]. Examples are:

- A purchase order (records a purchase activity).
- An email confirmation of an airline reservation.
- A stock tick message that reports a stock trade.
- A message that reports an RFID sensor reading.

2.1.2 Complex Event

A complex event is an event that is an abstraction of other events, or denotes a set of other events [1]. Composite events are a type of complex event composed of a set of simple events or complex events.

Following are few examples of complex events.

- Stock market crash in 1929 (an abstraction denoting many thousands of member events, including individual stock trades) [1]
- The South Asian tsunami in 2004 (an abstraction of many natural events) [1]
- A CPU instruction (an abstraction of Register Transfer Level (RTL) events) [1]
- A completed stock purchase by a client (an abstraction of the events in a transaction to purchase the stock) [1]

2.1.3 Event Queries

These queries are continuously evaluated against the set of events that are continuously arriving from event-generating sources. When the queries used in CEP are compared with database queries, database queries operate on a finite set of data [14] and complex queries operate on a dynamic set of data. Following are some operations used in complex queries. Selection and projection, join and split, aggregation, ordering.

2.1.4 CEP systems

CEP systems are capable of conducting operations such as complex pattern matching, event correlation, abstraction, hierarchical organization, causality, membership, and timing. CEP can provide an organization with the capability to define, manage and predict events, situations, exceptional conditions, opportunities and threats in complex, heterogeneous networks. These observations help to improve the operational and situational awareness in many business scenarios [14, 15, 16, 17].

Complex event processing systems are involved in many domains such as,

- Search documents for a set of specific keywords
- Air traffic scheduling
- Radio frequency identification analysis
- Financial market transaction pattern analysis
- Banking applications
- Health management
- Weather forecasting

Let us look at an example of CEP taken from a real-world scenario.

Consider a banking alert system that wants to provide information to its customers on account activities, stock market activities, and exchange rates. The bank can allow customers to customize the alerts, define a delivery mechanism for the alerts, and the frequency at which alerts are dispatched. CEP rules are used to identify the alerts and decide on the delivery mechanism. Following are some rules used in such systems.

Notify when there is ‘X’ number of Asynchronous Transfer Method (ATM) withdrawals with an amount greater than ‘A’ within ‘H’ hours, where ‘X’, ‘A’ and ‘H’ are the parameters defined when customizing the alert.

Notify when there are withdrawals higher than the total amount ‘X’ from an account within ‘H’ hours, where ‘X’ and ‘H’ are the parameters defined when customizing the alert.

2.2 Event Streams

A set of events generated by event sources is received by the event processing system one at a time or as a stream. In an event stream, the events are usually ordered by a timestamp, but it may vary depending on the event source. The volume of generated events is very high so it is normal to receive events as streams. An event stream can be homogeneous or heterogeneous. In a homogeneous event stream, all the events received belong to the same type. However, in a heterogeneous event stream, events are varied. That means event streams consist of different types of events like stock market data, weather forecast data, and RSS feeds. Typically, heterogeneous events are expected by event processing systems.

2.3 Complex Event Processing Engines

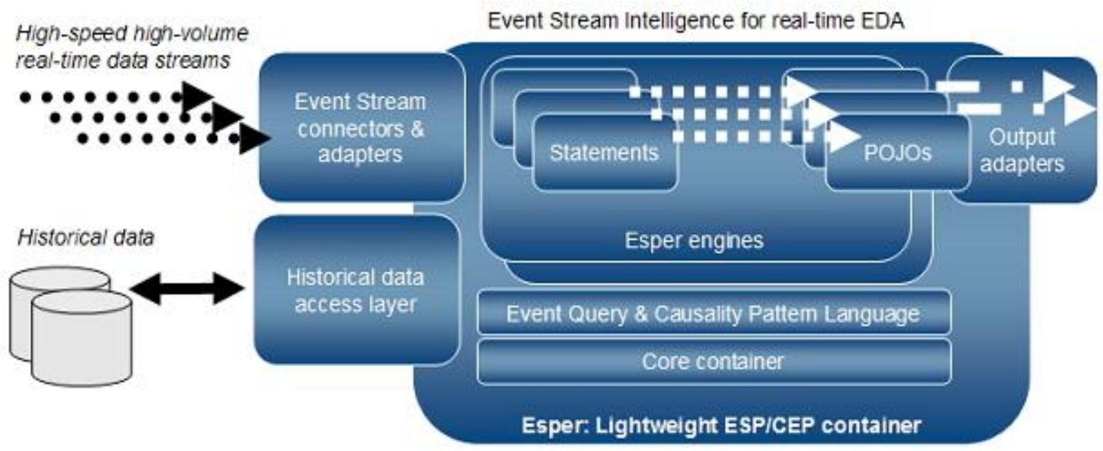
Several CEP products are available in the market, some of which are free and open source, while others are commercial products. CEP engines provide the runtime to do the complex event processing. The CEP engine accepts a set of queries provided by the users and those queries are then matched against the event streams. A notification is triggered if the events provided are matched against the condition satisfied in the query. The following subsections contain a more detailed description of some of the available CEP engines.

2.3.1 Esper

Esper [18] is an open source event processing engine that is used to detect event patterns and trigger actions on received events, based on the conditions met. Esper is designed for Java applications, and a separate engine called Nesper is used for .Net. It provides an Event Processing Language (EPL) that can be used for filtering, aggregation and to join operation multiple event series.

Esper supports a wide variety of event representations including Java beans, XML documents, and simple name-value pairs. Esper can be easily embedded into existing Java applications without doing a serialization of the message. Other than that, Esper is fully embeddable in existing Java application servers and ESBs. Moreover, Esper can run as a standalone container for any standalone application.

One of the problems of Esper is that its performance stops improving after the event rate exceeds a certain threshold [18]. Figure 2-1 [18] shows the Esper engine architecture.



High-speed high-volume real-time data streams
Historical data
Event Stream Intelligence for real-time EDA
Event Stream connectors & adapters
Historical data access layer
Statements
POJOs
Output adapters
Esper engines
Event Query & Causality Pattern Language
Core container
Esper: Lightweight ESP/CEP container

Figure 2-1 Esper Engine [18]

2.3.2 Simple Scalable Streaming System (S4)

Simple Scalable Streaming System [19] is a distributed, scalable, fault-tolerant and pluggable streaming platform developed to solve problems in search applications that use data mining and machine learning algorithms. More servers can be added to increase the throughput and scalability.

Following are the major design goals of the S4 project.

- Provide a simple programming interface for the users.
- Design a high available cluster that can scale using commodity hardware.

- Use local memory in processing nodes. This minimizes latency and avoids disk I/O issues.
- Use decentralized and symmetric architecture where each node shares the same functionality.
- Use pluggable architecture to make it easily customizable.

Figure 2-2 [19] shows the structure of the S4 processing node.

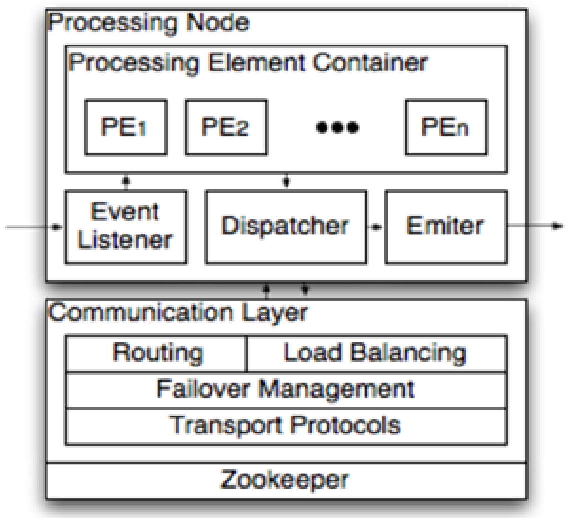


Figure 2-2 S4 Processing node [19]

The processing node contains multiple Processing Elements (PE). The task of the processing node is to read incoming events and perform operations on them. The processing element container is a cluster of PEs, which invokes the relevant PE to process events in the correct order. Events are sent to multiple processing nodes as well. The communication layer provides cluster management and failure management. It can automatically detect hardware failures and does the required functionality of notifying the failures.

2.3.3 Siddhi

Siddhi [20] is an open-source CEP engine that has overcome some of the problems in existing CEP engines such as S4 [19] and Esper [18]. Those existing CEP engines have not provided much support for complex queries, high memory consumption, and are not open source. Figure 2-3 [20] shows the Siddhi architecture. As shown in this figure, Siddhi receives events from event sources through the input adaptors. Then, all incoming types of events are converted into a common data model that is known to the Siddhi core. When users submit queries, those queries convert into a runtime representation and are deployed in the Siddhi core. The Siddhi core does all the processing consisting of event processors and event queues.

Inside the core, processors and event queues are placed as pipelines, after which, outputs are created accordingly and placed in output queues. In the pipeline, each processor has several executors that express the query conditions. For incoming events, executors produce a Boolean value stating whether that event is a match. Logical executor’s process is matching events and any others simply are discarded. The publisher/subscriber method is used to move data between the Siddhi, pipeline processor.

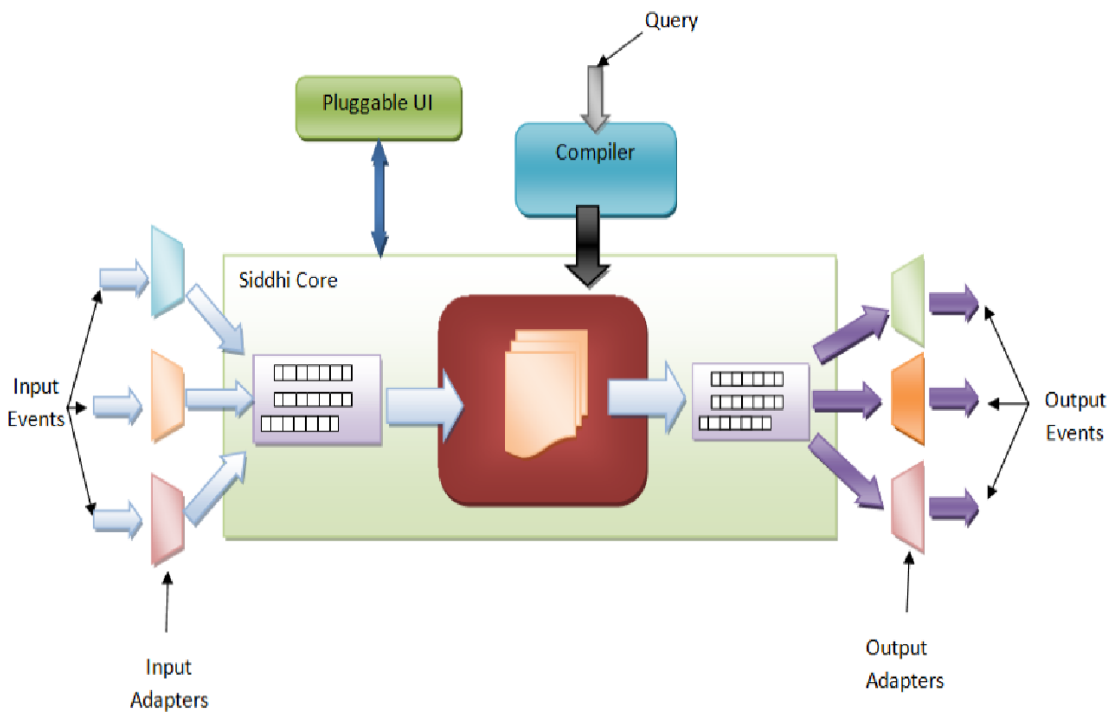


Figure 2-3 Siddhi Core [20]

Siddhi uses more Structured Query Language (SQL) like queries, and it uses optimization techniques to optimize the provided queries. Each Siddhi query produces an output stream that can be given as an input to another query. The complex queries can be modeled as combinations of simple queries that are given as inputs to another query. Manipulation of queries is also possible in Siddhi, where users can add/remove queries at runtime.

In the case of a processor requesting multiple streams, Siddhi uses a single event queue to multiplex multiple streams. This way, different event queues do not need to be checked all the time. Another important thing to note is that Siddhi uses state machines to support pattern queries and sequence queries. Pattern queries fire an event when the given IF conditions for the series are satisfied one after the other.

2.3.4 Cayuga

Cayuga [21] is a general purpose event monitoring system. It has six main operators; projection, selection, renaming, union, conditional sequence and iteration [21]. Cayuga supports online detection of complex patterns from event streams. Other than that, Cayuga uses custom heap management, indexing of operators and reuses shared automata instances. Cayuga can process complex events on a large scale and can detect complex patterns within the event streams online.

Cayuga has its event language designed for expressing queries over the event streams. The queries are the mapping of algebra operators to the SQL-like syntax.

2.4 Complex event processing architectures

This subsection talks briefly about complex event processing architectures. These are used to deploy CEP systems. Project Epzilla [1] is an example of CEP architecture. It provides a scalable fault-tolerant architecture to deploy CEP systems. This was a research project developed during the time where the concept of CEP was not so popular. The main plus points that can be taken from this project are how to deploy CEP engines in a distributed environment and fault-tolerant mechanisms that need to be considered.

2.5 Dynamic Load Balancing

Dynamic load balancing in CEP systems is not an easy task. It involves both event distribution, as well as query distribution. This is the same for dynamic load balancing in any distributed system. As mentioned by Ali et al. [23], a major issue in distributed systems in general is to design an efficient algorithm to perform dynamic load balancing, ultimately increasing the overall performance of the distributed system.

Following are some of the benefits of using load balancing in distributed systems [23].

- Improves the overall performance of the system
- Reduces the idle job time
- Smaller jobs will not suffer from the starvation
- Shorter response time
- High throughput
- High reliability
- Maximum utilization of the resources

Normally in a heterogeneous system, processing nodes are not the same. The processing speed of the node, the communication link speed, and memory available in the processing nodes, are different to each other. Therefore, the processing speeds of nodes and the communication link speeds need to be considered when doing dynamic load balancing. Otherwise, the slowest node in the system becomes a bottleneck and can limit the performance of the overall system.

Figure 2-4 [23] shows the graphical view of the dynamic load balancing function. According to the figure, load balancing happens by transmitting the load from the heavily loaded node to the lightly loaded node. Dynamic load balancing is not part of the process allocator, but a part of the overall system. So when a processing node becomes heavily loaded, the neighboring nodes can handle the load. Various algorithms are available to achieve this task. Following are some of the algorithms available in the literature [23].

- Nearest neighbor algorithm
- Random algorithm
- Cyclic algorithm

- Prioritized random algorithm
- Adaptive controlling with neighbor

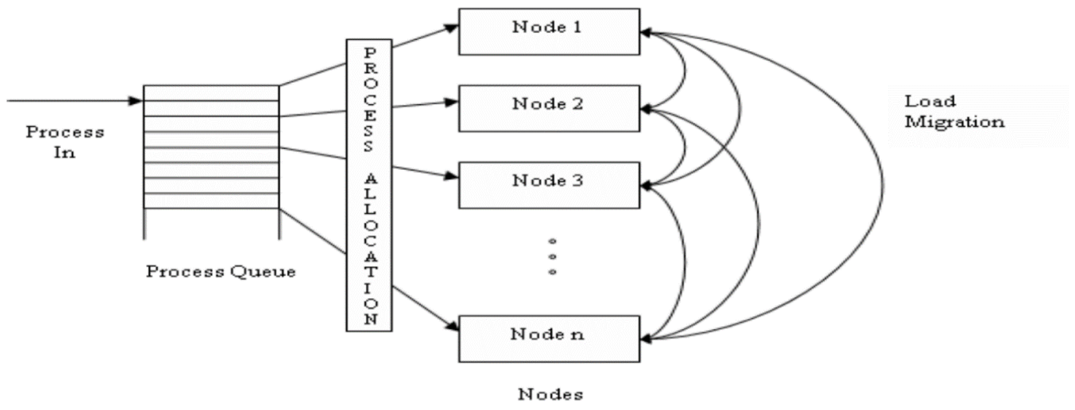


Figure 2-4 Dynamic Load-balancing [23]

2.6 Query Distribution Strategies

There are two main strategies used in CEP systems for query distribution and deployment; operator distribution and query distribution. Section 2.6.3 describe about existing operator distribution research. As mentioned earlier, the main focus in this research is on query distribution.

In query distribution, a set of queries is allocated to the processing nodes in the distributed CEP system by minimizing the network communication cost and load variation. Further, query distribution can be divided into two major parts; static query distribution and dynamic query distribution. This classification is based on the type of algorithm used by the CEP system to distribute the queries. The following subsections discuss existing distributed CEP systems.

2.6.1 Static query distribution

Static query distribution is the process of distributing queries among processing nodes in the system during deployment time[2]. The system needs to have a clear understanding of the queries required for CEP, mainly to do static query distribution. Following are some of the advantages and disadvantages of using static query distribution in CEP systems.

Advantages of using static query distribution:

- In a known context, it is very efficient in pre-identifying a query set
- Increased overall system performance

Disadvantages of using static query distribution:

- Difficult to add/edit queries at runtime
- Once configured, nothing can be done until the next restart of the system

The following subsections describe existing distributed CEP systems that use static query distribution.

2.6.1.1 Scalable Context Delivery Platform (SCTXPF)

Scalable context delivery platform [2] has been optimized for a large number of complex event processing rules and for huge event rates. Event Processors (EP), which operate independent to each other, achieve high throughput and scalability with CEP. Scalability is considered using the number of events received and the number of CEP rules. Event processors are parallelized by allocating a certain number of CEP rules for each. When allocating queries to the event processor nodes, this algorithm tries to allocate queries with the same attributes to the same node. This is to reduce event multicast at the event dispatcher level. If this is not taken into consideration, there will be a lot of event multicasting and the overall system performance will be degraded. Load imbalance problems that can occur due to this kind of configuration are also addressed in the system.

There are three major components in this system; Event Processor control unit, Event Processors (EP) and the Dispatcher. The task of the control unit of the event processor is to accept CEP rules from the application and allocate the rules to the event processing nodes. It also generates dispatcher rules that are used by the Dispatcher to distribute events to the relevant event processing node. Event processing nodes do the CEP tasks. EP nodes contain a rule engine that does the CEP. The dispatcher receives events from the event sources and forwards the events to the relevant EP nodes according to the dispatcher rules. If there is more than one EP node, it copies the events and multicasts the events to the EPs. If there is no match, that event is filtered out. This reduces the load on the EP nodes and the overall network traffic in the system.

An objective of the rule allocation algorithm is to minimize the number of event processing nodes that need the same event streams. This helps to minimize the number of multicasts at the dispatcher and minimize the event processing nodes that hold the same states of events. Therefore, the processing node is not overloaded while the other nodes are idle.

In the rule allocation algorithm, a set of all event processing nodes are created first and then the nodes that have queries more than or equal to the threshold value are removed. The EPs that are selected are those that have the most common necessary attribute values when compared with the new. If there are multiple nodes of the same type, one is selected randomly.

With this rule allocation algorithm, SCTXPF reaches an event rate of 2,700,000 events/second and high scalability [2].

Although SCTXPF has a simple and efficient way of distributing operators in the CEP system, it does not support dynamic adjustments in the system. Therefore, handling overloaded situations in SCTXPF may not be efficient for varying event loads in streams. In addition, there is a high probability of overloading the node due to computationally intensive operations performed on operators before distribution takes place.

2.6.1.2 Static query distribution in VISIRI

VISIRI [8] is a distributed complex event processing system designed for a large number of queries. This system consists of event dispatchers, CEP nodes, event sinks and sources that generate events. The three major algorithms described in this research are; the initial query distribution algorithm, the dynamic query distribution algorithm, and the transferable query selection algorithm. Initial query distribution is the static part of the query distribution.

The algorithm introduced in VISIRI aims to reduce network bandwidth by reducing event duplication and to maximize processor utilization by considering the complexities of the queries themselves. The initial query distribution algorithm takes the following inputs when deciding an efficient query distribution.

- Set of queries to be distributed
- Set of processing nodes and dispatchers
- Queries currently allocated for each node

Other than that, the algorithm considers factors like costs of the queries, number of existing queries in each node, and the number of common event types required for the query. These factors are not considered in the SCTXPF algorithm. Hence, VISIRI has a better outcome in initial query distribution when compared to SCTXPF.

2.6.2 Dynamic Query Distribution

Dynamic query distribution concerns the distribution of queries among the processing nodes during runtime. The dynamic distribution of queries for CEP is an NP-hard problem [1, 5]. Parameters such as the current system load, the current load of a given node, the availability of the nodes, and many more parameters need to be considered when distributing queries among the CEP nodes. The mentioned parameters have to be available globally, and there should be a mechanism to update required values at runtime with greater accuracy. Therefore, with the introduction of a dynamic query distribution mechanism, it is very difficult to work with CEP systems.

Following are some of the advantages and disadvantages of using dynamic query distribution in CEP systems:

Advantages of using dynamic query distribution:

- Add new queries at runtime
- Update the queries with the changing requirements
- Load balancing at runtime

Disadvantages of using dynamic query distribution:

- Performance degrades due to dynamic query distribution algorithm
- Possibility of losing events during the dynamic query distribution time

The following subsections describe the systems that are currently available for dynamic query distribution, dynamic operator distribution and query adjustment at runtime.

2.6.2.1 Dynamic query distribution in VISIRI

In the VISIRI architecture, a user can select any CEP node and deploy the queries in that node. That node then becomes the primary node, executing the query distribution algorithm and distributing the queries among all active CEP nodes. These queries are automatically deployed in the CEP nodes, and the information on the queries is notified to the event dispatcher. The Dispatcher maintains a forwarding table according to the queries allocated to the CEP nodes. The idea is to reduce network traffic by sending relevant event streams to the relevant CEP nodes. When a CEP node finishes the processing of an event stream, the resulting stream is sent to the event sink.

In the query distribution algorithm, the cost of the queries, the number of existing queries in each node, and the number of common event types required for the queries are considered. This is to make sure that the queries with higher costs are not deployed on the same node and to avoid overloading the CEP node.

Steps of the VISIRI dynamic query distribution algorithm are summarized below.

- First, place all nodes in the candidate list.
- Find the minimum utilization level and filter nodes above a certain threshold.
- Find the minimum total cost among the nodes and filter nodes above the cost threshold.
- Select the nodes with the maximum common event types.
- Choose a node randomly from the filtered candidate list.
- Distribute the queries to that selected node.

Figure 2-5 [8] shows the high-level system architecture of the VISIRI system.

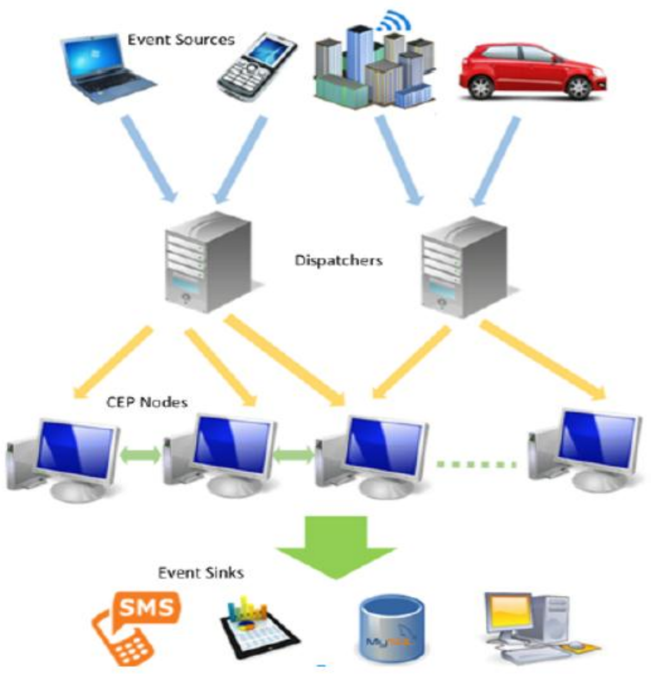


Figure 2-5 VISIRI high-level system architecture [8]

When adding new queries to the system, the dynamic query distribution algorithm is used to distribute the queries.

The transferable query selection algorithm is used to transfer queries in a CEP node when its utilization exceeds the defined threshold level. These queries are transferred to the nodes within the system to maintain the overall system utilization at a higher level and to maintain the system throughput at a higher level. This algorithm considers the incoming event rate for a particular query and the cost of the query. The most suitable transferrable queries are chosen by calculating the cost rate of the query [18, 24, 27]. Queries within the middle range of the cost rate are selected for the transfer. This is to maintain the utilization of the particular processing node and to make sure the receiving processing node is not overloaded due to the arrival of queries.

VISIRI does not consider the network communication cost when doing dynamic query distribution. There may be a possibility of losing the queries during the dynamic distribution period, which must be considered.

2.6.3 Operator distribution

In operator distribution, it divide a complex query into distinct sequence of steps and execute them as separate queries in different processing nodes.

2.6.3.1 Distributed Stream Processing in Borealis

Borealis [10, 11] is a distributed stream processing engine that describes the correlation-based load distribution algorithm. By minimizing the load variance, overload conditions are avoided, and the end-to-end latency of processing an event is reduced. The initial load distribution algorithm is used for initial operator placement, and the pairwise algorithm is developed for dynamic load distribution. Achieving the incremental scalability and high availability are two key reasons to distribute the stream processing load among multiple machines. Major differences between stream processing and CEP are that stream processing engines tend to be parallel and distributed, while CEP engines tend to be more centralized. Another thing is that CEP engines have quicker response time compared to stream processing engines.

The algorithm, as mentioned earlier, balances the load among the processing nodes and minimizes the load variance on each node. In their experiment, connected operators were placed in different nodes, and better performance was achieved than if the operators were placed on the same node. If the correlation coefficient of the load time series of two operators is small, then placing those operators in the same node can minimize the load variance. What they try to emphasize is that the average load level is not the only factor for load distribution but that the load variance is also a key factor in a push-based load distribution.

The architecture of a single node in the Borealis system is shown in Figure 2-6 [11].

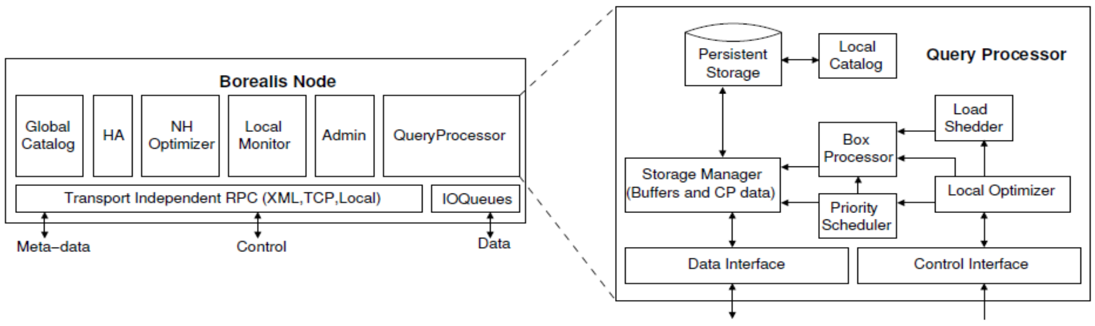


Figure 2-6 Borealis system architecture [11]

In the Borealis architecture, query execution occurs locally within the Query Processor (QP) module.

The Admin module is responsible for controlling the local QP. The Admin module coordinates with the Local Optimizer to find performance enhancements. The Local Monitor collects performance-related statistics as the local system runs, to report to local and neighborhood optimizer modules. Although this is based on operator distribution, the concepts of initial load distribution and dynamic load adjustment are important for the dynamic query distribution mechanism as well. A major limitation of Borealis is that it does not support query distribution.

2.6.3.2 FUGU Elastic complex event processing under varying query load

The use of elastic scaling in FUGU allows event processing systems to react to the dynamically changing queries and events. When moving queries dynamically between hosts, processing needs to pause until the movement queries are finished executing. If the query distribution latency is high, it causes a possible loss of events. FUGU [9] is a modeled latency-aware elastic operator movement algorithm. Minimizing the number of latency violations and maximizing system utilization are important considerations in the project.

Figure 2-7 shows the system architecture of FUGU [25].

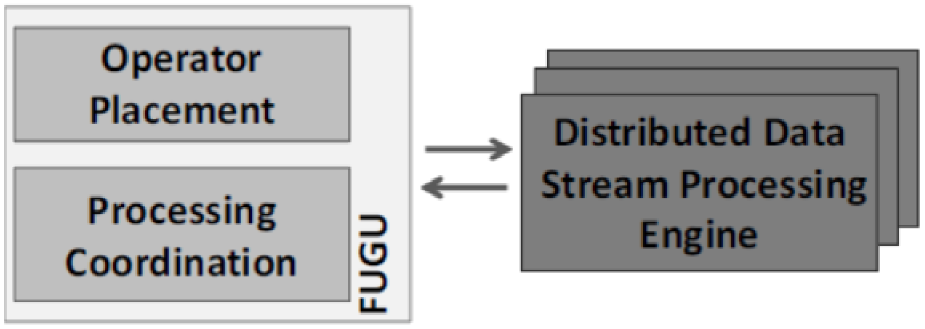


Figure 2-7 FUGU system architecture [25]

The CEP system consists of multiple instances running in parallel on different nodes. The CEP system can accept and process queries arriving continuously at these nodes. Query operators are distributed on different hosts, and the final result needs to be calculated based on the output from different hosts consisting of query operators. FUGU works as the centralized component that allocates queries to the different hosts by calculating the placement decisions. It uses the bin packing algorithm [26] when a new query is added, or an existing query is removed at runtime. In the bin packing algorithm, items are allocated to a bin in such a way that the minimum number of bins are used.

In the dynamic operator placement algorithm used in this system, a load model approach is used. This measures the CPU, memory and network consumption for each operator. When adding a new query at runtime, all the variables in the model are estimated assuming a worst-case scenario. The required CPU load for an operator is a major factor. Other than that, hosts with insufficient memory and high network bandwidth are removed from the target host list. The bin packing approach helps to scale the system in and out with dynamic addition or removal of queries from the system.

Although this research deviates a little from the dynamic query distribution concept, it still provides important factors that must be considered when doing dynamic query distribution. Here, it provides how to evaluate a processing node and its characteristics when doing dynamic adjustments of queries in the system. Project

FUGU has performance problems where frequent operator movements cause spikes in the end-to-end latency [9]. Therefore, performance improvements are required for the FUGU system.

2.7 Cost model development

It is essential to have a good cost model to perform an efficient query distribution operation. In literature, there are cost models found in different types of CEP systems [19, 27].

The NEXT CEP [27] system employs query rewriting. Query rewriting is a technique that is used to get the maximum out of available resources such as the CPU. The cost model development is one of the major outcomes of this research. It allows automated optimization of queries. As mentioned in the research paper, memory, CPU time and network bandwidth are the three significant resources in a distributed system. With the aid of query rewriting, they try to reduce the bottleneck that occurs due to limited CPU resources. The intention is to reuse existing operators to minimize the CPU usage, latency, and variance of CPU usage.

In the cost model, they made the following assumptions:

- Cost model gives asymptotic cost of operators
- Ignore the cost of predicate evaluation
- Ignore the selectivity of operators

The cost model calculates the cost of a query as a summation of costs of various operators in that query. Then, it does query rewriting to find the most efficient pattern.

The NEXT CEP system architecture, which is shown in Figure 2-8, consists of a central manager that receives, processes, and optimizes queries and instantiates queries on the available operator nodes. The node manager monitors the system and gets information of the operator nodes available for operator distribution. They chose a centralized design for simplicity, and it helps query optimization as well.

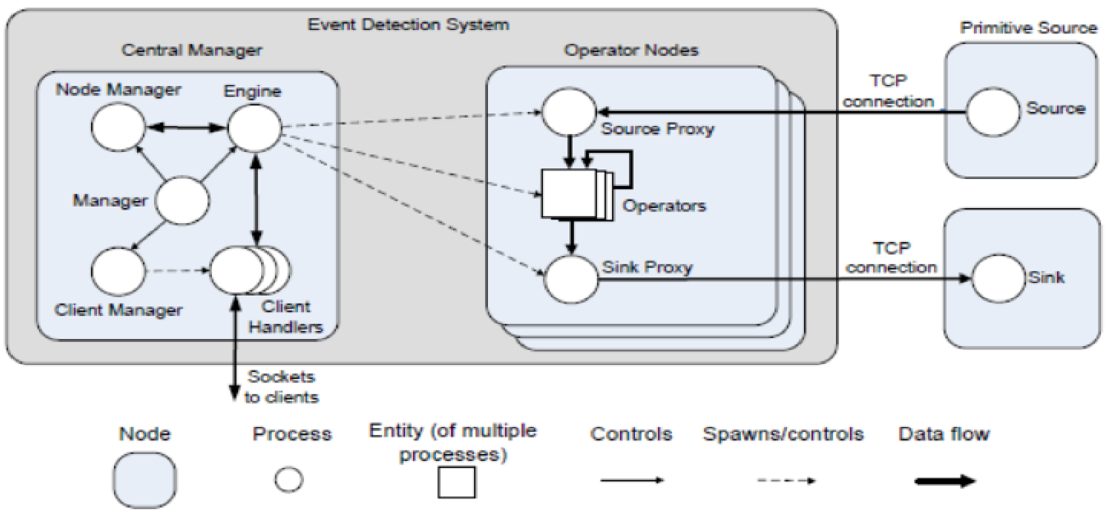


Figure 2-8 Next CEP system [26]

VISIRI [8] has developed a cost model for its dynamic query distribution mechanism. In that cost model, first the execution cost for a particular query is estimated, and the queries that require considerable resources are identified. The queries that require higher resources are not deployed on a single node. This is to avoid overloaded situations in a processing node. Following are the parameters used to assign the cost value for a query.

- Number of filtering parts in a query
- Number of attributes in input stream definition and output stream definition
- Number of input streams and output streams
- Window length of queries

2.8 Summary

The design concepts and system architectures mentioned in the above subsections have some drawbacks and limitations. Following is an analysis of the major system architectures discussed in the above sections.

VISIRI system implements the static query distribution. However, the dynamic query distribution part is not fully implemented in VISIRI. VISIRI has theoretical concepts such as the rule allocation algorithm mentioned in the SCTXPF architecture. The major drawback seen in VISIRI is that it does not consider the network communication cost when doing dynamic query distribution. There may be a possibility of losing the events during the dynamic distribution period. According to the VISIRI research [8], it has benefits like high performance and high throughput than SCTXPF architecture.

As mentioned in the subsections above, Borealis and FUGU systems have their limitations. However, they were designed for operator distribution. However, the concepts and approaches mentioned above are useful in coming up with a suitable architecture for designing a dynamic query distribution mechanism for the CEP system.

3 DYNAMIC QUERY DISTRIBUTION IN CEP SYSTEMS

In Chapter 1 under the objectives of this research, I mentioned three questions that need to be answered in this research.

a) When should queries be distributed?
b) What queries should be distributed?
c) How can queries be distributed?

This chapter discusses the dynamic query distribution system that answers the above questions.

3.1 Overview

The architecture of GATHIKA is influenced by the VISIRI [8], Borealis [10, 11], FUGU [9], and epzilla [1] projects.

VISIRI

The initial query distribution algorithm developed in VISIRI is used as the base for developing the GATHIKA system. This algorithm is modified in GATHIKA to achieve higher performance. The initial query distribution algorithm developed in GATHIKA that is described in Section 3.3.1 has higher performance than the algorithm used in VISIRI. Transferable query selection algorithm in VISIRI is not completed. Because it does not select appropriate queries to transfer. Hence, in GATHIKA project transferable query selection algorithm is redesigned as described in Section 3.4. The algorithm used for cost model development in VISIRI, which is described in Section 3.6, is reused without any modification.

Borealis

The concepts of having the initial query distribution algorithm and minimizing the load variance are important ideas taken from the Borealis project.

FUGU

Bin packing algorithm is an important algorithm implemented in FUGU. The concept of effectively adding new queries to processing nodes is taken from the bin packing algorithm. FUGU considers CPU and memory usage as important factors for operator placement. This concept, is taken when deploying queries in the GATHIKA system.

Project Epzilla

The concept of creating a cluster of processing nodes is taken from Epzilla [1]. Also, having a caching framework to achieve high availability and fault tolerance are concepts taken from Epzilla.

3.1.1 High-level architecture

Following are the major components of the architecture.

- Event Dispatcher – responsible for distributing the events to the processing nodes.
- Processing nodes – does the actual event processing based on the queries deployed on it. Processing nodes are arranged in a cluster where each cluster has a leader node.
- Accumulator – accumulates the final result from the individual results from the processing nodes

Figure 3-1 shows a high-level overview of how the different components interact with each other.

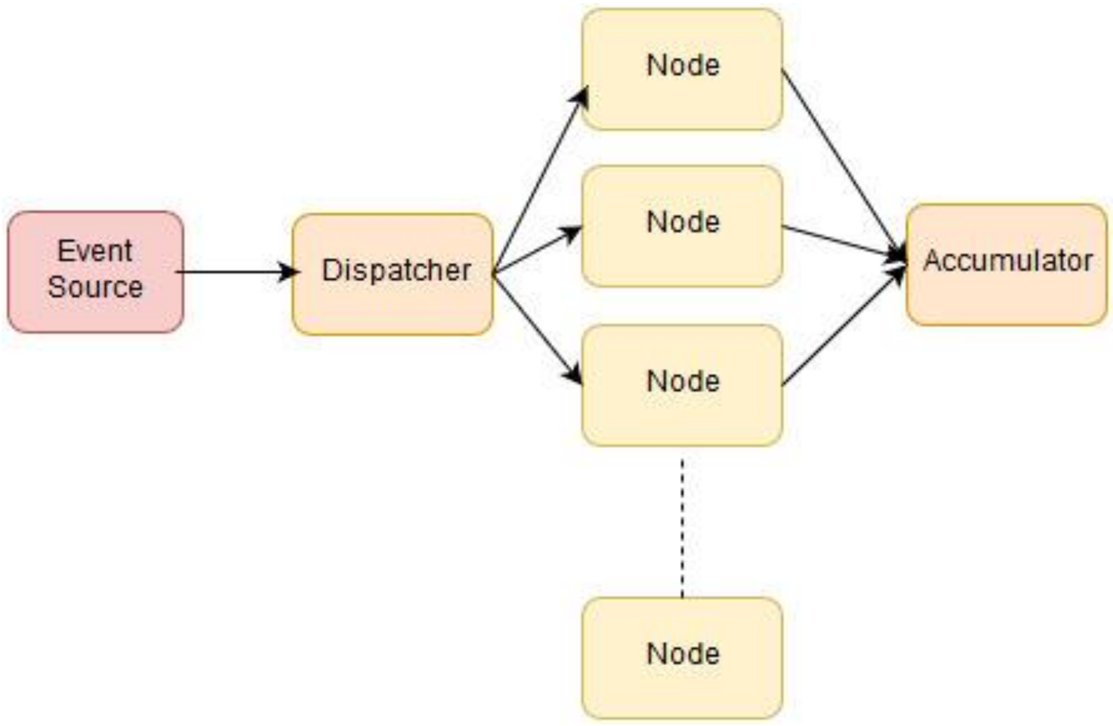


Figure 3-1 High-level architecture of the system

3.2 Components

Since this research uses VISIRI as the baseline, some of the features developed in the VISIRI project are reused in GATHIKA. Source code of environment, event processing node, event dispatcher, and accumulator components were taken from the VISIRI project, and they were modified when implementing the dynamic query distribution feature. Implementation of each component is listed in the subsections below.

3.2.1 Environment

'Environment' is a special component or caching layer that is responsible for keeping all the shared data among all the components in the system [8]. This implements the Hazelcast [28] message listener interface. Singleton pattern is used to make sure that only one shared single object is there to access the caching layer of this system.

3.2.2 Event Dispatcher

The dispatcher is the component used in this architecture to accept the events from external sources and route the events to the event processing nodes. The dispatcher can start in two different modes.

Buffering mode

This mode provides the option to buffer the incoming event stream at the dispatcher. This is useful when the dynamic query distribution is in progress within the event processing nodes. When dynamic query distribution starts, the leader node notifies the dispatcher to buffer the incoming event streams. Once the dynamic query adjustment is finished, it notifies the Dispatcher again.

No Buffering mode

In this mode, the dispatcher routes the receiving event stream to the event processing nodes. It does not check the status of the event processing nodes. In this mode, events may get lost.

Figure 3-2 shows the internal architecture of the dispatcher. The dispatcher has an engine handler, event listener and list of event clients.

Engine handler

When the dispatcher starts, the engine handler thread is started. The engine handler has a reference to the event server and list of event clients.

Event Listener

The task of the event listener is to listen on a given port and accept the events received from the external sources. Received events are passed to the engine handler for further processing.

Event Client

When event processing nodes start and join the cluster, each node notifies its status to the dispatcher. The dispatcher keeps a list of event processing nodes and starts an event client for each event processing node. The dispatcher routes the events to event clients, and the responsibility of the event client is to route the assigned events to the respective event processing node. As shown in the figure below, the dispatcher has an Engine handler. This engine handler has the event server and list of event clients started for each processing node.

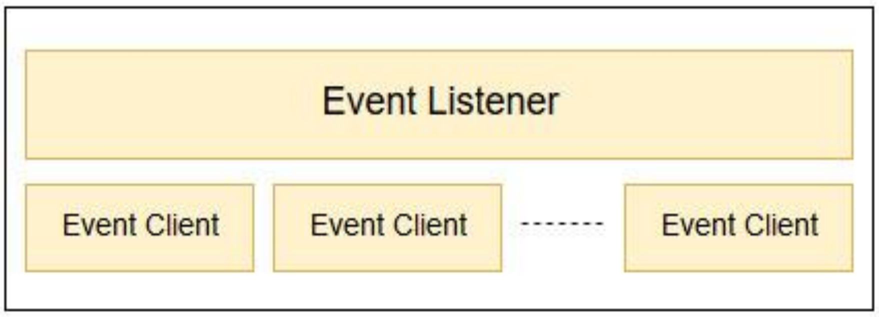


Figure 3-2 Internal architecture of Dispatcher

3.2.3 Processing Node

Processing nodes do actual event processing. There are two types of nodes in this architecture; the leader node and the normal processing node.

Figure 3-3 shows the internal architecture of the processing node. The node listener and Siddhi CEP engine are the major components inside the event processing node.

Node listener – accepts events from the Dispatcher

Siddhi CEP engine – does the event processing task inside the node

The number of CEP engines are equal to the number of queries deployed in the processing node.

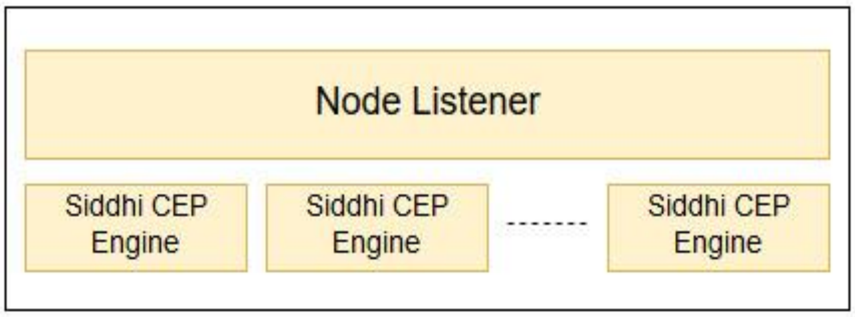


Figure 3-3 Internal architecture of Event processing node

3.2.4 Accumulator

The task of the accumulator is to collect the information sent by the event processing node and generate the final result or notification of the processed events.

3.2.5 Functioning of the system

Initially, the main node is responsible for deploying the user provided queries. During the runtime of the system, the user can add queries to any processing node. For evaluation, an option is given to generate the given number of random queries to a particular node.

Queries are deployed in the processing nodes and each processing node has a separate CEP engine inside it to handle a single query. If a node contains ten queries, that implies that it has ten CEP engines deployed in it.

A leader node exists within the cluster of event processing nodes. This leader node is responsible for collecting the statistics of the nodes within the cluster. Statistics include,

- Current CPU usage of the node
- Number of queries deployed in a node
- Frequency of query usage

These statistics are used in the dynamic query distribution algorithm. The algorithm is executed in a given cluster by the leader node to balance the behavior of the cluster.

There are two other parameters defined in the system

- Minimum time between two consecutive dynamic query distributions
- Minimum number of queries for dynamic query distribution

The above parameters are used when adding new queries during runtime of the system. This will avoid unnecessary dynamic query distributions that can occur in the system. Frequent dynamic query distributions degrade the overall system performance, which is discussed in the next chapter. Details of the query distribution algorithm will be discussed later in this chapter.

All the factors mentioned above have a defined threshold value. If the current value of a factor exceeds the threshold value, then it triggers the dynamic query distribution in the cluster. This is the answer to question a) mentioned above.

Given below is an example that demonstrates how the threshold value is used to determine when to distribute the queries.

- Threshold value of CPU usage of a node = 80%
- Due to heavy load on a given node let us say that the current CPU of the node > 80%
- When this happens, dynamic query distribution needs to be triggered.

The answer to question b) is queries that are deployed in the overloaded nodes. This is identified by using the transferable query selection algorithm that will be discussed later in this chapter.

The answer to question c) is to distribute queries using a dynamic query distribution mechanism. A detailed description of the algorithm will be discussed later in this chapter.

3.3 Query distribution algorithm

Query distribution is categorized into two parts called initial query distribution and dynamic query distribution. The following subchapters will provide a detailed description of algorithms.

3.3.1 Initial query distribution

VISIRI took SCTXPF algorithm as their baseline for development of the initial query distribution algorithm. VISIRI aimed to reduce the network bandwidth by reducing event duplication and maximizing processor utilization by considering the complexities of the queries. However, the VISIRI algorithm does not scale well when the number of queries increases.

In GATHIKA, the VISIRI algorithm is modified to consider the query type as well. Query type is identified based on the attributes of the query. That means that the complexity and the number of attributes in the query decide the query type. Each query in the system has a query type assigned to it. In this new algorithm, the query type is also considered when deciding query distribution. The same type of queries is deployed in the same processing node. This addition improves the performance of the initial query distribution algorithm by an inconsiderable amount. Performance comparison of initial query distribution algorithms will be discussed in the next chapter. Following is the detailed description of the initial query distribution algorithm.

The initial query distribution algorithm takes the following as its inputs.

- List of queries to be distributed
- List of processing nodes and dispatchers

The algorithm considers many factors in the process, including,

- Query type of the queries (depending on the complexity and number of attributes of a query)
- Number of existing queries in each node

Cost model calculator developed in the VISIRI project was reused since not much modification was needed. These costs may also depend on the underlying implementation of the complex event processing engine. The cost model will be discussed in detail in Section 3.7.

The major difference between the initial query distribution algorithm in VISIRI and that in GATHIKA is that VISIRI does not consider the similarities between queries. VISIRI randomly distributes queries among the processing nodes. The GATHIKA project checks the type of the queries deployed in the processing nodes and the type of the queries that are going to be deployed. This is known as similarity checking, as mentioned above.

3.3.2 Dynamic query distribution

Dynamic query distribution is the main contribution of this research. Dynamic query distribution only occurs at the runtime of the system. More details on dynamic query deployment are discussed in the next subsection. Although we discussed different types of dynamic load balancing algorithms in Section 2.5, those cannot be used directly for dynamic query distribution. If we use random query distribution or the nearest neighbor algorithm, queries get deployed in the processing nodes without considering the query cost and status of the processing node. Therefore, this may reduce the overall system performance.

The dynamic query distribution algorithm is a modification of the aforementioned initial query distribution algorithm. Listed below are the major steps involved in dynamic query distribution.

The dynamic query distribution algorithm takes the following as its inputs.

- Query to be distributed
- List of processing nodes and dispatchers
- Queries currently allocated for the nodes

The query to be distributed is selected using another algorithm called a transferable query selection algorithm, which is described in Section 3.4.

Following are the steps of the dynamic query distribution algorithm.

1. Initially, put all nodes on the candidate list
2. Calculate the total cost of queries deployed in each node and keep it in a separate list
3. Calculate the minimum memory utilization level of processing nodes and filter nodes above the defined threshold value
4. Find the minimum total cost among the nodes and filter nodes above the cost threshold
5. Find the nodes with same query type from the remaining candidate node list
6. Sort the candidate's map by similarity in descending order
7. Choose the node that comes at the top of the similarity order
8. Otherwise, select a random node from the candidate node list

Following is the pseudo code of the dynamic query distribution algorithm.

```
Input: Query q, Node[] processingNodes
Output: target_node
//assign all the nodes to the candidate_list
declare candidate_list = all the nodes
declare min_num_q_node = the minimum number of queries deployed in a node
//declare threshold value for query variability
declare QUERY_VARIABILITY =5
//if the query count in a node greater than min_num_q_node + QUERY_VARIABILITY then remove
node  for all the nodes in candidate_list do
        if node query count > min_num_q_node + QUERY_VARIABILITY then
          remove the node from the candidate_list
        end if
 end for
//declare minimum cost of a node to infinity
declare min_cost = ∞;
for all the nodes in candidate_list do
     cost = calculate the cost of all the queries in the node
     node_cost = cost
        if min_cost > node_cost then
           min_cost =  node_cost
        end if
end for
//define the cost variability threshold
declare COST_VARIABILITY=1
for all nodes in candidate_list do
     if node_cost> min_cost + COST_VARIABILITY then
          remove the node from the candidate_list
    end if
 end for
//define the similarity map to order the nodes with query type count
declare similarity_map = new_map
for all the nodes in candidate_list do
        query_list = get the query list in the node
//count queries in the node that have same query type with the given query
declare count=0
  for all the queries in query_list do
         if the query type of the node = query_type of the new query
```

```
            then count ++
        end if
  end for
        add the node and the value of count to similarity_map
end if
end for
if similarity_map ≠ {} then
        target_node = node having higher value for the count
else similarity_map = ∅ then
   if there are remain nodes in the candidate_ list then
          target_node = randomly select a node from candidate _list
  end if
end if
return target_node
```

Algorithm operates as follows. First new array list called candidate list is initialized. Then it assigns the node list to the candidate list. Then it finds the minimum number of queries deployed in a node and removes all nodes that have queries above the defined threshold + minimum number of queries. This will balance the overhead of having a large number of CEP engines in the same node.

Then it finds the minimum total cost of a node. This is done by taking the sum of the costs of queries deployed in a node. VISIRI cost model [8] is used to calculate the query cost. Nodes having a total cost exceeding the defined threshold value will be removed from the candidate node list. This is to balance the cost distribution among the nodes.

Then it finds the nodes that have a similar query type to the query given as the parameter and count the number of matching query types in the node. All the nodes are added to a map with the calculated count. Then, it orders the nodes by the query types. If the similarity map contains the nodes, then the algorithm takes the node having the highest count value. If not it takes a random node from the available candidate nodes. Query will be added to this selected node.

E.g.:

- Query type of the provided query is QT1.
- Node 1 has ten queries out of which five belong to QT1 and two belong to QT2.
- Node 2 has ten queries out of which two belong to QT1 and eight belong to QT3.
- After the sorting is finished, Node1 has the highest priority since it has five CEP engines with QT1.
- Therefore, Node1 becomes the candidate node to deploy the given query.

Therefore, the proposed algorithm tries to distribute queries targeting balanced CEP utilizations, balanced query distribution, and efficient network utilization.

3.4 Transferable query selection

The transferable query selection algorithm is used to select queries to be transferred from a processing node. As mentioned in previous topics, every node has an agent component deployed. This agent periodically checks the node utilization. If the defined threshold values exceed, then the agent selects queries that need to be removed from the processing node. For the selection of transferable queries, the agent executes the transferrable query selection algorithm. This research reused the algorithm developed in the VISIRI [8] project.

Following is the description of a transferable query selection algorithm. This algorithm selects the most suitable queries to be transferred. The query selection algorithm uses two factors;

1. Incoming event rate for a particular query
2. Cost value of the query

Pseudo code for transferable query selection algorithm is shown below.

```
Input: event_rates [], List query_list
Output: tansfer_query_list
declare size= query_list.size()
declare cost_rate_q_map = new Hashmap
declare query_array =new array [size]
then add query list to the query_array
  if  size of the event rates array ≠ size of the  query_list then
    return
  end if
for all the queries do
calculate the cost rate of queries and add to the cost_rate_q_map
end for
  sort the query array considering the event rates of queries
  //calculate the low index and high index of the middle 10%  of cost value array //
declare threshold value = middle 10%
        min_index=  size/2 – size* threshold
        max_index=size/2 + size* threshold
declare transfer_query_list;
for queries in the range between min_index to max_index do
  add queries to the transfer_query_list taken from the cost_rate_q_map
end for
 return transfer_query_list
```

Algorithm accepts the array of event rates to the CEP engines deployed and the list of CEP engines deployed in the processing node. As mentioned in Section 3.2.3, each query runs in a separate CEP engine.

The transferable query selection algorithm calculates the cost rate of each query. This is done by multiplying the event arrival rate of the query and the cost of the query. Then it adds cost rates to the ‘cost_rate_q_map and sorts the values in the map.

After that, the algorithm selects the middle 10% of the queries. If the algorithm selects the query set with high cost-rate values, then after transferring the queries to other available processing node, then there can be a possibility of overloading the receiving processing node. If the low cost-rate query set is selected, then it will not reduce the utilization level of that particular processing node. Therefore, we assumed that the best query set would be the middle 10% of the queries. Then the selected query set is allocated to the other processing nodes using the dynamic query distribution algorithm by the relevant processing node.

3.5 Dynamic query deployment process

There are two scenarios in the system, which triggers dynamic query distribution.

- When defined threshold values exceed in a processing node
- When adding new queries to the system

Any processing node can run the dynamic query distribution algorithm in the system.

There is an agent component deployed in each processing node and the task of that agent is to collect the CPU and memory utilization statistics of the node. As mentioned in Section 3.2.3, if the utilization value exceeds the threshold value, it executes the dynamic query distribution algorithm.

In this system, a user can submit queries to any of the processing nodes. That node informs its leader about the arrival of queries. The leader then evaluates the system with the statistics and selects the best possible node to allocate the queries. If there are multiple nodes selected for allocating the queries, the algorithm selects the minimum number of nodes with the best suitability. Once the queries are allocated to the respective nodes, the leader informs the dispatcher about the state of the processing nodes. This information is kept at the dispatcher level and is being updated periodically with the information provided by the leader node of the cluster. When the dispatcher allocates the events to the processing nodes, it uses this information to select the best possible node to dispatch the events. Following is the summary of the query deployment process.

- Agent checks its performance of the processing node periodically
- Agent detects that the node is overloaded
- Execute the ‘Transferable query selection’ algorithm to select queries that need to be transformed
- Get performance information of other nodes and current query distribution
- Execute the dynamic query distribution algorithm to decide destination nodes

- Send the event stream blocking message to the dispatcher
- Send the relevant queries and query states to the destination nodes
- Send a message to the dispatcher about the changes of the query distribution
- After new queries are deployed at destination nodes, send dynamic completed message to the dispatcher
- Dispatcher stops blocking such event streams and releases previously blocked events

3.6 Cost model

When new queries arrive to be deployed, the cost of that query needs to be calculated first. Before deciding the query distribution, it is important to know the cost of the query. The cost model developed in VISIRI [8] is reused in this project as well. Following is the description of the cost model calculation.

The Siddhi query language grammar is used to parse the query to get a list of tokens in the query. Using those tokens, queries that have filtering parts are identified and assigned a value to the cost depending on the number of filtering attributes. Then the cost value is assigned for the number of attributes in the input stream definitions and the output stream definition. Apart from that, the number of streams and the output streams count are added to the cost value. Queries with windows were given higher priority. The cost value was assigned depending on the window length and that value increases exponentially with the window length. The logarithmic value of the cost value is then taken to limit it to a certain range.

3.7 Discussion

Since the GATHIKA system is influenced by VISIRI [8], the algorithms developed for initial query distribution and transferable query selection were reused after modifications. The major outcome of GATHIKA is an implementation of a dynamic query distribution algorithm. Performance and throughput of the dynamic query distribution algorithm is included in Chapter 4.

4 MEASUREMENTS AND EVALUATION

Overview

This chapter discusses the measurements taken to describe the work in this thesis. How the research objectives are achieved is described in this chapter by analyzing the measurements taken.

All the measurements were taken in the Level 2 lab of the Computer Science Department at University of Moratuwa. The measurements were taken to capture different aspects of dynamic query distribution and initial query distribution in a CEP system.

Integration setup details

Processor: Intel Core i3-4150 CPU @ 2.60GHz 3.5 GHz

RAM: 4.0 GB

System type: 64-bit

Operating System: Ubuntu 14.04.2

Ethernet connection: 100 Mbps

Event source: mfg-uat-phoenix.mubashertrade.com

Assumption

All the deployed queries in the system are distinct to each other. Therefore, in all the experiments there will not be duplicated queries in the processing nodes.

4.1 Initial query distribution comparison with VISIRI

The initial query distribution algorithm was developed to demonstrate static query distribution in GATHIKA. It is very important to have high performance in this algorithm. Otherwise, with the increase of queries and processing nodes, it takes a considerable amount of time to distribute queries among the processing nodes. Both this project and VISIRI have initial query distribution algorithms implemented. For evaluation purposes, the performance of the initial query distribution is measured in two systems by varying the number of queries and increasing the processing nodes.

Three setups consist of 4, 8 and 12 processing nodes were used for the evaluation. Moreover, evaluation of both scenarios was done using the same set of processing nodes and the same set of queries. The time elapsed for initial query distribution is measured for the evaluation. The random query generation algorithm was used to generate large query sets. Following are sample queries used for distribution.

- from stock[Bid <= 55 and Bid>50] select Symbol,Date,Volume insert into stocks
- from stock#window.lengthBatch(10) select Symbol, max(Open) as open insert into stocks
- from stock[High < 21.96 and Open < 70.44 and Open < 48.16 and Close < 45.24 and Low < 67.69 and Close > 76.05 and Close > 53.39 and High > 67.92]#window.timeBatch(148 milliseconds) select max(Open) as Open, max(High) as High insert into result

Figure 4-1 shows the distribution of 1,000, 10,000 and 100,000 queries in the mentioned three setups. From the measurements taken, it is clear that the algorithm in GATHIKA has higher performance in all the experiments. In VISIRI when the number of processing nodes increases, the time elapsed for initial query distribution increases for a given set of queries. However, in GATHIKA, the time elapsed for initial query distribution does not show a large variance with the number of processing nodes increases. GATHIKA has low latency in compare to VISIRI for initial query distribution.

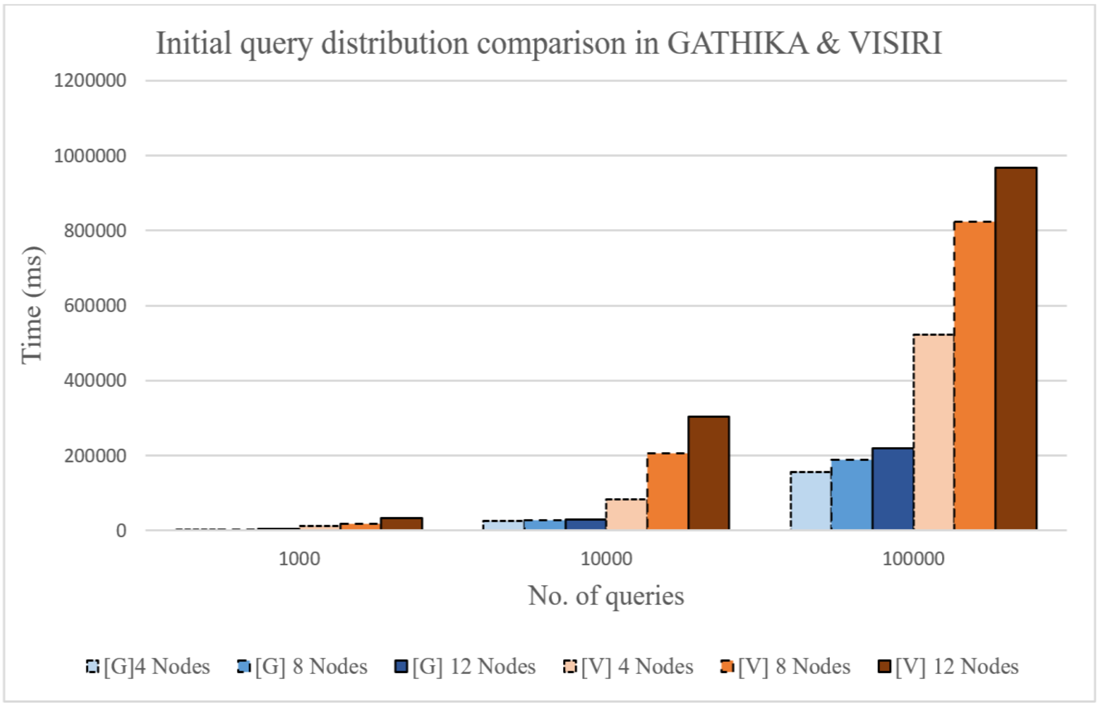


Figure 4-1 Initial query distribution comparison

4.2 Dynamically adding processing nodes to the system

The purpose of this experiment is to measure the overall system throughput and its variation when adding new processing nodes to the system. This experiment is further divided into two different test cases.

1. Measure system throughput when the dynamic query distribution is underway
2. Measure system throughput after the dynamic query distribution is triggered

The following dataset and configurations were used for the experiment.

Event source: mfg-uat-phoenix.mubashertrade.com

Stock Markets: DFM, ADX, KSE, TDWL, EGX, BSE, MSM

Time: 11.30 am – 4.00 pm

Initial query set in the leader node: 100

In every processing node, CEP engines do parallel processing of events. All the queries are distinct and none of the queries is duplicated in any of the processing nodes in the system. Therefore, the number of events processed by a single processing node is equal to the multiplication of number of deployed queries into number of events processed by the CEP engine.

The setup consists of an event dispatcher and a set of processing nodes. Only the event source is outside the system. The event source is a separate component that gets real-time stock market data and sends it to the Dispatcher. In addition, the setup is configured to run dynamic query distribution when adding new processing nodes. Following is the sequence in which the components start at setup.

1. Dispatcher
2. Leader processing node
3. Other processing nodes (one by one)

When the event source sends event data to the dispatcher, it routes data to the registered event processing nodes. The dispatcher continuously distributes events at a rate defined above. When processing nodes join the cluster, they receive the events from the dispatcher. Figure 4-2 shows the throughput comparison graph for the above two test cases.

Throughput of the system is measured using the following equation:

Throughput = Number of events processed per second

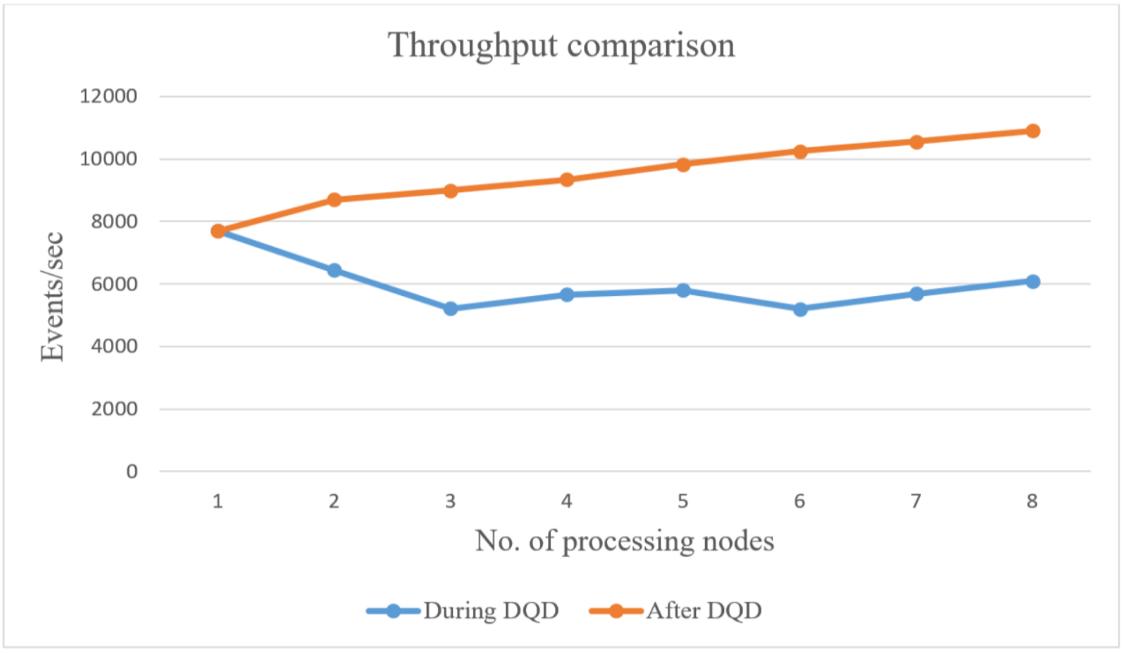


Figure 4-2 Throughput comparison with adding new nodes

Discussion

When the number of processing nodes increase and frequent dynamic query distributions are triggered, the overall system throughput degrades. This is discussed further in Section 4.2.1.

Once the system is steady after the dynamic query distribution, the overall system throughput increases. Having queries distributed among processing nodes is always better than keeping all the queries in a single processing node. Because if we keep all the queries in a single node, the number of resources required for the CEP engines is very high and the overall performance of the single node degrades. If the same set of queries is distributed among multiple processing nodes, then a single node requires fewer resources to run a CEP engine. Therefore, the performance of a single processing node increases. Finally, the collection of processing nodes gives a better throughput for the overall system.

4.2.1 Time elapsed for dynamic query distribution

The purpose of this experiment is to evaluate the impact of dynamic query distribution to the overall system. Figure 4-3 shows the time elapsed for query distribution for the mentioned experiment in Section 4.2. Here the X-axis shows the number of processing nodes added and the Y-axis shows the time elapsed for dynamic query distribution in milliseconds.

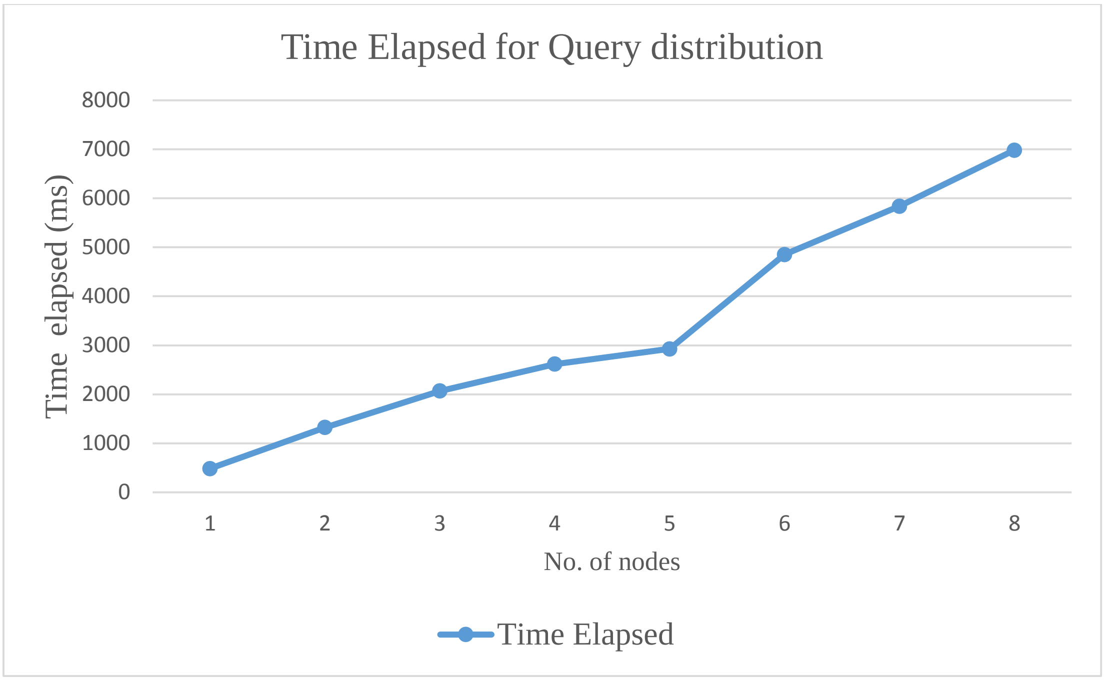


Figure 4-3 Time elapse for query distribution

Discussion

When the number of event processing nodes increase, the time taken for dynamic query distribution also increases. Due to this reason, the overall system throughout decreases during the dynamic query distribution time when the number of processing nodes increases. But after dynamic query distribution, system throughput is increased.

4.3 Event miss rate comparison

The purpose of this experiment is to measure the event miss rate when dynamically adding new queries to the system. The number of processing nodes is fixed in the cluster. Finally, the relation between dynamic query distribution and the number of missed events is evaluated. The following dataset and configurations were used for the experiment.

- System with four processing nodes

- System with eight processing nodes

Average event rate received at Dispatcher = 100 events/sec

Minimum number of queries to trigger dynamic query distribution = 10

Measure number of events missed during the time of dynamic query distribution.

The experiment starts with ‘0’ queries in the system; a set of queries is then added to measure the number of events missed. Since this experiment deals with a large number of queries, queries are randomly generated for this experiment. A random query generator validates that duplicate queries are not generated for this experiment.

Figure 4-4 shows the comparison between a four-node cluster and an eight-node cluster. Measurements were taken for a period of 120 seconds for each query set. The X-axis denotes the number of queries added during runtime and the Y-axis denotes the number of missed events.

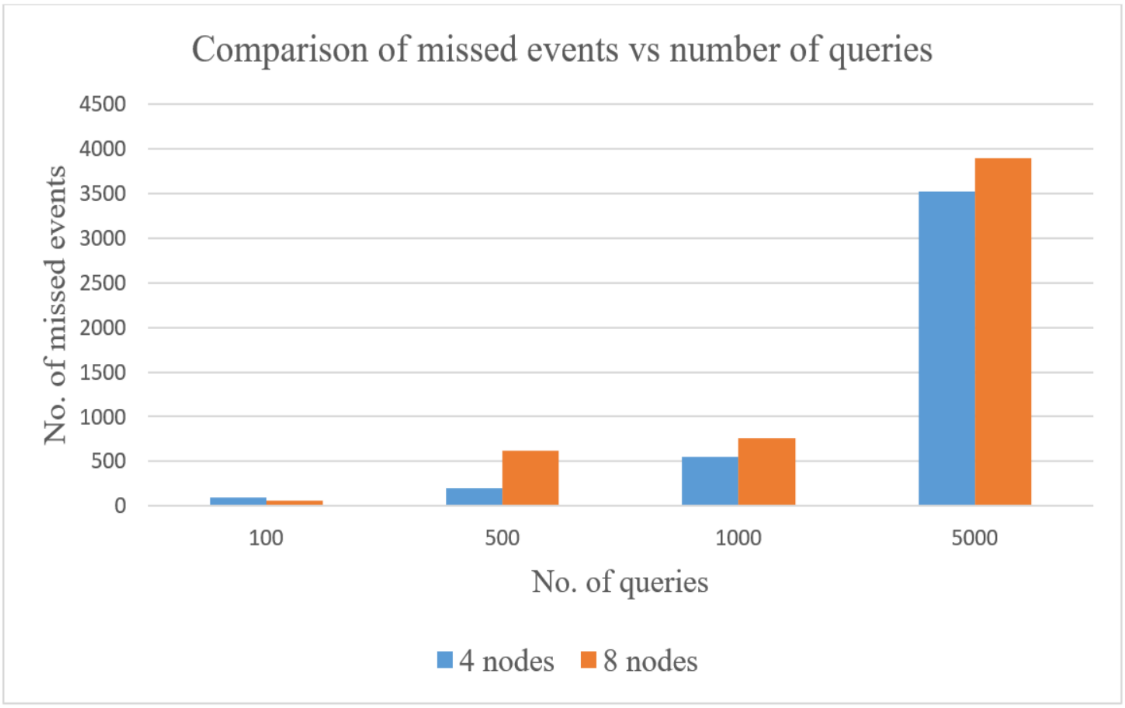


Figure 4-4 Events missed vs Number of queries

Discussion

Since the system is configured to trigger dynamic query distribution when adding more than ten queries, dynamic query distribution is triggered at all the test cases.

With the increase in the number of queries, the time taken for dynamic query distribution also increases. Because of that, the number of events missed also increases. Therefore, there is positive correlation between the number of queries and the number of missed events.

When comparing the system having four event processing nodes with the system having eight event processing nodes, it is clear that the time taken for dynamic query distribution increases when the number of queries increases. Therefore, we can conclude that when the number of event processing nodes is high, and the number of queries added is of a higher value, then the number of events that may be lost during the time of dynamic query distribution is high.

According to the measurements taken, the difference between the event miss count in a four-node system and an eight-node system is not high. Because of this reason, this will not affect the overall throughput of the eight-node system. Further details of this will be discussed in the next Section.

If frequent dynamic query distribution is triggered in the system, it loses events and degrades the overall system throughput.

4.4 Throughput comparison when adding new queries

The purpose of the experiment is to measure the throughput of the system when adding new queries during runtime. Same setup mentioned in Section 4.3 is used.

An event dump consisting of data records taken from the price feed received through the Mubasher Live price feed is used. Average Event arrival rate at the dispatcher is 500 events/second.

Throughput is measured during dynamic query distribution time and after dynamic query distribution.

4.4.1 Throughput during dynamic query distribution

The purpose of this experiment is to measure the relation between dynamic query distribution and system throughput. Figure 4-5 shows the graph created from the data collected from the baseline system, the system with four processing nodes and the system with eight processing nodes during the dynamic query distribution period. In the baseline system, dynamic query distribution is not triggered while in the other two systems dynamic query distribution is triggered when adding more than ten queries.

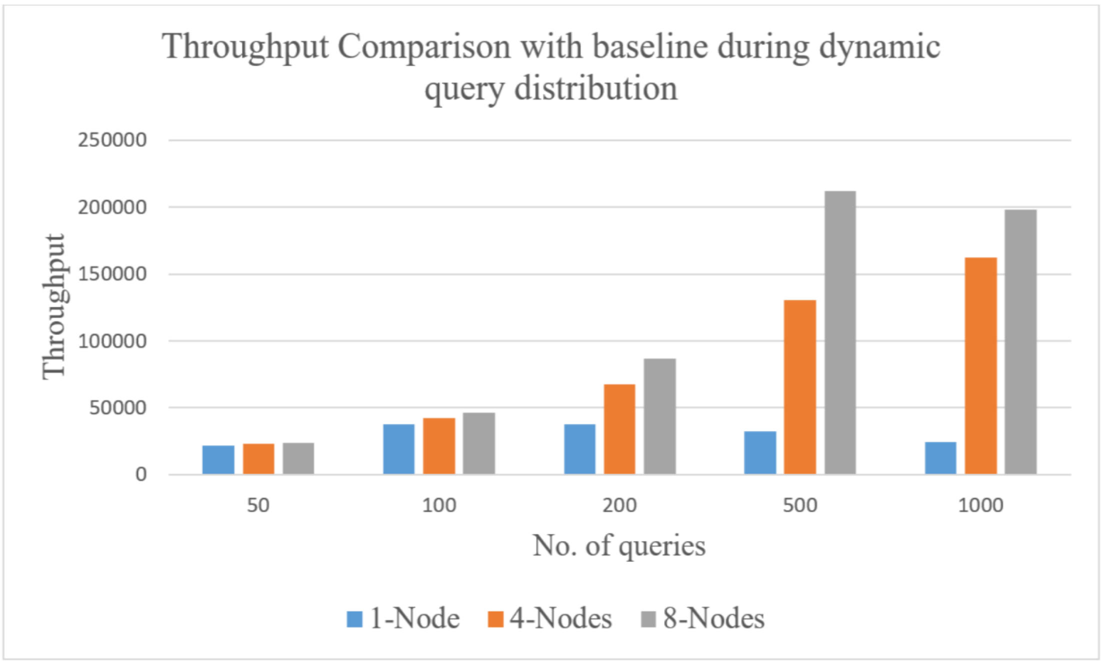


Figure 4-5 Throughput comparison with baseline during dynamic query distribution

Discussion

In the baseline system, the number of responses generated increases until the number of deployed queries is 100. When the number of deployed queries exceeds 100, system throughput reduces drastically. This is because for each deployed query, there is a separate CEP engine that starts to handle the events. When the number of running CEP engines increases, it causes a performance reduction of the node. Also, the number of events processed by the individual CEP engines goes down.

In the four-node system, the number of responses generated gradually increases. Although dynamic query distribution is triggered during the query addition period, it does not affect the performance of the overall system. The main reason is that the time taken for dynamic query distribution in a four-node system is of a low value.

In the eight-node system, the number of events processed increases until the deployed queries are 500. After adding a 500 query bundle at runtime, system throughput decreased during the measured period. The main reason for this is the time taken for dynamic query distribution in an eight-node system is a large portion of the measured period.

4.4.2 Throughput after dynamic query distribution

The purpose of this experiment is to measure the system throughput after dynamic query distribution triggers. Figure 4-6 shows the graph created from the data collected for the baseline system, the system with four processing nodes and the system with eight processing nodes, after the dynamic query distribution period.

Discussion

With a smaller number of queries, the difference between system throughputs is not high. When the number of event processing nodes is higher, the overall system throughput gets higher. This is seen from the number of responses in an eight-node system and a single-node system.

When the number of deployed queries increases, throughput decreases in the baseline, while the system throughput increases in the other two systems.

In the single node system, the number of responses generated degrades after the number of deployed queries exceeds 100. This can be identified as a threshold in a single-node system.

The throughput of the four-node system has overall throughput higher than a single-node system. However, the responses generated in the four-node system do not rapidly increased.

The throughput of the eight-node system increases at a rapid growth after increasing the number of queries in the system.

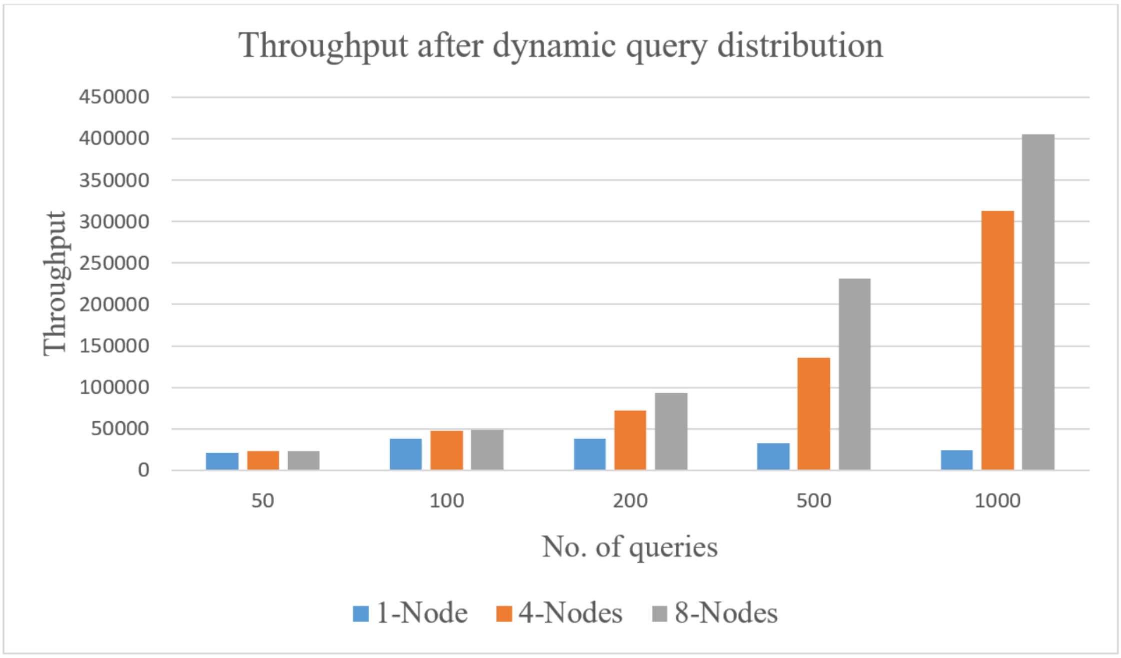


Figure 4-6 Throughput comparison of dynamic query distribution

4.5 Throughput comparison with event burst

The purpose of the experiment is to measure the system throughput by increasing the event arrival rate to the processing nodes. The number of processing nodes was fixed in the system. Queries are added to the system during runtime. Three test cases are mainly associated with this experiment.

For this experiment, an event dump was used to control the event rate. An event dump consists of data records taken from the price feed received through the Mubasher Live price feed. The dataset used for the experiment is discussed below.

Three systems were defined for this experiment.

- Base line system with a single processing node.
- System with four processing nodes
- System with eight processing nodes

First experiment dispatcher received events at rate =1000 events/sec

Second experiment dispatcher received events at rate =600 events/sec

4.5.1 Throughput in baseline system

Figure 4-7 shows the baseline system throughput with and without event burst.

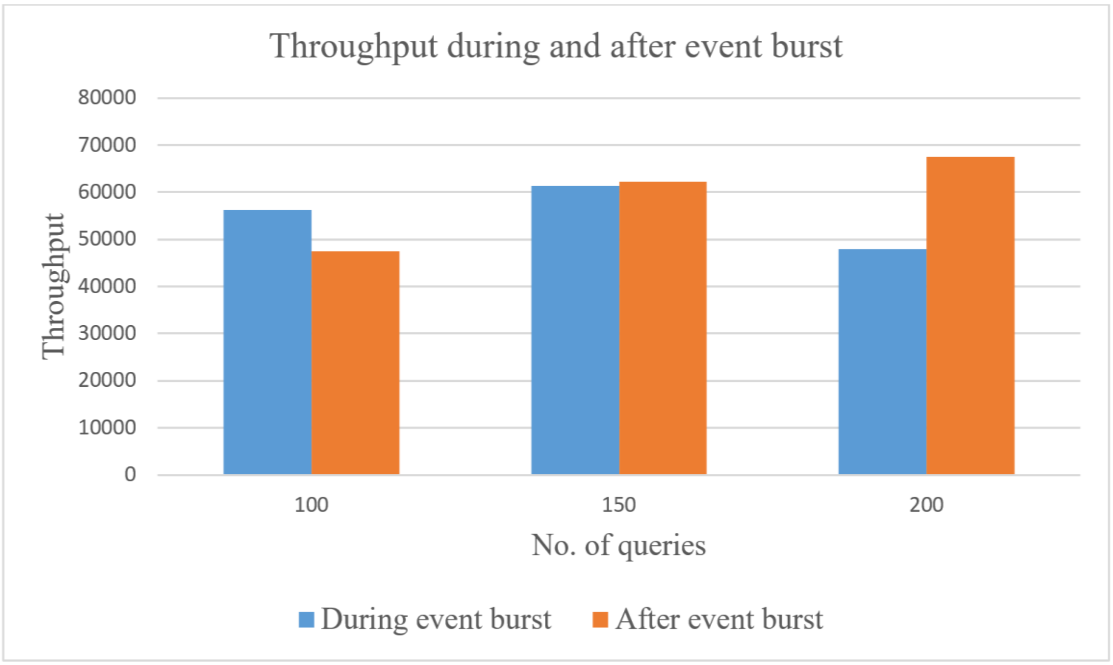


Figure 4-7 Baseline system throughput comparison during and after event burst

Discussion

In the baseline system, when the number of deployed queries increases, throughput is reduced. This is a known experimental factor from all the experiments conducted. This experiment especially focused on how the baseline system behaves when an event burst comes.

Initially, the event rate was set to 600 events per second and suddenly increased to 1000 events per second. At this point, the baseline system was overloaded with events and the CEP engines were busy. System throughput is measured for 100, 150 and 200 deployed queries. After the number of queries is increased to 150, baseline throughput drops during the event burst.

4.5.2 Throughput with four processing nodes

Figure 4-8 shows the four nodes system throughput with and without event burst.

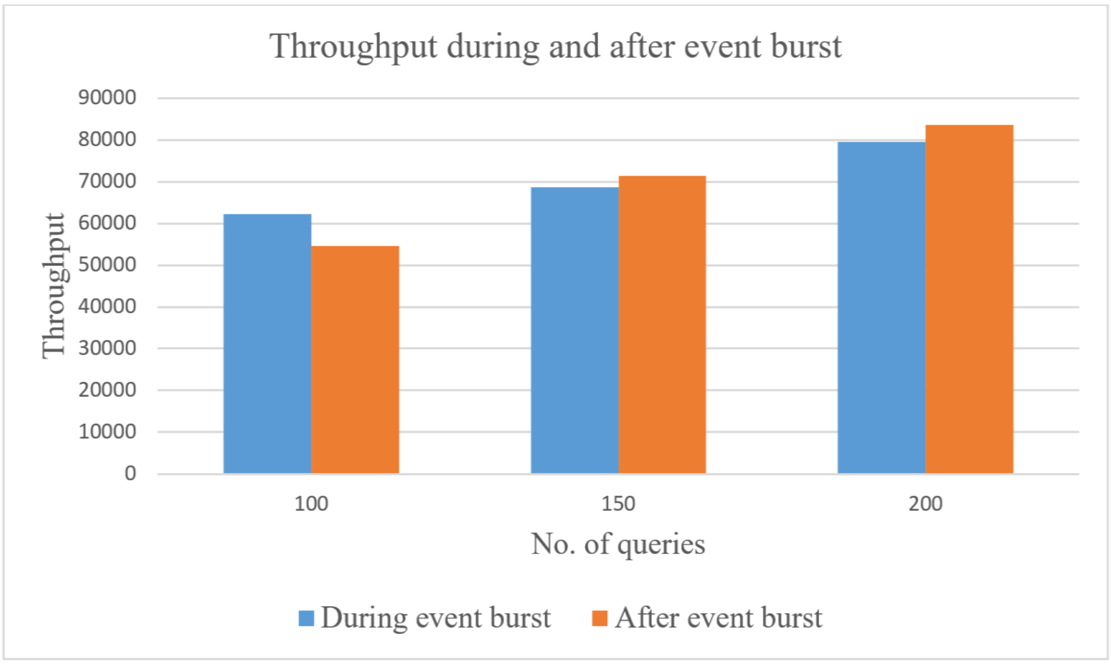


Figure 4-8 Throughput comparison during and after event burst in 4 nodes system

Discussion

The throughput of the system with four processing nodes increases when increasing the number of deployed queries. When the number of queries is 100, a sudden event burst increases the throughput. However, after the number of queries increases to 150 and beyond, event burst causes slight reduction in the throughput. This is because the performance of every individual processing node was reduced with the occurrence of event bursts and that affects the throughput of the overall system.

4.5.3 Throughput with eight processing nodes

Figure 4-9 shows the eight nodes system throughput with and without an event burst.

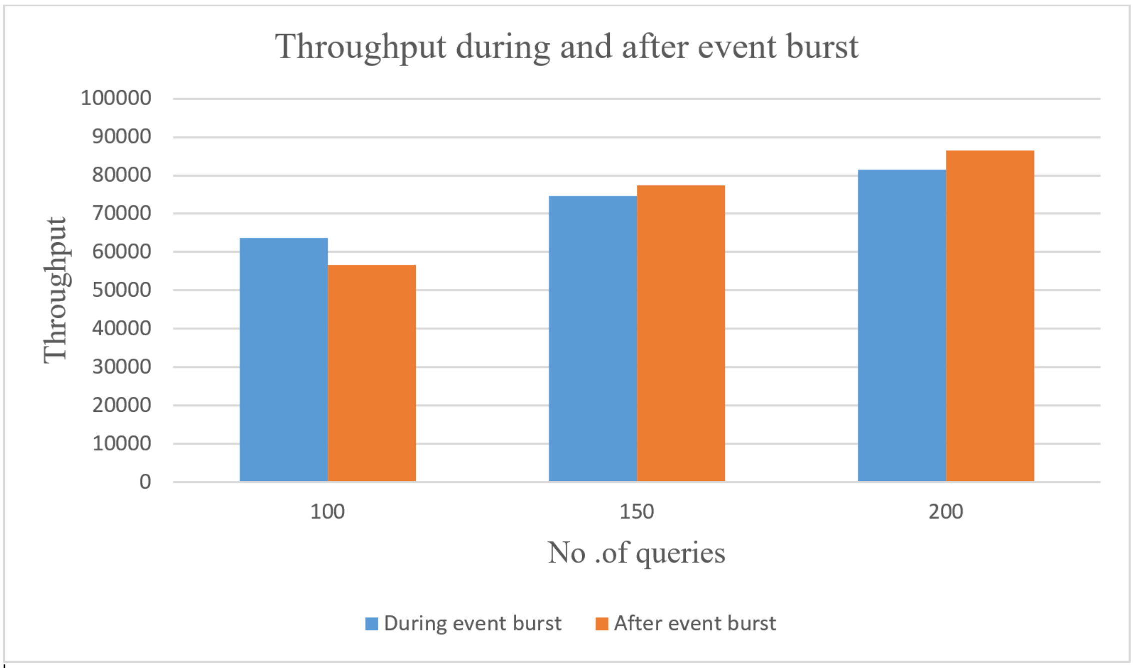


Figure 4-9 Throughput comparison during and after event burst in 8 nodes system

Discussion

Similar observation as in Section 4.5.2 is recorded. As discussed earlier, an event burst reduces the performance of the individual processing nodes. This affects the overall system throughput.

Figure 4-10 shows the throughput comparison of three systems during event bursts that happen with dynamic query distribution. In the baseline system, with the occurrence of event bursts, the overall system throughput degrades. However, in the four-node system, throughput does not degrade with the event burst. In the system using eight nodes, the rate of throughput increase got lower with the increase in the number of deployed queries. This is because the time taken for dynamic query distribution in an eight-node system is high compared to the four-node system during the measured time.

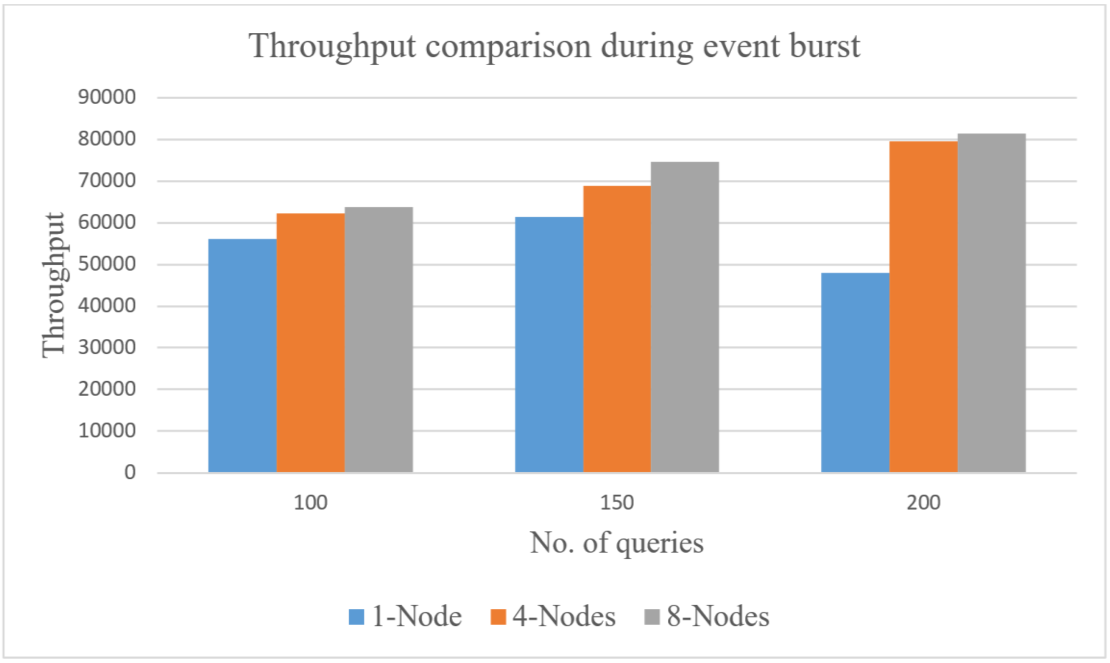


Figure 4-10 Throughput comparison of three systems during event burst

When comparing the baseline system with the other two systems, it is noted that when the number of processing nodes increases, the overall system throughput also increases.

However, after dynamic query distribution, throughput of both the eight-node system and the four-node system increases. In the baseline system, throughput is reduced with the increase of the number of queries.

5 DISCUSSION

5.1 Summary of experiments

As mentioned in the experiments section in Chapter 4, frequent dynamic query distributions negatively affect the overall system performance. Although dynamic query distribution has a negative impact on throughput during query distribution time, after that, system throughput increases. The requirement of having dynamic query distribution is to increase the overall system performance by distributing queries. Therefore, the major objective of this research is achieved.

5.2 Remarks of the experiments

System performance is proportional to the number of event processing nodes

For a smaller number of queries (i.e., less than 50, according to the baseline), it is not worth to use a clustered system. This is because there is no big difference in throughput achieved from using the baseline system and the four-node system. Another factor is that the cost of using four nodes exceeds the benefit of throughput that can be achieved from a four-node system.

If the system has a higher number of deployed queries (i.e., greater than 100 according to the baseline), it is worth to use a cluster of event processing nodes. Because of this, the overall system throughput increases. The benefit of using a cluster exceeds the cost of using four processing nodes.

Dynamic query distribution needs to be executed when the number of queries increase or when the number of resources available in the processing node decreases during runtime.

Frequent dynamic query distributions degrade the overall system performance

When the number of processing nodes in the cluster is large and the number of queries to distribute is large, then the time required for dynamic query distribution is high.

Dynamic query distribution not only depends on the number of queries to be deployed, it also depends on the available processing nodes. Therefore, the factors are not proportional to each other.

Frequent dynamic query distributions reduce the system throughput. Thus, it is important to have control over when to trigger dynamic query distribution. The experiment has few parameters to control the dynamic query distribution in the system.

- Time elapsed after last successful dynamic query distribution is an important factor
- The number of new queries to trigger dynamic query distribution

In the GATHIKA system, queries can be added to any of the nodes. However, in the testing environment, the leader node is used to add queries. Then, it is the leader that does the query distribution. If the number of added queries is less than the threshold value defined, there is no need to execute dynamic query distribution. This is because the cost of dynamic query distribution is a higher value. Therefore, in this kind of situation, the leader node keeps the new queries.

5.3 Evaluation of algorithms

Complexity of the Dynamic query distribution algorithm

As mentioned in Section 4.2.1 time elapsed for dynamic query distribution increases with the available processing nodes in the system. According to the measurements taken and as shown in Figure 4-3, the time taken for dynamic query distribution is proportional to the number of processing nodes. Moreover, time taken for dynamic query distribution increases linearly as the number of processing nodes increase. Therefore, the complexity of the dynamic query distribution algorithm is O(n), where n denotes the number of processing nodes. In summary, we can say that this is an efficient algorithm to perform dynamic query distribution.

Complexity of the Initial query distribution algorithm

As mentioned in Section 4.1, the time taken for initial query distribution increases when the number of queries to distribute increases. However, when the number of processing nodes increases, and the number of queries to be distributed is a constant value, the time taken for initial query distribution does not proportionally increase. Table 5-1 contains the time elapsed for initial query distribution in GATHIKA.

As shown in Table 5-1, when the number of queries is constant and the number of processing nodes increases, difference between time elapsed is not a higher value. Therefore, we can say that the time requirement is the same regardless of the number of processing nodes. So the time complexity is O(1) + DELTA.

According to the statistics in Table 5-1, time taken for initial query distribution linearly increases with the number of queries. Therefore, the complexity of initial query distribution is O(n).

In summary, we can say that this is an efficient algorithm to perform initial query distribution in a CEP system.

Table 5-1 Time elapsed for initial query distribution in GATHIKA

No. Queries	4 Nodes	8 Nodes	12 Nodes
10	129	160	188
100	548	559	616
1000	3235	3364	3926
10000	25322	27103	29661
100000	155351	189351	219346

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

Dynamic query distribution in a CEP system is an NP-hard problem. There is no exact solution on how to do query distribution dynamically. In this specific area, it is difficult to find similar or related research. Only few research projects are available but they do not address the exact problem.

The goal of this research is not to provide an exact answer on how to do dynamic query distribution. Instead, the contribution to the research problem is identifying when to do dynamic query distribution, what queries to distribute dynamically, and how the dynamic query distribution is done. The answers to the above questions will be discussed later in this chapter. The major observations of the experiments are,

- System performance is proportional to the number of processing nodes
- System throughput increases after the dynamic query distribution
- Frequent dynamic query distributions degrade the overall system performance

Finally, let us answer the three major questions in this research.

When to distribute queries

According to the experiments below, three occurrences are identified

1. When adding new processing nodes to the system
2. When adding new queries to the system
3. When the CPU and memory threshold exceeds in a processing node

What queries to distribute

In the first occurrence, existing queries in an overloaded node in the system need to be redistributed by executing the dynamic query distribution.

In the second occurrence, when there are new queries, they need to be checked and it needs to be identified whether those need to be distributed or kept in the processing node. A minimum number of queries are defined in GATHIKA to control this occurrence. If the query count exceeds the minimum value, dynamic query distribution is triggered and the queries are distributed.

In the third occurrence, the system checks and obtains transferable queries from the processing node and distributes those to other processing nodes using dynamic query distribution.

How to distribute queries

The answer is by using a dynamic query distribution algorithm. However, it is not a good idea to execute frequent dynamic query distributions. This is because it degrades overall system performance.

6.2 Future work

The GATHIKA project mainly targets events received from the stock market domain. It is better to have benchmark statistics for dynamic query distribution in different domains.

As mentioned in Section 3.4, transferable queries are selected based on the middle 10% of the queries. As future work, we can move beyond the middle 10% and check the impact of transferring those queries.

One of the problems in Hazelcast [28] (caching framework of the system) is that it takes a considerable amount of time to stabilize the system after a node failure. Therefore, it is good to benchmark this solution with different caching frameworks.

REFERENCES

[1] H. Randika, H. Martin, D. Sampath, D. Metihakwala, K. Sarveswaren and M. Wijekoon, "Scalable fault tolerant architecture for complex event processing systems", In *Proceedings of the International Conference on Advances in ICT for Emerging Regions (ICTer)*, pp. 86-96, 2010.

[2] G. Cugola and A. Margara, "Deployment strategies for distributed complex event processing", *Computing*, vol. 95, no. 2, pp. 129-156, 2012.

[3] Kazuhiko Isoyama, Yuji Kobayashi, Tadashi Sato, Koji Kida, Makiko Yoshida, and Hiroki Tagato. A scalable complex event processing system and evaluations of its performance. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pp. 123-126. ACM, 2012.

[4] Yongluan Zhou, Karl Aberer, and Kian-Lee Tan. Toward massive query optimization in large-scale distributed stream systems. In *Proceedings of the 9th ACM/I-FIP/USENIX International Conference on Middleware*, pages 326-345. Springer-Verlag New York, Inc., 2008.

[5] S. Perera, "How to scale Complex Event Processing (CEP) Systems?" [Online]. Available at: http://srinathsview.blogspot.com/2012/05/how-to-scale-complex-event-processing.html. [Accessed: 19- Apr- 2016]

[6] Cs.bu.edu. (2016). 4.2Complexity of NP Problems. [online] Available at: http://www.cs.bu.edu/fac/lnd/toc/z/node18.html [Accessed 24 Apr. 2016].

[7] S. Jayasekara, S. Kannangara, T. Dahanayakage, I. Ranawaka, S. Perera and V. Nanayakkara, "Wihidum: Distributed complex event processing", *Journal of Parallel and Distributed Computing*, vol. 79-80, pp. 42-51, 2015.

[8] M. Kumarasinghe, G. Tharanga, L. Weerasinghe, U. Wickramarathna and S. Ranathunga, "VISIRI - Distributed Complex Event Processing System for Handling Large Number of Queries", *Lecture Notes in Computer Science*, pp. 230-245, 2015.

[9] T. Heinze, Y. Ji, Y. Pan, F. J. Grueneberger, Z. Jerzak, and C. Fetzer, "Elastic complex event processing under varying query load" in *First International Workshop on Big Dynamic Distributed Data (BD3)*, pp. 25-31, 2013.

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

[11] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S.

B. Zdonik, "The Design of the Borealis Stream Processing Engine", *CIDR*, pp. 277-289, 2005.

[12] “Stock Market", *Investopedia,*2017. [Online] Available at: https://www.investopedia.com/terms/s/stockmarket.asp. [Accessed 10 Oct 2017].

[13] C. Li and R. Berry, "CEPBen: A Benchmark for Complex Event Processing Systems", *Performance Characterization and Benchmarking*, pp. 125-142, 2014.

[14] M. Eckert and F. Bry. Complex event processing (cep*). Informatik Spektrum*, 32(2): pp. 163–167, 2009.

[15] David C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, pp. 3-10, 2001.

[16] D. Luckham, *Event processing for business*. Hoboken, N.J.: John Wiley & Sons, 2012, pp. 101-134.

[17] Sase.cs.umass.edu. (2016). SASE - SASE Home. [online] Available at: http://sase.cs.umass.edu/ [Accessed 29 Apr. 2016].

[18] EsperTech. (2016). Home page - EsperTech. [online] Available at: http://www.espertech.com [Accessed 29 Apr. 2016].

[19] L. Neumeyer, B. Robbins, A. Nair and A. Kesari, "S4: Distributed Stream Computing Platform", In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, pp. 170-177, 2010.

[20] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments,* pp. 43-50, 2011.

[21] Demers, A. J.; Gehrke, J.; Panda, B.; Riedewald, M.; Sharma, V. & White, W. M. (2007), Cayuga: A General Purpose Event Monitoring System., *in* 'CIDR' , www.cidrdb.org, , pp. 412-422 .

[22] Demers, J. Gehrke, M. Hong, M. Riedewald and W. White, "Towards Expressive Publish/Subscribe Systems", *Lecture Notes in Computer Science*, pp. 627-644, 2006.

[23] M. Ali, "The Study On Load Balancing Strategies In Distributed Computing System", In *International Journal of Computer Science & Engineering Survey*, vol. 3, no. 2, pp. 19-30, 2012.

[24] M. Mendes, P. Bizarro and P. Marques, "A Performance Study of Event Processing Systems", *Lecture Notes in Computer Science*, pp. 221-236, 2009.

[25] T. Heinze, Z. Jerzak, G. Hackenbroich, C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems", In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pp. 18-22, 2014.

[26] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. 1996. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, Dorit S. Hochbaum (Ed.). PWS Publishing Co., Boston, MA, USA 46-93.

[27] N. Schultz-Møller, M. Migliavacca and P. Pietzuch, "Distributed complex event processing with query rewriting", *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems - DEBS '09*, pp. 4-12, 2009.

[28] Hazelcast.com. (2016). In-Memory Data Grid – Hazelcast IMDG. [online] Available at: https://hazelcast.com/use-cases/imdg/ [Accessed 3 Aug. 2016].