SCATTER-GATHER BASED APPROACH IN SCALING COMPLEX EVENT PROCESSING SYSTEMS FOR STATEFUL OPERATORS

Sriskandarajah Suhothayan

(168268V)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

February 2019

SCATTER-GATHER BASED APPROACH IN SCALING COMPLEX EVENT PROCESSING SYSTEMS FOR STATEFUL OPERATORS

Sriskandarajah Suhothayan

(168268V)

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

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

February 2019

DECLARATION

I declare that this is my own work and this MSc project report does not incorporate without acknowledgment any material previously submitted for 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 acknowledgment is made in the text.

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

Signature: .....
Date: .....
Name: Sriskandarajah Suhothayan

We certify that the declaration above by the candidate is true to the best of our knowledge and that this report is acceptable for evaluation for the CS6997 MSc Research Project qualifying evaluation.

Supervisors

.....	.....
Dr. H. M. N. Dilum Bandara	Dr. Srinath Perera
.....	.....
Date	Date

ABSTRACT

With the introduction of Internet of Things (IoT), scalable Complex Event Processing (CEP) and stream processing on memory, CPU, and bandwidth constraint infrastructure have become essential. While several related work focuses on replication of CEP engines to enhance scalability, they do not provide expected performance while scaling stateful queries for event streams that do not have pre-defined partitions. Most of the CEP systems provide scalability for stateless queries or for the stateful queries where the event streams can be partitioned based on one or more event attributes. These systems can only scale up to the pre-defined number of partitions, limiting the number of events they can process. Meanwhile, some CEP systems do not support cloud-native and microservices features such as startup time in milliseconds.

In this research, we address the scalability of CEP systems for stateful operators such as windows, joins, and pattern by scaling data processing nodes and connecting them as a directed acyclic graph. This enabled us to scale the processing and working memory using the scatter and gather based approach. We tested the proposed technique by implementing it using a set of Siddhi CEP engines running on Docker containers managed by Kubernetes container orchestration system. The tests were carried out for a fixed data rate, on uniform capacity nodes, to understand the processing capacity of the deployment. As we scale the nodes, for all cases, the proposed system was able to scale almost linearly while producing zero errors for patterns, 0.1% for windows, and 6.6% for joins, respectively. By reordering events the error rate of window and join queries was reduced to 0.03% and 1% while introducing 54ms and 260ms of delays, respectively.

ACKNOWLEDGMENT

I would like to take this opportunity to express my deep sense of gratitude and a profound feeling of admiration to my project supervisors. Many thanks go to all those who helped me in this work. My special thanks to the University of Moratuwa for giving an opportunity to carry out this research project.

I would like to gratefully acknowledge Dr. Dilum Bandara, the internal project supervisor, for his continuous guidance and support throughout the whole duration of the project, under whose supervision that I gained a clear concept of what I should do. I would also like to extend my heartfelt gratitude to Dr. Srinath Perera, the external supervisor of the project, for sharing the experiences and expertise with the project matters. Last but not least, I thank all those who like to remain anonymous although the help they provided to me was valuable.

Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ATM	Automated teller Machine
CEP	Complex Event Processor
CPU	Central processing unit
DBMS	Database Management System
ESB	Enterprise Service Bus
GC	Garbage Collection
IoT	Internet of Things
JMS	Java Messaging Service
NFA	Non-deterministic Finite Automata
RDD	Resilient Distributed Dataset
TCP	Transmission Control Protocol
TPS	Transactions Per Second
XA	eXtended Architecture

1. INTRODUCTION

1.1 Background

Processing data on the fly with Complex Event Processing (CEP) and stream processing systems are gaining popularity due to the introduction of the Internet of Things (IoT) [1], [2]. With increased network connectivity and lower costs, more sensors are being deployed to build smart connected systems. These systems often generate a massive volume of data in real time, where resource constraint systems now find it difficult to keep up with the expected performance. Surveillance through IoT devices, connected cars, monitoring critical distributed systems, monitoring financial transactions, digital control systems, and network security monitoring are some use cases which demand highly scalable real-time stream processing on resource-constrained environments.

CEP engine is a real-time, in-memory event processing system, which has the capability to receive events (from various sources via various transports), correlate them to identify meaningful insights on those events based on the user-defined queries, and notify them as alerts in many formats. In CEP engines, an *event* is a unit of data that contains a timestamp and set of attribute values according to a defined schema, and the sequence of events arriving on a particular type is called *stream*, on which users can perform complex analytical processing. This data in motion analytics can be provided to the system as query/rule that will be executed on each event as and when they arrive at the system. CEP engines support several common functions such as filtering events based on attributes, aggregation over sliding windows, joining multiple streams, pattern matching, and sequence detection.

1.2 Motivation

As processing data on distributed systems is gaining more attraction due to the increased availability of sensor networks and with the new trend of micro-services architecture [3], the focus is more on building distributed processing optimized to

utilize available resources in the ecosystem. In most of the real-world scenarios, a single CEP engine cannot process all the events with expected performance due to limited resources. Therefore, we need to scale the Complex Event Processing to overcome the following limitations [4]:

- Bandwidth bottlenecks in receiving and publishing events to CEP.
- Insufficient CPU cycles to handle multiple complex query operations.
- Memory limitations to store intermediate events of the queries.

To overcome some of these challenges multiple distributed deployment patterns are implemented by several CEP vendors including the following:

- Running multiple CEP nodes in a cluster fronted by Enterprise Service Bus (ESB) [5].
- Distributing different type of queries to different CEP nodes [6], [7].
- Distributing execution via Publish/Subscribe infrastructure [8], [9].
- Distributing events by partitioning each stream into isolated environments [10], [11].
- Distributing events as batches [12].

There are several hybrid implementations [11], [13], [14] that also tries to mix and match some of these techniques. Therefore, it is evident that scaling CEP systems into multiple nodes is effective in increasing availability and throughput while maintaining low response time by reduced bandwidth, CPU, and memory consumption of its single instance/node. While scalability have been discussed in the literature specifically for window and pattern matching by partitioning the events streams into non-overlapping partitions [15], [16], [17], [18], [19], not much work has been done on achieving scalability for non-partitionable streams which cannot be divided into partitions by any criteria such as an attribute key or range, and being able to process them in isolation, especially for CEP constructs like window, joins, and patterns. Therefore, there is still a need for building a fully-featured CEP engine that can be scaled dynamically, which will be useful in IoT, cloud, and microservices environments, that allow us to build

highly scalable event processing systems that can work both in device and cloud infrastructure.

1.3 Problem statement

The problem this research trying to address can be formulated as the following research question:

How scatter-gather based approach can be used for scaling Complex Event Processing systems having stateful operators?

such that it can be massively scaled by distributing memory, CPU, and bandwidth utilization over multiple nodes, without depending on user-defined partition (Group By) key. This become very useful in cloud and microservices environments such as Apache Mesos [20] and Kubernetes [21] where they provide on-demand scalability using low-resource instances.

1.4 Objectives

The objective of this research is to address the following on event streams that cannot be partitioned by a key:

- To build a system using scatter-gather based approach to scale stateful CEP operators without having any assumptions on the datasets.
- To develop a scalable solution focusing of following core CEP queries:
    - Large sliding time-based window queries.
    - Large sliding length-based window queries.
    - Pattern queries defined over large temporal period.
    - Achieving inter streams joins over various window sizes.
- To evaluate the scalability and accuracy of the proposed system using a fixed data rate workload.

1.5 Outline

Rest of the thesis is organized as follows. Chapter 2 presents the literature review. It covers areas related to CEP and scaling, as well as existing scaling approaches. Chapter 3 presents the proposed solution for scaling CEP operations such as windows, patterns, and joins. It also discusses the techniques that are used for scaling such as connecting distributed data processing nodes as a directed acyclic graph and processing streaming data using scatter and gather based approach. Details of how these techniques are implemented on top of Siddhi CEP engine are presented in Chapter 4. Chapter 5 presents the evaluation of the proposed approach using fixed data rate workload, on uniform capacity nodes, to understand the processing capacity of the deployment. Concluding remarks and suggestion for future works are presented in Chapter 6.

2. LITERATURE REVIEW

This section presents the related work on scaling Complex Event Processing (CEP) systems. Section 2.1 presents details about CEP systems. Common CEP functionalities that one could expect from CEP engines are discussed in Section 2.2. Section 2.3 characterizes single node CEP performance in terms of CPU, bandwidth, and memory utilization. Section 2.4 reviews distributed architectures for scaling CEP deployments. Finally, how CEP operations can be scaled on multiple CEP nodes is discussed in Section 2.5.

2.1 Complex Event Processing Systems

As the name implies, CEP engines (as seen in Fig. 2.1) are not only used for simple filtering where the events could be filtered by certain attribute value, but also used for more complex analysis such as time and length (number of events) based aggregations on sliding and tumbling (batch) windows, multi-stream joins, and for pattern matching to identify event occurrence order. As its primary goal is to provide real-time monitoring, CEP's main functional behavior includes being high performance in retrieving events, processing them to produce accurate results, and responding to the results quickly as possible in suitable message format and transport [22].

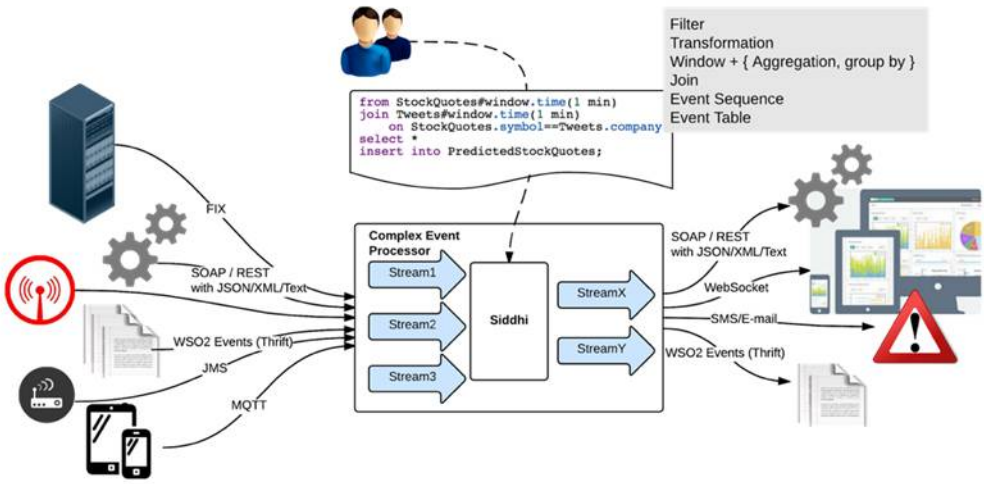

Fig. 2.1: Overview of CEP.

2.2 CEP functionalities

Following are some common CEP functionalities that all CEP systems supports and it is important that these functionalities are also available in distributed and scalable deployments. For the analysis of these functionality we are using Siddhi CEP Engine [22], [23]. The reason for choosing Siddhi is because it is lightweight high performing Java library such that it can be easily integrated as a core for a distributed CEP implementation, and it can run on multiple nodes with limited resource consumption while each node performing up to 100,000 events per second. Further it also provides easy adoption as it was also released under an open source Apache License v2 and provides and easy to use SQL-like query processing interface. Next, we discuss the common set of CEP functionalities.

2.2.1 Filtering events based on attributes

This provides the capability of removing uninteresting events in the stream and only outputting the events that match the filtering condition. For example, in Query 2.1 all events of the *TempStream* containing temperature events are filtered and only events having *temp* attribute value greater than 45 have been selected and from those events *roomNo* and *temp* attributes are sent to the stream *HighTempStream.*

```
from TempStream[temp > 45]
select roomNo, temp
insert into HighTempStream;
```

Query 2.1: Filter query.

2.2.2 Aggregation on sliding windows

The window is an important function provided by the CEP engine, where a predefined last few events based on time or length are considered and user defined aggregations operations can be executed on top of them. The window can be a sliding window where it shifts upon each event arrival and departure, such as last five minutes, last one hour, last 100 event windows. Aggregation operations such as *sum*(), *min*(), *max*(), and *avg*() can be performed effectively on windows as windows will provide a bound on the number of events we need to consider for the processing.

As seen in Query 2.2 on the stream *TempStream* per room average temperature is calculated for the events arrived in last 1 min. In Query 2.2 the results are inserted into the stream *AvgRoomTempStream*.

```
from TempStream#window.time(1 min)
select roomNo, avg(temp) as avgTemp
group by roomNo
insert all events into AvgRoomTempStream;
```

Query 2.2: Sliding time window query.

2.2.3 Joining multiple streams

Joining multiple streams based on given conditions is also a must to have functionality for CEP engine. In this case, CEP engines use windows to collect events and join the incoming events against them. Example in Query 2.3 demonstrates how CEP can be used to correlate *OrderStream* and *DeliveryStream* to detect time taken to deliver the order within 1 hour and notify the results via *DeliveryTimeStream*.

```
from DeliveryStream#window.length(1) as d join
     OrderStream#window.time(1 hour) as o
     where o.id == d.orderId
select d.time - o.time as timeToDeliver,
     o.id as orderId
insert into DeliveryTimeStream;
```

Query 2.3: Join query.

2.2.4 Pattern matching and sequence detection

Another most important aspect of the Complex Event Processor is its ability to detect events occurrences by identifying specific patterns. Non-deterministic finite automata are used by most of the CEP systems to accomplish this task, and they usually support regular expression related query syntax to achieve this.

Query 2.4 demonstrates how CEP can be used to detect potential credit card fraud, where when a thief steals a credit card, the thief tries whether it is working by using it for a low-value transaction at the grocery store and if it does work the thief might try to use that to for a huge purchase. Here in *CardStream*, if there is an event *a1* arrives having amount less than $100 and following that event (shown using '->'

symbol) if there in another event *b1* arrives which has amount greater than $10,000 within a one-day interval for the same ATM card, then CEP will insert the matching details to *PossibleFraudStream* for alerting.

```
from every a1 = CardStream[amount < 100]
    -> b1 = CardStream[amount > 10000 and
                            a1.cardNo == b1.cardNo]
    within 1 day
select a1.cardNo as cardNo,
     a1.cardHolderName as cardHolderName,
     b1.amount as lastPurchaseAmount,
     b1.location as location,
     b1.cardHolderMobile as cardHolderMobile
insert into PossibleFraudStream;
```

Query 2.4: Pattern matching query.

Filter, window and aggregation queries are at the heart of many continuous streaming analytics applications, CEP applications also support other complex queries involving multi-stream joins and pattern matching, and they support inserting the results back to other streams to chain multiple analytics.

2.3 Understanding characteristic of single node CEP

As a baseline we compare the scalable CEP characteristics against single node CEP for each of its query type, their results are presents in Table 2.2. The symbols used to analyze the CEP characteristics on this thesis are presented in Table 2.1.

Table 2.1 Symbols used to analyze CEP characteristics.

Symbol	Description
e	Number of events
g	Number of groups by keys
n	Number of nodes
s	Number of states
t	Time interval
λ	Rate of event arrival

Table 2.2 Baseline characteristics of a single node CEP engine.

Query type	Input Bandwidth	Output Bandwidth	Memory Consumption	CPU Utilization
Filter	$\Theta(\lambda)$	$\Omega(\lambda)$	0	$\Theta(\lambda)$
Sliding time window t with no group by	$\Theta(\lambda)$	$\Theta(\lambda)$	$O(\lambda t)$, contains all events.	$\Theta(\lambda)$, process each event twice one for arrival and one for expiry.
Sliding time window t with g group by keys	$\Theta(\lambda)$	$\Theta(\lambda)$	$O(\lambda t + g)$, contains all events and aggregates for each group by key.	$\Theta(\lambda)$, same as sliding time window without group by.
Sliding length window e with no group by	$\Theta(\lambda)$	$\Theta(\lambda)$	$O(e)$	$\Theta(\lambda)$
Sliding length window e with g group by keys	$\Theta(\lambda)$	$\Theta(\lambda)$	$O(e + g)$	$\Theta(\lambda)$
Pattern matching with s states. e.g., A → B → …	$\Theta(s\lambda) = \Theta(\lambda)$, assuming each stream associated with a state have input rate λ and s is constant.	$O(\lambda)$, output at last state	$< O((s-1)\lambda) = O(\lambda)$, as s is constant	$\Theta(s\lambda) = \Theta(\lambda)$, as s is constant
Joining 2 streams using windows and equijoin. Assuming all streams have input rate λ.	$\Theta(\lambda)$	$\Theta(\lambda)$	$O(e)$	$O(\lambda e)$

2.4 Distributed architectures for scaling CEP nodes

Even though single node systems are fast and have the ability to process all CEP functionalities they frequently encounter memory, CPU, and bandwidth bottlenecks forcing us to distribute CEP. This section focuses on possible architectures

for scaling CEP functionalities. Some of the distributed deployment architectures are following:

2.4.1 Running multiple CEP nodes in a cluster

This enables horizontal scalability by simply fronting the CEP cluster with a load balancer or an Enterprise Service Bus (ESB) [5], here each node in the cluster will contain exactly the same queries and all nodes will receive all events or in a round-robin manner. This mode is efficient for filtering kind of stateless queries but does not support stateful queries like windows and patterns as their states will be now distributed across multiple nodes. As a solution, sticky sessions can be used for some use cases but this will not work as the same stream can be joint it multiple ways with same attribute value and always sending a stream or an attribute value to a single node will not fix the problem. Alternatively, a central session store can be used to solve the problem but this will not be efficient as there will be a high number of concurrent updates to the central session store slowing the system drastically.

2.4.2 Distributing different type of queries to different CEP nodes

This approach focuses on distributing query network to multiple nodes as an acyclic graph helping to achieve vertical and horizontal scalability. As depicted in Fig. 2.2, this will allow us to place queries such that we can reduce the event rate by filtering incoming events as they go through the initial stages of the query topology and perform more Complex Event Processing at latter stages of the topology [7]. This can also further scale horizontally by having multiple filter nodes to handle the load appropriately.

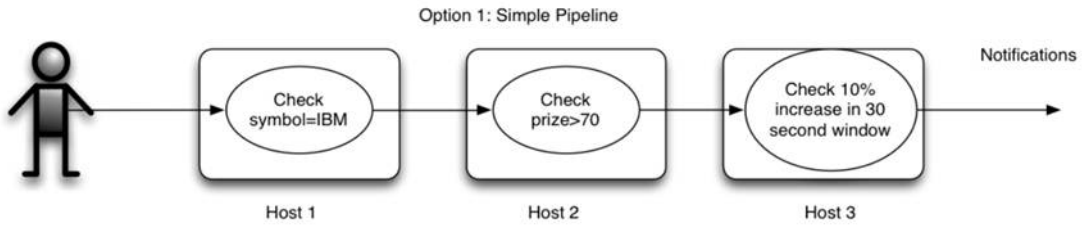

Fig. 2.2: Vertical scaling with multiple CEP nodes.

Most stream processing systems currently use the technique of distributing queries as an acyclic directed graph over connected nodes [24]. WSO2 Complex Event Processor [25], Medusa [26], Apache Storm [11], Apache Flink [14], and Apache Samza [13] are some systems who support such approaches. Apache Storm allows deploying multiple number of different type of nodes (Spouts and Bolts) and connect to each other as a directed graph using some well-defined patterns such as, Shuffle Grouping which shuffles the events among the receiving nodes, Field Grouping which does hash based partitioning based on the given event field, and Global Grouping where it sends all events to all the receiving nodes. There have also been attempts to do distributed processing using query rewriting [6] such that transparently distributing single query plan across multiple nodes distributing each query to one or more stream processing nodes. But most of the open source CEP systems [27], [28] like S4 [29] do not provide many capabilities in achieving automated distributed deployment and they rely on developer configuring the topology for distributed deployment.

The main drawback of these systems is fault tolerance, this is because if a node goes down or if a node is processing slowly the upstream nodes get to know about this via back pressure and they start collecting events in their buffers till the downstream system to recover. However, due to high event rate there is a high possibility the whole system to crash.

2.4.3 Distributing execution via Publish/Subscribe infrastructure

This allow CEP to achieve vertical and horizontal scalability while allowing it to work asynchronously and help tolerating event rate spikes. Each node sends events to the Publish/Subscribe infrastructure (supported by a message broker) and the interested nodes subscribe to the infrastructure to get relevant events based on the topic they are interested in [8], [9]. Java Messaging Service (JMS) is a popular approach that used for traditional CEP scaling use cases, where each processing node is manually configured with different JMS topics to achieve distributed processing. Currently with durable subscriptions they can now reliably communicate between multiple nodes and with XA Transaction (2-way commit) support of JMS they can also have exactly-once event processing. While this is suitable for a small distributed

deployment, it cannot scale vertically by chaining multiple processing nodes one after another using JMS topics as the latency of event processing increases drastically with each event going through the message broker.

If multiple nodes can also subscribe to the same topic then the events will reach each node in a random partitioning strategy, which is still not suitable for scaling stateful processing. To overcome this, as depicted in Fig. 2.3, Oracle CEP [30] engine uses JMS message selectors when subscribing to the topics that will filter only the events that the node interested in.

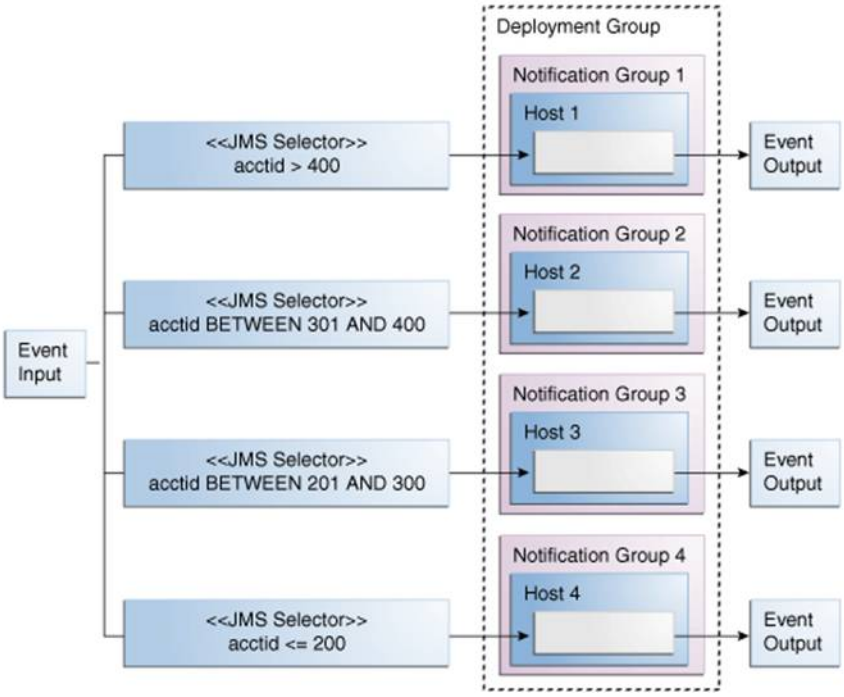

Fig. 2.3 Distributed deployment of Oracle CEP [30].

2.4.4 Distributing events by partitioning each stream

This allows horizontal and vertical scalability of the system by allowing it to run the same query in multiple nodes and by partitioning the stream by defined attribute and allowing each partition of the same stream to be processed in each node. Systems like Apache Storm [11] supports such processing.

For efficiently processing events via partitions, modern CEP systems are utilizing Apache Kafka [10], which is not a traditional message broker, but rather the

backend for such a broker. Kafka guarantee exactly-once processing and, as depicted in Fig. 2.4, it supports partitioning topics into multiple topic-partitions and assigning a sequence number for each of the events within a topic-partition. When writers write events, the data gets distributed among each partition based on a hash key, and the subscribers can either subscribe to the topic to consume events from all its topic-partitions or subscribe to one or more partitions to retrieve data. Through this it supports scalable executions of CEP nodes. Further as Kafka has the API to support reading historical (already processed) data using an offset within each of its topic-partitions, the same events can be read from a topic-partition multiple times like reading a file in the file system many times. The consuming system can keep track of these event offsets for each partition, and during failure recovery, it can decide from which offset it should read the messages again. Further with optimized disk write and read Kafka also have the ability to provide very high-performance persistence asynchronous data transfer compared to conventional JMS-based data transfer.

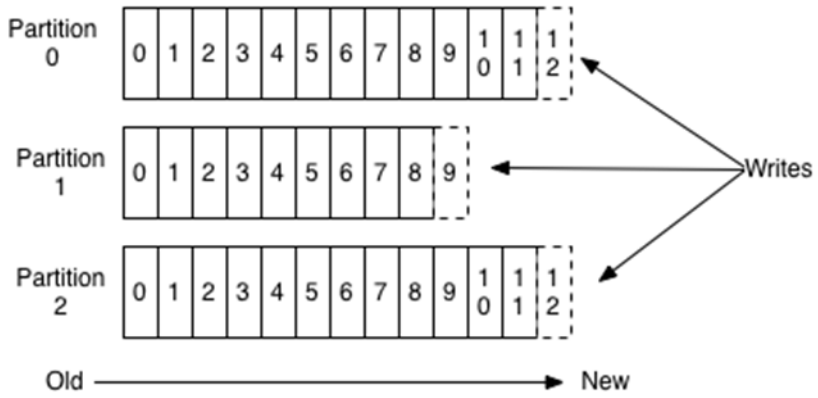

Fig. 2.4: Anatomy of Kafka topic.

Apache Samza [13] uses Kafka [10] to guarantee the messages are processed in the order they were written to a partition and that no messages are ever lost. Due to the use of Kafka in Samza, if one job goes slow and builds up a backlog of unprocessed messages, the rest of the system does not get affected. Though stream partition helps Samza to achieve high performance and helps to reduce latency, Samza can only process messages in the order they appear in a partition, and it does not guarantee message ordering across different input streams or partitions.

2.4.5 Distributing events as batches

Spark Streaming [12] treats streaming as a series of deterministic batch operations. Spark Streaming groups the stream into batches of a fixed duration (such as 1 second) and so each batch is processed one after another. Each batch in Spark represents a Resilient Distributed Dataset (RDD). The receiver receives data and stores it in Spark which then does transformations and performs the output operations. Spark Streaming guarantees ordered processing of RDDs within a stream. When using multithreading, each RDD will process in parallel and hence output event ordering is not guaranteed. Hence, multithreading is a tradeoff design Spark has made. Further, it does not provide any key-value access to its state data, and the only way to access the state is by iterating the whole dataset. The recommended minimum value of processing block interval is about 50ms [12], below which the task launching overheads may cause problems. Due to task launching overheads, event processing with less than 50ms latency may not be achievable with Apache Spark. Further Spark Streaming and Samza achieve end-to-end exactly-once semantics by using idempotent and transactional updates.

This section presented the current systems that support distributed processing deployment, where they either support distributing CEP queries if the stream can be partitioned or else provide an infrastructure for us to implement distributed processing logic that we want. Table 2.3 compare currently available such distributed architectures for scaling CEP nodes.

2.5 Distributing CEP operations over multiple CEP nodes

This section discusses the techniques used to scale stateless and stateful CEP operations over multiple CEP nodes.

Table 2.3: Comparison of distributed architectures for scaling CEP nodes.

Solution	Technique	Advantages	Disadvantages
Running multiple CEP nodes in a cluster.	Fronting nodes with Load Balancer.	Easy to implement and scale stateless queries.	Cannot scale stateful queries.
Distributing different type of queries to different CEP nodes.	Configuring nodes as an acyclic graph and using transient network communication.	Support horizontal and vertical scaling by reducing load on latter nodes.	Cannot horizontally scale stateful queries. Uses back pressure to control the flow, system is brittle.
Distributing execution via Publish/Subscribe infrastructure.	Configuring nodes as an acyclic graph and using pub/sub communications.	Support horizontal and vertical scaling by reducing load on latter nodes, also no need for back pressure as it can buffer events when nodes are down.	Cannot horizontally scale stateful queries.
Distributing events based by partitioning each stream.	Configuring nodes as an acyclic graph and also partition the streams based on partitioning key.	If the stateful query can be partitioned then it can scale horizontally and vertically.	Cannot scale stateful queries if the stream cannot be partition based on a partitioning key.
Distributing events as batches.	Process events as batches.	Support scalable processing with MapReduce.	High latency due to batching behavior.

2.5.1 Scalability of stateless operators

When distributing CEP, stateless queries like filters can only be a bottleneck of the system when their CPU or bandwidth is over utilized. Because as they are stateless they will not be constrained by memory. To overcome CPU and bandwidth limitations, as illustrated in Fig. 2.5, they can be scaled over multiple nodes both horizontally and as pipeline [31]. Here source can send events to the horizontally scaled nodes in a round-robin manner or using a hash function such that CPU load and bandwidth will be distributed equally across multiple nodes.

With the query horizontally scaled to n nodes, input and output bandwidth, and CPU utilization of each node will be (baseline values)/n. At the meantime when they are scaled as pipeline their CPU utilization will be reduced as $\Theta(\lambda/n)$.

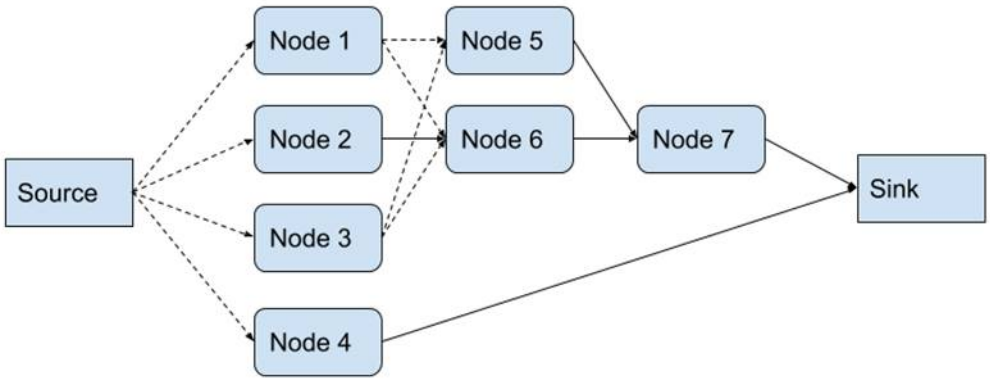

Fig. 2.5: Scaling stateless CEP queries.

2.5.2 Scalability of stateful operators

Stateful queries such as windows, joins, and patterns are the most complex in terms of distributing as they need to maintain a common state among the nodes to produce consistent results. There have been several attempts in achieving this and among them, the most efficient scaling approach is partitioning the queries [31], and as depicted in Fig. 2.6 the event streams will be partitioned based on event attributes that will allow the data to be distributed across multiple nodes while achieving approximately equal CPU, memory and bandwidth load on each node, provided that each partition have the approximately same number of events and the deployed queries match approximately equal number of events from each partition.

As depicted in Fig. 2.6 each source should send events falling into the same partition to the same node. Because partitions are usually calculated with hash functions, when data is skewed there is a high probability that lots of events fall into the same partition and load only some nodes in the system. In such cases, CEP systems become unable to handle the load or reach their maximum performance limit.

When the CEP query is horizontally scaled using equal partitions to n nodes, their input and output bandwidth, memory consumption, and CPU utilization of each node will be (baseline values)/n.

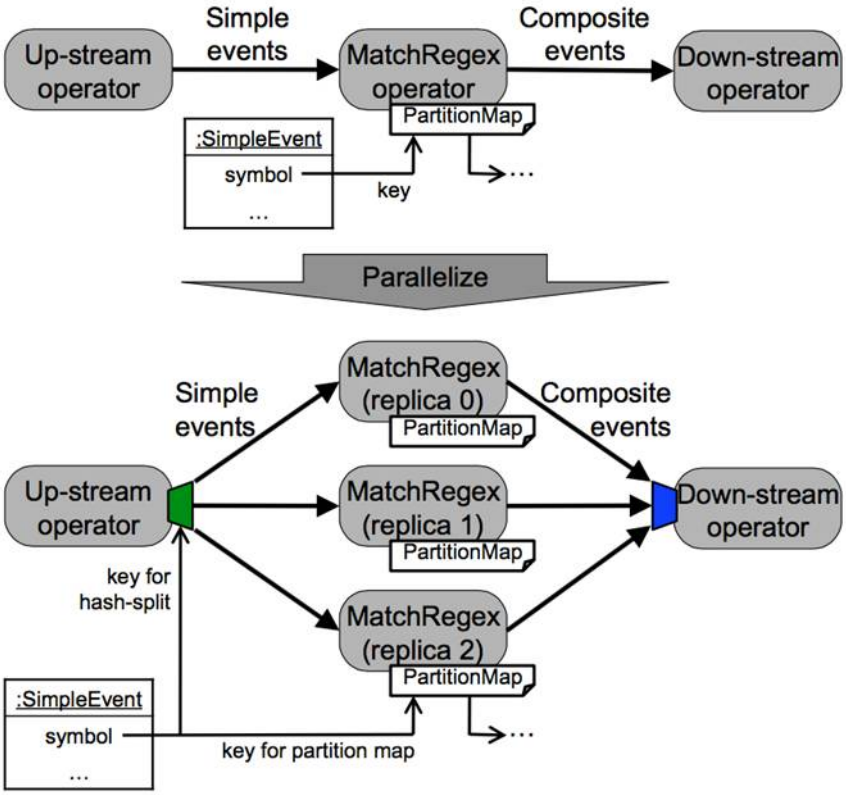

Fig. 2.6: Parallelizing operator graph using partitions [31].

Partitioning itself is not always effective, but when this is combined with pipelining (vertical scaling) as seen in Fig. 2.7, it helps to process events in a more scalable manner, because this approach filters out more events in the initial nodes and only concentrate on storing the states on relevant events in latter nodes for further processing [31].

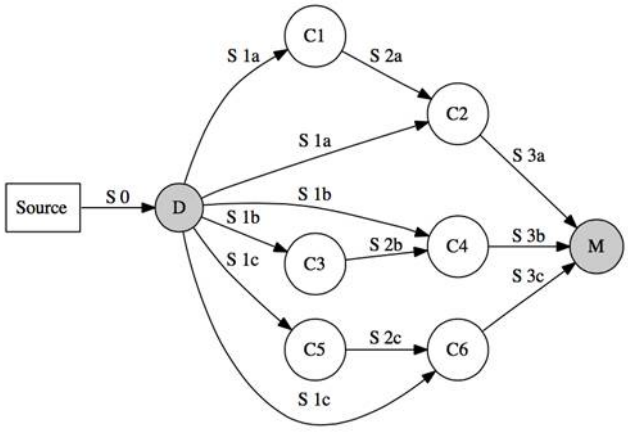

Fig. 2.7: Combining partitioning and pipelining [31].

Brenna et al. [31] illustrates two techniques for scaling pattern queries which can be expressed as Non-Deterministic Finite Automata (NFA) in CEP engines. Here they define scaling by partitioning the event streams and also by scaling by splitting NFA into smaller sub NFA's and distributing them in a pipeline manner. In the prior CPU, memory and bandwidth can be scaled based on number of partitions we have defined, here if we partition the query into n partitions and distribute them over to n nodes the CPU, memory, and bandwidth utilization will become (baseline values)/n. But in the latter we can only achieve CPU and memory scaling as (baseline values)/n where the bandwidth can become a bottleneck, where first node in the sequence will have $O(\lambda)$ bandwidth utilization and all other nodes in the sequence will have bandwidth of $O(\lambda)$. This is because in Brenna at el. deployment, not only all nodes receive all the events but the latter nodes will additionally receive matched events from their preceding nodes.

Pandey et al. [19] provides a MapReduce based streaming aggregation technique to scaling aggregations without depending on a partition key.

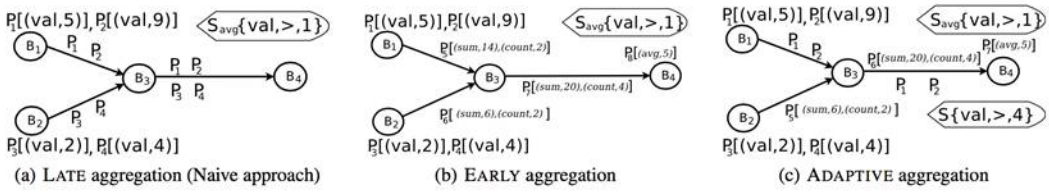

Fig. 2.8: Optimization on streaming aggregation [19].

As depicted in Fig. 2.8, the average is calculated by counting the sums and counts of the events and then dividing the sum of sums by the sum of counts at node B_4. The paper also focuses on optimizing these aggregations by an adaptive technique dynamically deciding when to aggregate the events in the event flow based on publication rate and the number of subscriptions. But this technique does not focus on managing the window period that the events need to be processed.

Similarly, for patterns that cannot be partitioned by a key, Leghari et al. [17] have proposed a system that can distribute data to multiple nodes based on the first matching events. This only supports when the pattern is defined with the *within*

operator. When the first message on the sequence is detected the system randomly send that event to a downstream node and it will continue to send all the following events to the same downstream node till the window period for the specific pattern ends. Though this approach scales the memory and CPU consumptions by (baseline values)/*n* it hinders bandwidth utilization as in the worst-case scenario almost all events will be sent to all the nodes where bandwidth being equal to the baseline values.

StreamCloud [9] has implemented a scalable infrastructure that scales CEP operators, here it uses both fan-out and pipelining techniques. As depicted in Fig. 2.9, the main finding of this work is that a single topology can be divided into multiple sub clusters such that each sub cluster will have multiple nodes where each node will have one stateful operator followed by number of stateless operators and here each sub cluster communicate to its downstream cluster by sending messages in a fan-out manner to each nodes of the sub cluster. In this case, the CPU and memory consumptions will be (baseline values)/*n* where n being the number of partitions and when it comes to bandwidth all stateful nodes will have bandwidth equal to baseline values and other nodes will only have (baseline values)/*n* bandwidth utilization.

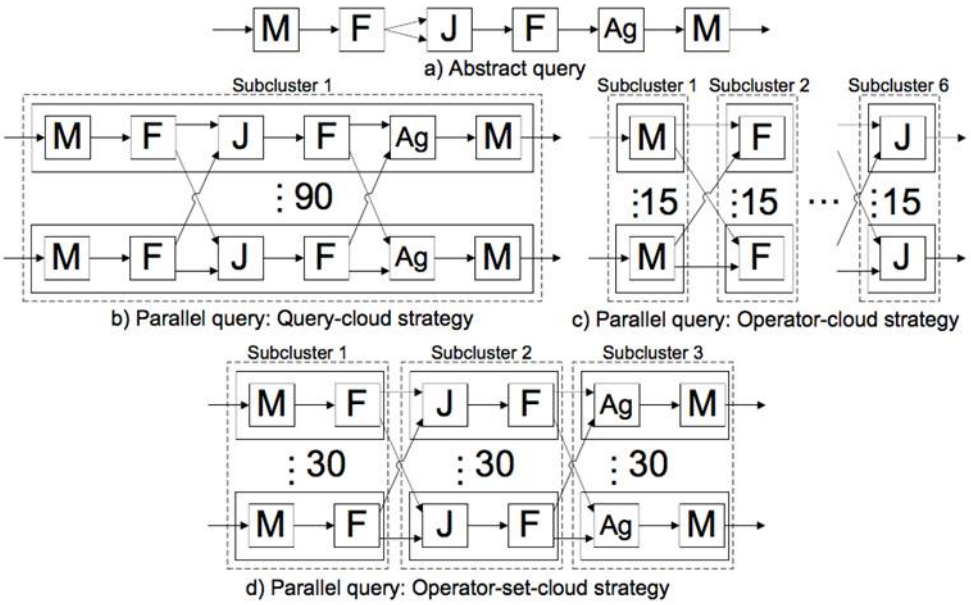

Fig. 2.9: StreamCloud query parallelization strategy [9].

Similarly, for joining events in a scalable manner map side joins and reduce side joins are some techniques that are used to perform join operations in MapReduce [32]. In map side join the data is joint at the map side where one table will be large and it will be distributed among all the map nodes while the other table will be small enough to keep in the mapper nodes memory. Events will be joint at the mapper and passed to the reducer. At the meantime for the reduce side join the mappers will map the data against its joining key and passes that to the reducer and the equijoin happens at the reducer by joining data with same keys from both tables. Google Photon [33] has also illustrated several approaches for scalable and fault-tolerant joining with low latency.

As discussed in the literature the major bottleneck in the CEP systems is its ability to scale beyond the number of partitions and especially on cases where streams cannot be partitioned by a key. In the following sections, we will discuss how we can scale CEP functions such as windows, patterns and joins without enforcing any restrictions on event streams.

2.6 Summary

This section presented related work on scaling CEP systems. Filter, window, pattern, and join are the common operations of a CEP system where these can be scaled by scaling CEP nodes or by scaling CEP operations. Running multiple CEP nodes in a cluster, distributing different type of queries to different CEP nodes, distributing execution via Publish/Subscribe infrastructure, distributing events by partitioning each stream, or by distributing events as batches are some approaches of scaling CEP nodes. In the meantime, CEP operations can be scaled by simply duplicating stateless operators such as filters. Whereas for scaling stateful queries such as windows, patterns, and joins techniques like parallelizing operator graph using partitions, combining partitioning and pipelining, optimizing streaming aggregations, and using StreamCloud's query parallelization strategy can be used. The key takeaway in this section is that the major bottleneck in the current CEP systems is their ability to scale

beyond the number of partitions, especially without enforcing any restrictions on event streams.

Table 2.4: Characterization summary of distributed CEP operations over multiple CEP nodes.

Query Type	**Input Bandwidth**	**Output Bandwidth**	**Memory Consumption**	**CPU Utilization**
Stateless Operators (Filter)	$\Theta(\lambda/n)$	$\Theta(\lambda/n)$	(none)	$\Theta(\lambda/n)$
Stateful Operators with partitioning (window, pattern, and join)	$\Theta(\lambda/n)$	$\Theta(\lambda/n)$	$O(\lambda t/n)$, for temporal processing.	$\Theta(\lambda/n)$
Pattern matching by combining partitioning and pipelining.	$\Theta(\lambda)$ for the first node, $\Theta(\lambda/n)$ for other nodes.	$\Theta(\lambda)$ for the first node, $\Theta(\lambda/n)$ for other nodes.	$O(\lambda t/n)$, for temporal processing.	$\Theta(\lambda/n)$
Pattern matching by time-based distribution.	$\Theta(\lambda)$	$\Theta(\lambda)$	$O(\lambda t/n)$, for temporal processing.	$\Theta(\lambda/n)$
StreamCloud	$\Theta(\lambda)$ for stateful nodes, $\Theta(\lambda/n)$ for other nodes.	$\Theta(\lambda)$ for stateful nodes, $\Theta(\lambda/n)$for other nodes.	$O(\lambda t/n)$, for temporal processing.	$\Theta(\lambda/n)$

3. PROPOSED SOLUTION

Complex Event Processing Engines (CEP) support stateless operators such as filters and stateful operators such as windows, patterns, and joins. Solutions to scale stateless operators are already discussed in the literature. Whereas we focus on how to scale stateful operators without enforcing any restrictions on partitioning the event streams. Section 3.1 proposes several solutions for scaling stateful CEP operations while Section 3.2 summarizes the methodology.

3.1 Proposed solution

As discussed in Section 2.4 and 2.5 stateful queries such as windows, joins, and patterns cannot be distributed in an efficient manner unless otherwise they are partitioned by one or more of their event attributes [31]. Partitioning the event streams based on event attributes will allow the data to be distributed across n nodes while achieving approximately equal (baseline values)/n of CPU, memory, and bandwidth load at each node. This is achieved when approximately the same number of events are sent to each partition, and queries deployed on the partitions also matches an approximately equal number of events. As discussed in the literature the major bottleneck in the CEP systems is its ability to scale beyond the number of partitions and especially on cases where streams cannot be partitioned by a partition key. Next, we discuss how we can scale CEP functions such as windows, patterns, and joins for streams that cannot be partitioned by a partition key.

3.1.1 Scaling window operators

CEP window queries can be categorized into time and length siding windows. *Time window* aggregates given attributes based on a given time period, e.g., last five hours. *Length window* aggregates the attributes based on the given number of events, e.g., last 10,000 events. Both of these windows move in a sliding manner and produce outputs upon each event arrival/departure to/from the window.

3.1.1.1 Scaling sliding time window

Sliding *time window* holds all events that fall within the time window in-memory. This is necessary because it has to emit the events upon the window expiry to adjust the time window aggregations accordingly. When it comes to scaling sliding time windows over non-partitioned streams, techniques such as MapReduce-style aggregations [8] can be used by modifying mapper to perform window execution in a distributed manner and reducer functions to aggregate the windows results in a streaming manner.

Given a time windows of size t and arrival rate λ, $t\lambda$ events can be split among n mappers such that each maintains $t\lambda/n$ events. Each mapper then needs to emit its local result such that a reducer could calculate the global result. To maintain a consistent view of CEP state, each mapper needs to emit its local state at the arrival or departure of an event while reducer needs to recalculate as soon as a new local result arrives. This style of reducer operations can be applied for queries such as sum, count, average, min, and max. For example, consider the Siddhi query for a sliding time window in Query 3.1.

```
from stockStream#window.time(h hour)
select sum(price) as sumPrice,
     count() as countEvents,
     avg(price) as avgPrice
Insert into outputStream;
```

Query 3.1: Example sliding time window query.

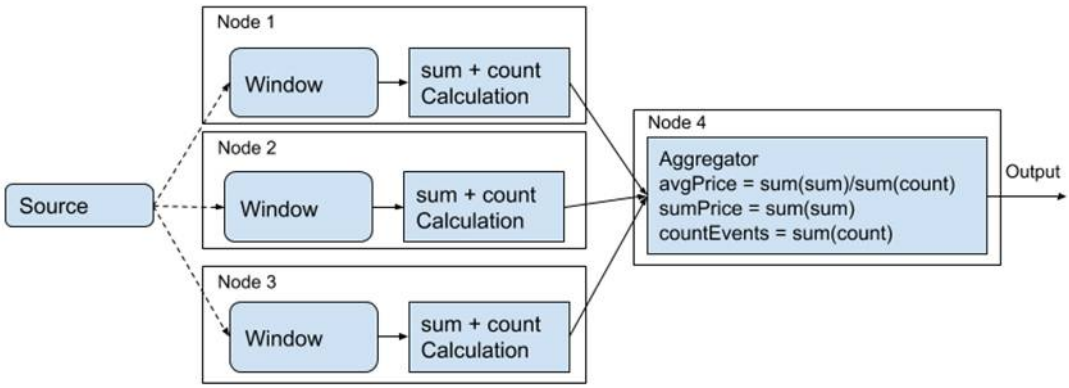

Fig. 3.1: Scaling sliding time window.

As seen in Fig. 3.1 Nodes 1, 2, and 3 mimics mapper functions while node 4 mimics reducer. As per Query 3.1, *sumPrice, countEvents*, and *avgPrice* have to be calculated over the last *h* hour. Based on the MapReduce techniques *avgPrice* will be calculated by separately calculating sums (*s*) and counts (*c*) at each window nodes and aggregating them at the reducer node.

As depicted in Fig. 3.1 events will be sent from source to Node 1, 2, and 3 in a round robin manner. Then each node uses its sliding window to keep the states of the last *h*-hour. All nodes will then send sums and counts of their windows to node 4 as a stream, node 4 calculates *sumPrice*, *countEvents*, and *avgPrice* and emits the output as a stream.

With this technique, there is no need of partitioning the streams and due to the round robin distribution to the nodes 1, 2, and 3, they will get an equal number of events approximately splitting the CPU, memory, and bandwidth among the nodes as (baseline values)/*n*. As node 4 get partially aggregated data it does not need high CPU, memory, and bandwidth. Therefore, node 4's input and output bandwidth will be the same as the single node case, and its CPU utilization and memory consumption will come down from O(baseline) to O(*n*).

3.1.1.2 Scaling sliding length window

While a similar idea could be adopted for length-based windows, a length-window query has an additional drawback. Each window node does not implicitly know when they should expire the window, as it depends on the windows size of other reducers and the source.

Let us discuss an example query based for sliding length as seen in Query 3.2. having a window of length *l*.

```
from stockStream#window.length(l)
select sum(price) as sumPrice,
     count() as countEvents,
     avg(price) as avgPrice
Insert into outputStream;
```

Query 3.2: Example sliding length window query.

By tagging each event with a sequence number at the source, this issue can be resolved. Here the windows in all nodes just need to ensure that the following condition is met:

$$latest\_Event\_sequence\_Number - oldest\_Event\_sequence\_Number < l.$$

For this to work correctly, the events from the source should be dispatched to the windows in a round-robin manner. This is because when other techniques like random or hashing are used there is a possibility of some windows not getting new events regularly; hence, they may fail to expire the events that fall out of the window.

As all the techniques used in sliding time window is also applied to the sliding length window, nodes 1, 2, and 3 will get an equal number of events approximately splitting the CPU, memory, and bandwidth among the nodes as (baseline values)/n. At the same time, node 4 provides the same input and output bandwidth as the single node case, and $O(n)$ CPU utilization and memory consumption.

3.1.2 Scaling pattern operators

As discussed in [31] there are two techniques for scaling pattern queries that can be expressed as Non-Deterministic Finite Automata (NFA). First, is scaling by partitioning the event streams. Second, is scaling by splitting NFA into smaller sub-NFA's and distributing them in a pipeline manner.

Let us consider the pattern query in Query 3.3. The query can be simply expressed as a small purchase (a) followed by another small purchase (b) followed by a large purchase (c) within a day. Based on Brenna et al. [31] this cannot be partitioned but can only be scaled as a pipeline by moving each pattern to a separate node and chaining them one after another as a pipeline such that the matching events of the previous state feed into nodes with later states as depicted in Fig. 3.2.

```
from every a = CardStream[amount < s]
    -> b = CardStream
           [amount < s and a.cardId==cardId]
    -> c = CardStream
           [amount > l and a.cardId==cardId]
    within 1 day
select a.amount as initialPurchaseAmount,
```

```
        c.amount as lastPurchaseAmount,
        c.location as location
insert into PossibleFraudStream;
```

Query 3.3: Example pattern query.

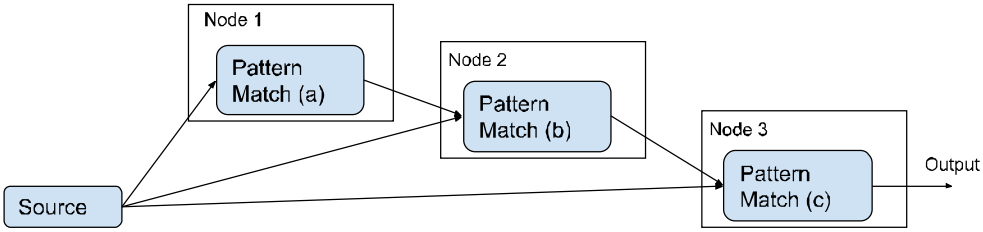

Fig. 3.2: Scaling pattern based on Brenna at el. [31].

As discussed in the literature this has a limitation in scaling bandwidth. This can be reduced to some extent by categorizing input streams into logical streams based on their types or by pre-filtering and publishing relevant streams only to relevant nodes. For example, in the above case based on the pattern query defined we know that node 1 and 2 are only interested in events having (price < s); hence, as depicted in Fig. 3.3, the source can filter those events having (price < s) into a logical stream (S) and publish only that stream to node 1 and 2. At the same time, we can filter all events having (price > l) into another logical stream (L) and publish that stream only to node 3. As this approach only sends part of the initial stream to each node, the initial node consumes $\Omega(\lambda)$ bandwidth while other nodes consume $\Omega(\lambda)$ bandwidth. This way, we can reduce the bandwidth constraint to some extent. But if the arrival rates are high theses nodes may even need more CPU and memory to process incoming events.

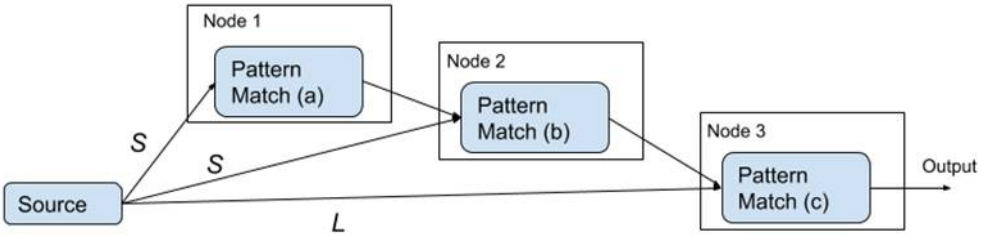

Fig. 3.3: Scaling patterns based on stream type.

In real-world deployments, patterns may also be defined across multiple stream types. For example, suppose a Tweet saying CEO of company X resigned followed by the stock price of company X falling. For these types of queries instead of generating logical streams, we can directly distribute the different type of streams to different nodes and scale distributed pattern matching.

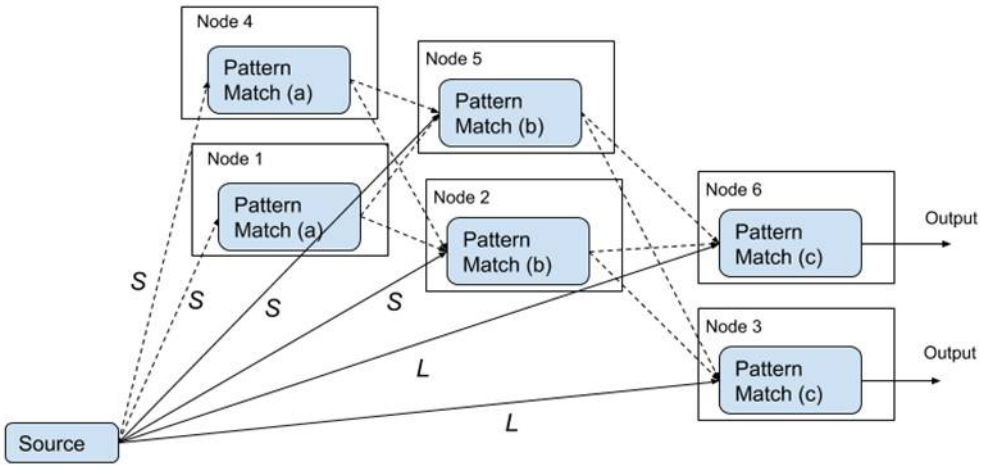

Fig. 3.4: Scaling pattern based on distributed streams.

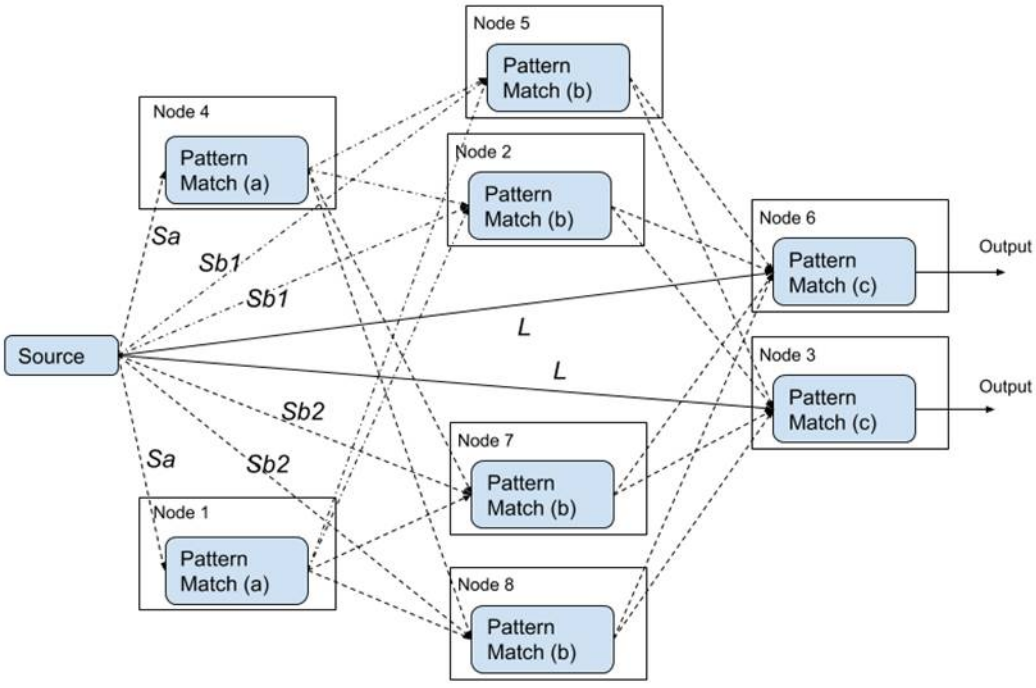

Fig. 3.5: Scaling pattern by replicating distributed streams.

Even with this approach, there is a possibility of nodes hitting maximum bandwidth, memory and CPU limits, especially for the nodes processing the initial part of the pattern. We can overcome this by replicating and horizontally scaling each pattern state into multiple nodes, chaining each previous state with each next state, and by sending events of the high bandwidth stream in a round-robin manner while broadcasting other streams to relevant nodes. For example, as depicted in Fig. 3.4 an instance of stream *S* is sending to nodes 1 and 4 in a round-robin manner, another instance of stream *S* is sending all events to nodes 2 and 5, and stream *L* is sending all events to nodes 3 and 6. Further, the matched events of node 1 and 4 are sent to nodes 2 and 5 and their matched events are then sent to nodes 3 and 6 in a round-robin manner. This will make sure that the NFA is initialized in either of the initial nodes 1 or 4 and those nodes pass the partially matched states to downstream nodes in a scalable manner. With this design, we reduce the CPU utilization, memory consumption and bandwidth of the initial pattern to (initial value when pattern states are not replicated)/*n* where n is the number of state-replicated nodes.

Though this approach scales the CPU and memory of all pattern states and the bandwidth of the initial state, it does not scale the bandwidth of the second and later states of the pattern. This can be resolved by replicating the states that receive high bandwidth into groups and send events in a round-robin manner to each group while broadcasting the events to each member within the group. For example, as depicted in Fig. 3.5 the bandwidth of each second pattern state node is now reduced by replicating the states to two groups of two nodes where node 2 and 5 in one group and node 7 and 8 in the other group, and by source broadcasting stream *Sb,* and the previous states broadcasting partially matched events to node 5 and node 2 and then to node 7 and node 8 in a round-robin manner. Through this bandwidth of the non-initial nodes can be reduced to $2\lambda/n$.

This approach is also better than Leghari et al.'s [17] implementation as this has only O(none scaled state node value)/*n* CPU, memory, and bandwidth utilization, while the Leghari et al.'s implementation can only scale up to (none scaled state node value)/3 in terms of CPU and memory utilization.

3.1.3 Scaling join operators

Joining streams can be categorized into two, namely joining a stream having a large window with another stream having a small window and joining two streams both having large windows. Both of these problems are already solved in the databases and in the MapReduce literature [32] and we can simply use the same techniques to achieve streaming joins in a scalable manner.

3.1.3.1 Joining large window with small window

To scalable joining of large and small windows, we have to have the instance of the small window in all the nodes and distribute the large window across all the nodes. When scaling the large window, we need to employ the techniques that are used for scaling windows in Section 3.1.1. This way we can make sure that windows contain only the data relevant to the window period and all events are properly joint against one another. Finally, as depicted in Fig. 3.7, all events should be sent to a reducer for final aggregations if there are any. As depicted in Fig. 3.6 events of *StreamA* (large window) is sent in a round robin manner to multiple nodes (like how we have scaled the window) and events of *StreamB* (small window) is sent to all the nodes. The joining of the *StreamA* window with the *StreamB* window happens at each node.

Consider the example query in Query 3.4.

```
from StreamA#window.length(l) join
     StreamB#window.length(s)
     on StreamA.symbol == StreamB.symbol
select sum(price) as sumPrice,
     count() as countEvents,
     avg(price) as avgPrice
Insert into OutputStream;
```

Query 3.4: Example joining small and large window query.

This will help to scale CPU and memory on all window nodes. When we scale the window to n nodes, as each incoming event need to be processed against all events of the other stream's window, its CPU utilization will be $O((\lambda_A/n)e_B + \lambda_B e_A/n)$ where e_i is the number of events in stream i's window. Whereas memory consumption, input bandwidth utilization, and output bandwidth utilization will be $O(e_A/n + e_B)$,

$\Theta(\lambda_A/n + \lambda_B)$, and $\Omega(\lambda_A/n + \lambda_B)$, respectively. As the aggregation node 3 has to aggregate all the incoming events at any given time, its CPU and bandwidth load will be $\Omega(\lambda_A/n + \lambda_B)$ and the memory utilization will be $O(n)$.

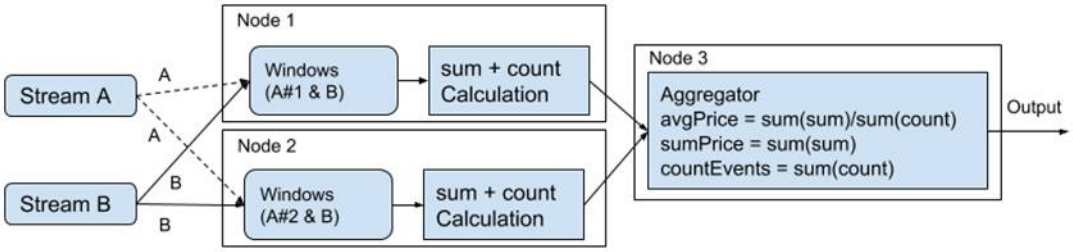

Fig. 3.6 Scaling join of small and large windows.

3.1.3.2 Joining two large windows

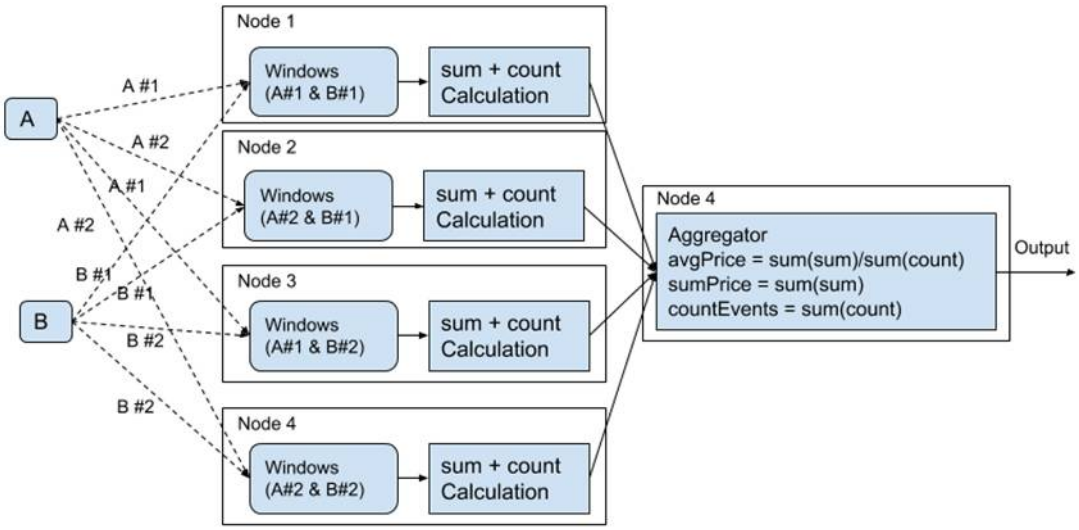

Fig. 3.7: Scaling join of two large windows.

For scalable joining of two large windows, as described in the database literature we can distribute both the windows such that each part of the window will be matched against the other as depicted in Fig. 3.7.

Though this scale based on CPU, memory, and bandwidth, the drawback of this approach is that it needs number of nodes equal to the power of two. This will help to scale CPU, memory, and bandwidth on all window nodes where the CPU utilization

will be $O((\lambda_A e_B + \lambda_B e_A)/n)$ where e_i is number of events in Stream i's window, memory consumption will be $O((e_A + e_B)/n)$, input bandwidth utilization will be $\Theta((\lambda_A + \lambda_B)/n)$, and output bandwidth utilization will be $\Omega((\lambda_A + \lambda_B)/n)$. At node 3, the CPU and bandwidth load will be $\Omega((\lambda_A + \lambda_B)/n)$, while memory load will be $\Theta(n)$.

Table 3.1: Summary of distributed stateful CEP operations over multiple CEP nodes for streams that cannot be partitioned by a key.

Query Type	**Input Bandwidth**	**Output Bandwidth**	**Memory Consumption**	**CPU Utilization**
Scalable sliding time and length windows operators	Window Node $\Theta(\lambda/n)$, Aggregator Node $\Theta(\lambda)$ which is $\leq$ to single node baseline $\Theta(\lambda)$.	Window Node $\Theta(\lambda/n)$, Aggregator Node $\Theta(\lambda)$ which is $\leq$ to single node baseline $\Theta(\lambda)$.	Window Node $\Theta(e/n)$, Aggregator Node $O(n)$ which is $\leq$ to single node baseline $\Theta(e)$.	Window Node $\Theta(\lambda/n)$, Aggregator Node $\Theta(\lambda)$ which is $\leq$ to single node baseline $\Theta(\lambda)$.
Scalable pattern operators based on distributed streams. Assuming all streams have input rate λ and all states have equal number of events.	$\Omega(\lambda)$	$\Omega(\lambda)$	First node $O(1)$, Other nodes $\Theta(e/(s-1))$ which is $\leq$ to single node baseline $O(e)$.	$\Theta(\lambda e/s)$ which is $\leq$ to single node baseline $O(\lambda e)$.
Scalable join operators for joining small and large windows. Assuming all streams have input rate λ.	Join Node $\Theta\left(\lambda + \frac{\lambda}{n-1}\right)$, Aggregator Node $O(\lambda)$ which is = to single node baseline $O(\lambda)$.	Join Node $\Theta\left(\lambda + \frac{\lambda}{n-1}\right)$, Aggregator Node $O(\lambda)$ which is = to single node baseline $O(\lambda)$.	Join Node $\Theta\left(e + \frac{e}{n-1}\right)$, Aggregator Node $O(n)$ which is = to single node baseline $O(e)$.	Join Node $\Theta\left(\lambda e + \frac{\lambda e}{n-1}\right)$, Aggregator Node $\Theta(\lambda + \lambda/n)$ which is = to single node baseline $O(\lambda e)$.
Scalable join operators for joining 2 large windows. Assuming all streams have λ input rate.	Join Node $O(\lambda/n)$, Aggregator Node $O(\lambda)$ which is $\leq$ to single node baseline $O(\lambda)$.	Join Node $O(\lambda/n)$, Aggregator Node $O(\lambda)$ which is $\leq$ to single node baseline $O(\lambda)$.	Join Node $\Theta(e/n)$, Aggregator Node $O(n)$ which is $\leq$ to single node baseline $O(e)$.	Join Node $\Theta(\lambda e/n)$, Aggregator Node $\Theta(\lambda/n)$ which is $\leq$ to single node baseline $O(\lambda e)$.

3.2 Summary

We discussed how stateful CEP operations such as windows, patterns, and joins can be distributed and scaled across multiple nodes while understanding how each of the approaches affects CPU, memory, and bandwidth utilization of the system. In our proposed approach events are sent from the source to various processing nodes connected to each other as a directed acyclic graph. As discussed in the methodology Table 3.1 summarizes the input and output bandwidth, memory consumption, and CPU utilization of the scalable stateful CEP operations for streams that cannot be partitioned by a key while comparing it against single-node base cases. The workload characteristics such as the rate of event arrival of each stream, number of processing nodes, the total number of events stored in memory for processing, and the total number of states in a pattern.

4. IMPLEMENTATION

The proposed solution for scaling windows, patterns and joins are to be implemented using Siddhi CEP engine and Kubernetes [21] container orchestration system. Siddhi is selected as it is open-source, has low latency, highly extensible, and capable of analyzing millions of events per second [23]. Kubernetes is selected as it is a widely accepted container orchestration system that can host scalable microservices. Section 4.1 presents how to practically implement the proposed solution discussed in Section 3.1, and the implementation summary is presented in Section 4.2.

4.1 Scaling window operators

To evaluate the stability of each window component, a non-scalable implementation and scalable distributed implementations are tested. To achieve fair comparison across all use cases the data publisher and the consumer are separated from the CEP processing logic. For example, for the cases of scalability of sliding time window as given in Query 4.1 we have implemented two implementations as shown in Fig. 4.1 and 4.2.

```
from stockStream#window.time(1 hour)
select sum(price) as sumPrice,
     count() as countEvents,
     avg(price) as avgPrice
Insert into outputStream;
```

Query 4.1: Example sliding time window query.

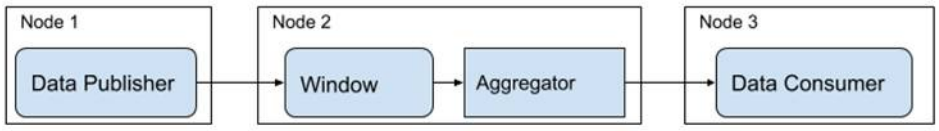

Fig. 4.1: Deployment of standard sliding time window test.

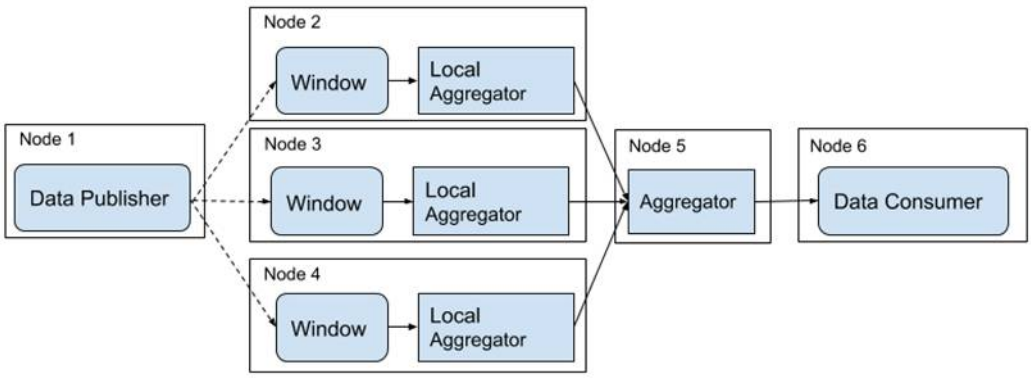

Fig. 4.2: Deployment of scalable sliding time window test.

The data publisher, data consumer, and the intermediate data processing nodes are implemented using Siddhi. Each segment of the distributed logic is also written as Siddhi queries, where each node runs a single Siddhi Manager and deploys a service to consume, process, and publish messages.

For publishing, application given in Query 4.2 is deployed and events are pushed to *StockEventStream*. This will send the events to multiple internal data processing nodes in a round-robin manner.

```
@app:name('publisher')
@sink(type='tcp', sync='true', @map(type='binary')
      @distribution(strategy='roundRobin',
      @destination(
         url='tcp://url1/time-window/StockEventStream')
      @destination(
         url='tcp://url2/time-window/StockEventStream')))
define stream StockEventStream(symbol string,
                               price float, volume long);
```

Query 4.2: Data publisher Siddhi Application.

To consume the final output, the application provided in Query 4.3 is deployed. When the processed events are pushed to its endpoint the application calculates the throughput and accuracy of the system.

```
@app:name('consumer')
@source(type='tcp', @map(type='binary'))
define stream AggregateStockStream (symbol string,
                 totalPrice double, avgVolume double);
```

Query 4.3: Data consumer Siddhi application.

In terms of the standard sliding time window test, the processing node 2 (see Fig. 4.1) will contain the application in Query 4.4 to consume the events, process the sliding time window, and to publish the events to the consumer.

```
@app:name('time-window')
@source(type='tcp', @map(type='binary'))
define stream StockEventStream (symbol string,
                                price float, volume long);
@sink(type='tcp',
      url='tcp://url3/consumer/AggregateStockStream',
      sync='true', @map(type='binary'))
define stream AggregateStockStream (symbol string,
                  totalPrice double, avgVolume double);
@info(name = 'query1')
from StockEventStream#window.time(1 hour)
select symbol, sum(price) as totalPrice,
           avg(volume) as avgVolume
group by symbol
insert into AggregateStockStream;
```

Query 4.4: Standard sliding time window Siddhi Application.

When implementing the scalable sliding time window, nodes 2 to 4 of Fig. 4.2 will contain the following application as shown in Query 4.5. Here each node will have time window and local data aggregations to achieve scalability as we have discussed under Section 3.1.1 scaling window operators. These will also send the node ID in each of its output event for accurate global aggregation.

```
@app:name('time-window')
@source(type='tcp', @map(type='binary'))
define stream StockEventStream (symbol string,
                                price float, volume long);
@sink(type='tcp',
url='tcp://url5/aggregation/PartialAggregateStockStream'
,sync='true', @map(type='binary'))
define stream PartialAggregateStockStream (symbol string,
                  totalPrice double, totalVolume long,
                  countVolume long, id string);
@info(name = 'query1')
from StockEventStream#window.time(1 hour)
select symbol, sum(price) as totalPrice,
           sum(volume) as totalVolume,
           count(volume) as countVolume, '1' as id
group by symbol
insert into PartialAggregateStockStream;
```

Query 4.5: Scalable pre aggregation of sliding time window Siddhi Application.

The global aggregator of the scalable sliding time window depicted in node 5 of Fig. 4.2 will contain the Siddhi application provided by Query 4.6. This matches the latest event of each node by their unique node ID and performs the aggregation to get the global aggregation across the given sliding time period. When the events are arriving out of order, they are buffered and reordered using the K-Slack [34] algorithm to improve the accuracy of the system. In this case, the K-Slack algorithm is engaged just before the *unique:ever()* window to reorder the events before aggregation.

```
@app:name('aggregation')
@source(type='tcp', @map(type='binary'))
define stream PartialAggregateStockStream (symbol string,
          totalPrice double, totalVolume long,
          countVolume long, id string);
@sink(type='tcp',
     url='tcp://url6/consumer/AggregateStockStream',
     sync='true', @map(type='binary'))
define stream AggregateStockStream (symbol string,
          totalPrice double, avgVolume double);
@info(name = 'query1')
from PartialAggregateStockStream
     #window.unique:ever(id, symbol)
select symbol, sum(totalPrice) as totalPrice,
     sum(totalVolume)*1.0/sum(countVolume) as avgVolume
group by symbol
insert into AggregateStockStream;
```

Query 4.6: Global aggregator of scalable sliding time window Siddhi Application.

Data publisher and data consumer are separated, and the data is transferred between nodes using blocking TCP calls using Siddhi's TCP [35] data publisher. Blocking TCP calls are selected as they give low latency data transfer and provide back pressure to the event sources if the system is having performance bottlenecks downstream. Due to this behavior, there would not be any events pile-ups in the network buffers. Therefore, we can better evaluate the memory consumptions of stateful CEP operators.

All the nodes used for testing are build using Docker [36] and deployed as pods in Kubernetes, and for this implementation, Kubernetes Google Cloud is used. Further, the Kubernetes deployment is also configured such a way that it kills the nodes who

consumes more memory than allocated. This helped us to detect out of memory situations without affecting the performance of the system.

In terms of the scalable implementation, each distributed node is given a unique node ID and it is passed to the aggregation node along with each message, through this the latest event of each node was uniquely identified and aggregated. Sliding length window operations are also implemented and tested using the same methodology and guidelines.

4.2 Scaling of pattern operators

To evaluate the scalability of pattern component, like in windows, a non-scalable implementation and a scalable distributed implementation are tested. Here also the data publisher and the consumer are separated from the CEP processing logic. To test the scalability of the pattern operation, Query 4.7 is selected as discussed in Section 3.1.2 and its implemented in three different ways as given in Fig. 4.3, 4.4, and 4.5. Implementation in Fig. 4.3 represents the standard single-node pattern deployment. Implementation in Fig. 4.4 is used to evaluate how patterns can be scaled by executing each pattern state in a dedicated node and the final implementation shown in Fig. 4.5 is used to evaluate how each pattern state can be further scaled.

```
from every a = CardStream[amount < 100]
    -> b = CardStream
           [amount < 100 and a.cardId==cardId]
    -> c = CardStream
           [amount > 10000 and a.cardId==cardId]
    within 1 day
select a.amount as initialPurchaseAmount,
     c.amount as lastPurchaseAmount,
     c.cardId as cardId,
     c.location as location
insert into PossibleFraudStream;
```

Query 4.7: Example pattern query.

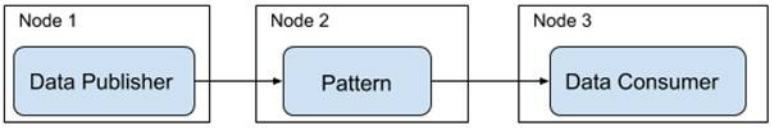

Fig. 4.3: Deployment of standard pattern test.

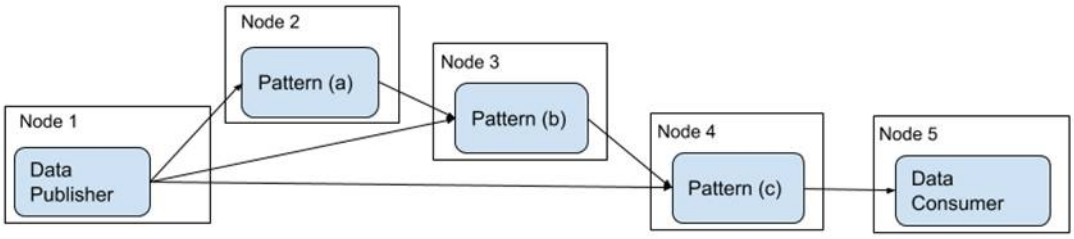

Fig. 4.4: Deployment of scalable patterns based on streams.

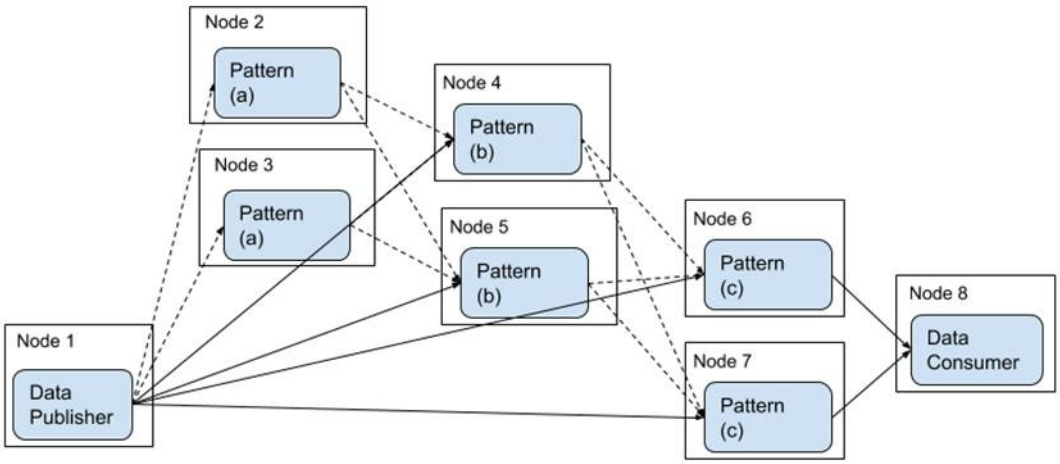

Fig. 4.5: Deployment of scalable pattern based on distributed streams.

The data publisher, data consumer nodes are implemented similarly to the scalable window operator implementation in Section 4.1. The standard pattern test implementation for the processing node 2 as shown in Fig. 4.3 contains the application in Query 4.8 to consume the events, process the pattern query, and to publish the events to the consumer.

The first condition used in the scalable pattern depicted as nodes 2 of Fig. 4.4 and node 2 and 3 of Fig. 4.5 is implemented using Query 4.9 to perform basic input event filtering. The second condition of the pattern implementation of node 3 of Fig.

4.4 and node 4 and 5 of Fig. 4.5 contains the Siddhi application in Query 4.10, having basic pattern matching with the previous event occurrences.

```
@app:name('pattern')
@source(type='tcp', @map(type='binary'))
define stream CardStream (cardId string, amount float,
                          location string);
@sink(type='tcp',
      url='tcp://url1/consumer/PossibleFraudStream',
      sync='true', @map(type='binary'))
define stream PossibleFraudStream (
      initialPurchaseAmount float,
      lastPurchaseAmount float,
      cardId string, location string);
@info(name = 'query1')
from every a = CardStream[amount < 100]
      -> b = CardStream[amount < 100 and
      a.cardId==cardId]
      -> c = CardStream
                [amount > 10000 and a.cardId==cardId]
      within 1 day
select a.amount as initialPurchaseAmount,
           c.amount as lastPurchaseAmount,
           c.cardId as cardId
           c.location as location
insert into PossibleFraudStream;
```

Query 4.8: Standard pattern Siddhi Application.

```
@app:name('pattern1')
@source(type='tcp', @map(type='binary'))
define stream CardStream (cardId string, amount float,
      location string);
@sink(type='tcp',
      url='tcp://url2/pattern2/PossibleFraudStream1',
      sync='true', @map(type='binary'))
define stream PossibleFraudStream1 (
      initialPurchaseAmount float,
      cardId string, timestamp long);
@info(name = 'query1')
from every a = CardStream[amount < 100] within 1 hour
select a.amount as initialPurchaseAmount,
      currentTimeMillis() as timestamp, a.cardId
insert into PossibleFraudStream1;
```

Query 4.9: First condition of the scalable pattern Siddhi Application.

```
@app:name('pattern2')
@source(type='tcp', @map(type='binary'))
define stream CardStream (cardId string, amount float,
      location string);
@source(type='tcp', @map(type='binary'))
define stream PossibleFraudStream1 (
      initialPurchaseAmount float,
      cardId string, timestamp long);
@sink(type='tcp',
      url='tcp://url2/pattern3/PossibleFraudStream2',
      sync='true', @map(type='binary'))
define stream PossibleFraudStream2 (
      initialPurchaseAmount float,
      cardId string, timestamp long);
@info(name = 'query1')
from every a=PossibleFraudStream1
      -> b = CardStream [amount < 100 and
      (currentTimeMillis() - a.timestamp) < 60000 and
      a.cardId == cardId] within 1 hour
select a.amount as initialPurchaseAmount,
      a.timestamp, a.cardId
insert into PossibleFraudStream2;
```

Query 4.10: Second condition of the scalable pattern Siddhi Application.

Like the second condition, all other non-initial pattern conditions can also be implemented using the same technique discussed for the second pattern state implementation. In the pattern implementation to handle out of order event arrival, the conditions are written in such a manner that they will always produce the expected output when either of the streams arrives to the node first. The implementation of nodes, deployment, and data transfer technique of patterns are similar to the window implementation and follows the same implementation techniques and principles.

4.3 Scaling join operators

The join operations are also implemented like the windows having a non-scalable implementation and scalable distributed implementations. To test the scalability of the join query given in Query 4.11 is selected as discussed in Section 3.1.3 and implemented as given in Fig. 4.6 and 4.7. The standard join is similar to the window and pattern implementations but the data producer, in this case, publishes both the joining streams (*StreamA* and *StreamB*) at the same time.

```
from StreamA#window.length(10000)
     join StreamB#window.length(20)
     on StreamA.symbol == StreamB.symbol
select sum(price) as sumPrice,
     count () as countEvents,
     avg(price) as avgPrice
Insert into OutputStream;
```

Query 4.11: Example join query.

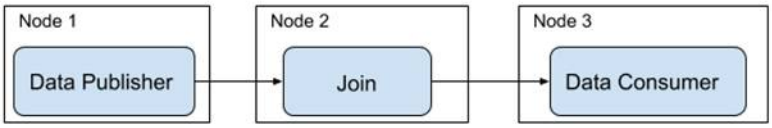

Fig. 4.6: Deployment of standard join test.

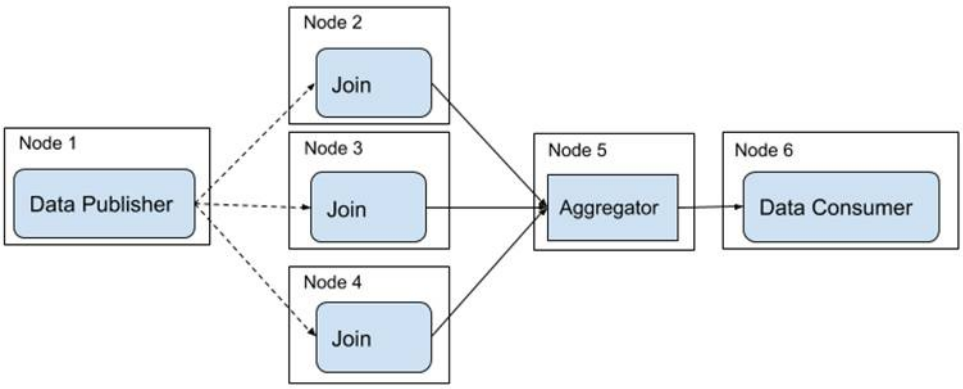

Fig. 4.7: Deployment of scalable join test.

The scalable join is implemented as illustrated in Fig. 4.7 and it caters both joins across small and large windows and joins across two large windows. These are facilitated by sending the data streams as discussed in Section 3.1.3. The implementation, deployment, and data transfer between nodes of the join implementation also follows the same techniques and principles followed by the window and pattern implementation.

4.4 Summary

We discussed how stateful CEP operations such as windows, patterns, and joins can be implemented and scaled across multiple nodes. In the proposed approach

events are sent from a data publisher to various processing nodes to perform standard and scalable complex event processing. Finally, when the data is processed by the data processing nodes the final output will be pushed to the data consumer to calculate performance and accuracy. Here each node is implemented using Siddhi, containerized using Docker, and deployed as Kubernetes pods. The communication between the nodes happens using TCP transport.

5. PERFORMANCE EVALUATION

This section provides details about the scalability analysis of the proposed stateful CEP operation. Section 5.1 discusses how the data is generated and how it is used for the evaluation. Experimental setup and hardware configurations are presented in Section 5.2. Section 5.3 analyzes the scalability of the proposed solution compared to the single nodes operation of the CEP engine considering system resource (CPU, memory, and bandwidth) utilization. Section 5.4 presents more in-depth information on accuracy and latency while Section 5.5 provides the applicability of the results of other CEP systems and Section 5.6 presents the summary.

5.1 Data set

As this thesis focuses on scalability and related performance implications, data generation for testing also focuses on introducing the worst-case scenarios for all the use cases as and when applicable. Due to this fact, no existing benchmark was used but rather the data is generated in a way that stresses the system.

Data generation is done using a Java client and published using a Siddhi Application using TCP transport. For testing window and join operations stock quote data stream is generated containing a *symbol* of type string, *volume* of type long, and *price* of type float. To test the pattern operation a credit card data stream is generated comprising *cardID* of type string, *amount* of type float, and *transaction location* of type string. All events are also published with the event generated timestamp value.

To maintain consistency when testing standard and scalable distributed setups the data publisher client is modified to publish events at a consistent rate. For testing purposes, this value is constantly maintained at 1000 Events per Second (TPS). By having a consistent data flow rate, the calculations of CPU, memory, and bandwidth become straightforward and they also become comparable across various test cases. Here, the bandwidth of the nodes is approximated by calculating the corresponding

nodes the event arrival and event publishing rates. For testing purposes, the data publisher's upload bandwidth for all test cases is constantly maintained at 1000 TPS.

5.2 Experimental setup

The test setup is configured on Kubernetes [21] container orchestration system offered by Google Cloud. Each node used for testing is modeled as a Docker [36] Container running necessary Java code and Siddhi application and deployed in Kubernetes running on Google Cloud.

The prototype of the experimental implementation is built using Open JDK 1.8.0_152 and Siddhi version 4.2.35. For achieving TCP based event transmission Siddhi IO TCP version 2.0.20 is used. The data publisher, data consumer, and the processing nodes are built as Docker images using OpenJDK 8-jre-alpine3.8 base image. These Docker images are then configured using Kubernetes deployment and service configurations and deployed on the Kubernetes version 1.9.7-gke.11 [21] provided by the Google Cloud.

As we are focused on the scalability of the system, resource constraint nodes are selected, and for testing purposes, each node is also allocated an equal amount of CPU and memory. Here each node of the window operation tests was given 0.5 virtual CPU and 256MB of memory, and each node of the pattern and join operation tests was given 1 virtual CPU and 512MB of memory. Further Kubernetes deployment is configured such a way that it kills the nodes who consumes more memory than allocated. This facilitated detecting out of memory situations without affecting the throughput of the system.

The system is also implemented and tested according to the following guidelines:

- All nodes have an equal amount of memory and CPU.
- The data publishing throughput is constantly maintained at 1000 events per second.

- Events with constant size are published across standard and scalable stateful CEP operations for the test.
- For each test, the system ran for more than 20 minutes and the results are measured at steady state.
- The results for each scenario was calculated by running the tests three times and averaging the readings to improve the accuracy.

5.3 Analysis on system scalability

As discussed in Section 3.1 different implementation was done for standard and scalable versions of each stateful CEP operation and tested for scalability of the system. During this phase, the scalability of the system is tested by changing the duration considered for real-time analytics from 5 seconds windows to 1-hour windows while maintaining the input message rate and keeping the hardware configuration of the nodes unchanged. During the evaluation CPU, memory and bandwidth of each node are measured.

5.3.1 Analysis on scalability of window operation

Sliding time, and length windows are considered for this evaluation. For evaluation purposes, each test has been carried out with a selected query with a given time or length window, as these query parameters do not affect the scalability properties of the solution.

5.3.1.1 Sliding time window

As our primary objective is to determine the scalability of the system, the standard single-node time window and deployment of the scalable time window (as discussed in Section 4.1) are tested against various time window intervals with a varying number of nodes and their throughput, memory, and CPU utilization. The throughput of the single-node system stayed at a constant level up to a 30-second window test, and on the 35-second window test, the Kubernetes killed the node due to high memory consumption. This can be viewed by the throughput and memory values depicted in Fig. 5.1 and 5.2, respectively. During this phase, the CPU of the node has

been the same over various time intervals as depicted in Fig. 5.3. This can be justified as that the amount of work that needs to be done only depends on the input rate and they do not depend on the size of the sliding window that is being processed. This is because the Siddhi CEP engine always maintains running aggregations and as events arrive and leave the window, it updates running aggregations only by considering the deltas produced by the new and expired events, and not by iterating through the whole window every time.

Fig. 5.1: Throughput of 1, 3, 5, 9, and 17 node time windows.

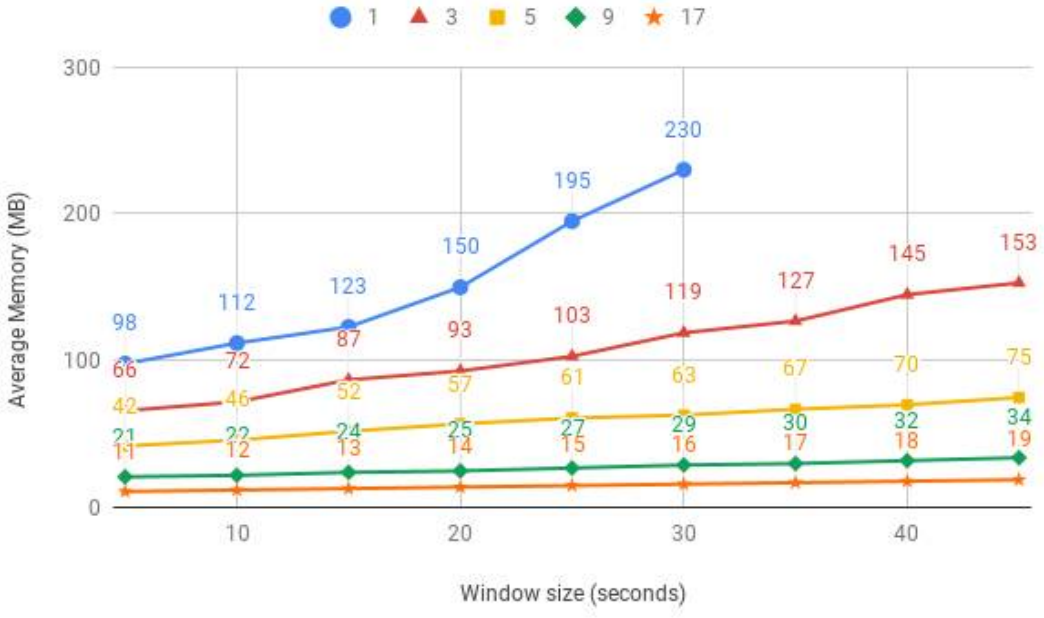

Fig. 5.2: Memory consumption of 1, 3, 5, 9, and 17 node time windows.

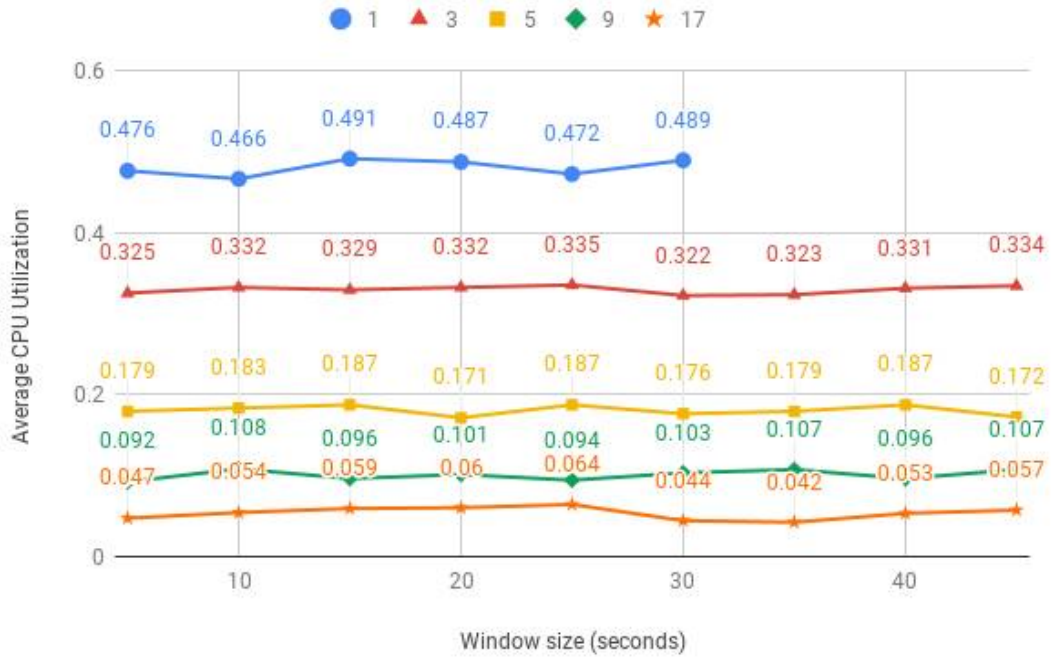

Fig. 5.3: CPU utilization of 1, 3, 5, 9, and 17 node time windows.

From Fig. 5.1, 5.2, and 5.3 we are able to understand how average system throughput, memory, and CPU varies as we scale the number of processing nodes. To understand how much the system can scale, the system was stressed to find out the maximum time interval it can support when the system is at steady state by varying number of processing nodes. For 3, 5, 9, and 17 nodes systems, one node was used as the aggregator and other nodes were used as the window nodes. Corresponding results are shown in Fig. 5.4. It can be seen that the time interval that the system can process has increased exponentially when the number of nodes increases. This is in contrast to the expected linear scalability of $\Theta(e/n)$ (see Table 3.1). This is probably due to not having sufficient aggregated memory when the number of nodes is less. To further analyze the behavior of the system, the number of events stored in each node, the bandwidth of each window processing node, and their CPU utilization were measured while scaling the system. The results are depicted in Fig. 5.6, 5.7, and 5.8.

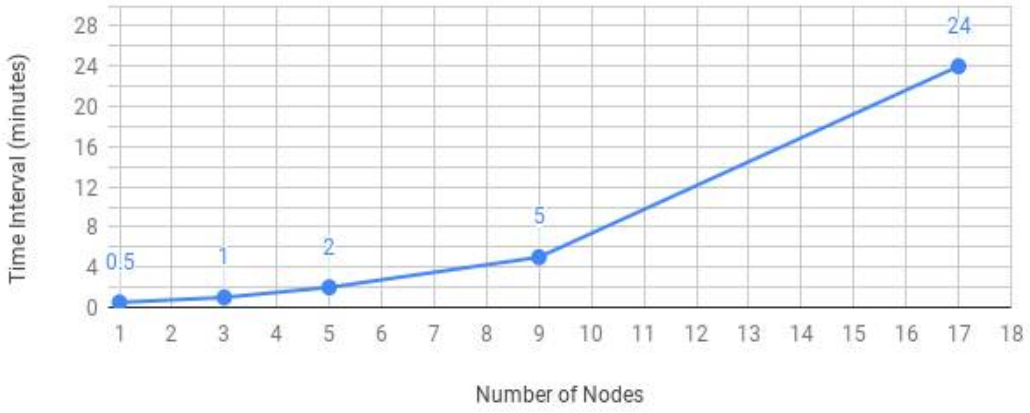

Fig. 5.4: Maximum time interval supported by the number of nodes.

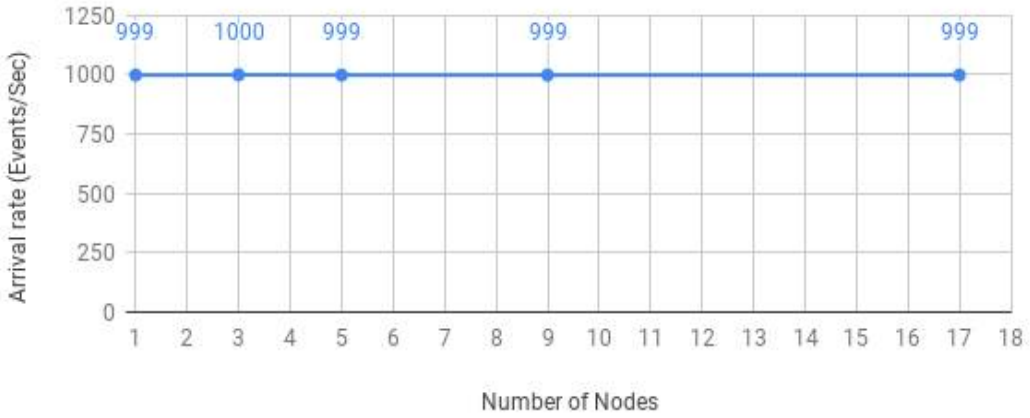

Fig. 5.5: Event consumption throughput by the number of nodes while supporting maximum time interval.

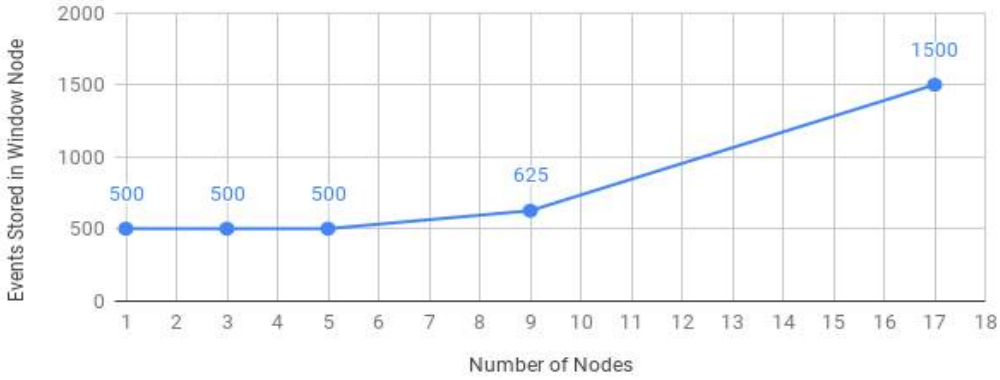

Fig. 5.6: Average number of events stored in each window node.

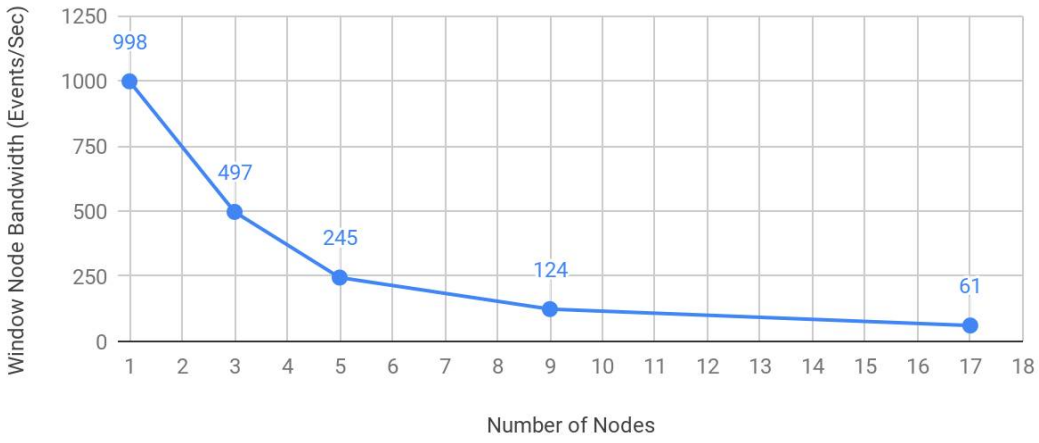

Fig. 5.7: Average bandwidth of window processing nodes (events/Sec).

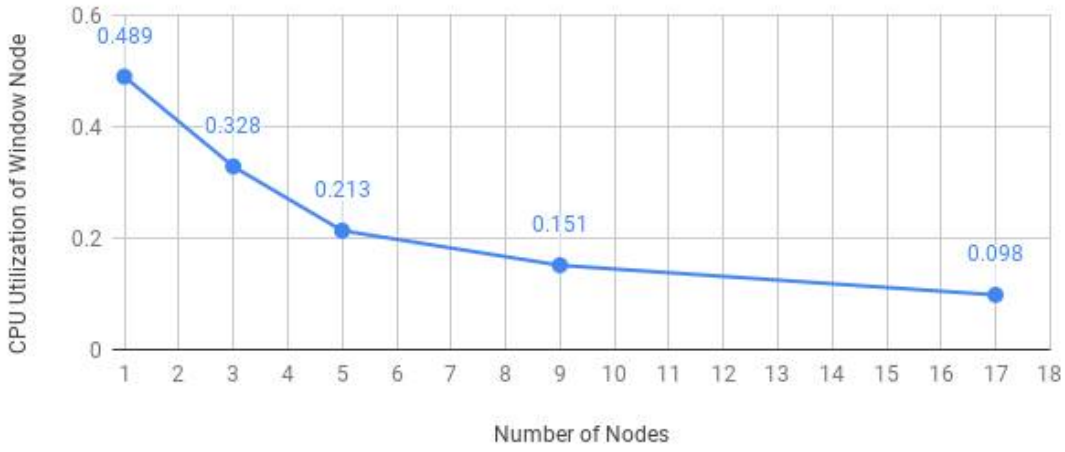

Fig. 5.8: Average CPU utilization of window processing nodes.

While the results show that the bandwidth and CPU utilization do not adhere to theoretical expected linear input bandwidth of $\Theta(\lambda/(n\text{-}1))$ and linear CPU utilization of $\Theta(\lambda/(n\text{-}1))$, we were able to see a clear positive correlation between CPU utilization and the input bandwidth of the system where both are decreasing when the number of nodes increases. This is evident because our publishing bandwidth is kept at a constant rate of 1000 events/Sec and hence when the number of nodes increases each node will receive a fraction of the published bandwidth. Because each node is consuming only a fraction of the published bandwidth their CPU utilization has also decreased corresponding to the processing event rate. This behavior can also be correlated against the number of events stored in each node, where when the CPU utilization of the node

decreases the system was able to keep more events in the window. This is because when the CPU utilization decreases the system will have enough cycles to perform accurate thread scheduling for event expiry and have cycles for garbage collection. Due to this behavior, the system will not unnecessarily waste its memory because it will not contain many garbage-collectible objects on its heap at any given time. Hence, it can use that space to accommodate the events that are actually needed to be stored in sliding time window. Note that this conclusion only holds when the system is implemented in Java and deployed in Kubernetes, as Kubernetes does not impose additional garbage collection cycles but rather it kills the nodes who consuming more memory without providing back pressure and affecting the throughput of the system. If this system is implemented on top of a non-Java based CEP system which does not require garbage collection, there is a possibility of achieving more linear scalability as we increase the number of nodes.

5.3.1.2 Sliding length window

The evaluation of the sliding length window depicted in Fig. 5.9 shows approximately linear behavior than the sliding time window because its Siddhi implementation does not require any additional threads for event expiry like time windows. Hence, it will consume lesser CPU and processing memory even for systems having a lower number of nodes.

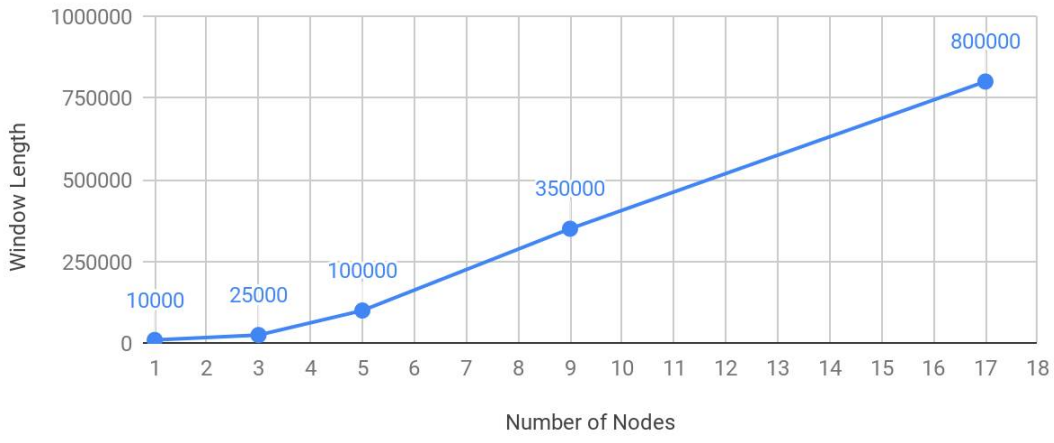

Fig. 5.9: Maximum window length supported by the number of nodes.

5.3.2 Analysis of pattern operation scalability

Standard single-node pattern, scaling pattern by distributing each of its states to a dedicated node, and scaling pattern by replicating the states on multiple nodes are evaluated. To understand the worst-case behavior of the system subject to an uneven rate of input streams, we tested the system in such a way that the system will create lots of partially matched internal states but none of them ever qualifying as full matches. This is done by sending only *CardStream* events having an amount less than 100 for each *cardID* for the pattern Query 4.7, these events matched the pattern conditions at state 1 and state 2 while none of them ever matched the condition at state 3. This lets the system to accumulate lots of partially matched internal events at state 2. To make sure the system does not go out of memory, Query 4.7 restricts the pattern matching duration such that the first and last event of a particular match should be within a given time period. Therefore, if there are no matching events to a partially matched event within the given time period it will get cleaned by the system. This worst-case behavior is analyzed by capturing the maximum supported pattern matching duration as given in Fig. 5.10 for all three types of deployments while scaling the number of nodes.

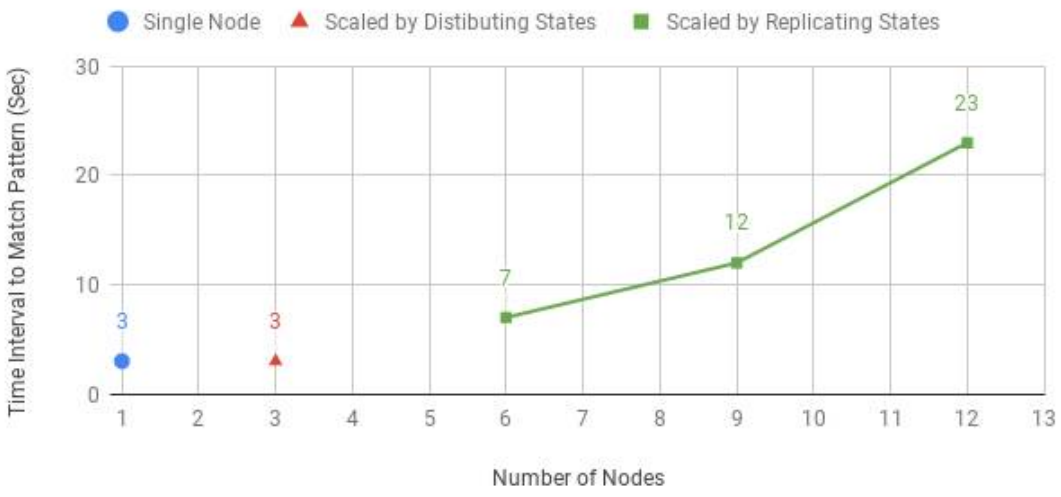

Fig. 5.10: Maximum supported pattern matching duration for worse-case workload.

In this case, as we are sending the input events at a constant rate, the pattern matching time interval will have a positive correlation to the number of internally

created partially matched events. Therefore, we can consider the system is not scalable when we distribute the states to three dedicated nodes by increasing the number of nodes from 1 to 3, as in both cases the system can only sustain three seconds of data. This can be categorized as the state 3 in pattern becoming the bottleneck as it was holding all the data objects that need to be matched against the state 3 of the query. To further analyze this an average-case scenario was introduced. Here the data is published in such a way that nodes having state 2 and state 3 will be holding an equal number of objects that need to be matched against their respective states. This is achieved by sending two events having *amount* < 100 for 50% of the *cardIDs* and sending only one event having *amount* < 100 events to rest of the 50% *cardIDs* this make sure the prior *cardIDs* matches both state 1 and 2 while the latter matches only state 1. The output of this is also analyzed by capturing the maximum supported pattern matching duration as in Fig. 5.11. It can be seen that the supported pattern matching time interval increase with the increasing number of nodes, especially when we move each state to individual nodes. This behavior also correlates with the theoretical analysis of linear memory distribution of $\Theta(e/(s-1))$ between all the nodes except the node containing the initial state. Further, while using the worse-case workloads, we can also conclude that the scalability of the system is limited by the partially matched event objects accumulated at state 3.

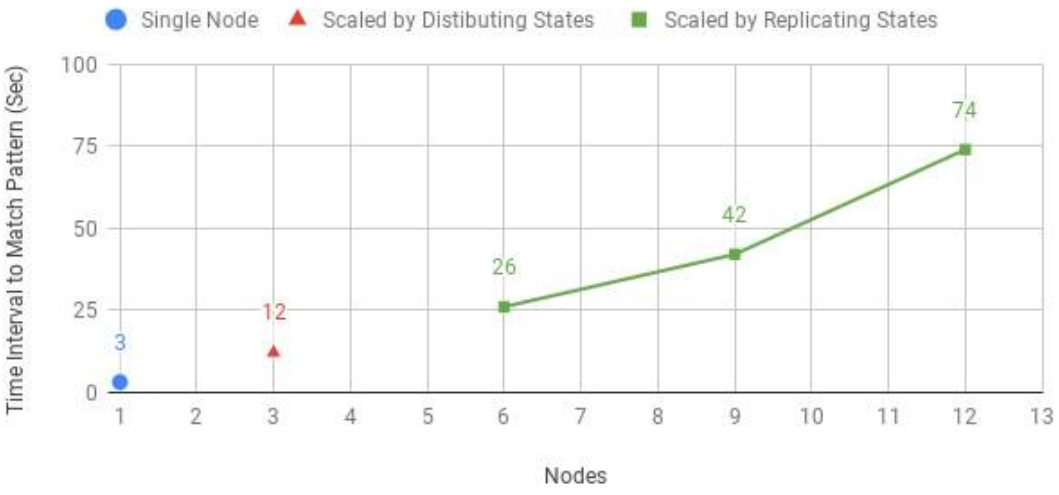

Fig. 5.11: Maximum supported pattern matching duration for average-case workload.

To further analyze the pattern's scalability both bandwidth and CPU utilization were analyzed for the average-case. The results of the analysis are presented in Fig. 5.12 and 5.13, respectively.

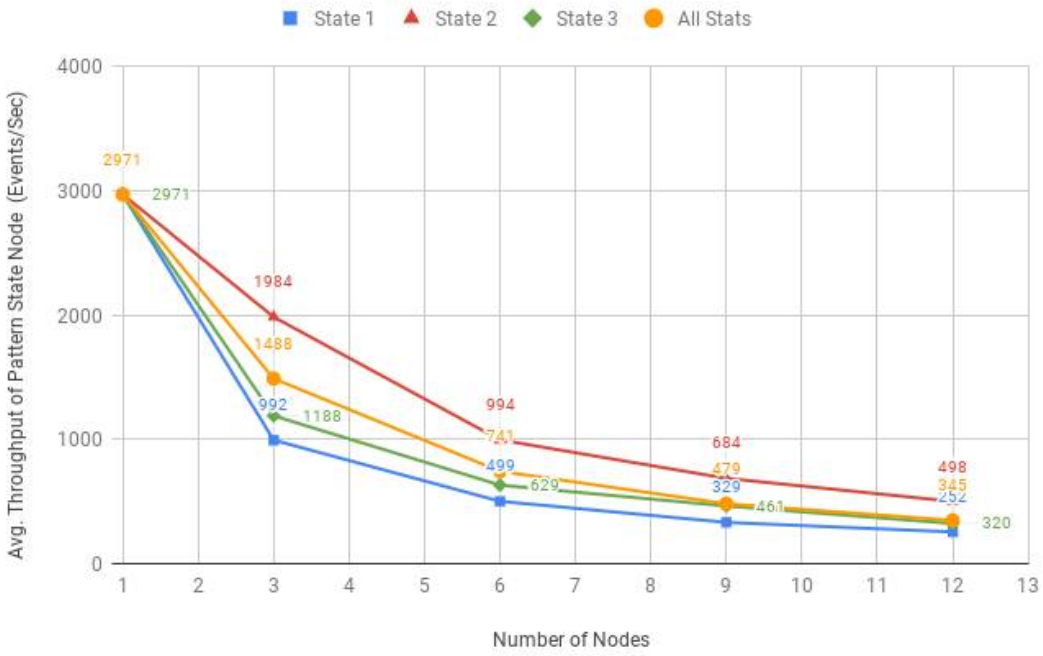

Fig. 5.12: Average throughput of each pattern state node.

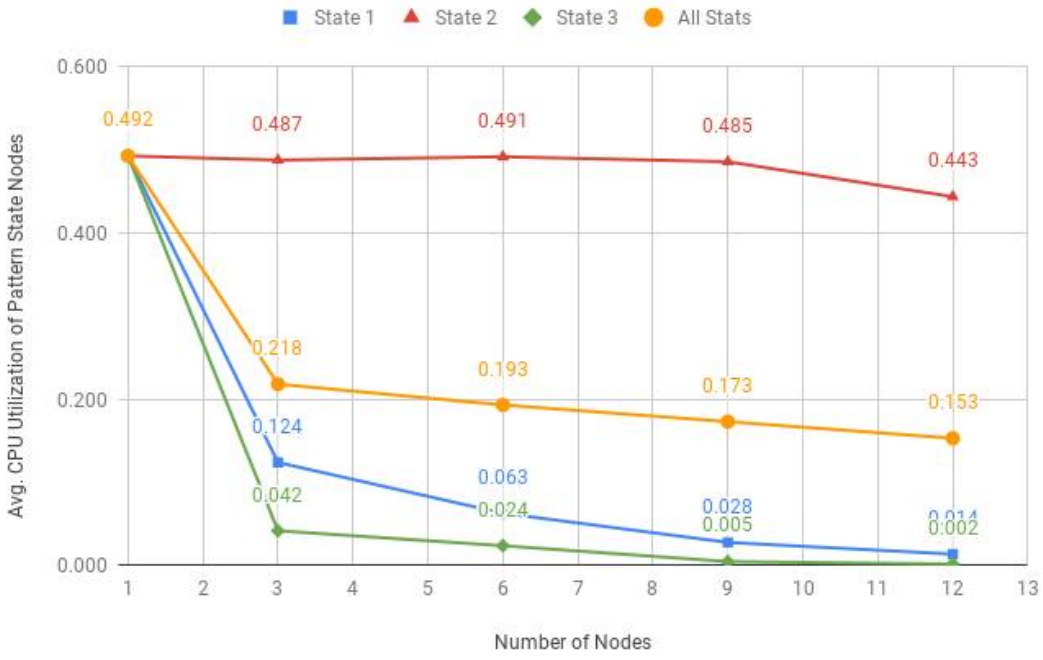

Fig. 5.13: Average CPU utilization of each pattern state node.

Aligning to the expected per node bandwidth $\Omega(\lambda)$, the input bandwidth to the nodes is decreasing as we increase the number of nodes (see Fig. 5.12). Even though the average CPU utilization calculated by considering all the three states together is decreasing as depicted by the All-State series of Fig. 5.13 corresponding to the expected linear theoretical $\Theta(\lambda e/s)$ behavior, the CPU consumption of state 2 is high. This is because state 2 nodes hold more partially-matched events compared to other states, and they need to process all incoming events against those. This is because based on our workload, 50% of the events that do not get matched at state 2 will be stagnated at state 2, and the other 50% of the events not getting matched at state 3 will also get stuck at state 2 for some time before they get matched and reach state 3.

5.3.3 Analysis on scalability of join operation

We conducted an evaluation similar to the window and pattern operations to evaluate the scalability of join operations as well. We evaluated both joining a small window against a larger window and joining two large windows. As we are interested in the worst-case scenarios we simulated events in such a way that all events in the one window will always match against the events in the other window.

5.3.3.1 Joining a small window with large window

To evaluate join against small and large window we constantly kept the small window length size as 20 and only changed the large window size for testing purposes. Here three, five, and nine node deployments were tried and in all the deployments one node is used as an aggregator and other nodes are used to hold the windows and to perform join operations. Through this, we tried to find the maximum large window size the system can hold while we increase the number of nodes as depicted in Fig. 5.14. In the meantime, the average bandwidth of the join nodes is also captured and its reported in Fig. 5.15.

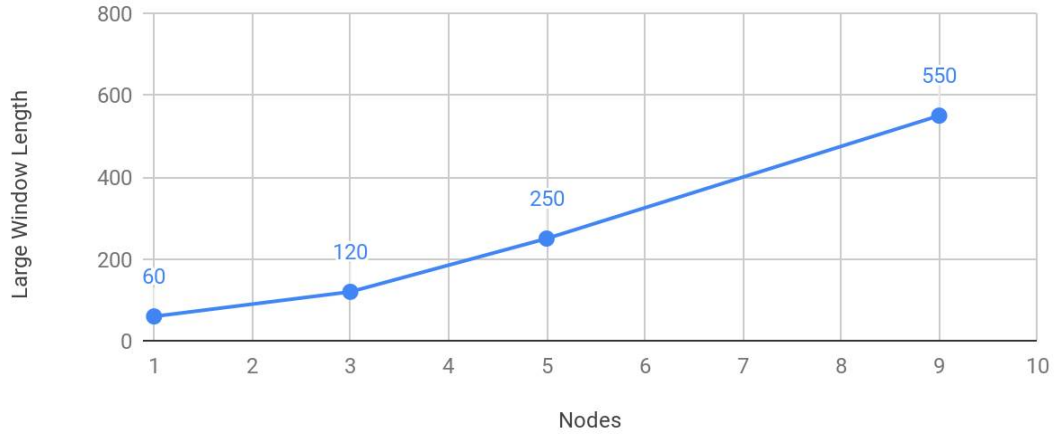

Fig. 5.14: Maximum large window length of the join nodes.

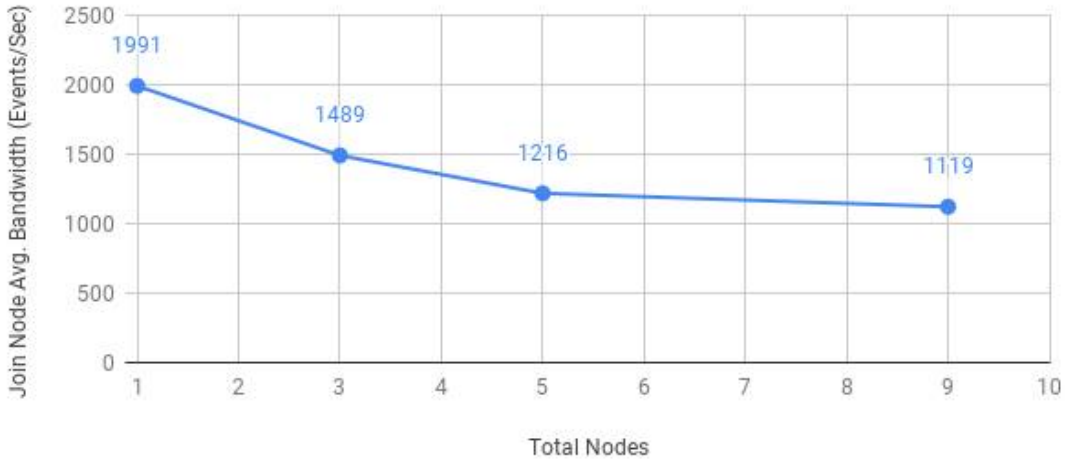

Fig. 5.15: Average bandwidth of the join nodes.

As seen in Fig. 5.14, the proposed system is able to almost linearly scale as we increase the number of nodes, adhering to the theoretically expected scalability of $\Theta(e + e/n)$. This clearly shows that the proposed system was able to distribute the events of the large window evenly across available join nodes. Further, as depicted in Fig. 5.14 the average bandwidth of join nodes is also constantly decreasing aligning to the expected theoretical bandwidth distribution of $\Theta(\lambda + \lambda/n)$, as we are distributing one of the streams across the nodes. Further, as join queries internally contain windows, the behavior of the distributed join correlates with the behavior of the windows.

5.3.3.2 Joining two large windows

Similar to the above use case, two large window joins are also evaluated. Here both the windows are increased at the same level to evaluate the scalability of the system. Here five and ten node deployments are used for testing. In all cases, one node is used as an aggregator and other nodes are used to hold the windows and perform join operations. For the five node case each joining stream is divided into two equal sections such as A1, A2 and B1, B2, and on the four available nodes A1B1, A1B2, A2B1 and A2B2 combination of the sub streams are joined, at the same time in the 10 node case each stream is divided into three equal subsections and they are cross joint in the available nine join nodes.

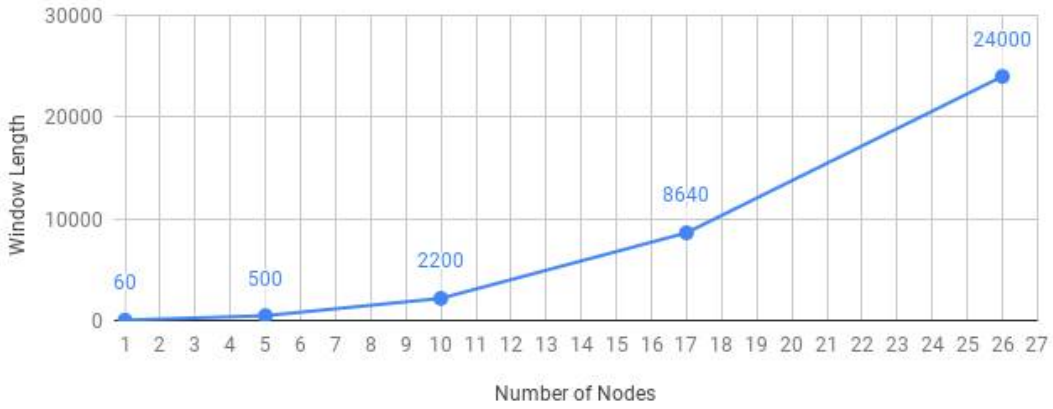

Fig. 5.16: Maximum length of each join window.

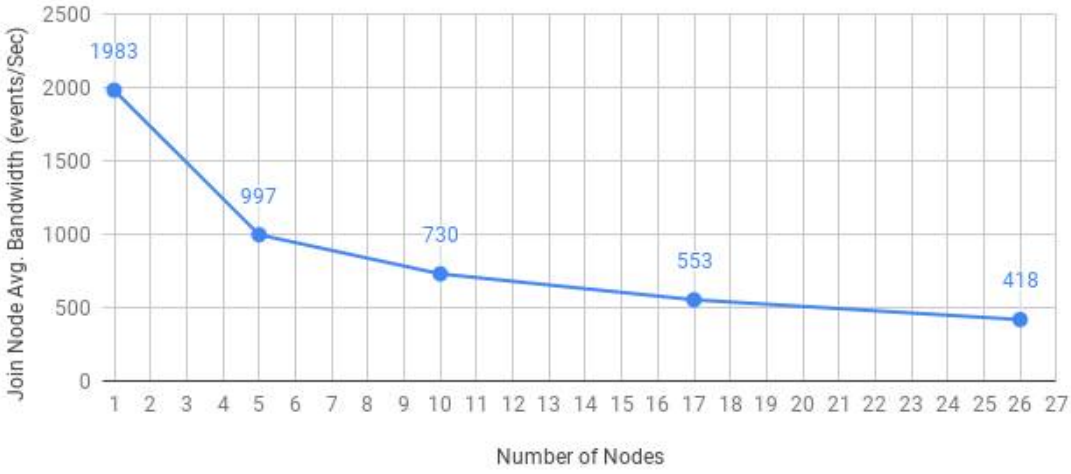

Fig. 5.17: Average join node bandwidth while holding the largest possible length window.

The maximum large window size, the system can hold and the average bandwidth of the join nodes at that time are evaluated to understand the scalability of joins and they are reported in Fig. 5.16 and 5.17, respectively. From these figures, we can see that the system is able to evenly distribute the events of both the join windows across all the nodes and scale the system corresponding approximately to the expected linear theoretical value of as $\Theta(e/n)$. The average bandwidth of the join nodes is also reducing while approximately correlating to the expected linear theoretical value of $O(\lambda/n)$. These results also match the behavior of small-window joins and the scalability characteristics of the windows. Hence, we can conclude joins can be approximately linearly scaled as we increase the number of joining nodes.

5.4 Analysis on latency and accuracy

The latency of all operators is also calculated during the tests. In this test, the latency is calculated by setting the data publishing timestamp to the event and by finding the time difference between the published time and the final event arrival time. Therefore, the calculated latencies also include the time taken to send the events over the network. Fig. 5.18, 5.19, and 5.20 depict the end-to-end latency of sliding time and length window, simple pattern, large and small window, and two large-window-join use cases.

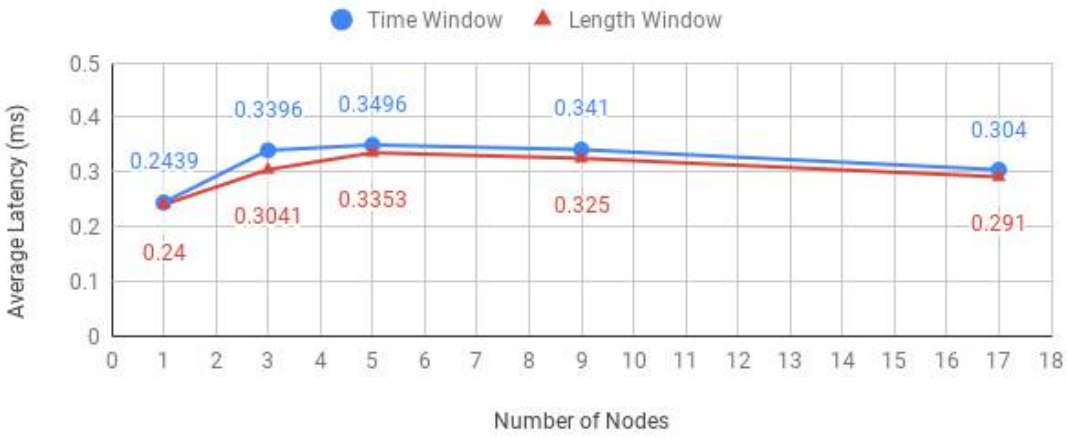

Fig. 5.18: Average latency of the sliding time and length windows.

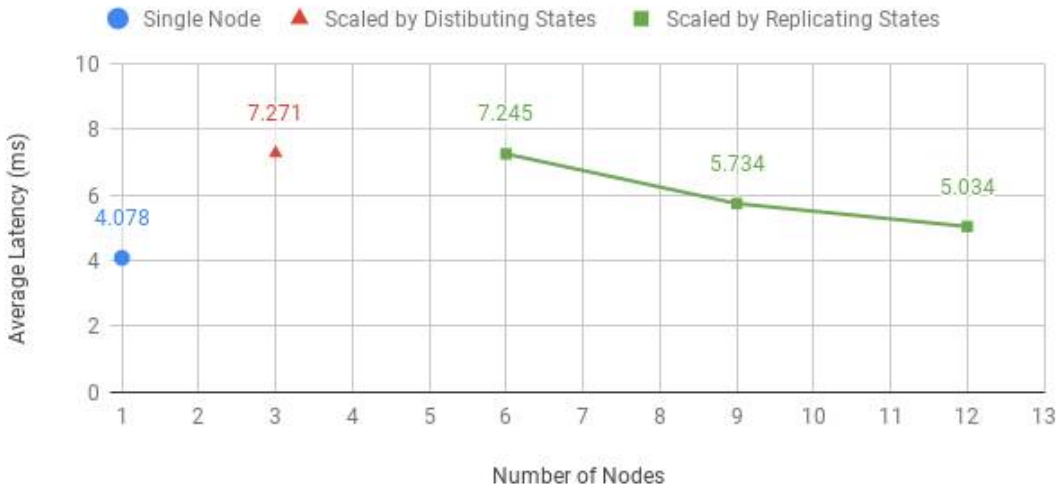

Fig. 5.19: Average latency of simple pattern.

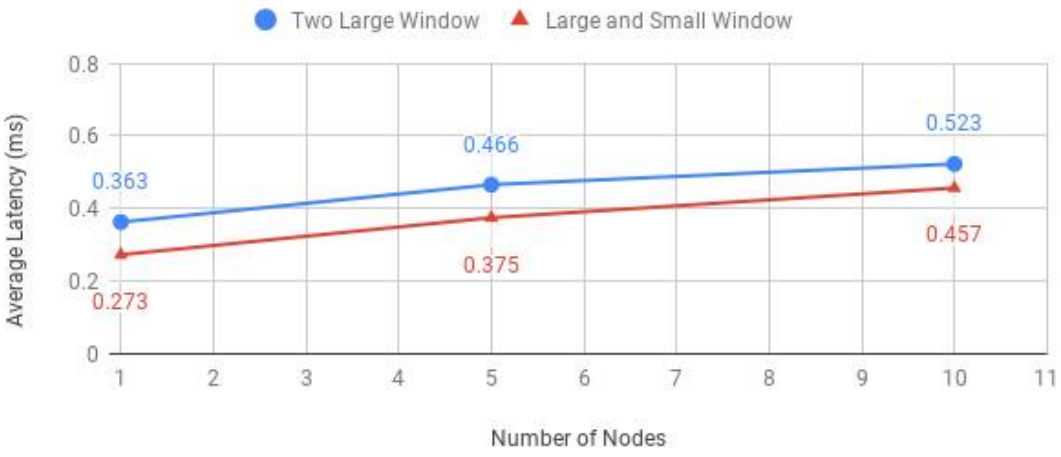

Fig. 5.20: Average latency of small and large window and two large-window joins.

When it comes to the window use case when the system is scaled from one to three nodes the latency increased. But when its scaled beyond the latency gradually reduces. This is because from three nodes onwards the hop count of the end to end event flow has not changed but at the same time overall system load decreases while reducing the latency. Similar behavior is also observed with the pattern use case. In terms of joins, the latency increased between 44% to 67% with the number of nodes. This is because we are storing more events when we scale the nodes the number of events each node needs to match is also increasing and this is contributing to latency.

To understand the accuracy of the final CEP results produced by the system, one test out of each CEP operator was taken and evaluated. In each case, they are tested against the corresponding single node counterpart. Here, five-node scalable time window, six-node scalable simple pattern, and five-node scalable two large window join were evaluated for accuracy. For window and join cases the error is defined by the difference between final aggregation results of the expected single node and scaled solutions, and in the evaluations, all events were given the same weight to make the error reproducible. In the pattern case, the error is defined by the difference in the number of patterns detected by the single node and the scaled solution.

When five node scalable window and five node scalable join are evaluated, we were able to find an out of order event arrival behavior. Such out of order arrivals to the aggregation node caused incorrect aggregation results, and therefore, the implementation was enhanced to reorder events at the aggregation nodes using the K-Slack [34] algorithm. Here the K-Slack algorithm is used, as it buffers events and orders them based on their origin time by limiting the buffer size to worse-case out of order events seen so far. To understand the output accuracy and performance of the system single node, scalable five node, and scalable K-Slack based five node setups are tested for both windows and joins. During the tests as presented in Table 5.1 and 5.2, the percentage of events arriving out of order, the maximum out of order arrival delay, the end to end event latency, and the error in final aggregation are measured while maintaining the throughput at 1000 events per second.

Table 5.1: Accuracy and performance analysis of window queries.

	Single Node Standard Time Window	**5 Node Scalable Time Window**	**5 Node Scalable Time Window with K-Slack**
Max window time interval	15 sec	2 min	2 min
Out of order %	0.00%	0.58%	0.07%
Max out of order (ms)	0	92	30
Avg latency (ms)	0.243	0.349	55.58
Error %	0.00%	0.08%	0.03%

According to Table 5.1 when the system is scaled there is about 0.58% out of order event arrivals coursing 0.08% error as the incorrect output window aggregation values, and when K-Slack is used the out of order event arrivals has been reduced to 0.07% and the incorrect output window aggregation values has been dropped to 0.03% with the additional cost of ~54ms delay in output. As depicted in Table 5.2 a similar behavior was also observed with joins where we were able to improve the accuracy by 5.6% by adding around 260ms of latency.

Table 5.2: Accuracy and performance analysis of join queries.

	Single Node 2 Large Window Join	**5 Node Scalable 2 Large Window Join**	**5 Node Scalable 2 Large Window Join with K-Slack**
Max window length	100	500	500
Out of order %	0.00%	3.40%	0.05%
Max out of order (ms)	0	33	202
Avg latency (ms)	0.366	0.504	261.06
Error %	0.00%	6.60%	1.00%

When accuracy is tested for the pattern, as its scalable implementation already handles pattern matching for out of order event arrival as discussed in Section 3.1.2, there were no errors identified when we scale the system.

5.5 Applicability to other CEP systems

As the system is designed to scale by rewriting the CEP operators in a generic way by defining independent CEP queries and distributing them to various nodes, we will be able to achieve almost identical results to the experimental results produced by Siddhi even if we implement the system using any other CEP engine. In all cases, the system should identify the expected queries that need to run on each distributed node and implement them using an available CEP engine. This approach also allows us to use a different type of CEP engines at various nodes if necessary, and this can become handy if we are using multiple runtime environments in scenarios like IoT. Based on the

analysis we believe if we have used a CEP engine that is implemented on a language that has better garbage collection than Java or that has no garbage collection at all (e.g., C++) the overall performance would have been better.

5.6 Summary

We simulated a dataset for worse-case scenarios for all the operators, and we simulated a dataset with an average-case scenario for patterns. In all cases, we were able to get almost linear scalability when we increase the number of processing nodes. For window and pattern cases we were able to achieve this with no increase in the overall end to end latency while for join we were able to achieve this with lower than linear increase in latency. Most importantly the system was able to produce results with zero errors for the pattern, less than 0.1% of error for windows and with 6.6% of error for joins. By using K-Slack based event reordering algorithm the error rate of the window was able to be reduced to 0.03% just by adding 54ms of delay, and the error rate of join was decreased to 1% by adding 260ms delay.

6. SUMMARY

This section presents the summary of the thesis. Section 6.1 presents the conclusion on the finding of how stateful operations of Complex Event Processing can be scaled when event streams are not partitioned by a key. The research limitations are provided in Section 6.2 and Section 6.3 presents possible future work that can be done in this area.

6.1. Conclusion

Scaling the Complex Event Processing (CEP) is an essential requirement in the data analytics space. Scaling enables handling a large number of CEP queries, running queries that need large working memory, handling a large number of events, complex queries that might not fit within a single machine, and handling a large number of events. In this thesis, we discussed the most common and available scaling approaches that can be used for CEP scaling purposes. They are, running multiple CEP nodes in a cluster, using different types of queries to different CEP nodes, scaling execution via publish/subscribe and scaling events by partitioning each stream and as batches. We have evaluated each of these scaling approaches and analyzed their pros and cons. Based on the analysis, we identified several bottlenecks while scaling stateful CEP operations such as windows, patterns, and joins especially when the streams are not partitioned by a partition key. Scaling such queries on such streams are difficult, as in-memory states need to be maintained between multiple nodes in the cluster.

As the major bottleneck of scaling windows, patterns and joins is storing temporal events at the processing nodes in a scalable manner, we proposed a scatter and gather based solution. In the proposed approach events are sent from a data publisher to various CEP nodes who are connected as a directed acyclic graph. For window and join cases the data is processed in a distributed manner, and the results are collected and aggregated by a single aggregation node to produce the final output. In the case of patterns, the pattern matching and the aggregation happens at each node used in the pattern matching pipeline. Finally, the processed output of windows, joins,

and patterns are sent to a consumer node to calculate the performance and the accuracy of the system. Here the communication between the nodes happened through TCP transport and the distributed data processing is implemented using Java and Siddhi CEP engine and deployed as Docker images in Kubernetes pods. When accuracy-related issues are identified K-Slack algorithm is used to reorder the events.

We conducted various tests to verify the scalability of the proposed solution with a various number of processing nodes. The tests were carried out for a fixed data rate, and with the uniform capacity nodes, to understand the maximum processing capacity of the deployment as we scale the number of nodes. Based on this analysis, in all cases, we were able to achieve almost linear scalability when we increase the number of processing nodes. For window and pattern cases we were able to achieve this with about ~25% increase in the overall end to end latency while for join cases we were able to achieve this with 44% to 67% increase in the latency. Most importantly the system was able to produce this result with zero errors for the pattern, less than 0.1% of error for windows and with 6.6% of errors for joins. By using K-Slack based event reordering algorithm the error rate of the window was able to be reduced to 0.03% just by adding 54ms of delay, and the error rate of joins was decreased to 1% with by adding 260ms delay.

6.2. Research limitations

The proposed solution is implemented using the Siddhi CEP engine which is a Java-based system. Due to Java's garbage collection and threading behavior the system was not able to achieve the calculated theoretical values. This can be overcome by using a CEP engine that is implemented on a language that has better garbage collection than Java or that has no garbage collection at all (e.g., C++).

The current system can only support windows, patterns, and joins that can be implemented using scatter and gather approach, where we split the data, perform summarizations or pattern matching in a distributed manner, and then we combine them for further summarization or pattern matching. Because of this, all MapReduce based aggregation algorithms such as sum, count, min, max, average, and standard

deviation can be easily implemented. However, we cannot process algorithms such as median which does not fit into the scatter and gather approach. Therefore, this approach has a limitation on the algorithms supported. Also, the supported algorithms need to be written in scatter and gather way to support distributed processing possible.

The window and join implementations have a possibility of producing out of order events even when the K-Slack algorithm is used for event reordering. Therefore, the solution may not be useful when we need 100% accuracy for output data. But this system will be useful in most real-world use cases because most of the use cases will be able to tolerate the 1% error produced by the system.

The proposed solution increases the per-event latency compared to the default Siddhi engine. Moreover, this latency depends on the number of network hops and the corresponding data reordering that need to be done. Due to this, our solution might not be the most suitable to produce results instantly. However, in most cases the latency is increased by only a few millisecond ranges; hence, the proposed solution is still applicable in many applications that need near real-time event detection.

Finally, the proposed solution is not implemented to handle data recovery when there are node failures or network failures. Therefore, during failures even though the system was able to recover the nodes using Kubernetes due to data losses it can produce wrong results.

6.3. Future work

As the proposed solution is implemented using Java, due to the garbage collection and threading behavior was not able to achieve the calculated theoretical values. One possible extension is to implement the system using a language that has better garbage collection and threading behavior than Java.

Currently, the distributed processing topology and the sub queries need to be given by the user to scale the system, and as future work, we can implement a compiler that can take a simple single node query and automatically build the distributed query and the connection between the nodes for deployment.

For this thesis, we were using the K-Slack algorithm for reordering out of order events and that was enforcing up to 260ms delay on some cases, at the meantime the system was still producing up to 1% error on the output of window and join use cases, as future work better algorithms should be evaluated and implemented to improve the overall latency and to eliminate the output error of the system.

Even though the system was implemented using Kubernetes the current implementation does not use any of its auto-scaling capabilities to scale the system, and therefore, the system is currently static. As future work, we can implement elasticity to the system such that it can scale based on the load.

The current solution is not implemented to handle data recovery when there are node failures or network failures. Therefore, as future work, this can be eliminated by using periodic data persistence and using data playback techniques during recovery.

REFERENCES

[1] A. Shukla and Y. Simmhan, “Benchmarking distributed stream processing platforms for IoT applications,” in Proc. Technology Conf. on Performance Evaluation and Benchmarking, Springer, July 2016, pp. 90–106.

[2] T. H. R. Munige, “Real time stream processing for Internet of Things and sensing environments,” in Proc. IEEE Intl. Parallel and Distributed Processing Symposium, Sep. 2016, pp. 1143-1152.

[3] “Microservices architecture.” [Online]. Available: http://microservices.io/. [Accessed: 24-Dec-2016].

[4] S. Perera, “Handling Large Scale CEP Usecase with WSO2 CEP” [online], Available: http://srinathsview.blogspot.com/2014/07/handling-large-scale-cep-usecase-with.html [Accessed: 23-Dec-2014].

[5] A. Aalto, “Scalability of Complex Event Processing as a part of a distributed Enterprise Service Bus,” Ph.D. dissertation, Dept. Science., Aalto University, Espoo, Nov 2012.

[6] N. P. Schultz-Mller, M. Migliavacca, and P. Pietzuch, “Distributed complex event processing with query rewriting,” in Proc. 3rd ACM Int. Conf. on Distributed Event-Based Systems, July 2009, pp. 4.

[7] S. Perera, “Srinath’s Blog: My views of the World: How to scale Complex Event Processing (CEP) Systems?” [Online]. Available: http://srinathsview.blogspot.com/2012/05/how-to-scale-complex-event-processing.html. [Accessed: 24-Dec-2016].

[8] J. Dean and S. Ghemawat. “MapReduce: simplified data processing on large clusters.” Communications of the ACM 51, no. 1, Jan. 2008, pp. 107-113.

[9] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. “StreamCloud: An Elastic and Scalable Data Streaming System,”

in Proc. IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 12, Jan. 2012, pp. 2351-2365.

[10] "Apache Kafka." [Online]. Available: http://kafka.apache.org/. [Accessed: 29-Sep-2016].

[11] "Apache Storm." [Online]. Available: http://storm.apache.org/. [Accessed: 29-Sep-2016].

[12] "Spark Streaming — Apache Spark." [Online]. Available: http://spark.apache.org/streaming/. [Accessed: 29-Sep-2016].

[13] "Samza." [Online]. Available: http://samza.apache.org/. [Accessed: 29-Sep-2016].

[14] "Apache Flink: Scalable Batch and Stream Data Processing." [Online]. Available: https://flink.apache.org/. [Accessed: 24-Dec-2016].

[15] V. Govindasamy and P. Thambidura, "An Efficient and Generic Filtering Approach for Uncertain Complex Event Processing," In Proc. Intl. Conf. on Data Mining and Computer Engineering, Dec. 2012, pp. 211-216.

[16] O. Poppe, C. Lei, S. Ahmed, and E. A. Rundensteiner. "Complete Event Trend Detection in High-Rate Event Streams," In Proc. 2017 ACM Int. Conf. on Management of Data (SIGMOD '17). May 2017, pp. 109-124.

[17] A. K. Leghari, M. Wolf, and Y. Zhou, "Efficient Pattern Detection Over a Distributed Framework," in Proc. 8th Int. Workshop on Business Intelligence for the Real-Time Enterprise, Sep. 2014, pp. 133–149.

[18] M. Hirzel, "Partition and compose: parallel complex event processing." In Proc. 6th ACM Int. Conf. on Distributed Event-Based Systems (DEBS '12), July 2012, pp. 191-200.

[19] N. K. Pandey, K. Zhang, S. Weiss, H. Jacobsen, and R. Vitenberg. "Distributed event aggregation for content-based publish/subscribe systems,"

In Proc. 8th ACM Int. Conf. on Distributed Event-Based Systems (DEBS '14), May 2014, pp. 95-106.

[20] “Apache Mesos.” [Online]. Available: http://mesos.apache.org/. [Accessed: 24-Dec-2016].

[21] “Production-Grade Container Orchestration," Kubernetes.io. [Online]. Available: https://kubernetes.io/. [Accessed: 24-Nov-2018].

[22] S. Suhothayan, K. Gajasinghe, I. L. Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, “Siddhi: A Second Look at Complex Event Processing Architectures,” in Proc. ACM Workshop on Gateway Computing Environments, Nov. 2011, pp. 43-50.

[23] “wso2/siddhi,” GitHub. [Online]. Available: https://github.com/wso2/siddhi. [Accessed: 24-Dec-2016].

[24] M. A. Fardbastani, and M. Sharifi, “Scalable complex event processing using adaptive load balancing,” Journal of Systems and Software, Mar. 2019, vol. 149, pp. 305-317.

[25] “Complex Event Processor | WSO2 Inc.” [Online]. Available: http://wso2.com/products/complex-event-processor/. [Accessed: 24-Dec-2016].

[26] M. Cherniack et al., “Scalable Distributed Stream Processing,” in Proc. CIDR Conf., Jan. 2003, vol. 3, pp. 257-268.

[27] V. Govindasamy, and P. Thambidurai, “Complex Event Processing - A Survey,” Journal of Computing, April 2013, vol. 5.

[28] K. Vikram, “Finger Lakes: A Distributed Event Stream Monitoring System,” 2007.

[29] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, “S4: distributed stream computing platform,” in Proc. IEEE Int. Conf. on Data Mining Workshops, Dec. 2010, pp. 170-177.

[30] “Understanding Scalability,” Docs.oracle.com. [Online]. Available: https://docs.oracle.com/cd/E23943_01/dev.1111/e14301/scalunder.htm#CEPED1975. [Accessed: 07-Dec-2018].

[31] L. Brenna, J. Gehrke, M. Hong, and D. Johansen. “Distributed event stream processing with non-deterministic finite automata”. In Proc. ACM Int. Conf. on Distributed Event-Based Systems (DEBS '09), July 2009.

[32] K. H. Lee, Y. J. Lee, H. Choi, Y. D. Chung, and B. Moon, “Parallel Data Processing with MapReduce: A Survey,” ACM SIGMOD Record, vol. 40, no. 4, Jan. 2012, pp. 11–20.

[33] R. Ananthanarayanan et al., “Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams,” in Proc. ACM SIGMOD Int. Conf. on Management of Data, June 2013, pp. 577–588.

[34] M. Li et al., “Event Stream Processing with Out-of-Order Data Arrival,” in Proc. 27th Int. Conf. on Distributed Computing Systems Workshops, June 2007, pp. 67–74.

[35] “Siddhi IO TCP,” Wso2-extensions.github.io. [Online]. Available: https://wso2-extensions.github.io/siddhi-io-tcp/. [Accessed: 30-Nov-2018].

[36] “Docker.” [Online]. Available: https://www.docker.com/. [Accessed: 29-Sep-2016].