Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 0202-0010

Data Parallelization in Complex Event Processing Without a Dedicated Splitter

Qing Lu

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dipl.-Inf. Ruben Mayer
Commenced:	January 18, 2016
Completed:	July 19, 2016
CR-Classification:	C.2.4 Distributed applications

Abstract

With the popularity of Internet of Things(IoT), Complex Event Processing (or CEP) shows its power in detecting specified patterns from input event stream. There are existing parallel CEP architectures to improve the capacity of CEP system. The major data parallel CEP architecture is the Split-Process-Merge architecture, which is able to provide unbounded parallelism degree. However, it has limitation when the splitting decision becomes computational heavy, which leads the splitter becoming a bottleneck. E.g. splitting decision depends on comparing two images to check if they contain the same object such as a person. The result is that the single splitter, instead of operator instances, is doing the computational expensive job. To help analyze the cause of "heavy" splitting decision, this thesis proposes an Extended SNOOP query language, which combines features from both SNOOP and TESLA, two of the leading event specification languages. Then this thesis derives an architecture, which avoids the splitting decision, from Split-Process-Merge architecture. The Split-Process-Merge architecture splits the input event stream into sub-streams and each operator instance handles one or more sub-streams. Instead, the new architecture creates Tasks by combining every incoming event to all existing Partial Matches, and operator instances process the Tasks. The Task Creation Algorithm is content independent. It won't check the content, like the image data, in events. Therefore, the computational heavy splitting decision is avoided. Together with this thesis, an example implementation of new architecture for a specific query is given. The Evaluation results of implementation show the new architecture obtains a good scalability as number of CPU cores increasing and as the cost of operation increasing.

Acknowledgment

I would like to take this opportunity to thank the Distributed Systems department represented by Prof. Kurt Rothermel for giving me the chance to write my Master Thesis.

A deep appreciation for my supervisor Ruben Mayer who guided me the direction and gave me much inspiration during my work.

I am also grateful for the assist from my family and friends. They encouraged me a lot when I had hard time.

It would be impossible to finish this thesis if I didn't received the support from them. I would like to express my gratitude here again.

Qing Lu
Esslingen, 12. July. 2016

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

The idea of Complex Event Processing (or CEP) was raised decades ago. The concept CEP is derived from discrete event simulation, and active database area. The functionality of active database is limited since it is based on a database system. Data needs to be stored in database before they can be processed, which will lead to high I/O latency. The expressiveness is limited in active database because there are only limited operators supported to express the relationship between events. Maybe these operators were enough for the scope of database, but they are far from enough for the world outside database. Therefore, CEP focus on processing event stream instead of database and it needs more operators to express better relationship between events. The source of events can be anywhere and everywhere. In addition, the size of an event stream is unbounded. It is impossible to store all data first and then to process them. CEP needs to process the events as soon as they arrive. Also, CEP doesn't keep the working data in hard disk, but just keep them in memory which has much higher I/O speed. Once the events have been processed, these events can be just discarded. Of course, it is possible to record the events for other purpose, but for CEP system, these data are useless after being processed.

CEP is used mainly in Business Intelligence or Operational Intelligence. A major usage of CEP is to detect certain patterns in an event stream. These patterns are identified as opportunities or threats by domain experts. A very good example of CEP application is algorithmic stock-trading[Luc02][Lun06]. The domain experts describe the patterns when to trigger a "sell" or "buy" activity and the CEP system finds out all these opportunities by detecting the patterns from event stream.

In last ten to twenty years, the Internet of Things(IoT) had its boom times. With the rise of IoT, CEP shows its power in analyzing the data from RFID, and/or sensor networking because of the capability of processing large amount of continuous event stream.[WSK08][Dun09][YCL11] CEP is used to detect patterns in the field of environment monitoring, traffic management, and also business process management. Domain experts only need to define the pattern rules, CEP will find the complex events from hundreds of thousands simple raw events and then present the valuable information to the user. Nowadays, the artificial intelligence has been developed rapidly. Maybe in future, the pattern rules will be generated from artificial intelligence more than human beings. This is another story beyond this thesis.

Besides the growth of IoT, the Internet itself grows explosively as well. More accurately, the amount of content on the Internet keeps exploding in recent years. [Jam12] shows the data created every minute and that was in 2012, 4 year ago. CEP has already shown its power in pattern detection. Is it possible for CEP to detect patterns from the whole Internet instead of just IoT? Or how dose CEP perform if incoming events containing images, audio, and video, etc.? E.g. [HLR+13] gives examples about "vehicle tracking" and "traffic monitoring" using Mobile CEP. To achieve this target, CEP needs to handle "heavier" data than before. Previously, the data CEP processed are mainly primary data types, i.e. integers, floating numbers, boolean values, and strings, etc. The operations against these data types,e.g. adding two numbers, finding a sub-string in a string, are quite "cheap" since only several nanoseconds are needed. On the other hand, the content on the Internet is much "heavier",e.g. articles, images, audios, and even videos. Computer needs much more time, from several milliseconds to hundreds of milliseconds, to process these data. To handle a large amount of "heavy" content, a parallel architecture is always preferred. However, this thesis will show most current popular parallel CEP architectures are not suitable for this kind of "heavy" content and a new parallel CEP architecture for "heavy" content will be proposed in later chapters.

Thesis Structure

The structure of this thesis is as following:

Chapter 2 – Background: In this chapter, some CEP systems as well as parallel CEP architecture will be introduced.

Chapter 3 – "Heavy" Splitter Problem: In this chapter, "heavy" splitter problem in the Split-Process-Merge architecture will be put forward and analyzed. To help analyze the problem, a pseudo CEP language as well as some application examples will be given.

Chapter 4 – Approach to Create New Architecture: This chapter will show the approach how to obtain the major components of new architecture from Split-Process-Merge architecture step by step.

Chapter 5 – Details of New Architecture: This chapter will show the details about the components of new architecture. Related data structure, algorithms, and APIs will also be introduced.

Chapter 6 – Evaluation: Evaluations about the performance of new architecture will be in chapter 6.

Chapter 7 – Conclusion and Outlook: The last chapter will list some possible direction of further work.

2 Background

2.1 Background of CEP Systems and Query Languages

As mentioned in previous chapter, CEP has become popular in last tens of years. The survey paper from [CM12b] has excellent historical stories about different CEP systems. At very beginning, CEP is derived from active database, e.g. HiPac[DBB+88], Ode[LGA96], Samos[GGD91], and SNOOP[CM94]. People wanted to detect specified patterns from the modification of database. For instance, people wanted the database could automatically raise an alert when some values in database matched certain rules, instead of inquiring the database manually. This is the function of active database. Most of active database only supports internal events, i.e. the modification on database. Samos and SNOOP support both internal events and external events as the input. They can detect patterns from events inside database as well as outside database, such as events from sensors. SNOOP is also independent from the database model, which makes it easy to modify to use it outside the database management system. SNOOP provides selection and consumption policy in addition, while the other three do not support this function. If multiple combination of events matches the pattern, the selection policy specifies which combination to choose. For example, there are four events in the event stream, A_1, A_2, B, and C. The index indicates the order of events so as to distinguish two A events. Both A_1BC and A_2BC will match the pattern ABC. Then with *first occurrence* selection policy, a matched result A_1BC will be fired, while with *last occurrence* selection policy, A_2BC will be fired. After a matched result is fired, the consumption policy defines what to do with the events in a matched result. E.g. Selected consumption policy will cause every event in a matched result to be consumed. None of them will ever appear in other later matched results. SNOOP achieves this by constructing and traversing a tree. However, this limits the possibilities of selection and consumption policy. For instance, SNOOP doesn't support to specify selecting the first event A and the last event B. Also SNOOP cannot consume part of selected events, e.g. just consume event A but not event B.

Later, Complex Event Processing Systems are proposed. CEP system is designed to detect patterns from one or more event streams rather than events in a database. The source of events can be anywhere, e.g. stock market, sensors or the Internet. There are lots of

CEP system been present. I would like to emphasis three of them: Amit, Cordies, and TESLA. Amit[AE04] inherits many features from SNOOP, which is a very expressive and flexible event detecting language. Amit also introduces the $lifespan$, a time window for detecting a pattern, which is not supported in SNOOP. However, compared to Cordies and TESLA, Amit, as a commercial product, doesn't support the functionality that allow the user to define and implement an arbitrary evaluation operation. Cordies[KKR10], another CEP system, has good expressiveness in describe arbitrary relationship between events. In addition, Cordies provides functionality for user to define new operators, although in the original paper, these operators are simple arithmetic operations or logic operations. However, the syntax of Cordies seems more complex than other CEP query languages, such as SNOOP. While, TESLA[CM10], as a general purpose CEP system as well, has less expressiveness then SNOOP but better flexibility. TESLA doesn't support every operator in SNOOP, which decreases the expressiveness, but the pattern detecting algorithm allows it to perform a more flexible selection and consumption strategy.

Since SNOOP has higher expressiveness in query language, besides SNOOP is quite similar to SQL which is also a very popular query language, I will derive a pseudo CEP language, Extended SNOOP, by adding the features from TESLA and T-REX ([CM12a] an implementation of TESLA), which has a better detecting algorithm, into SNOOP. This pseudo CEP language will be described in Chapter 3.1 and it can help us to express the examples in later chapters. Considering the pattern detecting algorithm, TESLA and T-REX show a better mechanism to do so. Therefore, in later chapters, the pattern detecting mechanism will be based on TESLA and T-REX model.

2.2 CEP Architecture

Considering the data rates in high frequency trading(HFT), a stock-trading algorithm, large amount of trades and quotes can be generated per second. Since New York Stock Exchange stops publishing the volume of contracts made by stock-trading algorithm, the real amount becomes mysterious. However, [Aga12] reports that in the May 6-th Flash Crash in 2010, HFT contributed 49% out of 27,000 contracts within 14 seconds. That still is a large event rate for a single machine. Also in the field of IoT, even if the network has 100 sensors which send 1 event per millisecond per sensor, the total event rate would be 100,000 event per second. In the era of data explosion, it is hard for a single machine to process such an event stream. Therefore, efficient parallel CEP architecture is necessary to enable CEP system to handle an event stream with high data rate.

There are works about parallel CEP architecture. These works can be distinguished between intra-operator parallelization, and data parallelization. [WDR06] and [GJP+12] show the approaches to obtain good parallelization for specific operators. [BDWT13]

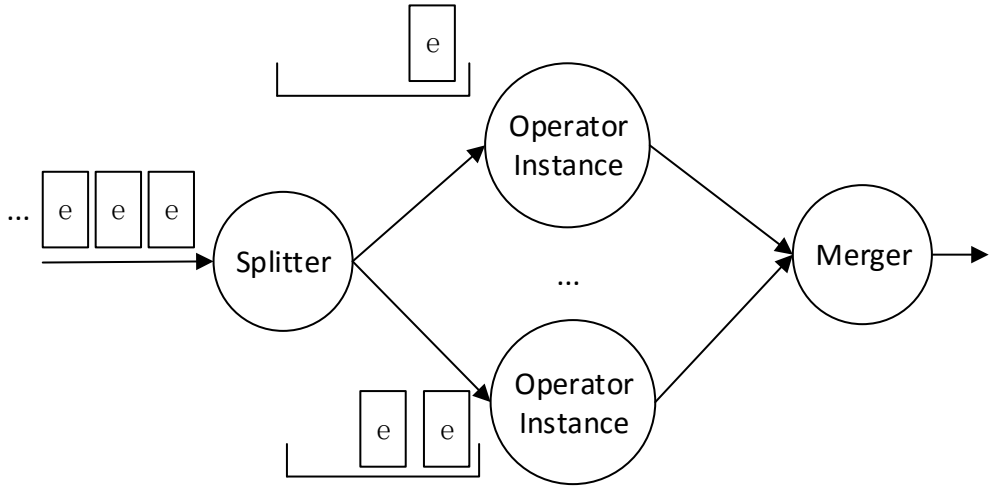


Figure 2.1: Split-Process-Merger Architecture

also mentions a state-based parallelization method. The interesting idea of these three works is to convert a query in event specification language, e.g. SNOOP, to a finite state machine. This leads to the state-based parallelization, which is easy to pipe-lined parallel execute. Each processing unit handles one state in the finite state machine model. However, the intra-operator parallelization has its limitation. The parallelism degree is bounded by the number of states in the finite state machine model, which is derived from the query. Therefore, the parallelism degree is query dependent.

Another parallelization method is data parallelization. One of the most popular data parallelization architecture is the Split-Progress-Merge architecture(Figure 2.1). The general idea of Split-Process-Merger architecture is to split the event stream into partitions (i.e. sub-streams, a.k.a. selection) by a "splitter". For instance, paper[KMR+13] shows a way to define the start and end of a partition. Each partition can be independently executed against the query by an operator instance. Therefore, all processing units are identical and arbitrary number of operator instances can be assigned to one event stream. Thus, the parallelism degree is no longer bounded by query.

To keep the results from the Split-Process-Merge architecture consistent as the results from sequential processing architecture, a "merger" is needed as well as some rules must be obeyed by splitting algorithm. The "merger" sorts the results from each operator instance so that the matched results from the Split-Process-Merge architecture have the same order as the matched results from a sequential processing architecture. Of course, the union of matched results from each operator instance should be no more (i.e. no false positive) or no less (i.e. no false negative) than the matched results from a sequential processing architecture. To satisfy this requirement, the "splitter" must

assign the events exactly needed to detect a pattern to each partition. If some events are missing in a partition, it may cause false negative. If some extra events are assigned to a partition, false positive may happen. If all requirements above are satisfied, then the parallelization is transparent to outside.

However, most data parallelization CEP system splits the event stream based on key-based fission models[BEH+10][ASF16][IBY+07]. Dynamic changes to the assignment of key values to operator instances is hard or not allowed at all. Also for some operators, a key to group events belonging to a pattern of interest is hard to find. There are works, [BDWT13][MKR15][MKR14], to improve this issue by using a pattern-sensitive fission model in splitter and, in addition, there are works, e.g.[MMTR16][OKRR13], to improve the performance of operator instance so that the overall performance can be improved. While, it's hard to find works discussing about a computational heavy splitting decision.

For example, in a scenario that we need to put two events containing the same object into the same partition. It doesn't cost much time for the "splitter" to split events based on meta-data such as time-stamp or event id. However, it costs much time for the "splitter" to split events based on the content, such as similarity between two images. Because, when an event containing an image comes in, it is hard for the splitter to decide what is the other event to compare with the incoming event. In most cases, the incoming event should be compared to all existing events in the system. Besides, the computation of similarity between two images itself is already computational expensive.

Therefore, the computational heavy splitting decision will lead the single "splitter" to the bottleneck of the architecture, which is this thesis focus on. This thesis will try to find out the cause of a "heavy" splitting decision and then propose a new architecture to avoid such "heavy" splitter in following chapters.

3 "Heavy" Splitter Problem

In this chapter, I will try to find out the cause of a "heavy" splitter. To help analyze the problem, first I will introduce a pseudo CEP language, extended SNOOP. This pseudo CEP language tries to combine the features from both SNOOP[CM94] and TESLA[CM10]. Then several example applications are given and analyzed by using the pseudo CEP language. Some of them have a "heavy" splitter while the others do not. At the end of the chapter, I will try to classify the "heavy" splitter problem according to the examples and then give the goal of this thesis.

3.1 Pseudo CEP Language "Extended SNOOP"

In this section, I will analyze the syntax of query languages from SNOOP and TESLA. The differences between these two query languages will be pointed out. Then, the Extended SNOOP query language will be proposed. The Extended SNOOP query language can help us to understand the cause of a "heavy" splitter in later sections.

3.1.1 Syntax of SNOOP and TESLA

Algorithm 3.1 and 3.2 are the syntax of SNOOP and TESLA query. There are five major differences as following.

Algorithmus 3.1 Syntax of SNOOP

```
SELECT *
FROM Pattern
WHERE
    Constraints (Key-based|Batch-based|Content-based)
WITHIN
RETURN
PARAMETER-CONTEXT
```

Algorithmus 3.2 Syntax of TESLA

```
DEFINE ComplexEvent
FROM
    SimpleEvents with Constraints
WHERE
    Assign values to Complex Event
CONSUMING Consumed Events
```

1. The pattern to be detected is explicitly written in the $FROM$ clause of SNOOP, but hidden in $FROM$ clause of TESLA.

2. TESLA introduces event composition operators, $each - within$, $first - within$, and $last - within$. These operators can nicely express event selection strategy.

   SNOOP only offers very simple global selection strategy by PARAMETER CONTEXT. If a pattern $SEQ(A; B; C)$ needs to be detected and we would like to fire a complex event with first A, last B, and each C, then this is not supported in SNOOP.

3. Although TESLA introduces event composition operators which can nicely express event selection strategy, as discussed in number 2, TESLA has limitation in expressing an order-irrelevant query. For example, the ANY operator in SNOOP: $ANY(2, A, B, C)$.

   We don't care the order of events A, B, and C. We just want to detect if any two of these three events happened. In TESLA we need write the query as algorithm 3.3.

   If we require more, we want to select last A, first B, and each C with a total time windows size $5\ seconds$ in the event stream. We already know this cannot be supported in SNOOP as discussed in number 2, but in TESLA, the query will be so complex as algorithm 3.4.

   In this scenario, both query language has their limitation.

4. TESLA offers "hierarchies of events" and "iterations". A complex event definition can be reused in the constraints of another complex event definition. SNOOP doesn't support this feature.

5. TESLA can explicitly express the consuming policy in $CONSUMING$ clause. Individual event in a set of involved events can be consumed respectively.

   SNOOP can only consume the whole set of involved events. If a pattern $SEQ(A; B; C)$ would like to be detected and only A is need to consume, then this is not supported in SNOOP.

Algorithmus 3.3 A complex example query of TESLA

```
DEFINE AnyTwoOfABC
FROM
    (A AND B) OR
    (B AND C) OR
    (A AND C)
```

Algorithmus 3.4 A more complex example query of TESLA

```
DEFINE AnyTwoOfABC
FROM
    ((A
    first B within 5 seconds from A AND
    each C within 5 seconds from A) OR
    (B
    last A within 5 seconds from B AND
    each C within 5 seconds from B) OR
    (C
    last A within 5 seconds from C AND
    first B within 5 seconds from C)) AND
    A within 5 seconds from B AND
    B within 5 seconds from C AND
    C within 5 seconds from A AND
```

3.1.2 Extended SNOOP Query Language

Both SNOOP and TESLA has their limitation, but either of them could be prefect complement to the other one. Therefore, I would like to combine both features of SNOOP and TESLA.

As I have mentioned before, I would like to use the pattern detecting algorithm from TESLA, which uses finite state machine, it is easy for me to directly derive a state machine model from query with an explicit “PATTERN” clause. Therefore, I would like to focus on SNOOP and extend it with some features from TESLA. Also SNOOP has SQL style syntax, which is another advantage compared to TESLA. Thus, the issue mentioned in difference number 1 is solved with no more efforts.

The two features I would like to extend first are the issues mentioned in difference number 2 and difference number 5. Because SNOOP supports some similar strategies but very simple ones. It can increase expressiveness if these features are extended. The selection strategy can also solve the issue in difference number 3.

Algorithmus 3.5 Syntax of Extended SNOOP

```
SELECT *
FROM Pattern
WHERE
    FILTER
        Constraints Part1
    Constraints Part 2
    PICK each-within-from| first-within-from| last-within-from
WITHIN
RETURN
CONSUMING the events need to be consumed
```

Issue mentioned in difference number 4 does not interest me very much since I think this feature can be achieved by a chain of pattern definition in SNOOP.

Algorithm 3.5 shows the syntax of Extended SNOOP. The whole structure of query is kept still as the syntax of SNOOP, except three modifications are made.

1. I split the original constraints in $WHERE$ clause into two parts. One part is $FILTER$ clause. I want put all constraints that semantically indicate two events representing the same object into this clause. In most cases, these constraints are checking if two fields in two events are equal or checking the type, source, etc. of the event. E.g. $e1.id = e2.id$, $e1.photoData\ matches\ e2.photoData$, $e1.source = ABC$. Semantically, this means event e1 and event e2 carry the information about the same object.

   The purpose of this modification is that, in Split-Process-Merge architecture, the splitter distributes the events based on the FILTER-style constraints in most cases. However, there are some cases the splitter doesn't work fine. I introduce the $FILTER$ clause so that I can analyze what kind of constraints limits the performance of splitter.

   From the functionality view, the $FILTER$ hasn't any function. It just helps us to analyze the cause of a "heavy" splitter in later sections.

2. I introduce the $PICK$ clause. This clause is taken from event selection strategy in TESLA. It should have the same function as in TESLA. Each constraint in $PICK$ clause indicates a selection strategy for an event.

   Since there is already a key word $SELECT$ in SNOOP, I use $PICK$ instead.

3. The $CONSUMING$ clause is as same as the $CONSUMING$ clause in TESLA. This clause indicates what sub-set in selected events should be consumed.

In next section, I will try to use some application examples to demonstrate the Extended SNOOP.

3.2 Application Examples

In this section, five application examples are given. In each example, a target is defined and one or more queries are given. The first two applications are more complex than the rest. Therefore, more details are described in first two applications. The next section following application examples, is the analysis of these five examples.

3.2.1 Application 1

Object Recognition and behavior pattern detection for auto-driving/alarm system

Target:

Designed for Auto-driving car or component of alarm system in car. The system should avoid collision or give alarm when the behavior pattern of others leads to a potential dangerous situation.

Example:

When the traffic light turns red, an object (a human being, car, or bicycle, etc.) is still moving fast than 5km/h (this speed is w.r.t human walking speed.) and is really close to crossroad. The system should give warning about a potential dangerous situation that the object may break the traffic law and invade into your normal path. You should execute emergency break or evasion.

Solution:

(HD) Cameras are need to catch real time photos of environment around the car. Then the photos are pre-processed into small photos. Each small photo contains one object (e.g. a human, car, bicycle, traffic sign, etc.). Each small photo should also contain the information about the position in the original photo. A CEP system can be used to analyze these small photos. Each small photo is one event. Those events contain the same object should be linked together to calculate the speed and direction of that object. A user-defined function “match” is needed. This function gives the possibility if two events contain the same object. Then these two events are used to calculate the speed and direction of the object they represent. Now a complex event “One Object” is created. “One Object” is processed with other events. In the event stream, within 5 seconds from the timestamp of “One Object”, if the CEP find one event which contains the traffic light

Algorithmus 3.6 TESLA Query for Application 1

DEFINE $OneObject(speed[3], direction[3])$
FROM
$Event_2(PhotoData = \$x)$ AND
last $Event_1(PhotoDatamatch\$x)$ within 5 $seconds$ from $Event_2$
WHERE
$calculate\ the\ speed[3]\ and\ direction[3]$
CONSUMING
$Event_1$
DEFINE $MayCollide()$
FROM
$OneObject(Speed > 5km/h,\ direction\ towards\ my\ path)$ AND
last $ItsTrafficeLight(Value = Red)$ within 5 $seconds$ from $OneObject$

Algorithmus 3.7 Extended SNOOP Query for Application 1

SELECT $*$
FROM $ALL(SEQ(E_1; E_2), E_3)$
WHERE
FILTER
$E_1.photoData\ matches\ E_2.photoData$
$E_3.photoData\ matches\ "TrafficLight"$
$CalculateSpeed(E_1, E_2) > 5km/h$
$CalculateDirection(E_1, E_2)\ towards\ my\ path$
$E_3.light = "Red"$
PICK last E_1 within 5 $seconds$ from E_2
WITHIN 5 $seconds$
RETURN
$MayCollideEvent$
CONSUMING E_1

for “One Object” and the traffic light is red. At the same time, “One Object” is moving faster than 5km/h and towards to your normal path. A complex event “May Collide” is created.

Algorithm 3.6 and 3.7 show the TESLA Query and Extended SNOOP Query can be used in this Scenario respectively. In TESLA, “last-within-from” is used to select traffic light event. But in this case we can only select the traffic light event happens before OneObject Event. In Extended SNOOP, we are using “ALL” pattern and with global “WITHIN” clause to constraint the size of time window. Then we can cover both situation that traffic light event E_3 happens either before or after events E_1 and E_2.

Algorithmus 3.8 TESLA Query for Application 2

DEFINE $SimilarNews(Content_1, Content_2, Similarity)$
FROM
$Event_1(Content = \$x)$ AND
each $Event_2(CalculateSimilarity(Content, \$x) > 80\%)$ within 1 $hour$ from $Event_1$
WHERE
$Content_1 = Event_1.Content$
$Content_2 = Event_2.Content$
$Similarity = CalculateSimilarity(Event_1.Content, Event_2.Content)$

3.2.2 Application 2

Fetch common content from different web source.

Target:

A system tries to monitor different web content sources. When common content appears w.r.t. certain rules, the system executes some reactions.

Example:

A system gets information from certain major News sources (e.g. BBC, CNN, ABC). When these major News sources publish similar main news within a short range of time, and the news hasn't been published before, then the system could assume there is some big (global/ domestic) news just happened.

Solution:

In this case, articles from major News sources can be the events of CEP. Each News source can be one event stream. CEP system analyzes the similarity between those events in different streams. If CEP finds similar events in different streams within a short range of time, then CEP may find a newly happened (global/ domestic) news.

Algorithms 3.9, 3.8, and 3.10 show the algorithms can be used in this scenario. Function $CalculateSimilarity(Content_1, Content_2)$ is a user defined function. Domain knowledge is needed to implement it. Function $IsFirstTimeAppear(Event.newsContent)$ could be implemented by calculating similarity of new content against history news.

Algorithmus 3.9 SNOOP Query for Application 2

SELECT $*$
FROM $ALL(E_1, E_2, E_3)$
WHERE
    FILTER
        $E_1.source \neq E_2.source$
        $E_1.source \neq E_3.source$
        $E_2.source \neq E_3.source$
        $CalculateSimilarity(E_1.newsContent, E_2.newsContent) > 80\%$
        $CalculateSimilarity(E_2.newsContent, E_3.newsContent) > 80\%$
        $CalculateSimilarity(E_1.newsContent, E_3.newsContent) > 80\%$
    $IsFirstTimeAppear(E_1.newsContent) > 50\%$
    $IsFirstTimeAppear(E_2.newsContent) > 50\%$
    $IsFirstTimeAppear(E_3.newsContent) > 50\%$
WITHIN 1 *hour*
RETURN
    Recently happened news E_1, E_2, E_3

Algorithmus 3.10 Extended SNOOP Query for Application 2

SELECT $*$
FROM $ALL(E_1, E_2, E_3)$
WHERE
    FILTER
        $E_1.source \neq E_2.source$
        $E_1.source \neq E_3.source$
        $E_2.source \neq E_3.source$
        $CalculateSimilarity(E_1.newsContent, E_2.newsContent) > 80\%$
        $CalculateSimilarity(E_2.newsContent, E_3.newsContent) > 80\%$
        $CalculateSimilarity(E_1.newsContent, E_3.newsContent) > 80\%$
    $IsFirstTimeAppear(E_1.newsContent) > 50\%$
    $IsFirstTimeAppear(E_2.newsContent) > 50\%$
    $IsFirstTimeAppear(E_3.newsContent) > 50\%$
WITHIN 1 *hour*
RETURN
    Recently happened news E_1, E_2, E_3
CONSUMING E_1, E_2, E_3

Algorithmus 3.11 Extended SNOOP Query for Application 3

SELECT $*$
FROM $SEQ(E_1; E_2; E_3)$
WEHER
    FILTER
        $E_1.receiver = E_2.receiver$
        $E_1.receiver = E_3.receiver$
        $E_1.accountID = E_2.accountID$
        $E_1.accountID = E_3.accountID$
    $E_1.amount < 100$ AND
    $E_2.amount < 100$ AND
    $E_3.amount > 250$
WITHIN $72\ hours$
RETURN
    $Probablefraud$
CONSUMING E_1, E_2, E_3

3.2.3 Application 3

Detect credit card fraud pattern, given certain detecting rules.

Example:

One of usual credit card fraud pattern is that a group of small amount of transferred money followed by one large amount of transferred money.

Algorithm 3.11 shows a simple Extended SNOOP Query w.r.t. the credit card fraud pattern in example.

3.2.4 Application 4

Detect if one car over takes another car in the forbidden area.

Example:

Set two cameras to catch the license plate of car. Detect the sequence of appearance of cars. If the sequence from camera 2 is different from the sequence from camera 1, then overtaken happens.

Algorithm 3.12 gives the simple Extended SNOOP Query for the example above.

Algorithmus 3.12 Extended SNOOP Query for Application 4

SELECT $*$
FROM $SEQ(SEQ(E_1; E_2); SEQ(E_3; E_4))$
WHERE
    FILTER
        $E_1.carID = E_4.carID$
        $E_2.carID = E_3.carID$
    $E_1.source = camera1$
    $E_2.source = camera1$
    $E_3.source = camera2$
    $E_4.source = camera2$
RETURN
    $Overtakenhappened.$

Algorithmus 3.13 Extended SNOOP Query for Application 5

SELECT $*$
FROM E_1
WHERE
    FILTER
        $E_1.name = "IBM"$
    $E_1.price > \$85$
RETURN
    $IBM\ price\ is\ greater\ than\ \$85$

3.2.5 Application 5

Monitor one stock price if it is greater than a value.[DGP+07]

Example:

Monitor if the price of IBM is greater than $85.

Algorithm 3.13 is the algorithm used in this scenario, which should be very similar to the original SNOOP Query.

3.3 Analysis of Application Examples

In previous section, five application examples are given. Some of them are suitable for the existing Split-Process-Merge architecture, while the others are not.

Application 4 and Application 5 are quite suitable for Split-Process-Merge architecture. The splitter can distribute the events based on the attributes id and $name$.

Application 3 can also be handled by Split-Process-Merge architecture. The attribute $accountID$ can be used to distribute the events. But what if in some cases it is hard to distribute the events based on certain attributes. This could happen when the splitting decision is heavy. (e.g. Application 1 and Application 2)

Application 1 and Application 2 are the examples of computational heavy splitting decision. Because the splitter needs to check if two photo data are match or to calculate the similarity of two news articles. These operations are not so simple operations as to check if two integers or two strings are equal. Then this leads to a heavy splitter since the splitter handles the work which should be done by operators.

The "heavy" splitting decision is one issue in Split-Process-Merge architecture when the splitter decides to open (usually happens in opening, but also possible in closing) one partition, which will be CPU bottleneck for a single splitter.

Another issue of Split-Process-Merge architecture is that the splitter needs to go through all partitions to check if it should close certain partition and at the same time, there are many (overlapped) partitions.

In Application 2 and Application 3, in the worst cases to CEP system, the credit card fraud and the global/domestic news never happen. However, it is still necessary to check if all incoming events match the pattern.

In Application 2 the splitter needs to start a partition for *EACH* incoming event, which will lead to a huge amount of overlapping partitions. For a popular news source, the rate of incoming events could be a very high value. At the same time, each partition needs to keep for 1 hour. The situation will become worse if the time window needs to be longer. When a new incoming event has received, the splitter needs to check through all these existing partitions if a certain partition should be closed. This would be another CPU bottleneck for single splitter.

In Application 3 there is the same problem. If the partitions need to be open for a long time, there will be huge amount of partitions waiting to be checked by the single splitter if the partition should be closed or not. In these two scenarios, the single splitter will have CPU bottleneck when closing the partitions.

In simple words, the single splitter will have CPU bottleneck in following two situations.

- The splitter needs to execute expensive operation when opening (sometimes closing) a partition.

- The splitter needs to go through huge amount of partitions to check if a partition should be closed or not.

In next section, I will try to find out the cause why some applications are suitable for Split-Process-Merge architecture while the others will leads to a "heavy" splitter.

3.4 Problem Classification

As mentioned in last section, Application 1 and Application 2 are not suitable for Split-Process-Merge architecture. Application 3 will also have performance issue. On the other hand, Application 4 and 5 are quite suitable for Split-Process-Merge architecture. To find the cause, we need to look into the differences among their queries.

In Application 1 and Application 2,there are $e_1.photoData\ matches\ e_2.photoData$ and $CalculateSimilarity(E_1.newsContent, E_2.newsContent)$ in "FILTER" clause, which is quite different than other applications. The operation $matches$ and $CalculateSimilarity$ are the cause of an heavy splitting decision.

I have a category for these kind of applications as Application 1 and Application 2: Content-Based. Other applications are Key-Based.

The difference between Key-based and Content-based is as following.

- Key-based: Using meta-data from event as the key. E.g. the ID of the event, the time-stamp, the counter. These meta-data are independent from the content that event carries.
- Content-based: Using the content that event carries as the key. E.g. the photo of an object, the whole text of a news article.

But in some cases, it is hard to distinguish Key-based from Content-based. As in Application 4, the $carID$ could be the meta-data as well as the content.

However, in most cases if the splitter wants to split the events by content, then the splitter needs to process the content, which should be the job of operator. This will cause a "heavy" splitter and leads to the first situation in last section, i.e. "The splitter needs to execute expensive operation when opening (sometimes closing) a partition."

Another issue of Split-Process-Merge architecture is the usual result of a complex pattern with large "WITHIN" time windows size. Usually a more complex pattern will lead to a larger partition size, because more events are involved. A longer time window also leads to a larger partition size because more events in time-line are involved.

Some semantic aspects will also influence the overlapping between partitions. For example, in Application 3 the events can be distributed by $accountID$, so there won't be overlapping between partitions. Because semantically, those events having different $accountID$ are irrelevant to each other. However, in Application 2 the shift-size of partition is one event, so there will be many overlapping between partitions. Because every event is potentially relevant to each other (i.e. each news may be similar to the other).

As a conclusion, in most cases Content-based "FILTER" will cause heavy splitting decision and a heavy splitter, which will become a CPU bottleneck in whole architecture. Because, we don't want the splitter to do the operator's job.

Complex pattern with large "WITHIN" windows size will cause huge amount of partitions. Some semantic aspects will influence the overlapping between partitions as well. Huge amount of partitions takes splitter lots of time to go through and to check the closing condition, which will also become a CPU bottleneck in whole architecture.

3.5 Thesis Goal

From the analysis and classification above, the cause of a "heavy" splitter has been found: the splitter needs to execute expensive operations against the content carried by events. In the idea of Split-Process-Merge architecture, these "heavy" jobs should belong to operator instance. Therefore, a new architecture is required to avoid a "heavy" splitter while at the same time, the features of Split-Process-Merge architecture should also be maintained: data parallelization and unbounded parallelism degree.

For the beginning, the developed architecture shall be tailored towards shared-memory machine, i.e. multi-core host.

4 Approach to Create New Architecture

In this chapter, I will show the approach how I obtained the major components of new architecture from Split-Process-Merge architecture step by step. Then I will explain each component of new architecture in details in next chapter.

As described in previous chapters, Split-Process-Merge architecture provides a very valuable feature in parallelism CEP: unbounded degree of parallelism. The splitter distributes the events according to certain rules. Each operator instance detects the pattern against a given set of events, i.e. partition or selection. Then merger receives results from operator instances and serializes the results for further processing.

These event sets may be overlapped, but they are independent from each other. I.e. Different selections may contain same events, but the result derived from one selection does not depend on other selections. Since selections are independent from each other, the operator instances are also independent, because they are detecting the same pattern but against different event set (selection) and no communication between operator instances is needed.

The independence between selections as well as between operator instances is the basis of data parallelism and basis of unbounded parallelism. This feature should be kept in the new architecture. Therefore, somehow in new architecture there will be a component maintaining unbounded number of operator instances, and operator instances should be independent from each other.

According to the analysis in previous chapter, one reason that the CPU bottleneck appears in the splitter in Split-Process-Merge architecture is because splitter needs to execute expensive distribution decision.

The splitter needs to check:

1. If the incoming event starts a new selection? (i.e. $P_s()$ operation)
2. If the incoming event closes some selections? (i.e. $P_c()$ operation)
3. What selections should this new event be sent to?

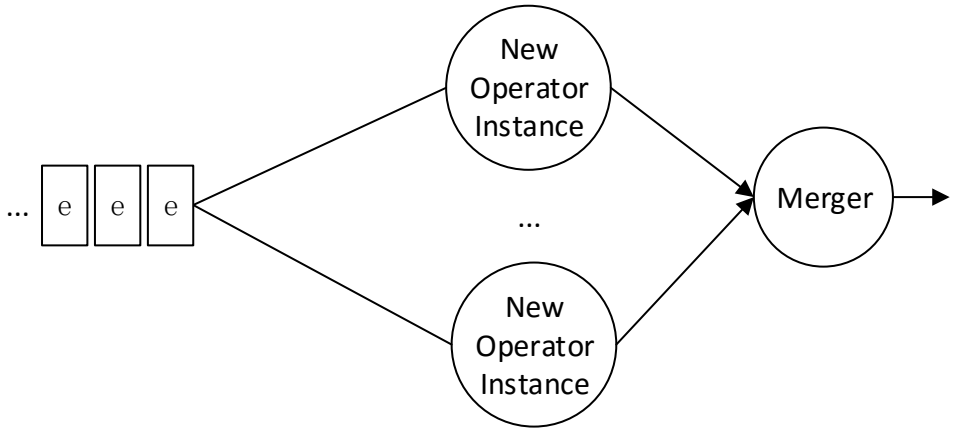


Figure 4.1: Merged Splitter and Operator

Algorithmus 4.1 Simple Example Query

SELECT $*$
FROM $SEQ(E_1; E_2; E_3)$
WHERE
FILTER
$(Constr.1)$ $E_1.type = A$
$(Constr.2)$ $E_2.type = B$
$(Constr.3)$ $E_3.type = C$
$(Constr.4)$ $E_1.photoData\ matches\ E_2.photoData$
$(Constr.5)$ $E_1.photoData\ matches\ E_3.photoData$

All these three tasks in splitter need to look into the content or meta-data of event. To some extent, the splitter is doing the job of operators. Therefore, I would like to move this part of function from splitter to operators. I don't want the splitter check the content or meta-data of event at all.

What I am doing is to merge the splitter and operators in Split-Process-Merge architecture. After merging splitter and operator, the architecture looks as Figure 4.1.

However, this leads into losing the parallelism. Because there is no splitter now, and the event stream flows directly into the operator instances. Every operator instance is handling the same event set (or no sets at all, just the same event stream) and gives same results. Parallelism should be created at some point of the architecture.

Now I look into the query to find where can we get parallelism. Algorithm 4.1 is used as a simple example query. Obviously, the major computational cost in detecting the pattern

is in evaluating the constraints ($Constr.1$ to $Constr.5$), especially the matches operation ($Constr.4$ and $Constr.5$). Therefore, it is reasonable to parallelize the constraints evaluating part, which of course makes the architecture state-based intra-operator parallelism. In later sections, data parallelization will be introduced into the architecture again so that the architecture will become hybrid parallelism later.

4.1 Chapter Organization

First, an improved Finite State Machine will be introduced in Section 4.2. Then Section 4.3 gives a general but brief view that how does the new architecture finally look like. To describe the new architecture in a formal way, Section 4.4 gives the terminology used both in Chapter 4 and in Chapter 5. Following two sections are about improvement of parallelism degree and generalization of the new architecture. The last section is the summary about new architecture obtained. In next Chapter, details about every component in new architecture will be introduced.

4.2 Improved Finite State Machine

4.2.1 Finite State Machine with Conditions

The first idea to parallelize the constraints evaluating part would be state-based parallelization. (Chapter 4.1 in [BDWT13]) Since state machine has following features.

- Easy to implement
- Easy to pipeline (intra-operator parallelism)
- Universal approach to convert from pattern to state machine model (This is also the reason why I focus on extending SNOOP rather than TESLA)

The example query can be modeled by a Finite State Machine as shown in Figure 4.2. The original idea of this State Machine Model is from TESLA automaton model[CM10]. TESLA makes some modification to the Finite State Machine. To transfer from one state to next state, the state machine does not depend on input, but depends on if conditions are met. I.e. the state machine will go to next state as long as there is an input event and conditions are met.

Figure 4.3 gives the syntax of State Machine Model. There are also two special cases: unconditional transfer, and transfer without an input. Figure 4.4a shows a situation

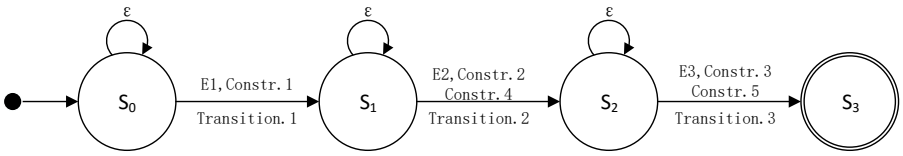


Figure 4.2: Finite State Machine Model of Query Algorithm 4.1

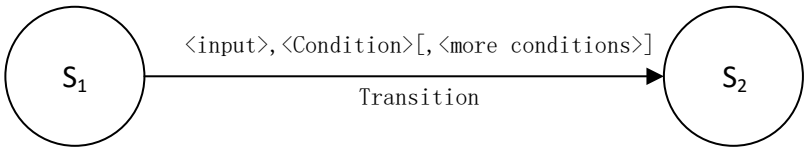
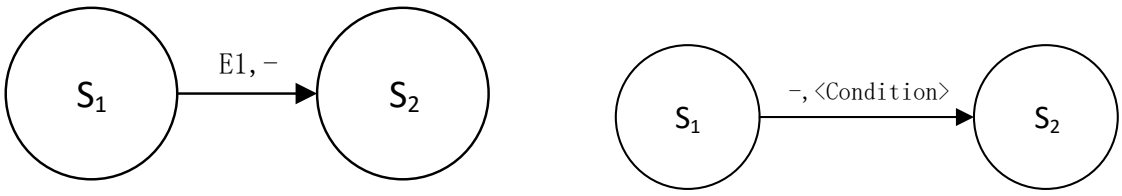


Figure 4.3: Finite State Machine Syntax

that unconditional transfer as long as there is an input. Figure 4.4b shows a situation that conditional transfer without an input. The state machine will go to next state if condition is met, but no input is needed.

With the help of conditions, it is possible to check constraints between two nonadjacent events. Because, to transfer to next state, the original Finite State Machine only depends on input value, which means it can only check the constraints on current input value, maybe plus the value of predecessor input if current state stores some more information.

In the Query Algorithm 4.1, with the help of conditions, the state machine will transfer from state S_2 to S_3 if there is any input E_3 as well as conditions $Constr.3$, and $Constr.5$ are met. In $Constr.5$ we would like to check if $E_1.photoData\ matches\ E_3.photoData$.



(a) Unconditional Transfer **(b)** Conditional Transfer Without Input

Figure 4.4: Special Transfer

However, the original Finite State Machine cannot achieve this, because E_1 and E_3 are not adjacent in event stream and the transfer only depends on current input event. It can only check if $Constr.3$ is met, i.e. if $E_3.type = C$ but not $Constr.5$, because at state S_2, there is no information about event E_1 any more.

By using the conditions instead of input value in transition, the functionality of Finite State Machine is extended. In next section, a replication mechanism is introduced to extend the Finite State Machine further.

4.2.2 Finite State Machine with Replication

Let's still look at same state machine in Figure 4.2.

Even with help of conditions, this state machine still has limitation. It can only detect one ABC match from $AABBCC$. For explanation, I will distinguish the input event A,B,C by index. For a given event stream A_1 A_2 B_1 B_2 C_1 C_2, the state machine can only detect A_1 B_1 C_1 and loses A_2 B_1 C_1, A_1 B_2 C_1, A_1 B_1 C_2, etc.

This is because there is only one "state machine instance". When A_2 arrives, state S_1 can only represent one event, either A_1 or A_2, the other one is then dropped. To keep both A_1 and A_2 events, two state instances S_1 are needed. It is the same for other events.

This is the idea of replication in TESLA: as soon as the state machine goes into next state, current state machine instance is replicated and kept still, the new state machine instance will go into next state with the input event. Therefore, the state of every incoming event is stored in a certain state instance. A sequence of state instances makes up the state machine instance.

In RIP paper[BDWT13], each transition will output a middle result, which is called partial match because it is a part of final match. In TESLA paper[CM10], it is called automaton instance.

The difference is that in RIP paper, they don't have replication mechanism so that each partial match will be consumed in next state and a new partial match is generated, or equivalently, the partial match is updated. While in TESLA paper, they introduce the replication mechanism, for each transition, the automaton instance is replicated, and the old one is kept still, the new one is updated.

In this paper, "Partial Match", "automaton instance", and "state machine instance" are equivalent and the replication mechanism is also used. I.e. each transition will output a Partial Match, which is replicated from an old Partial Match and applied the new incoming event to. A detailed definition about Partial Match in this paper can be found

in Section 4.4.6. Following is the formula representing the generation of a new Partial Match.

(4.1) $pm_n = TransitionCondition_n(pm_{n-1}, e_n)$ or $pm_n = TC_n(pm_{n-1}, e_n)$

pm_n is the new Partial Match generated.

e_n is the next incoming event.

$TransitionCondition_n()$ is the evaluation operation of the constraints w.r.t. state S_n, and the evaluation result must be true, otherwise, there won't be the transition triggered and no new Partial Match created. In the diagram, this process is represented by the edge with arrow(i.e. Transition) between two states.
$TC_n()$ is used as the short form of $TransitionCondition_n()$.

For $n = 1$, $pm_0 = \varnothing$, $pm_1 = TransitionCondition_1(pm_0, e_1)$.
pm_0 is used to represent the output Partial Match when the state machine enters the start state. (I.e. entering into the start state is treated as a transition, which also outputs a Partial Match pm_0.)
pm_n is generated from pm_{n-1}, and pm_{n-1} is kept still. This the implementation of replication mechanism.

Next, I will prove the necessity to replicate the Partial Match. To prove the necessity, I need an assumption as following.

Assumption: It is always possible to detect a Partial Match in event stream. I.e. the pattern to be detected will finally appear in the event stream.

always $\exists e_1$ such that $pm_1 = TC_1(pm_0, e_1)$
always $\exists e_n$ such that $pm_n = TC_n(pm_{n-1}, e_n)$

Therefore, $\forall pm_i$, $\exists e_j$ and $\exists e_k$ such that $pm_j = TC_{i+1}(pm_i, e_j)$ and $pm_k = TC_{i+1}(pm_i, e_k)$
If $e_j \neq e_k$, then $pm_j \neq pm_k$.

If pm_i is consumed in $pm_j = TC_{i+1}(pm_i, e_j)$ and not replicated, then $pm_k = TC_{i+1}(pm_i, e_k)$ is not executable since there is no pm_i anymore, so that pm_k is lost, which may cause false negative. Hence, pm_i should be replicated. Both pm_j and pm_k should be updated from the replica of pm_i. pm_i itself should be kept still.

4.3 General View of New Architecture

Following sections are the approach how did I obtain the new architecture from extended Finite State Machine step by step as well as the definition of terminology. Before

going into the details of approach and terminology, I would like to introduce the new architecture briefly so that you could have a general view about how does the new architecture finally look like.

There are three major components in the new architecture: A Centralized Data Structure, a group of Operator Instances, and a Merger.

Centralized Data Structure maintains all incoming Raw Events and Partial Matches in the system. It also takes over the work-flow of the whole system by creating Tasks and scheduling Tasks. Besides, Centralized Data Structure creates new Partial Matches according to the Task Results from Operator Instances.

Operator Instances are the worker. They receive Tasks from Centralized Data Structure, evaluate the Tasks, generate Task Results, and then returns the Task Results back to Centralized Data Structure.

Merger is responsible for sorting the Final Matches and firing them. The order of Final Matches should be consistent to the result from a sequential processing architecture, as described in previous chapters. In addition, if any fired Final Match needs to consume Raw Events according to consumption policy, Merger should inform the Centralized Data Structure about the consumed Raw Events.

In next section, the definition of terminology used in the new architecture will be given.

4.4 Terminology Definition

To explain the new architecture in a more accurate and mathematical way, this section lists the terms and their definition used in following sections.

4.4.1 Path

Definition 4.4.1
$path = (TC_1, TC_2, TC_3, \ldots, TC_m)$ *means a path starting from the start state* S_0*, passing through* TC_1*, followed by* $TC_2\ TC_3 \ldots$ *in sequence, and ended at state* S_m*.*

A path is a sequence of Transition Conditions in State Machine Model. Each Transition Condition connects two states, starting from the start state. The path is order-sensitive. $path = (TC_1, TC_2)$ and $path = (TC_2, TC_1)$ are different paths.

4.4.2 Equality of Path

Existing two paths $path_i$ and $path_j$, $path_i$ contains n Transition Conditions and $path_j$ contains m Transition Conditions.

Definition 4.4.2
Two paths are equal when they have the same size and elements. The order of elements should also be the same.

$$path_i \equiv path_j \Leftrightarrow \\ n = m \text{ and} \\ (1 \leq \forall k \leq n)\ TC_{k,i} = TC_{k,j}$$

4.4.3 Raw Event

Definition 4.4.3
A Raw Event is an event in the event stream.

General notation: e

Notation with index: e_i means the i-th Raw Event in the event stream.

Definition 4.4.4 (Consumed Raw Event (or invalid Raw Event))
A Raw Event has been consumed by a Final Match when the Final Match is fired. Once a Raw Event is consumed, it will have no more impact on following Raw Events in Raw Event Stream as well as the Partial Match Data Structure.

A Raw Event can only be consumed no more than once.

Definition 4.4.5 (Valid Raw Event)
A Raw Event has not been consumed yet.

4.4.4 A Sequence of Raw Events

To represent a sequence of n Raw Events, following notation is used.

Definition 4.4.6

$$(e)_{size=n} \doteq (e_1, e_2, \ldots, e_n)$$

When the size of sequence is not important, $size = n$ will be omitted.

Sequence of Raw Events is order-sensitive. $(e) = (e_1, e_2)$ and $(e) = (e_2, e_1)$ are different sequences of Raw Events.

The sequence of Raw Events must have the same order the Raw Events are in the event stream.

4.4.5 Equality of Sequence of Raw Events

Existing two sequences of Raw Events $(e)_i$ and $(e)_j$. $(e)_i$ contains n raw events. $(e)_j$ contains m raw events.

Definition 4.4.7
Two sequences of Raw Events are equal when they have the same size and elements. The order of elements should also be the same.

$$(e)_{i,size=n} \equiv (e)_{j,size=m} \Leftrightarrow$$
$$n = m \text{ and}$$
$$(1 \leq \forall k \leq n) e_{k,i} = e_{k,j}$$

4.4.6 Partial Match

Definition 4.4.8
Partial Match is the median result between states.

Partial Match can be used to represent the state machine instance. Therefore, it is totally the same as the "automaton instance" in TESLA paper[CM10]. Partial Match in this paper has small difference from the partial match in RIP paper[BDWT13]. In RIP paper, partial match will be consumed or updated in following state. In this paper, Partial Match can only be generated or discarded, but never be updated because of the replication mechanism.

The generation formula of Partial Match is $pm_n = TC_n(pm_{n-1}, e_n)$. This is the iteration expression, and the expanded expression is as following.

$$pm_n = TC_n(TC_{n-1}(TC_{n-2}(\dots TC_2(TC_1(pm_0, e_1), e_2) \dots, e_{n-2}), e_{n-1}), e_n) \tag{4.2}$$

I format the formula and use following notation to represent the formula above.

$$pm_n \doteq ((TC_1, TC_2, \dots, TC_{n-1}, TC_n); (e_1, e_2, \dots, e_{n-1}, e_n)) \tag{4.3}$$

Since start state pm_0 is constant, pm_0 is omitted in the notation. $(TC_1, TC_2, \ldots, TC_{n-1}, TC_n)$ is a path and $(e_1, e_2, \ldots, e_{n-1}, e_n)$ is a sequence of Raw Events, therefore,

(4.4) $pm_n = (path_n; (e)_n)$

From the formula, it is obvious that Partial Match contains the information about all history Raw Events and the path starting from the start state, which can be used for consumption problem later.

4.4.7 Equality of Partial Match

Definition 4.4.9
Two Partial Matches are equal when they have the same path and same Raw Event Sequence.

$$pm_i \equiv pm_j \Leftrightarrow \\ path_i \equiv path_j \textit{and} \\ (e)_i \equiv (e)_j$$

4.4.8 Final Match

Definition 4.4.10
A Final Match is a Partial Match that fully matched to the query. I.e. Starting from the start state in Finite State Machine, the last Raw Event in Final Match ends at a end state.

4.4.9 Order of Two Partial Matches

Definition 4.4.11
A Partial Match W is earlier than another Partial Match X if one of following rules is satisfied.

1. *The last Raw Event in Partial Match W is earlier than the last Raw Event in Partial Match X.*
2. *The last Raw Event in Partial Match W is equal to the last Raw Event in Partial Match X, and the first Raw Event in Partial Match W is earlier than the first Raw Event in Partial Match X.*

3. *The last Raw Event in Partial Match W is equal to the last Raw Event in Partial Match X, and the first Raw Event in Partial Match W is equal to the first Raw Event in Partial Match X. Then compare the sub-sequences of both Partial Matches. I.e. Both Partial Matches W and X removes the first Raw Event. If the sub-sequence of Partial Match W is earlier than sub-sequence of Partial Match X.*

To be more intuitive, Figure 4.5 shows the order of two Partial Matches. The Partial Matches with number 1, 2, 3, 4, 6, and 9 (with a dot in front of number) are earlier than Partial Match X. For Number 7, the order depends on the sub-sequences of both Partial Matches. (I.e. remove both first Raw Events)

4.4.10 Order of Two Final Matches

Definition 4.4.12
A Final Match W is earlier than another Final Match X if one of following rules is satisfied.

1. *The last Raw Event in Final Match W is earlier than the last Raw Event in Final Match X.*
2. *The last Raw Event in Final Match W is equal to the last Raw Event in Final Match X, and the first Raw Event in Final Match W is earlier than the first Raw Event in Final Match X.*
3. *The last Raw Event in Final Match W is equal to the last Raw Event in Final Match X, and the first Raw Event in Final Match W is equal to the first Raw Event in Final Match X. Then compare the sub-sequences of both Final Matches. I.e. Both Final Matches W and X removes the* last *Raw Event. If the sub-sequence of Final Match W is earlier than sub-sequence of Final Match X.*

To be more intuitive, Figure 4.5 shows the order of two Final Matches. The Final Matches with number 1, 2, 3, 4, 6, and 9 (with a dot in front of number) are earlier than Final Match X. For Number 7, the order depends on the sub-sequences of both Final Matches. (I.e. remove both *last* Raw Events)

4.4.11 Dependency between Raw Events and Partial Matches

Definition 4.4.13
If a Raw Event e was an input parameter in a Partial Match, then this Partial Match depends on Raw Event e.

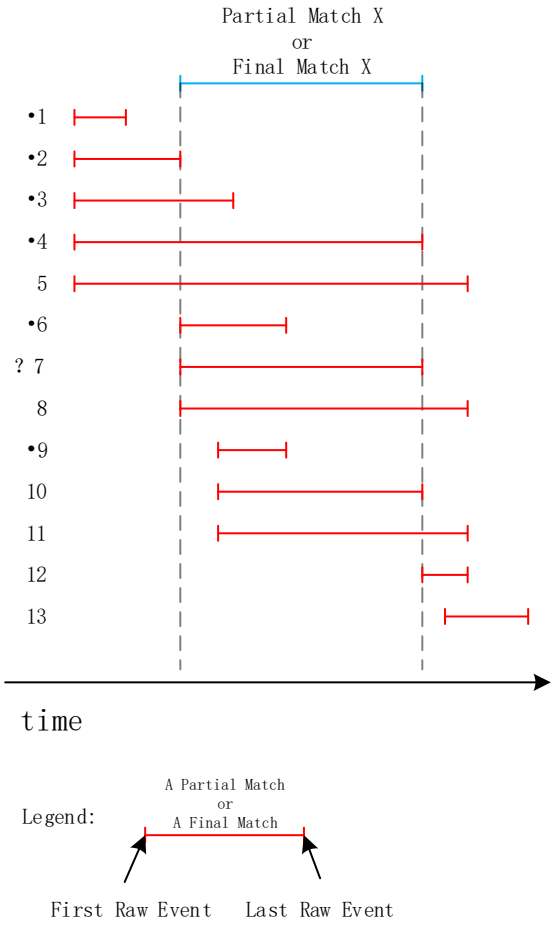


Figure 4.5: Order of Two Partial Matches/Final Matches

Obviously, a Partial Match $pm_i = (path_i; (e)_i)$ depends on a Raw Event e_k iff $e_k \in (e)_i$. The dependency between Raw Events and Partial Matches is used in later chapter for data structure, consumption problem, and optimization.

4.4.12 Dependency between Partial Matches w.r.t. Raw Events

Definition 4.4.14
If a Partial Match pm_i was an input parameter in another Partial Match pm_j, then pm_j depends on pm_i w.r.t. Raw Events.

According to formula $pm_n = TC_n(pm_{n-1}, e_n)$, pm_n depends on both pm_{n-1} and e_n, since they are the input parameters. The dependency between Raw Event and Partial Match is

introduced in previous section. Now we look at pm_{n-1}. Since pm_{n-1} can be expressed as $pm_{n-1} = (path_{n-1}; (e)_{n-1})$, the dependency between pm_n and pm_{n-1} is related to two aspects: Transition Condition and Raw Event. In this section, the dependency w.r.t Raw Events is discussed. In next section, the dependency w.r.t Transition Condition will be introduced.

The dependency w.r.t. Raw Events between pm_n and pm_{n-1} has following property.

Property: pm_j depends on pm_i if and only if $path_{pm_i} \subset path_{pm_j}$ AND $(e)_{pm_i} \subset (e)_{pm_j}$.

This property will be used in data structure and consumption problem in later chapter. In simple words, consumption of pm_i will cause the consumption of pm_j, and the algorithm knows this relationship because of this property.

Now I will prove that pm_j depends on pm_i if and only if $path_{pm_i} \subset path_{pm_j}$ AND $(e)_{pm_i} \subset (e)_{pm_j}$ from both necessity and sufficiency aspects.

Necessity Prove

Since

$path_{pm_i} = (TC_i, TC_{i-1}, \ldots, TC_1)$,

$path_{pm_i} \subset path_{pm_j}$,

$(e)_{pm_i} \subset (e)_{pm_j}$,

$path_{pm_j}$ can be written as

$$path_{pm_j} = (TC_j, TC_{j-1}, \ldots, TC_{j-k}, TC_i, TC_{i-1}, \ldots, TC_1)$$

$(e)_{pm_j}$ can be written as

$$(e)_{pm_j} = (e_1, e_2, \ldots, e_{i-1}, e_i, e_{j-k}, \ldots, e_{j-1}, e_j)$$

Therefore,

$$pm_j = ((TC_j, TC_{j-1}, \ldots, TC_{j-k}, TC_i, TC_{i-1}, \ldots, TC_1); (e_1, e_2, \ldots, e_{i-1}, e_i, e_{j-k}, \ldots, e_{j-1}, e_j))$$

Since

$$pm_i = ((TC_i, TC_{i-1}, \ldots, TC_1); (e_1, e_2, \ldots, e_{i-1}, e_i)),$$

formatting the equation yields

$$pm_j = TC_j(TC_{j-1}(\ldots TC_{j-k}(pm_i, e_{j-k}) \ldots, e_{j-1}), e_j)$$

Thus, pm_i was an input parameter of pm_j at some point in the past, which means pm_j depends on pm_i.

Necessity proved.

Sufficiency Prove

Since pm_j depends on pm_i, then at some point in the past, pm_i was the input parameter.

At the same time, pm_j can be represented by $pm_j = TC_j(pm_{j-1}, e_j)$.

Expending pm_{j-1} yields

$$pm_j = TC_j(TC_{j-1}(\dots TC_{i+1}(pm_i, e_{i+1}) \dots, e_{j-1}), e_j)$$

Formatting this equation yields

$$\begin{aligned} pm_j =&((TC_j, TC_{j-1}, \dots, TC_{i+1}, TC_i, TC_{i-1}, \dots, TC_1); \\ &(e_1, e_2, \dots, e_{i-1}, e_i, e_{i+1}, \dots, e_{j-1}, e_j)) \end{aligned}$$

Compare this equation to

$$pm_i = ((TC_i, TC_{i-1}, \dots, TC_1); (e_1, e_2, \dots, e_{i-1}, e_i))$$

Obviously, $path_{pm_i} \subset path_{pm_j}$ AND $(e)_{pm_i} \subset (e)_{pm_j}$.

Sufficiency proved.

4.4.13 Independency between Partial Matches w.r.t. Transition

Definition 4.4.15
Given a same Transition, if two Partial Matches pm_{i+1} and pm_{j+1} are generated from pm_i and pm_j respectively, and pm_{i+1} has no dependency to pm_{j+1} w.r.t. Raw Event, then pm_i are independent from pm_j w.r.t Transition

According to $pm_n = TC_n(pm_{n-1}, e_n)$, each Transition only takes one Partial Match as input parameter. Therefore, from the respect of Transition, it is no difference to take which Partial Match pm_i or pm_j first. Because w.r.t. Raw Event, pm_{i+1} only depends on pm_i and pm_{j+1} depends on pm_j. This is the independency between Partial Matches w.r.t. Transition. This independency will be used for data parallel to achieve unbounded parallelism degree in later section.

4.4.14 Task

Definition 4.4.16
A Task is an entity contains all information for an operator instance to evaluate a certain Transition Condition. Then a Task Result, which will be introduced later, should be derived from this Task and returned from the operator instance.

Notation $Task_{pm}$ is used to represent a Task corresponding to pm.

Definition 4.4.17 (Valid Task)
A valid Task is a Task such that when it is taken from the Task Queue by an operator instance, ALL *Raw Events involved in the Task are* NOT *consumed yet at that point.*

Definition 4.4.18 (Invalid Task)
An invalid Task is a Task such that when it is taken from the Task Queue by an operator instance, at least one Raw Event involved in the Task has already been consumed at that point.

Definition 4.4.19 (Necessary Task)
A necessary Task is a Task leads to a Final Match.

A Task generating a Partial Match, which is contained in a Final Match, is also a necessary Task. If one necessary Task is missing, the outcome of the system will NOT be consistent to a sequential processing architecture.

Definition 4.4.20 (Unnecessary Task)
An unnecessary Task is a Task doesn't lead to a Final Match.

If one unnecessary Task is missing, it will not influence the outcome.

Definition 4.4.21 (Order of two Tasks)
The order of two Tasks is defined by the order of two Raw Events that these two Tasks are carrying.

4.4.15 Task Result

Definition 4.4.22
A Task Result is the corresponding outcome of the evaluation of a Task from operator instance.
A Task Result can be either True or False. Please see definition of True Task Result and False Task Result.

Algorithmus 4.2 Example Query 2

SELECT $*$
FROM $SEQ(OR(E_1, E_2, E_3); E_4)$
WHERE
(Constr.1) $E_1.type = A$
(Constr.2) $E_2.type = B$
(Constr.3) $E_3.type = C$
(Constr.4) $E_4.type = D$

Definition 4.4.23 (True Task Result)
The result of the evaluation on a Task is true, which means a Partial Match is detected and should be added into the Partial Match Data Structure.

Definition 4.4.24 (False Task Result)
The result of the evaluation on a Task is false, which means no Partial Match is detected.

4.5 Parallelize the Improved Finite State Machine

So far, the Improved Finite State Machine is able to detect the pattern against arbitrary constraints and nonadjacent events in the event stream. In this section I will explain how to parallelize the Improved Finite State Machine.

4.5.1 Pipeline the Improved Finite State Machine

The state machine has the nature to be pipe-lined easily. Each state can be assigned to a processing unit. For instance, to detect a query such as Algorithm 4.2. The state machine model will be as Figure 4.6. The architecture will be as Figure 4.7a.

Each state will be assigned to one processing unit. Each Raw Event in event stream will be forwarded to all processing units. Each processing unit has an output Partial Match queue. The Partial Match in output queue will be sent to the input queue of all succeeding processing units.

To decide the destination input queue of Partial Match, which is output from the processing unit S_0, S_0 has to evaluate three constraints, $Constr.1$, $Constr.2$, and $Constr.3$. The other three processing units S_1, S_2, and S_3 have to evaluate constraint $Constr.4$. Obviously, there are unbalanced work here. One processing unit S_0 has more work than others. While at the same time, evaluation of constraints is the heavy job in detecting

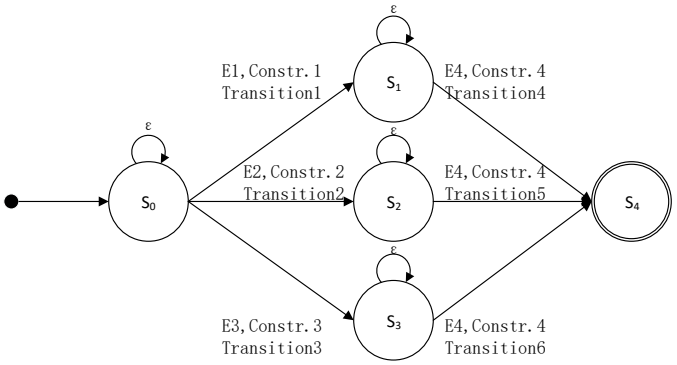
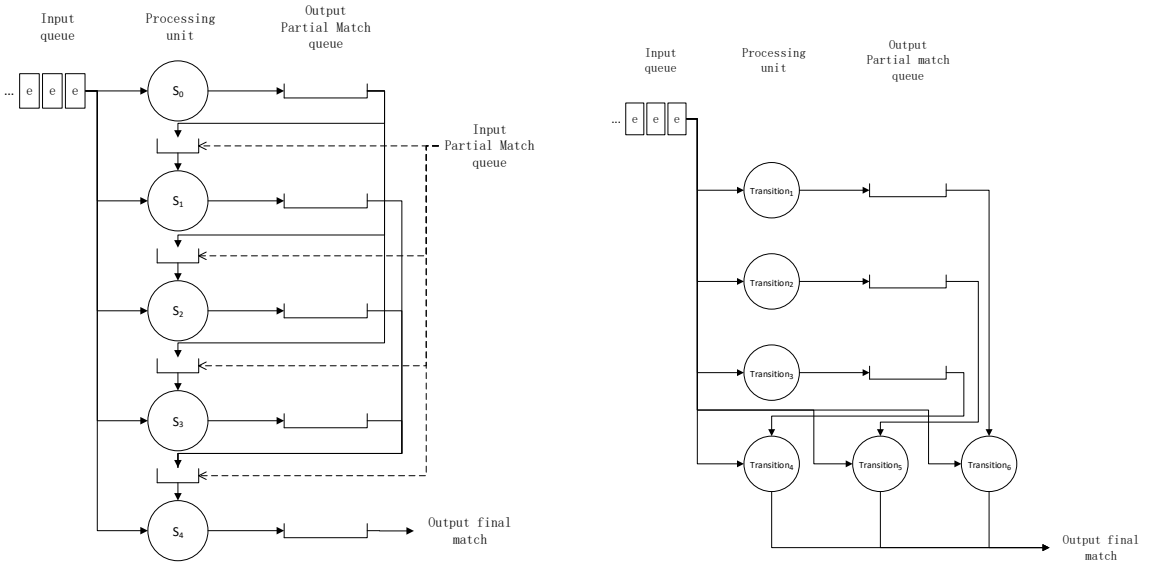


Figure 4.6: State Machine Model for Example Query 2



(a) State as Processing Units

(b) Transition as Processing Units

Figure 4.7: Architecture for Example Query 2

the pattern. A state with many outgoing edges(i.e. Transitions) will be the bottleneck of whole architecture.

Therefore, assigning the state to the processing unit is not a good idea. Instead, it is more reasonable to assign the transition to the processing unit, i.e. each processing unit handles one transition.

Now the architecture looks like as Figure 4.7b. For simplicity, the input queue of each processing unit is omitted unless it is necessary to show it. Please just remember, each

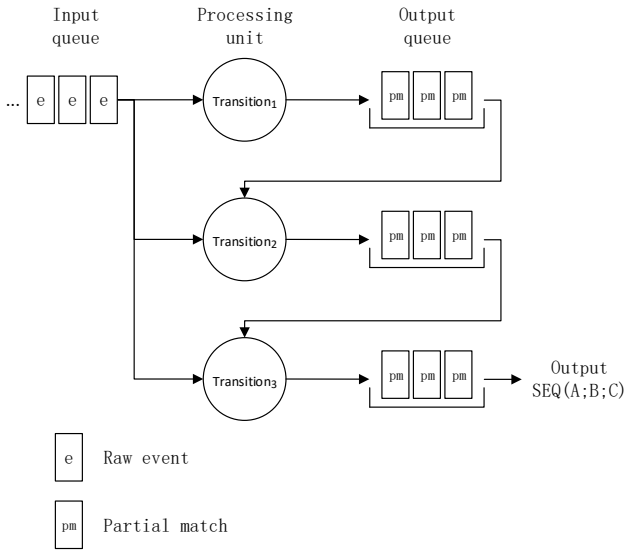


Figure 4.8: Architecture for State Machine Model in Figure 4.2

processing unit will have an input queue to receive the Partial Matches from output queue of predecessors.

The number of processing units is now proportional to the workload, i.e. Transitions amount. Although it seems there are three processing units working on the same constraint ($Transition4$, $Transition5$, and $Transition6$), this won't be a problem. I will try to generalize the processing units later to achieve a more balanced workload distribution.

Now, let's look back to the state machine model in Figure 4.2, which is used to detect pattern ABC. The architecture for Figure 4.2 is shown in Figure 4.8.

At this point, we have already achieve the parallelism with bounded parallelism degree by using pipe-lining. The degree of parallelism is bounded by the amount of Transitions. In next step, the parallelism degree will be improved.

4.5.2 Data Parallelization

Although we have parallelized the Transitions, the parallelism degree is still bounded. Since evaluation of constraints $Constr.4$ and $Constr.5$ has high execution cost, this architecture will have the bottleneck at processing unit $Transition_2$. (Because it is the first processing unit to evaluate $matches$ operation.)

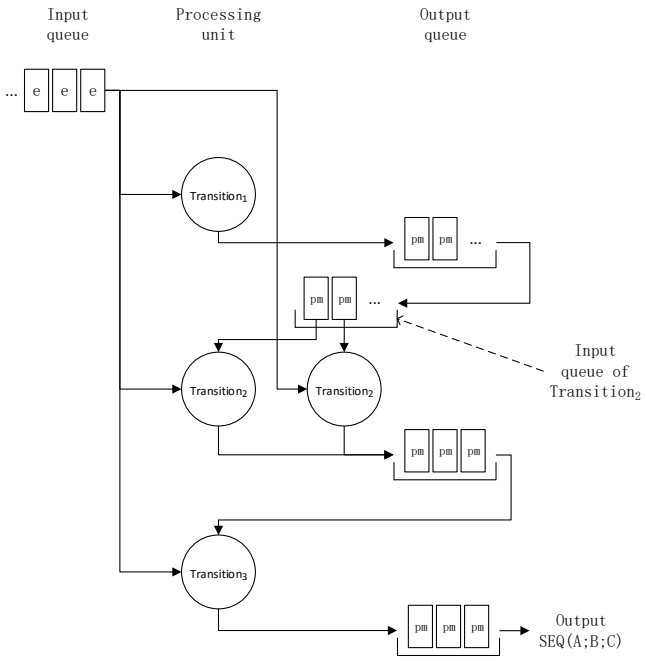


Figure 4.9: Architecture with Improved Parallelism Degree

The first idea to increase the parallelism degree will be to increase the number of processing unit $Transition_2$. The input queue is shared between all processing units of $Transition_2$. The architecture becomes as Figure 4.9.

Because Partial Matches in the input queue of $Transition_2$ are independent to each other w.r.t. $Transition_2$ (See Section 4.4.13, Page 50), if processing units of $Transition_2$ are identical then every Partial Match in the input queue can be distributed to an arbitrary processing unit of $Transition_2$.

Therefore, the additional processing unit of $Transition_2$ should be identical to the first one, the input queue should be shared between the processing units of $Transition_2$, and the Partial Matches in input queue can be distributed to either processing unit. However, please notice, each Partial Match should only be passed to *ONE* processing unit of $Transition_2$, *NOT* both.

Since the Partial Matches in one input queue are distributed in parallel to multiple processing units, the merging problem should be concerned. The issue about Merger will be discussed in Section 4.7.

By this approach, the parallelism degree has been increased by 1. In next step, more processing units will be added to achieve higher parallelism degree.

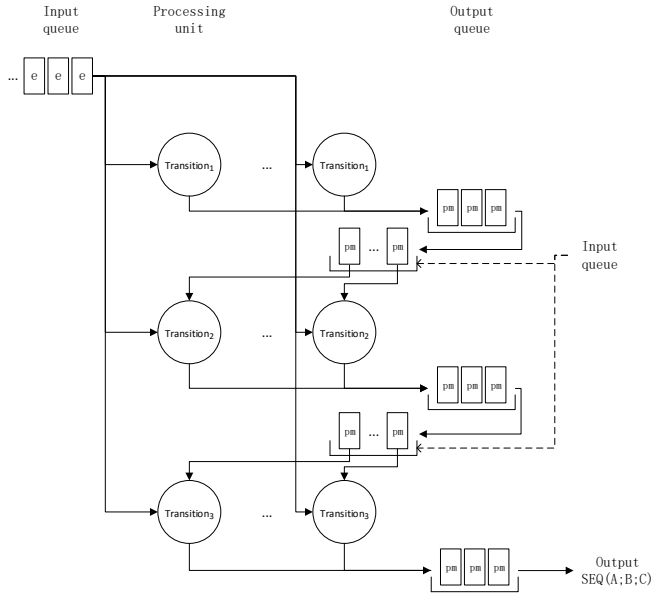


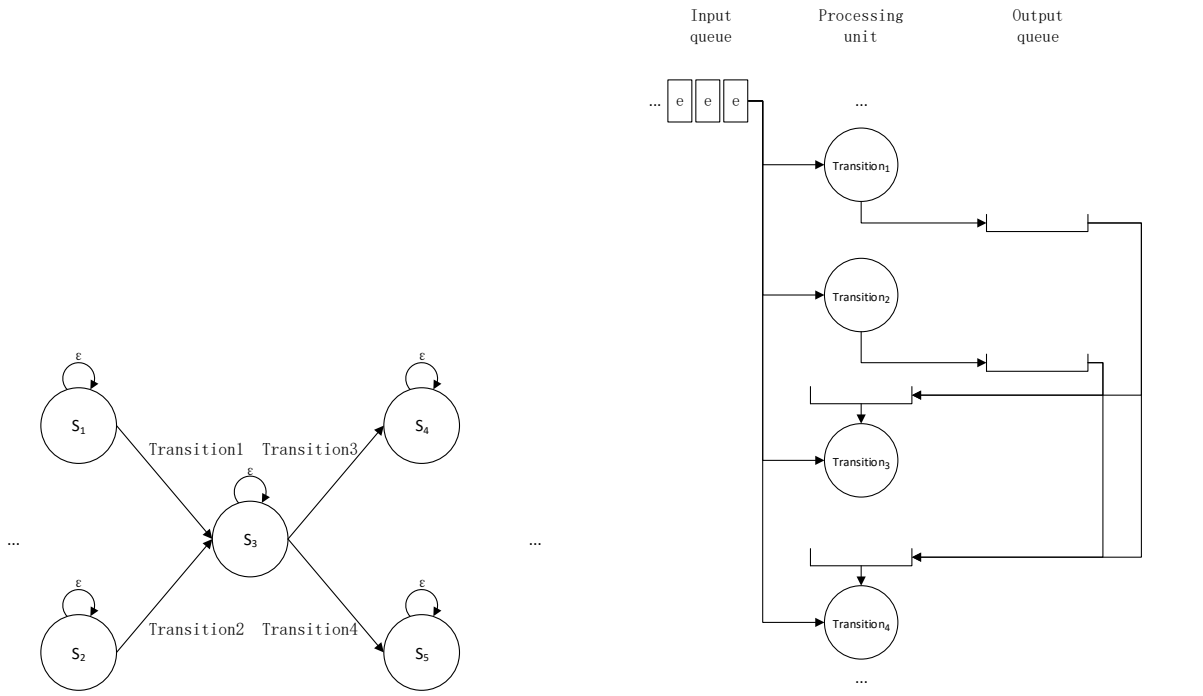
Figure 4.10: Architecture of Unbounded Parallelism Degree

Since the Partial Matches in the input queue of $Transition_i$ are independent to each other w.r.t. Transition, Partial Matches can be distributed to replicas of processing unit $Transition_i$, i.e. instance of operators. Therefore, the architecture, as shown in Figure 4.10, achieves unbounded degree of parallelism by increasing the number of processing units for each Transition. The input queue and output queue of each Transition are shared among all processing units of such Transition.

4.6 Generalization of the architecture

In current architecture, each Transition will have *ONE* output queue and *ONE* input queue and multiple identical processing units. These queues need to be maintained somewhere. In the example above, the output queue of $Transition_i$ can be the same input queue of $Transition_{i+1}$. In other cases, it is also possible that an input queue receives Partial Matches from two output queues and an output queue forwards Partial Matches to two input queues. For instance, a part of state machine model as described in Figure 4.11a. Then the architecture will be as in Figure 4.11b. Unrelated components are omitted.

The input queues of $Transition_3$ receives Partial Matches from two output queues, output queues of $Transition_1$ and $Transition_2$. The same as the input queue of $Transition_4$.



(a) State Machine Model Example 3 **(b)** Architecture for State Machine Model Example 3

Figure 4.11: Single Input Queue from Multiple Output Queues and Single Output Queue to Multiple Input Queues

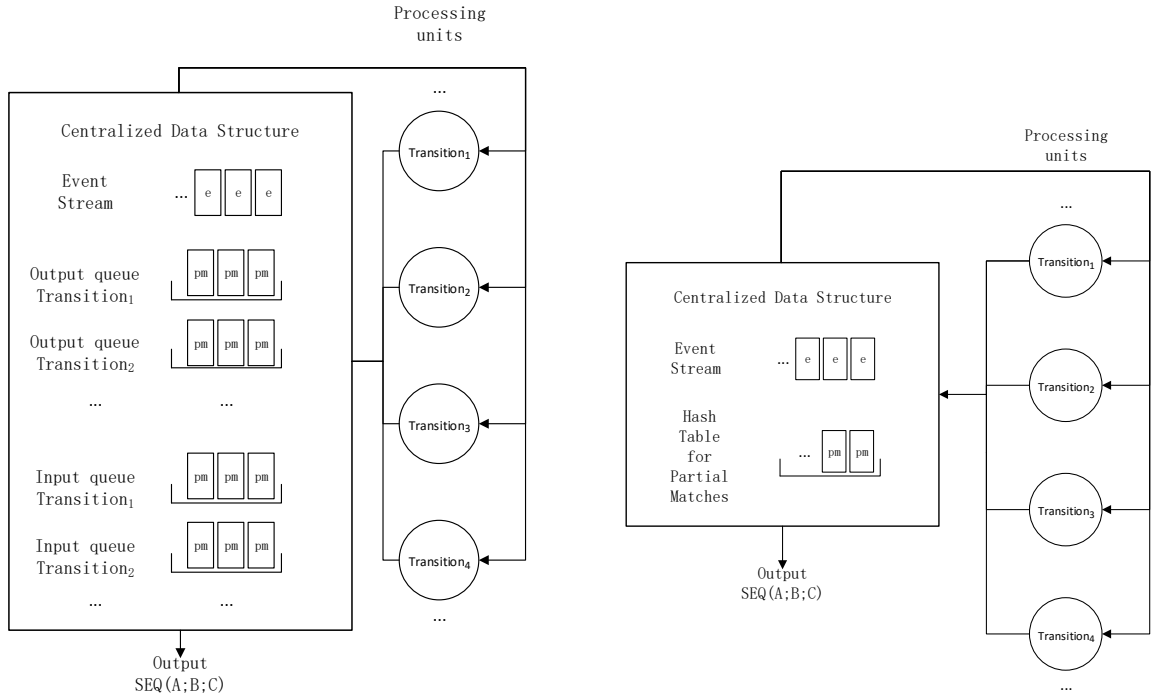
Also, Partial Matches in output queues of $Transition_1$ and $Transition_2$ are sent to two input queues.

The problem is that if the output queue is maintained in the processing unit, then the processing unit needs to know the topology of all processing units. Because it needs to know which processing unit it should send the Partial Match to or should get Partial Match from. Adding extra processing unit will be expensive because all existing processing units need to update the topology.

4.6.1 Centralized Data Structure

To eliminate the dependency between processing units, I would like to introduce a Centralized Data Structure, which maintains all queues.

Each processing unit gets/puts back Partial Match as well as Raw Events from/to the Centralized Data Structure. Therefore, the dependency between processing units



(a) Keep Queues **(b)** Use Hash Map instead of Queues

Figure 4.12: Centralized Data Structure

are eliminated. Only the Centralized Data Structure needs to know the topology of processing units. Figure 4.12a shows such architecture.

Also, pm_i is the state machine instance, which contains the current state S_i in the state machine model, because pm_i contains $path_i$, whose last state is the current state. Therefore, it is unnecessary to maintain so many queues in the Centralized Data Structure. To send the Task $Task_{pm_i}$ to a processing unit, Centralized Data Structure only needs to look up the outgoing edges of state S_i in the state machine model, and send $Task_{pm_i}$ to a processing unit of each Transition. Thus, input queues are no longer needed and a hash table to store the Partial Match in output queues in Centralized Data Structure is enough. The Centralized Data Structure generates new Partial Match according to Task Result returned from processing units and add Partial Match into the hash table. Then the architecture looks like in Figure 4.12b.

Now there is no more output queues and input queues, every Task will be sent to corresponding processing units by looking up the outgoing edges of current state in Partial Match. However, a Task should also contain a Raw Event as an input parameter. So, which Raw Event in event stream should be contained in the Task sent to the

Algorithmus 4.3 Task Creation Algorithm of $RawEventReceiver$

```
procedure CREATE TASKS
    for all existing pm do
        e_{t-1} ← last Combined Raw Event of pm
        e_t ← next Raw Event after e_{t-1} in event stream
        while e_t exists do
            for all outgoing Transition of current state of pm do
                CREATE A TASK(Transition,pm,e_t)
            end for
            e_t ← next Raw Event after e_t in event stream
        end while
    end for
end procedure
```

processing units? In next section, Task Creation Algorithm will be introduced to solve this issue.

4.6.2 Task Creation Algorithm

TESLA gives the operating algorithm in single-thread environment. Every automaton instance should be handled one-by-one. Also a new incoming event should wait unless there is no more automaton instance that needs to be handled. To apply the replication mechanism in multiple-thread environment, it is necessary to keep the result consistent as in the single-thread environment. Therefore, the simplest approach is that a new incoming event waits until all Tasks have got the Task Results back. However, this obviously unacceptable, because it limits the parallelism, since the Raw Event now depends on the result of previous ones.

To solve this problem, each Partial Match should save a record indicating the last Raw Event it combines with to create a Task. For a newly created Partial Match, the initial value of this record is the last Raw Event in Partial Match.

Then the Centralized Data Structure needs to regularly traversal all existing Partial Match to check if the record of each Partial Match meets the last Raw Event in event stream so far. I call the thread doing this job $RawEventReceiver$.

To decrease the workload of $RawEventReceiver$, operator instances are also allowed to create Tasks when a True Task Result is returned. Otherwise, it is very likely that operator instances are waiting for Tasks while the $RawEventReceiver$ is so busy in creating Tasks.

Algorithmus 4.4 Task Creation Algorithm of Operator Instance

procedure CREATE TASKS($TaskResult_{pm_i}$)
  if $TaskResult_{pm_i}$ is True **then**
    for all Raw Event e after the last Raw Event in $(e)_i$ **do**
      for all outgoing $Transition$ of current state of pm_i **do**
        CREATE A TASK($Transition$,pm_i,e)
      end for
    end for
  end if
end procedure

The Task Creation Algorithm is as following.

1. The Raw Events should be saved in a list in the order of arriving.
2. $RawEventReceiver$ regularly traversal all existing Partial Matches, apply Algorithm 4.3
3. When an operator instance returns a Task Result $TaskResult_{pm_i}$, apply Algorithm 4.4

4.6.3 General Processing Units

If we look back to the Split-Process-Merge architecture, we will find that the operator instances are totally identical. This makes it easy to add a new operator instance or remove an old one. It is also easy to pick an operator instance to detect the pattern against the selection given by the splitter, i.e. just to pick any idle operator instance.

Therefore,the processing units in new architecture should also be totally identical.

In current architecture, for each incoming Raw Event e and every pm_i in the Centralized Data Structure, the Centralized Data Structure needs to look up all outgoing Transitions of current state S_i in pm_i, e.g. an Transition $Transition_{i+1}$, and passes the pm_i as well as e to a specific processing unit $Transition_{i+1}$. Thus, this is a Task for specific processing unit $Transition_{i+1}$ and the passed parameters are pm_i and e. The processing unit will return if pm_{i+1} should be generated or not, which is the Task Result.

If the processing unit becomes general, then $Transition_{i+1}$ should also be contained as part of parameters in the Task.

For example, previously, Centralized Data Structure tells the specific processing unit $Transition_{i+1}$ to process pm_i and e. The Task is $Task_{pm_{i+1}} =< pm_i, e >$ and it is sent

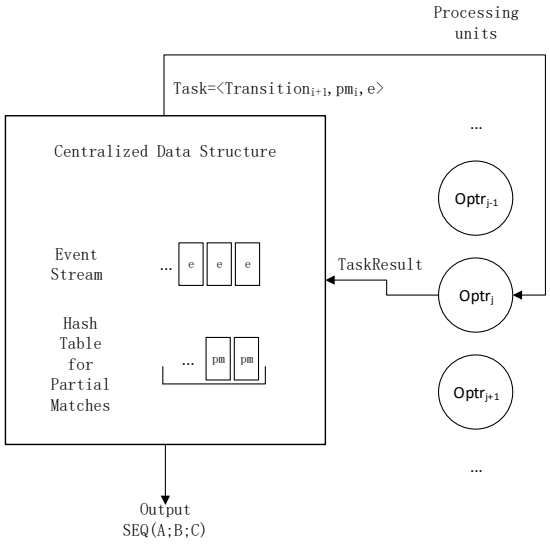


Figure 4.13: Architecture with General Processing Units

to processing unit of $Transition_{i+1}$. The processing unit gives $TaskResult_{pm_{i+1}}$ back to Centralized Data Structure. It can only evaluate $Transition_{i+1}$. Transition other than $Transition_{i+1}$ are not supported.

Now, Centralized Data Structure tells the general processing unit to act as $Transition_{i+1}$ to process pm_i and e. Thus, the Task becomes $Task_{pm_{i+1}} =< Transition_{i+1}, pm_i, e >$. It also requires $TaskResult_{pm_{i+1}}$ back to Centralized Data Structure. But the processing unit can also evaluate other Transition than $Transition_{i+1}$ as far as the role is given in the Task.

Therefore, Transition $Transition_{i+1}$ as well as pm_i and e will be the parameters in a Task sent to the general processing unit. Figure 4.13 gives the new architecture with General Processing Units.

Now the architecture has totally identical operator instances. The Centralized Data Structure creates Task $Task_{pm_{i+1}} =< Transition_{i+1}, pm_i, e >$ according to Task Creation Algorithm, which has been introduced in previous section, and sends the Task to one of operator instances. The operator instance evaluates the condition of $Transition_{i+1}$ against pm_i and e, then returns the Task Result to the Centralized Data Structure.

At this step, the components of architecture are nearly finished. Before introducing the details about algorithms and Centralized Data Structure, I would like to discuss about the Merger, which I skipped in previous section.

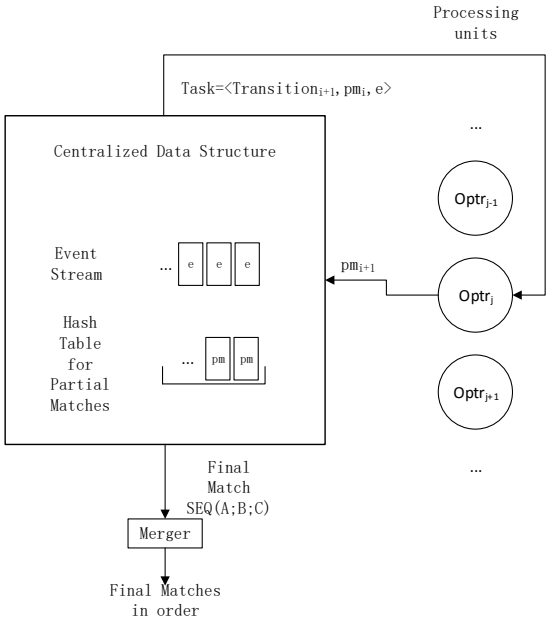


Figure 4.14: Architecture with Merger

4.7 Merger

In Section 4.5.2, multiple processing units of the same Transition are introduced into the architecture. For further processing, a merger is needed to merge the Partial Matches. But the Merger is only needed when the Final Match is fired.

Prove:
For each $Transition_i$ operation, only two input parameters are required: one Partial Match pm_{i-1} and one Raw Event e. There is no second Partial Match required. Therefore, for each operator instance, there is no need to merge two Partial Matches. Thus, no merger is needed before the final state in state machine model is reached.

However, for further processing beyond current architecture, it is possible that some operations required two or more Final Matches as input. Therefore, a merger is needed to merge the Final Matches. Figure 4.14 shows the architecture with merger. The details about Merging Algorithm will be introduced in Section 5.2.

4.8 Summary about Architecture

Till now, all components in the architecture are introduced. There are three major components in this architecture: Centralized Data Structure, Operator Instances, and Merger. Centralized Data Structure is responsible for maintaining all Partial Matches and all Raw Events. It also creates Tasks according to Task Creation Algorithms, accepts Task Results from Operator Instances, and fires Final Matches to Merger. The Centralized Data Structure controls work-flow of the whole system. The Operator Instances are responsible for evaluating the given Tasks and generate a Task Result for each Task. The Operator Instance also take participate in creating Tasks according to Task Creation Algorithm to decrease the workload of $RawEventReceiver$. The Merger is responsible for sorting the Final Matches and informing the Centralized Data Structure which Raw Events are consumed when a Final Match is fired. Since each operator instance handles different transition of the extended Finite State Machine, this is a intra-operator parallelism architecture. Also, because of independency between Partial Matches w.r.t. Transition, data parallelization has been introduced into the architecture, which makes the architecture a data-parallelism architecture as well. The parallelism degree is nearly unbounded. (Yes, it is still bounded by the amount of Tasks can be created, but after the system runs for some time, that value would be very large, compared to the number of states, and the value will grow as the size of data structure grows.) Therefore, "heavy" splitter is avoided, since Centralized Data Structure now has nothing to do with the content of events, and the features of Split-Process-Merge architecture have been kept. In next chapter, the details about each component, algorithms, and API will be introduced. There are also some optimization can be achieved for the new architecture..

5 Details of New Architecture

In this chapter, the details about each component in new architecture will be introduced. Operator Instance will be the first one, since it is the simplest component to describe. The second one will be Merger. Centralized Data Structure will be the last component to explain. After three components, some optimization will be introduced. The last section will be the approach to convert some example SNOOP query to State Machine Model.

Due to the time limitation of this thesis, this chapter will focus on query $SEQ(A; B; C)$ first, which is the major example I used in this thesis. The first occurrence selection strategy, and all selected consumption strategy are also performed. However, the idea is suitable for arbitrary query.

5.1 Operator Instance

The job of Operator Instance is to evaluate a given Task and generates the corresponding Task Result.

To evaluate a given Task, Operator Instance needs to evaluate the Transition Conditions against an old Partial Match together with a new incoming Raw Event, which are all included in the given Task.

This is where the user defined function should be implemented, since Transition Conditions are the only places in the architecture where allow to invoke user defined functions. For example, if the user wants to have $matches$ or $calculateSimilarity$ functions in Transition Condition, then they need to implement these two functions here. Because it is operator instance that invokes user defined functions to evaluate the Transition Conditions.

In later Evaluation Chapter, to simplify the implementation of user defined functions, I used a face detection function from OpenCV to simulate the CPU complexity of evaluating a Task. The details will be shown in Evaluation Chapter.

Algorithmus 5.1 Merging Algorithm

```
procedure RECEIVE FINAL MATCH(finalMatch)
    add into a sorted queue
end procedure
procedure FIRE FINAL MATCH(finalMatch)
    oneFinalMatch ← the first Final Match in the sorted queue
    if CENTRALIZEDDATASTRCTURE.ISEARLIESTFINALMATCH(oneFinalMatch) then
        CENTRALIZEDDATASTRUCTURE.CONSUME(oneFinalMatch)
        fire oneFinalMatch
    else
        put oneFinalMatch back to the sorted queue
        sleep for some time
    end if
end procedure
```

5.2 Merger

The job of Merger is to sort the Final Matches (the definition of order of Final Matches, see Section 4.4.10). The simplest way to sort the Final Matches would be starting to sort Final Matches after all Final Matches arrived, which will definitely lead to huge latency. That is why the simplest way is not acceptable. Besides, for an event stream, it is hard to tell when Final Matches have all arrived.

Therefore, as soon as a Final Match has been detected, the Final Match is stored in a sorted queue. The Merger needs to check if the first Final Match in the sorted queue is the correct one to fire. To achieve this purpose, the Merger needs to know if the first Final Match is the earliest Final Match in the system. I.e. there won't be an earlier Final Match than this one in future. The Merger needs to ask the Centralized Data Structure if this Final Match is the earliest Final Match in the system. One of APIs, $isEarliestFinalMatch()$, in Centralized Data Structure is designed for this purpose. If the Final Match indeed is the earliest Final Match, then the Merger informs the Centralized Data Structure that this Final Match is consumed and fires it. If the Final Match may not be the earliest one, the Merger will put the Final Match back to the sorted queue, wait for some time, pick the first Final Match in the queue, and repeat this procedure again. Algorithm 5.1 shows this Merging Algorithm. The Merger repeatedly invokes FIRE FINAL MATCH method unless the Merger is shutdown.

Yes, busy waiting is used here to avoid additional synchronization. Otherwise, Merger needs to synchronize with Centralized Data Structure every Task Result is returned.

Listing 5.1 APIs supported by Raw Event List

```
add(RawEvent):void
getNextawEvent(RawEvent):RawEvent
consume(RawEvent):void
cleanUp(RawEvent, CounterWindowSize):void
setSynchronizationPoint(RawEventSeqId):void
```

5.3 Centralized Data Structure

The Centralized Data Structure is the most complex component in the architecture. Because it maintains all Partial Matches and all Raw Events as well as controls the work-flow. There are two Data Structures in this component. One Data Structure, Raw Event List, is for maintaining all Raw Events. The other one, Partial Match Data Structure (or PMDS) is used to maintain all Partial Matches.

Since there are different ways to implement the Data Structure, I first list the APIs supported by Centralized Data Structure in Listing 5.1 and Listing 5.2. Following section is the explanation of each API's usage. The implementation details are introduced later. Listing 5.1 gives the APIs supported by Raw Event List.

- $add(RawEvent)$ is used when a new Raw Event arrives from event stream. The new incoming Raw Event is added to the end of Raw Event List.
- $getNextRawEvent(RawEvent)$ returns the next valid Raw Event after the given Raw Event in the Raw Event List.
- $consume(RawEvent)$ will remove the given Raw Event from the Raw Event List. I.e. the given Raw Event will become invalid.
- $cleanUp(RawEvent, CounterWindowSize)$ will remove all Raw Events whose sequence Id are earlier than the sequence Id of given Raw Event minus Counter Window Size. For instance, if the Raw Event Sequence Id is 100, and Counter Window Size is 25, then this method will remove all Raw Events whose sequence Id are earlier than 75 (100 minus 25). This is invoked when a Final Match is consumed. Any Raw Events, whose ids are earlier than the id of first Raw Event in the Final Match minus counter window size, should be useless so far. Otherwise, the $isEarliestFinalMatch(FinalMatch)$ should prevent this Final Match to fire because there may be an earlier Final Match than this one.
- $setSynchronizationPoint(RawEventSeqId)$ is used to set a synchronization point. The synchronization point well be introduced in Section 5.4.2.

Listing 5.2 APIs supported by Partial Match Data Structure

```
iterator():Iterator<PartialMatch>
receiveTaskResultAndGetNewTasksAndPartialMatch(TaskResult):Tasks and PartialMatch
isEarliestFinalMatch(FinalMatch):boolean
consume(FinalMatch):boolean
cleanUp()
```

Listing 5.2 gives the APIs supported by Partial Match Data Structure.

- $iterator()$ returns an iterator to traversal all Partial Matches in the Data Structure. The iterator is used in Task Creation Algorithm(See Section 4.6.2) to reach all Partial Matches.
- $receiveTaskResultAndGetNewTasksAndPartialMatch(TaskResult)$ is also used in Task Creation Algorithm(See Section 4.6.2). The Task Creation Algorithm says when a Task Result is returned, the Centralized Data Structure should generate a new Partial Match or not according to the Task Result. It also needs to create new Tasks if a new Partial Match is indeed created.
- $isEarliestFinalMatch(FinalMatch)$ is used by Merger to check if a Final Match is the earliest Final Match in the system.
- $consume(FinalMatch)$ is used by Merger to consume all related Raw Events according to Consumption Strategy when a Final Match is fired.
- $cleanUp()$ is used for optimization purpose, which will be introduced in Section 5.4.2.

5.3.1 Details of Raw Event List

Since Raw Event List needs to be a list and it also needs to have the feature of fast access (because of $getNextRawEvent(RawEvent)$), the ConcurrentSkipListMap in java is an excellent choice for this Data Structure. The ConcurrentSkipListMap uses a concurrent variant of skip list to store the keys[Ora16]. The nodes in the skip list uses compare-and-set algorithm for concurrent modification. The Raw Event Sequence Id can be used as a key for fast accessing. The time complexity for $add(RawEvent)$, $getNextRawEvent(RawEvent)$, and $consume(RawEvent)$ is $log(n)$. The time complexity for $cleanUp(RawEvent, CounterWindowSize)$ is also not expensive because it only needs to remove the head sub map or maybe just reset the header pointer if implement the Raw Event List by oneself.

5.3.2 Details of Partial Match Data Structure

Currently a hash table is used in the Centralized Data Structure to maintain all Partial Matches. This works fine if a Partial Match will never become invalid. (E.g. because of consumption problem or counter window size constraints)

However, when consumption problem, which will be discussed in details in later sections, is taken into consideration, a Partial Match pm_j will become invalid if another Partial Match pm_i which pm_j depends on is consumed. Also, a Partial Match pm_j may become invalid because of counter window size constraints. For example, if the query gives a counter window size as 25 events, then all Partial Matches earlier than the 25-th Raw Event and those Partial Matches depend on these ones should become expired.

All expired Partial Matches should no longer take participate in creating new Tasks and they should wait for deleting at a certain time point (See Section 5.4.2).

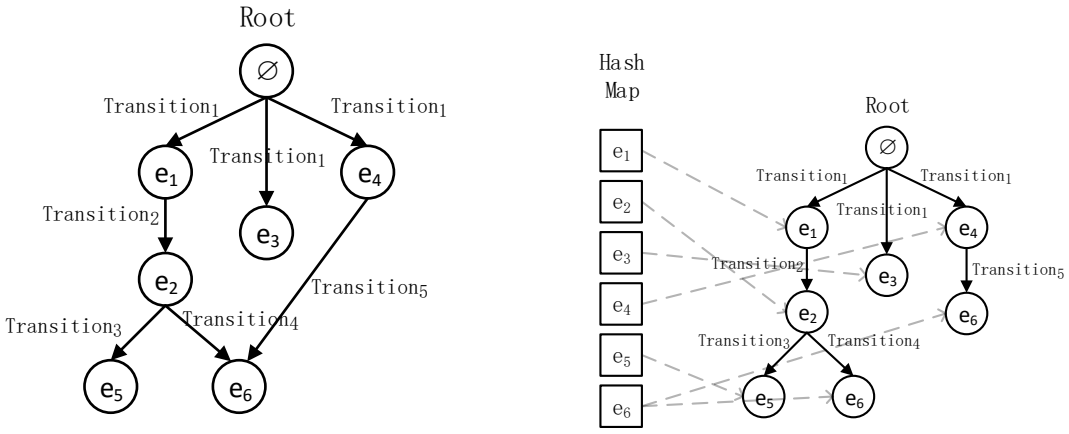
If hash table is still used under this scenario, it is a huge execution cost to discard all invalid Partial Matches because the Centralized Data Structure needs to go through all Partial Matches to discard them according to the dependencies introduced in Section 4.4.11 and Section 4.4.12.

According to the dependency between Partial Matches w.r.t. Raw Events, pm_i depends on pm_j is equivalent to $path_j \subset path_i$ and $(e)_j \subset (e)_i$. Therefore, to store pm_i, only the additional information other than pm_j, i.e. additional Transition and Raw Events, are needed to store in the data structure. There are two ways to achieve this: tree or graph.

Partial Match Graph

Figure 5.1a is an example of Partial Match Graph. The graph is built up by nodes and edges. Each node contains one Raw Event. Also each Raw Event only appears in one node. Each edge connects two nodes and edge has direction. The graph has a “Root” node. The Root node is a special case. Root node represents the initial Partial Match pm_0, thus Root node contains $\varnothing$ as the Raw Event. Edge represents a transition from one state to another state, which results a new Partial Match. Therefore, a path from Root node to another node represents a Partial Match.

In Figure 5.1a, there are 6 nodes, excluding the Root node. There are also many paths in this graph. Each path represents a Partial Match. E.g. path $\varnothing \rightarrow e_1$ represents a Partial Match $pm_1 = ((Transition_1); (e_1))$; path $\varnothing \rightarrow e_1 \rightarrow e_2$ represents a Partial Match $pm_2 = ((Transition_1, Transition_2); (e_1, e_2))$.



(a) Example of Partial Match Graph

(b) Example of Partial Match Tree

Figure 5.1: Two Partial Match Data Structure

Obviously, pm_2 depends on pm_1 w.r.t. Raw Events. The dependency is expressed in the graph in such way that one node is a successor of another node.

A detailed discussion about consumption problem and algorithm is in later part of this section. But in simple words, to consume one Raw Event or to discard an invalid Partial Match, the Data Structure only needs to delete a certain node and then check if there are any unreachable nodes from the Root node. E.g. in this graph, if e_2 is consumed, e_5 will become unreachable so that e_5 should also be deleted.

Partial Match Tree

Figure 5.1b is an example of Partial Match Tree. The tree is built up by nodes and edges. Each node contains one Raw Event, but each Raw Event may appear in different nodes. Each edge connects two nodes. The tree has a Root node, which contains $\varnothing$ as Raw Event. Since, a path from the Root node can represent a Partial Match, and the path to each tree node is unique, each tree node in the tree is actually a Partial Match.

In Figure 5.1b, there are 7 tree nodes, but there are only 6 Raw Events. Raw Event e_6 appears in two tree nodes, which means it appears in two Partial Matches. To explain "each tree node is a Partial Match", let's see some examples. Node e_1 is a Partial Match $pm_1 = ((Transition_1); (e_1))$. Node e_2 is a Partial Match $pm_2 = ((Transition_1, Transition_2); (e_1, e_2))$. There are two Nodes e_6. Therefore, there are

two Partial Matches $pm_{6.1} = ((Transition_1, Transition_2, Transition_4); (e_1, e_2, e_6))$ and $pm_{6.2} = ((Transition_1, Transition_5); (e_4, e_6))$.

Obviously, pm_2 depends on pm_1 w.r.t. Raw Events. The dependency is expressed in the tree in such way that one tree node is a successor of another tree node.

To access a specific Raw Event e_i in the tree in a fast way, a hash map is needed to store the location on each Raw Event in the tree. This is the extra cost compared to the tree without hash map and to the graph, because the hash map needs to update when new Partial Match is generated. In the tree without hash map, no hash map needs to maintain. In the graph, the hash map only needs to update when a new node is created.

So far, data structure has been introduced. In following sections, the algorithms of API will be explained.

5.3.3 Iterator

The iterator is used to traverse the Data Structure to get all existing Partial Matches. This can be done by depth-first-search or breadth-first-search. Also, there is not much space to optimize the traversal algorithm.

5.3.4 Receive Task Result

This method is used to receive Task Result and to create a new Partial Match and Tasks if the Task Result is $true$. The complexity of this algorithm is mainly in searching the parent Partial Match in the data structure. To avoid searching, the parent Partial Match stores a pointer pointing to the node, which represents the parent Partial Match, so that the node can be directly accessed when the Task Result is returned. The complexity of this algorithm becomes O(1).

5.3.5 isEarliestFinalMatch

This method is used by Merger to check if a given Final Match is the earliest Final Match in the system.

For Partial Match Tree, only the left sub trees of given Final Match needs to check. There are two conditions need to check:

1. All Partial Matches in the left sub trees of given Final Match should have already been combined to a Raw Event whose sequence Id meets one of following conditions:
   a) higher than the sequence Id of last Raw Event in Final Match.
   b) equals to the synchronization point.
   c) reaches the maximum counter windows size.
2. All Partial Matches in the left sub tree of given Final Match should have no earlier Tasks than a Task contains the last Raw Event of given Final Match.

If both conditions are met, then the given Final Match is the earliest Final Match in the system.

For Partial Match Graph, all Partial Matches containing a Raw Event which is earlier than the last Raw Event in Final Match need to check. There are also two conditions need to check. The two conditions are the same as the ones in Partial Match Tree. If both conditions are met, then the given Final Match is the earliest Final Match in the system.

5.3.6 Consumption

One of APIs of Partial Match Data Structure is $consume(FinalMatch)$, which means no matter tree or graph, with or without hash map, they should support to consume a Final Match. However, so far, consumption problem is never considered, because once consumption problem is taken into consideration, the dependency between Partial Matches will be changed.

The definition of consumption is such that once a Raw Event e or a Partial Match pm_i is consumed, this Raw Event e or Partial Match pm_i will never show up in further processing and have no more impact to the system.

Without consumption, pm_j depends on pm_i if and only if $path_{pm_i} \subset path_{pm_j}$ AND $(e)_{pm_i} \subset (e)_{pm_j}$. Also, a new generated Partial Match has no influence on existing ones. (i.e. a future result will not influence a past result.)

If the consumption problem is taken into consideration, the system becomes acausal system (a term in control theory, means current result not only depends on previous result but also depends on future result), because pm_j also depends on the *FUTURE* consumption result of pm_i. I.e. existence of pm_j depends not only on current pm_i but also on the fact that pm_i will not be consumed in future. This is what makes the problem complex.

Algorithmus 5.2 Consumption Algorithm for Graph

```
procedure CONSUME(list of Raw Events to consume)
    for all Raw Event e in list of Raw Events to consume do
        node_i ← get the node contains e
        remove all edges pointing to node_i
        remove node_i
    end for
    REMOVE UNREACHABLE NODES
end procedure
```

Algorithmus 5.3 Consumption Algorithm for Tree

```
procedure CONSUME(list of Raw Events to consume)
    for all Raw Event e in list of Raw Events to consume do
        setOfNodes ← get the nodes contain e
        for all Node n in setOfNodes do
            remove the sub-tree of Node n
        end for
    end for
end procedure
```

To solve this problem in an easy way is to apply following two steps

1. Assume Partial Match will never be consumed in future, which makes the system back to causal system again and behave as previously.
2. Once a Partial Match does be consumed in future, invalidate all existing Partial Matches which depend on this consumed Partial Match.

Now a consumption algorithm is needed to invalidate all existing Partial Matches which depend on a consumed Partial Match.

Consumption Algorithm

The algorithm is used to invalidate a set of Partial Matches when a Raw Event or a Partial Match is consumed. However, the nature of consumption of a Partial Match pm_i actually is the consumption of a sequence of Raw Events $(e)_i$ rather than one Raw Event. Therefore, the problem becomes "how to invalidate a set of Partial Matches when one or more Raw Events are consumed".

With the help of the data structure introduced in Section 5.3.2 the algorithm becomes more efficient, compared with going through all Partial Matches in a hash table.

Algorithm 5.2 is for Partial Match Graph. To consume a list of Raw Events in Partial Match Graph, following steps need to execute in sequence.

1. Find the nodes containing the target Raw Events
2. Remove all edges pointing to these nodes
3. Remove these nodes, which containing the target Raw Events
4. Check the graph to see if there are any unreachable nodes. If yes, then remove all unreachable nodes in the graph.

Algorithm 5.3 is for Partial Match Tree. To consume a list of Raw Events in Partial Match Tree, following two steps need to execute in sequence.

1. Find the nodes containing the target Raw Events
2. Remove the sub-tree of these nodes

The complexity of these two algorithms will be discussed in next section.

5.3.7 Complexity Analysis of Data Structure

The Partial Match Data Structure is to store and maintain Partial Matches. There are three data structures to be analyzed: Tree w/o Hash Map, Tree with Hash Map and Graph. Each of them has its own properties, which will be discussed now.

There are three major operations in Partial Match Data Structure:

1. Traversal
2. Add new Partial Match
3. Consume a Final Match

As described in Section 5.3.3, the complexity of traversal are almost same and the complexity depends on the number of nodes in data structure. Obviously, the tree will have more nodes than graph, but the complexity still remains in the same complexity class. The complexity of adding a new Partial Match can be improved to O(1) as described in Section 5.3.4. The most difference of three data structure is in consuming a Final Match.

Partial Match Tree w/o Hash Map

For Tree w/o Hash Map, consumption contains two steps: Search and Remove.

- Search: Search for a group of tree nodes which contain the Raw Event included in a given set of Raw Events.
  Complexity: O(All nodes in tree) i.e. O(Size of tree)
- Remove: Remove the group of nodes from the tree.
  Complexity: size of group $\times O(1) = O$(size of group)

Thus, the overall complexity is $O(|Tree|) + \sum_{i=1}^{\text{size of Raw Event Set}} (O(|group_i|))$

Partial Match Tree with Hash Map

For Tree with Hash Map, consumption contains three steps: Remove, Update and Check dead link.

- Remove: Get the set of links from hash map. Remove the nodes attached to the links.
  Complexity: size of links $\times O(1) = O$(size of links)
- Update: Remove the <key, value> pair from hash map. Complexity: $O(1)$
- Check dead link: Remove from hash map the links pointing to the nodes which are in the sub-tree of deleted node.
  Complexity: size of sub-tree $\times O(2) = O$(size of sub-tree)

Thus, the overall complexity is $\sum_{i=1}^{\text{size of Raw Event Set}} (O(\text{size of links}) + O(1) + O(\text{size of sub-tree}))$

Partial Match Graph

For Partial Match Graph, consumption contains these steps: Remove, Check unreachable nodes, and Update.

- Remove: Get the link to the node according to Hash Map. Remove the node. I.e. delete a given node and its all incoming links.
  Complexity: $O(1) + O$(size of incoming links)

- Check unreachable nodes: Check if there is any isolated nodes or isolated sub-graph in graph. This is done by traversing the whole graph first and the nodes not reached are unreachable nodes.
  Complexity: $O(\text{size of graph})$
- Update: Remove unreachable node from Hash Map
  Complexity: $O(\text{size of unreachable nodes})$

Thus, the overall complexity is $O(\text{size of graph}) + O(\text{size of unreachable nodes}) + \sum_{i=1}^{\text{size of Raw Event Set}} (O(1) + O(\text{size of incoming links})_i)$

Summary

First, we compare Tree w/o Hash Map and Tree with Hash Map.

$O(\text{size of group})$ and $O(\text{size of links})$ are in the same complexity class. It depends on the number of occurrence of *ONE* Raw Event.

In Tree w/o Hash Map, $O(|Tree|)$ is independent to the size of given Raw Event Set. The corresponding component in Tree with Hash Map is $O(\text{size of sub-tree})$. In worst case, $\sum O(\text{size of sub-tree})$ equals to $O(|Tree|)$.

$O(1)$ in Tree with Hash Map is the extra overhead to maintain the hash map.

Then, we compare Tree with Hash Map and Graph.

$O(1)$ is the overhead to maintain the hash map.

$O(\text{size of links})$ and $O(\text{size of incoming links})$ are in the same complexity class.

In Graph, $O(\text{size of graph})$ is independent to the size of given Raw Event Set. The corresponding component in Tree with Hash Map is $O(\text{size of sub-tree})$, but the complexity is still in the same class as $O(\text{size of graph})$.

However, please notice that it is unnecessary to check unreachable nodes in graph for every consumption. Because unreachable nodes won't influence the system except they cost some memory. Therefore, the clean-up of unreachable nodes can be delayed to the synchronization point(See Section 5.4.2) Then the overall complexity for each consumption in graph will become $\frac{O(\text{size of graph})+O(\text{size of unreachable nodes})}{\text{number of consumption invoked between two synchronization points}} + \sum_{i=1}^{\text{size of Raw Event Set}} (O(1) + O(\text{size of incoming links})_i)$.

5.4 Optimization

So far, the functionality of the architecture has been introduced. However, there are possibilities to improve the performance. In this section, two approaches are provided to improve the performance: Task Scheduling (or Priority Task Queue) and Synchronization Point. Task Scheduling (or Priority Task Queue) is to prioritize a Task which is more likely to be a necessary Task. Synchronization Point is to synchronize the progress between Task Creation, Task Evaluation, and Final Match Consumption.

5.4.1 Task Scheduling (or Priority Task Queue)

Task Scheduling is to use a Priority Task Queue to prioritize a Task which is more likely to be a necessary Task. The scheduling strategy is query dependent.

For example, given the query $SEQ(A; B; C)$, a "longer" Task is more likely to be a necessary Task as well as an "earlier" Task is more likely to be a necessary Task. A "longer" Task is a Task has more Raw Events in the Partial Match it carries. E.g. Given a Raw Event e, a Task carrying Partial Match AB is longer than a Task carrying Partial Match A. The first Task is more likely to become a Final Match. An "earlier" Task is a Task carries an earlier Partial Match. For instance, if A_1 is earlier than A_2 in event stream, then a Task carrying Raw Event A_1 should have higher priority than a Task carrying Raw Event A_2 because of first occurrence selection policy. Therefore, for query $SEQ(A; B; C)$, the strategy "longer Task, higher priority; earlier Task, higher priority" can be used.

While in another example, given the query $OR(SEQ(A; D), SEQ(A; B; C))$, previous scheduling strategy won't work as well as previously. Because given a Raw Event e, a Task with Partial Match A and a Task with Partial Match AB have the same probability leading to a Final Match.

Thus, the Task Scheduling Strategy is query dependent. The domain expert needs to give a suitable Task Scheduling Strategy according to the query.

Since the implementation is based on query $SEQ(A; B; C)$, the strategy "longer Task, higher priority; earlier Task, higher priority" is used as default strategy in following chapters.

It is also possible not to use Task Scheduling Strategy. Then the Task Queue is called Simple Task Queue, which applies FIFO rules.

5.4.2 Synchronization Point

Synchronization point is to divide the incoming Raw Events into blocks. The end of each block is the Synchronization Point. The amount of Raw Events between two Synchronization Point is the Raw Event Block Size. At the end of each block, the system forces all components to rendezvous at that point. This mechanism is used to avoid such situation that the Tasks are created in a very fast rate while all unnecessary Tasks have higher priority in Task Queue because of a bad Task Scheduling Strategy. (Actually, it is very hard to get a perfect Task Scheduling Strategy.) In such situation, it is very likely that no Final Match can be fired, the system is working on all unnecessary Tasks, and the Partial Match Data Structure keeps growing until the system is out of memory.

Thus, to avoid such situation, a Synchronization Point is introduced. When the synchronization point arrives, no more Tasks will be created unless all created Tasks are processed and all potential Final Matches are fired. Also, the Synchronization Point can be used to indicate the time point to clean up the Partial Match Data Structure. E.g. Clean up all unreachable nodes in Partial Match Graph.

5.5 Improved Finite State Machine

As described in Chapter 4, the new architecture is established on the Improved Finite State Machine, however, the user is more similar with the CEP query language. Therefore, an approach is needed to convert the CEP query language into Improved Finite State Machine. The Improved Finite State Machine generated from a query is called State Machine Model. A State Machine Model maps to a unique query, while a query can map to several State Machine Model. In this section, I will show some examples how to convert the SNOOP query into a State Machine Model.

5.5.1 SEQ Operator

For a given pattern $SEQ(E_1; E_2; E_3; \ldots ; E_n)$, the State Machine Model is shown in Figure 5.2.

A Partial Match pm_1 entered into State S_1 is created if the initial Partial Match pm_0 receives event E_1 and meets related Transition Constraints. Similarly, Partial Matches entered into State S_{12},S_{123},$S_{123\ldots n}$ are created if pm_1 receives event E_1,E_2,E_3, ... ,E_n and meets Transition Constraints respectively.

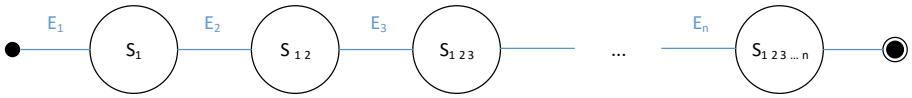


Figure 5.2: State Machine Model of SEQ Operator

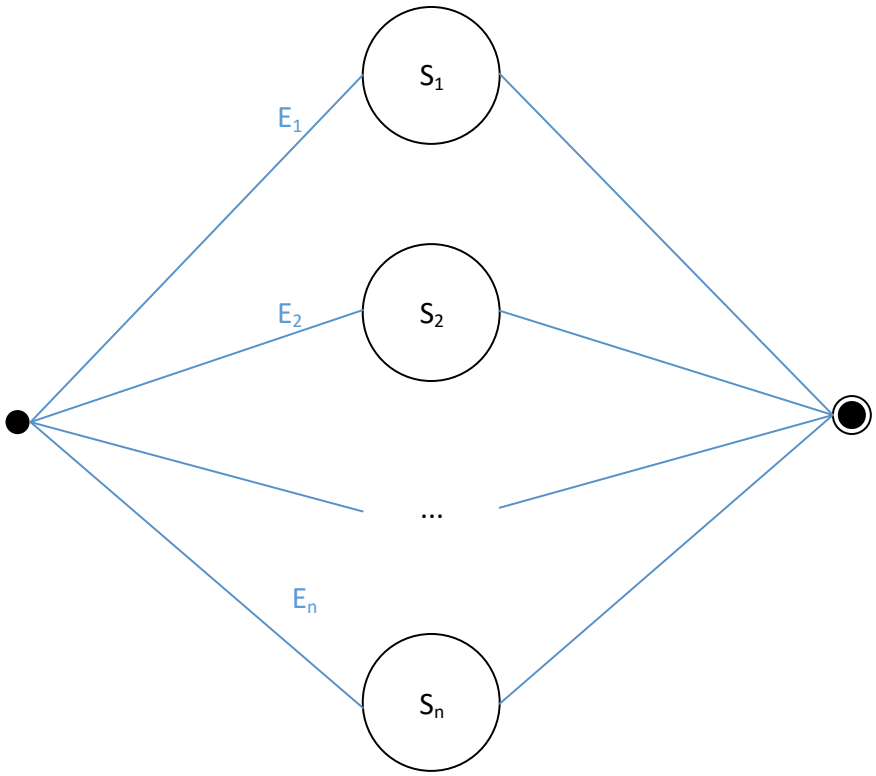


Figure 5.3: State Machine Model of OR Operator

5.5.2 OR Operator

For a given pattern $OR(E_1, E_2, \ldots, E_n)$, the State Machine Model is shown in Figure 5.3. Different Raw Events will lead to different states according to Transition Condition.

5.5.3 ANY and ALL Operator

Since “ALL” operator is a special case of “ANY”, “ALL” operator will be shown first. The model of “ANY” operator is a brief version of the model of “ALL”.

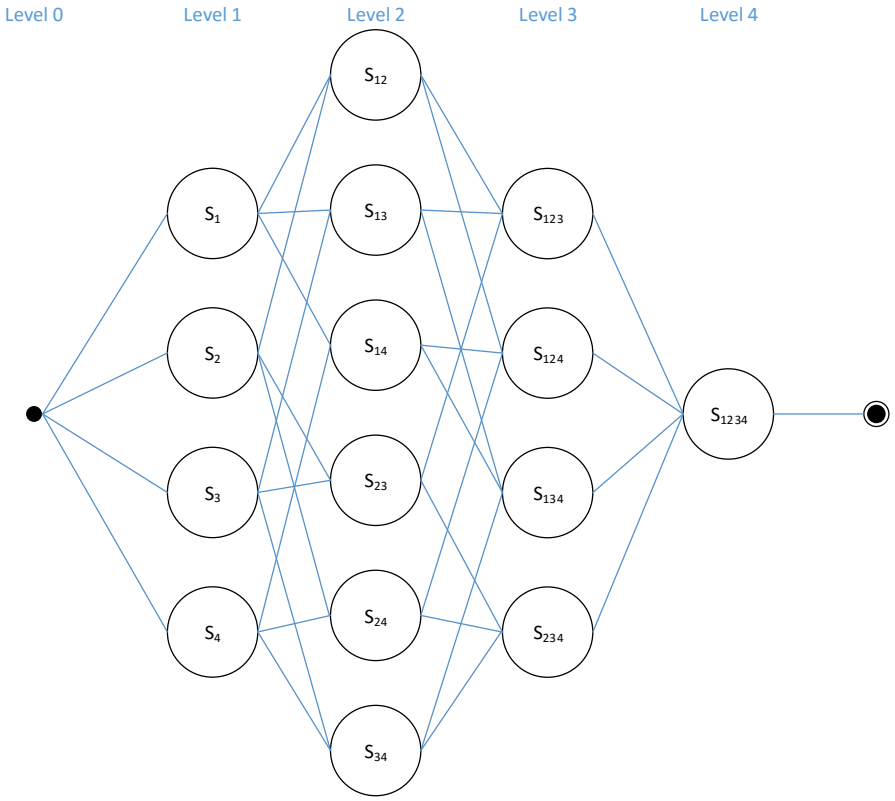


Figure 5.4: State Machine Model of $ALL(E_1, E_2, E_3, E_4)$

To explain "ANY" and "ALL" operator in a more detailed way, a detailed pattern $ALL(E_1, E_2, E_3, E_4)$ will be used as an example first.

The State Machine Model of $ALL(E_1, E_2, E_3, E_4)$ is shown in Figure 5.4

A Partial Match pm_1 entered one of States S_1,S_2,S_3, or S_4 is created when pm_0 receives one of Events E_1,E_2,E_3, or E_4. Then Partial Match pm_2 which enters into next level of states is created if pm_1 receives second Raw Event. For example, if the incoming Raw Event stream is as following sequence: E_2;E_3;E_1;E_4, then pm_1 which enters into S_2 in the first level is created, pm_2 which enters into S_{23} in the second level is created, pm_3 entering into S_{123} in the third level is created, and finally pm_4 entering into S_{1234} in the last level is created.

Then the general State Machine Model of "ALL" operator is shown in Figure 5.5

For a given $ALL(E_1, E_2, E_3, \ldots, E_n)$, the model will have n levels. At level k, there will be $\binom{k}{n}$ (binomial coefficient) states. Notes: The naming of States indicates what Raw

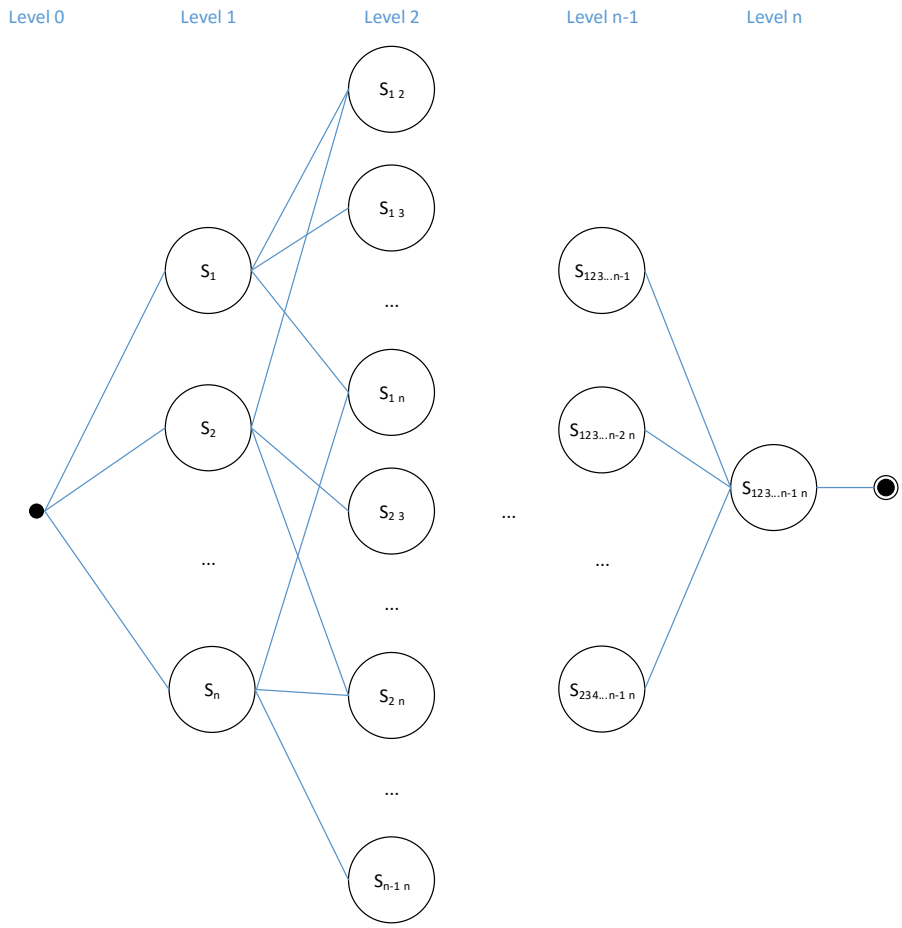


Figure 5.5: State Machine Model of ALL Operator

Events have already been received. E.g. State S_{123} means the Partial Match has already received Events E_1,E_2,E_3.

Since $ALL(E_1, E_2, E_3, \ldots, E_n)$ is a short form of $ANY(n, E_1, E_2, E_3, \ldots, E_n)$, for a given pattern $ANY(m, E_1, E_2, E_3, \ldots, E_n)$, the State Machine Model is the first m levels of the model of $ALL(E_1, E_2, E_3, \ldots, E_n)$.

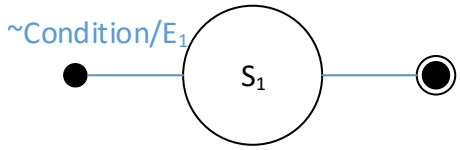


Figure 5.6: State Machine Model of NOT Operator

5.5.4 NOT Operator

The State Machine Model of “NOT” operator is to set the negation of related condition as shown in Figure 5.6

6 Evaluation

In this chapter a group of evaluation results will be shown. These evaluations are taken place on Google Cloud Compute Engine. Section 6.1 is the general view of the evaluation. The hardware and software environment information will be listed in section 6.1.1. Section 6.1.2 will show the evaluation configuration, i.e. what parameters are evaluated. Before the detailed discussion about the evaluation results in Section 6.3, some terminology will be introduced in Section 6.2.

6.1 Environment Setup and Evaluation Configuration

6.1.1 Environment Setup

The evaluations are taken place on Google Cloud Compute Engine. In the evaluation, several types of Google Cloud Compute Engine are used:“n1-standard-1”, “n1-standard-2”, “n1-standard-4” “n1-highcpu-8”, and “n1-highcpu-16”. The last number indicates the number of vCPU cores. I.e. “n1-standard-1” has 1 vCPU. “n1-highcpu-16” has 16 vCPUs. The type “High CPU”, as described by Google, is “High-CPU machine types are ideal for tasks that require more virtual CPUs relative to memory. High-CPU machine types have 0.90 GB of RAM per virtual CPU.”[Goo16]. While the “Standard” type has the same CPU performance but with a larger RAM. Because for the program, a minimum RAM should be guaranteed. If a “High CPU” type with small amount of vCPU cores is used, the RAM is too small for the program. Therefore, for 1, 2, and 4 vCPU cores, the type “Standard” is chosen.

The OS in the evaluations uses Ubuntu 16.04 LTS, provided by Google together with the Compute Engine. The JVM uses JavaSE8.

There are two third party components, OpenCV 3.1[Its16] and Sigar 1.6.4[Hyp16]. OpenCV is used to simulate the heavy weight operation. The heavy weight operation, to be more detailed, is to detect a face in an image. The cost time of heavy weight operation depends on the size of image and the classifier used. Sigar is used to measure CPU Load during evaluation.

Listing 6.1 Evaluation Configuration Parameters

```
Number of vCPU cores and Event Generation Rate:
      see Table 6.1 and Table 6.2
Query:
      SEQ(A;B;C)
Counter Window Size:
      25 (events for each Event, i.e. the max Global Counter Window Size is 75)
Image Size:
      80x60, 160x120, 320x240
Event Generating Time:
      1 hour for Latency test
      30 min for Throughput test
Task Queue Type:
      Simple Task Queue (FIFO)
      Priority Task Queue (Longer Task, higher priority; Earlier Task, higher priority)
Partial Match Data Structure:
      Tree w/o Hash Map
      Tree with Hash Map
      Graph
Raw Event Block Size:
      4, 16, 64, 256, 1024, 4096, 16384, 65536, 131072
```

6.1.2 Evaluation Configuration

During the evaluations, two important variables are measured, throughput and latency. Since latency is meaningless if the program is overloaded, because all Events will be queued up and waiting for processing, the longer running time, the larger latency, therefore a throughput test is executed first to determine the throughput against all configuration parameters. Then an Event Generation Rate, which is slightly smaller than the minimum throughput in the throughput test, is picked to use in the latency test to avoid overloading the system. Thus, the first two evaluation configuration parameters are Event Generation Rate and number of vCPU cores. The other parameters are listed in Listing 6.1. All these parameters will be explained in following subsections.

Number of vCPU cores and Event Generation Rate

In the throughput test, a large enough Event Generation Rate(Unit: events per second) is used to overload the system, as shown in Table 6.1. The program only exits when all Raw Events generated have been processed. Any larger value is also acceptable, but it will take too much time for the program to finish its job. Therefore, the throughput can be calculated by dividing the $total\ events\ generated$ by $total\ running\ time$, i.e. $throughput = \frac{total\ events\ generated}{total\ running\ time}$.

# of vCPU cores	Event Generation Rate
1	30
2	30
4	100
8	120
16	200

Table 6.1: Event Generation Rate in throughput test w.r.t number of vCPU cores

# of vCPU cores	Event Generation Rate
1	9
2	15
4	25
8	45
16	75

Table 6.2: Event Generation Rate in latency test w.r.t number of vCPU cores

In the latency test, the Event Generation Rate is chosen slightly smaller than the minimum throughput in throughput test, as shown in Table 6.2, so that the machine won't be overloaded.

Example Scenario and Query, Counter Window Size

In Section 3.2.2, Application 2 shows a scenario about object recognition and behavior pattern detection for auto-driving/alarm system. Here, to simplify the scenario a little bit more, I use query $SEQ(A; B; C)$ instead of $ALL(SEQ(E_1; E_2), E_3)$. Instead of looking for sequence E_1, E_2, E_3, or E_3, E_1, E_2, I just look for sequence A, B, C, i.e. one of two cases in Application 2. In Application 2, the user defined function $matches$ should do object recognition. Here I use OpenCV face detection function to simulate the CPU complexity of object recognition. The CPU complexity can be adjusted by resizing the image size or using other classifiers. I also set Counter Windows Size as 25, which means an event will expire after 25 events if there is no Partial Match detected. For instance, an event A will expire after 25 events unless there is B event in these 25 events. If there is a sequence A and B has been detected, then the expiration of this Partial Match is determined by the event B. I.e. if there is no C event in 25 events after B event, the Partial Match sequence AB will be expired.

The threshold 25 is chosen according to the throughput. If the Counter Window Size is too small, then the Partial Match Data Structure will always remain small because the

query is too simple. The Event Generation Rate needs to be very large to overload the program. If the Counter Windows Size is too large, then the query will be too complex and Partial Match Data Structure will be large, which leads to a very small throughput and it is hard to compare among different configurations if all throughput are small and close to each other.

Together with the query $SEQ(A; B; C)$, the first occurrence selection strategy is used. I.e. If there are many A events followed by a B event and a C event, the first A event occurred is selected to generate a Final Match ABC. The consumption strategy is that all selected events will be consumed. I.e. In sequence ABC, all of three events will be consumed once the Final Match ABC is fired.

Image Size

There are three image sizes: 80x60, 160x120, and 320x240. Different image sizes can be used to simulate different CPU complexities because the OpenCV needs more time on a singe CPU to detect a face in larger image. Image size 80x60 and 320x240 are only used for testing the performance w.r.t. different CPU complexity operations. In all other performance tests, the image size is fixed to 160x120. These image size 160x120 is also chose according to the throughput so that a suitable throughput can be obtained for comparison. The other two are just half or doubled image size.

To achieve a suitable throughput baseline, for image size 160x120, the face detection classifier uses “lbpcascade_frontalface.xml” which comes together with OpenCV package. The image size 320x240 uses the same classifier. While for image size 80x60, another classifier “haarcascade_frontalface_alt_tree.xml” is used so as to achieve a more less CPU complicity operation.

Event Generating Time

Event Generating Time is the time duration that how long the events are generated at a given Event Generation Rate for. E.g. 30 min for throughput test at Event Generation Rate 120. Events will be generated at 120 events per second for 30 minutes in throughput test.

Notice: Event Generating Time is NOT the $total\ running\ time$. $Total\ running\ time$ is the time duration that how long does the program need to process all events generated. In throughput test, the $total\ running\ time$ will be much longer than Event Generating Time. While in latency test, the $total\ running\ time$ obtained should be equal to Event Generating Time.

Task Queue Type

See Section 5.4.1. Task Queue Type is the scheduling strategy used in Task Queue. A Simple Task Queue uses FIFO strategy, i.e. a Task put into Task Queue earlier will be processed by operator instance earlier. A Priority Task Queue uses "Longer Tasks, higher priority;earlier Task, higher priority" strategy. For example, there are two Tasks and each Task containing one Partial Match. If PM_1 is longer than PM_2, i.e. PM_1 contains more Raw Events than PM_2, then the Task containing PM_1 will be processed by operator instance earlier. If PM_1 and PM_2 have the same length, then the Task containing earlier Raw Event will be processed earlier.

Partial Match Data Structure

There are three Partial Match Data Structure introduced in this thesis: Partial Match Tree w/o Hash Map, Partial Match Tree with Hash Map, and Partial Match Graph. Please see Section 5.3.2 for details.

Raw Event Block Size

The Raw Event Block Size is the number of Raw Events between two Synchronization Points. The CEP system will process the Raw Events block by block. Please see Section 5.4.2 for details.

6.2 Terminology Used in Evaluation

Definition 6.2.1 (Raw Event Processing Latency)
The elapsed time between the Raw Event put into Raw Event List and it is totally processed. A Raw Event is considered as totally processed when it has no more impact on the Partial Match Data Structure.

A new incoming Raw Event should be combined to all existing Partial Matches to create new Tasks. Therefore, after this combination is finished, this Raw Event has no more influence to the Partial Match Data Structure. Because following Raw Events will combine to the Partial Matches, not to this Raw Event.

However, it is hard to check whether a Raw Event has finished combining all existing Partial Matches in parallel system. I use “last taken time” as the time point when the Raw Event is

totally processed. Whenever a Raw Event has been accessed to create a new Task, the "last taken time" will be updated.

Definition 6.2.2 (Raw Event Queuing Time)
The time between the Raw Event put into the Raw Event List and it is first time accessed by an operator instance.

Definition 6.2.3 (Raw Event Life Time)
The time between the Raw Event put into the Raw Event List and it is removed because of consumed or expired.

Definition 6.2.4 (Task Queuing Time)
The time between the Task is created and it is taken from Task Queue by an operator instance.

Definition 6.2.5 (Task Evaluation Cost Time)
The elapsed time between the Task taken from Task Queue and all Transition Conditions are evaluated.

Definition 6.2.6 (Task PMDS Processing Latency)
The time used to manipulate the Partial Match Data Structure. I.e. Add a new Partial Match and create new Tasks.

Definition 6.2.7 (Task Processing Latency)
The time between the Task is created and it is totally processed.

A Task can only have following ending.

1. Invalid Task

An invalid Task will not be evaluated. It will be treated as a False Task Result immediately after it is taken from the Task Queue by an operator instance. The operator instance removes the Task from the tracking list in the Partial Match Data Structure

2. Valid Task with False Task Result

The Task is valid, but the evaluation gives a False Task Result, which means it doesn't pass the evaluation. The operator instance removes the Task from the tracking list in the Partial Match Data Structure.

3. Valid Task with True Task Result

The Task is valid and the evaluation gives a True Task Result, which means it passed the evaluation. The operator instance creates a new Partial Match and new Tasks in the Partial Match Data Structure as well as removes the Task from the tracking list.

The time when the operator instance finished manipulation in the Partial Match Data Structure is considered as the Task is totally processed. Therefore, Task Processing Latency contains different components according the ending of Task.

1. Invalid Task

Task Processing Latency only contains the Task Queuing Time

2. Valid Task with False Task Result

Task Processing Latency contains the Task Queuing Time plus the Evaluation Cost Time.

3. Valid Task with True Task Result

Task Processing Latency contains the Task Queuing Time, the Evaluation Cost Time, and PMDS Processing Latency.

6.3 Evaluation Results

Following are three subsections. In Section 6.3.1, the Evaluation Results are about the Throughput and Raw Event Processing Latency w.r.t. different Raw Event block size, Task Queue, and Partial Match Data Structure. The evaluation is run on a 8 vCPUs machine and use 160x120 image. Explanations about how these three configuration parameters impact on the Throughput and Raw Event Processing Latency will also be given. Section 6.3.2 will discuss about the architecture scalability as the number of CPU cores increases. Section 6.3.3 is about the performance of architecture w.r.t. different CPU complexity operations.

6.3.1 Performance about Raw Event Block Size, PMDS, and Task Queue

The Figure 6.1 shows the throughput test result on a 8 vCPUs cloud instance. Obviously, Partial Match Data Structure with Priority Task Queue generally obtains a better performance. With Priority Task Queue, the throughput of all Partial Match Data Structures are nearly same at block size 4, approximately 52 events per second. As the block size increasing, all data structures shows improvement, but at different rates. The Graph reaches its optimum area 90 events per second at block size between 256 and 1024, achieved 73% improvement . The Tree without Hash Map has its highest throughput 90 events per second at block size 1024, achieved 73% improvement as well. The Tree with Hash Map reaches its optimum area also around block size 1024, but with a lower throughput, only 85 events per second, which is 63% improvement. After the optimum area, the throughput becomes worse as the block size increasing. At block size

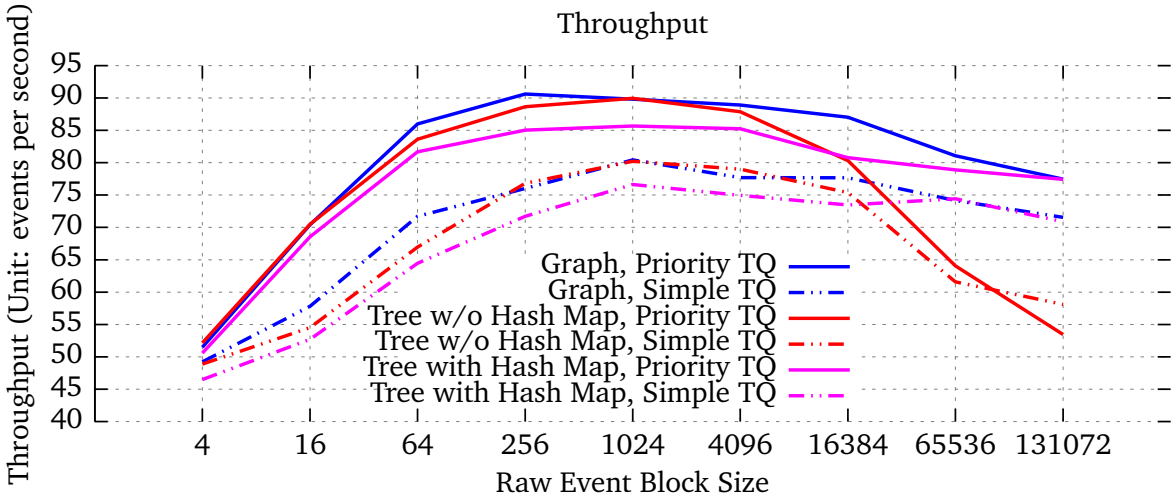


Figure 6.1: Throughput

131072, the throughput of Graph and Tree with Hash Map are both around 77 events per second. The Tree without Hash Map has the worst throughput, only about 54 events per second.

The throughput of three Partial Match Data Structures with Simple Task Queue (i.e. FIFO Queue) have the same trend, except the throughput value is lower. At block size 4, the throughput only achieves 45 to 50 event per second. The highest throughput of Graph and Tree without Hash Map are 80 events per second. The highest value of Tree with Hash Map is about 76 events per second. Then at block size 131072, the throughput decreased to 72, 58, and 72 events per second respectively.

All lines shows there is an optimum Raw Event block size range for each configuration. This optimum point mainly depends on the Raw Event Processing Latency, which will be described in Figure 6.3.

Before looking into Figure 6.3, I would like to the show the legend of boxplot first. Figure 6.2 shows the legend used in boxplot charts. There are six values in each box: 95%, 75%, 50%, 25%, 5%, and mean value. E.g. 95% means 95% results are lower than this value. The X mark is the mean value of all results. Now we continue to look into Figure 6.3.

Figure 6.3 described the Raw Event Processing Latency. At block size 4, all configurations show high Raw Event Processing Latency. With Priority Task Queue, the Raw Event Processing Latency keeps around 100ms. With Simple Task Queue, the Raw Event Processing Latency keeps from 270 to 370ms. As block size increasing, all configurations obtain the optimum area around block size 256 or 1024. With Priority Task Queue, the

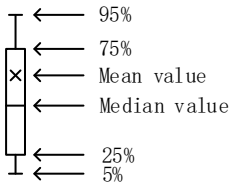
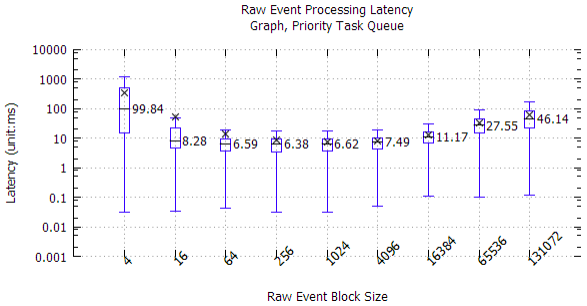


Figure 6.2: Legend of Boxplot Chart

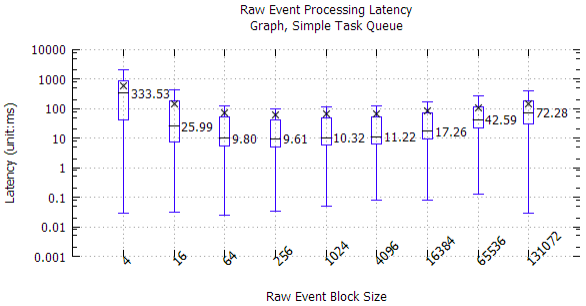
Graph has optimum point 6.38ms at block size 256. The Optimum point of Tree w/o Hash Map is 5.83ms, at both block size 256 and 1024. The lowest Raw Event Processing Latency of Tree with Hash Map is 6.05ms at block size 1025. While with the Simple Task Queue, the Raw Event Processing Latency is a little bit higher than the Priority Task Queue. The Graph has 9.61ms at block size 256. The Tree w/o Hash Map has 8.87ms at block size 1024. The Tree with Hash Map has 8.52ms also at block size 1024. As the Raw Event block size keeps increasing, the Raw Event Processing Latency grows back. At block size 131072, the Graph with Priority Task Queue has Raw Event Processing Latency 46.14ms. With Simple Task Queue, the Raw Event Processing Latency of Graph grows to 72.28ms. With Priority Task Queue and Simple Task Queue, the Tree w/o Hash Map has Raw Event Processing Latency 13.77ms and 24.03 respectively, and the Tree with Hash Map has 11.07ms and 19.18ms respectively.

Figure 6.4 is the Raw Event Queuing Time. Compared with Figure 6.3, at block size 4, the Raw Event Queuing Time almost takes 90% of Raw Event Processing Latency. Even at optimum block size, the Raw Event Queuing Time still makes 60% to 70% of Raw Event Processing Latency. As the block size keeps increasing after the optimum point, the Raw Event Queuing Time also grows, but at a lower speed than Raw Event Processing Latency. At block size 131072, the Raw Event Queuing Time is higher than at the optimum point, but it takes less percentage in the Raw Event Processing Latency.

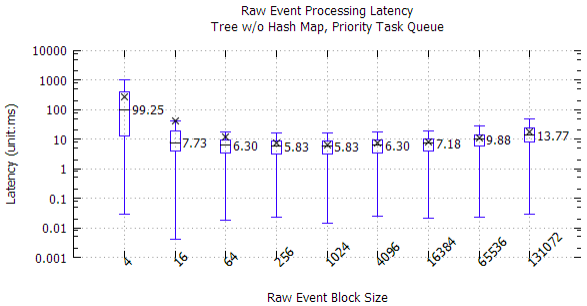
Between Raw Event block size 4 and 1024, the Raw Event Queuing Time plays an important role in the Raw Event Processing Latency. The improvement of Raw Event Queuing Time leads to a better Raw Event Processing Latency, and eventually leads to the growth of throughput. The queuing time decreases as block size increasing is because more Raw Events can be processed in parallel. For example, in block size 4, the 5th to 8th Raw Event have to wait all first four Raw Events finished processing before they can be processed, even though they arrive as early as the 1st Raw Event. While in block size 16, all first sixteen Raw Events can be processed as soon as they arrive. However, after optimum Raw Event block size, the throughput and Raw Event Processing Latency become worse again, while the Raw Event Queuing Time only grows a little, which means the Raw Event Queuing Time doesn't influence so much to the



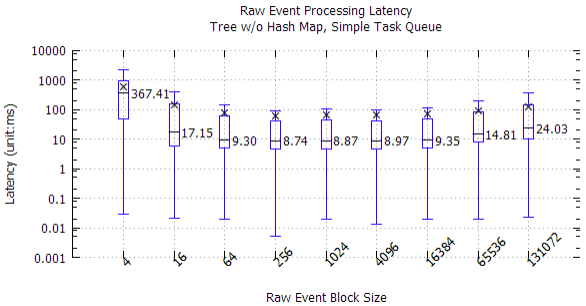
(a) Graph, Priority Task Queue



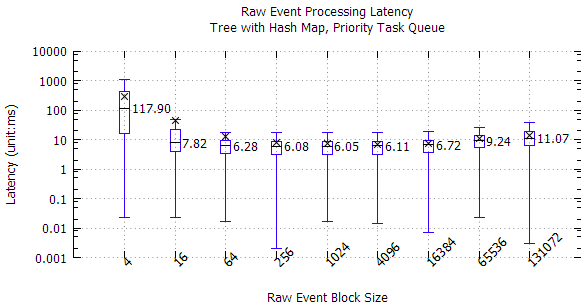
(b) Graph, Simple Task Queue



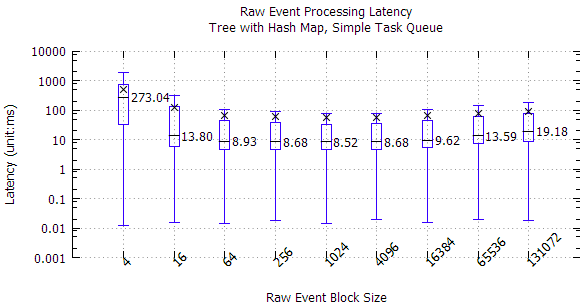
(c) Tree w/o Hash Map, Priority Task Queue



(d) Tree w/o Hash Map, Simple Task Queue

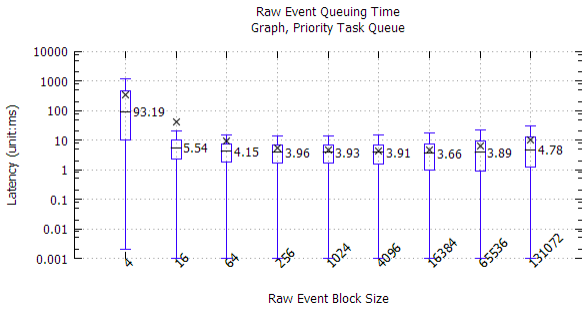


(e) Tree with Hash Map, Priority Task Queue

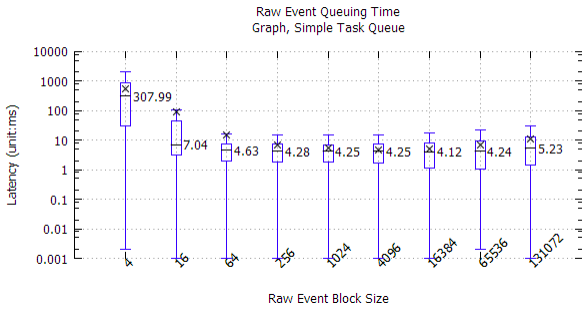


(f) Tree with Hash Map, Simple Task Queue

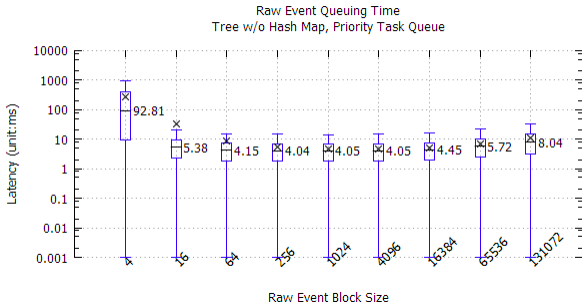
Figure 6.3: Raw Event Processing Latency w.r.t. Raw Event block size



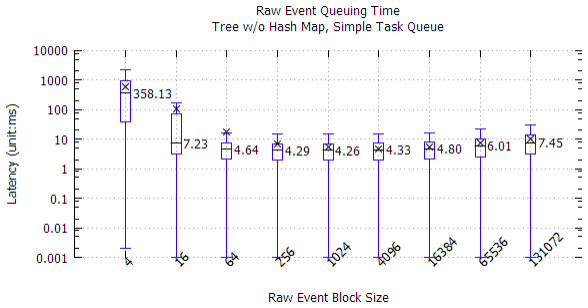
(a) Graph, Priority Task Queue



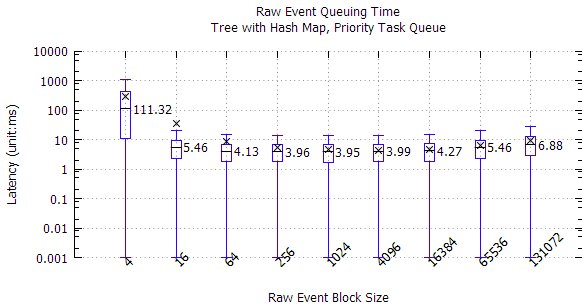
(b) Graph, Simple Task Queue



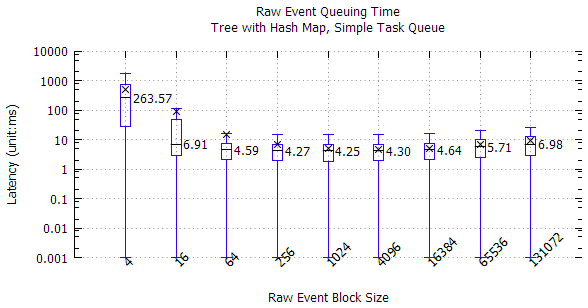
(c) Tree w/o Hash Map, Priority Task Queue



(d) Tree w/o Hash Map, Simple Task Queue



(e) Tree with Hash Map, Priority Task Queue



(f) Tree with Hash Map, Simple Task Queue

Figure 6.4: Raw Event Queuing Time w.r.t. Raw Event Block Size

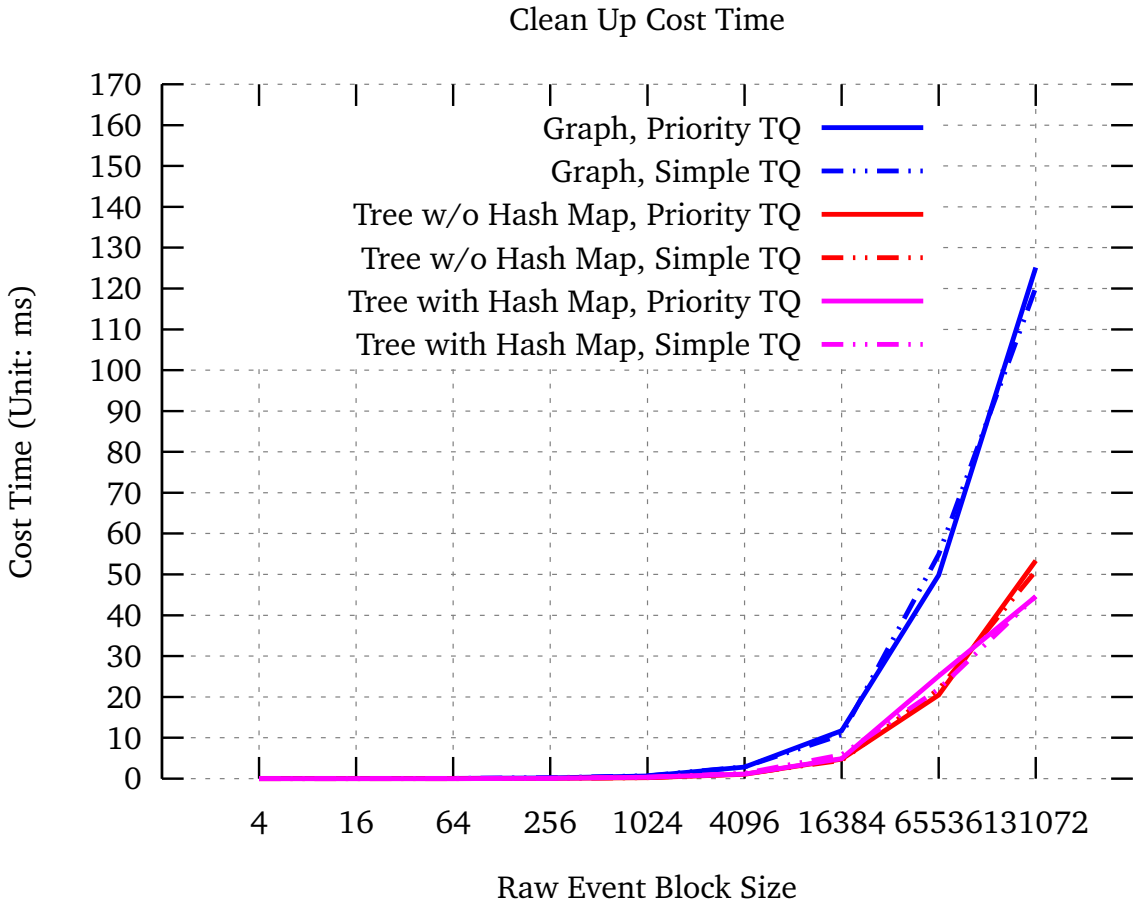


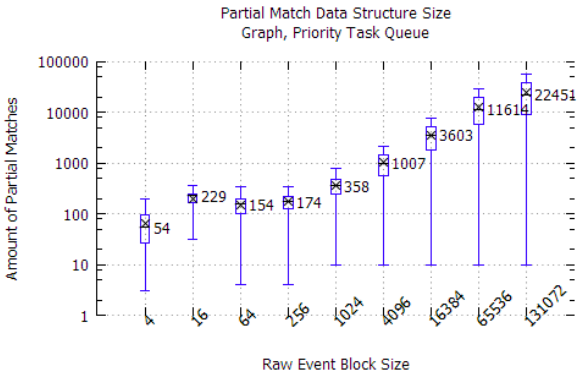
Figure 6.5: Clean Up Cost Time

system. Another point needs to mention is the reason why Raw Event Queuing Time at Raw Event block size 4 is much larger than the value at other block sizes. This is because at block size 4, the best throughput can be achieved is already very low, which is very close to the Event Generation Rate used in latency test. Therefore, some events will be queued up a little bit in the latency test.

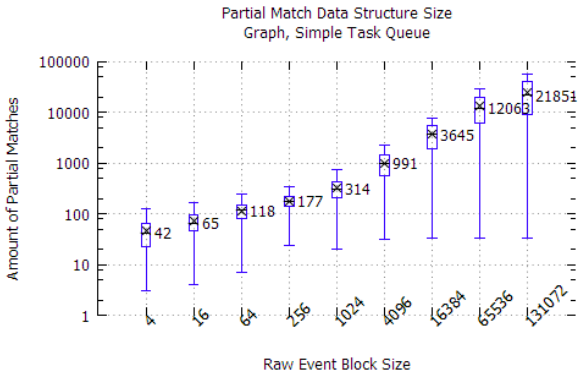
As the Raw Event block size increasing, other factors influence the system more and more, e.g. Partial Match Data Structure size and clean-up cost time. Figure 6.5 and Figure 6.6 show the clean-up cost time and the Partial Match Data Structure size. The median block size is around 30 to 80 Partial Matches in the data structure at block size 4. The amount of Partial Matches in the data structure exponentially increases as the block size increasing. At block size 131072, the median block size reaches more or less 20,000. As the Raw Event block size keeps increasing, the Partial Match Data Structure will become larger and larger. The time to traversal and to clean-up also increase, especially

the clean-up cost time. In Figure 6.5, between block size 4 and 1024, the clean-up cost time almost equals to none. However, from block size 1024, it shows the exponentially increment. Considering the Partial Match Data Structure size and clean-up cost time, the benefit of large Raw Event block size will eventually be neutralize, just as the Figure 6.1 shown.

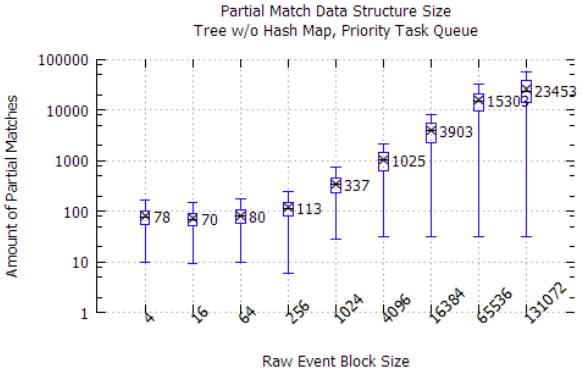
Another feature shown in Figure 6.1 is that the throughput of Tree w/o Hash Map drops more violently than other two data structures. This is because other two data structures use hash map to fast access the Partial Matches during the consumption. Figure 6.7 shows the Final Match Consumption Cost Time. In 6.7a, 6.7b, 6.7e, and 6.7f, the consumption cost time remains same and only costs 0.1ms for each Final Match, while the consumption cost time for Tree w/o Hash Map increases as the Raw Event block size increasing. The consumption cost time for Tree/wo Hash Map grows from 0.1ms to 25ms and 18ms, as shown in 6.7c and 6.7d. This result matches Figure 6.6 because Tree w/o Hash Map doesn't have a mechanism to fast access a Partial Match containing a given Raw Event and a larger Partial Match Data Structure needs more time to traversal.



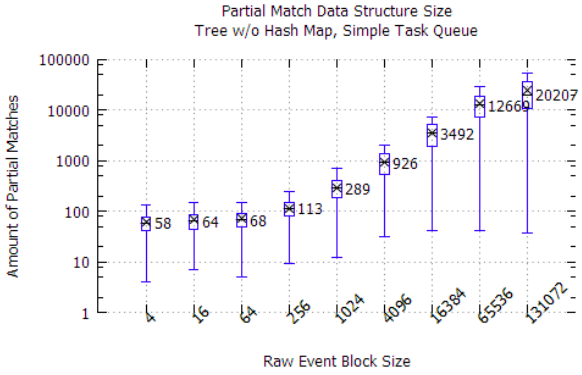
(a) Graph, Priority Task Queue



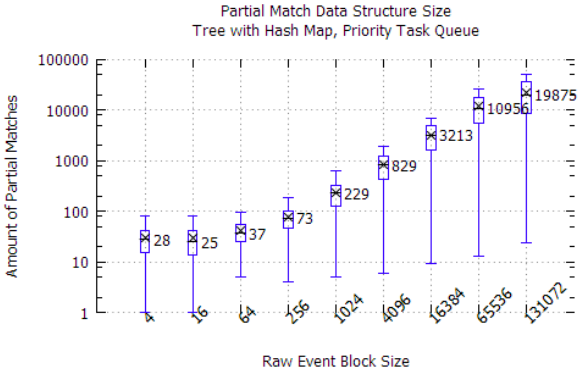
(b) Graph, Simple Task Queue



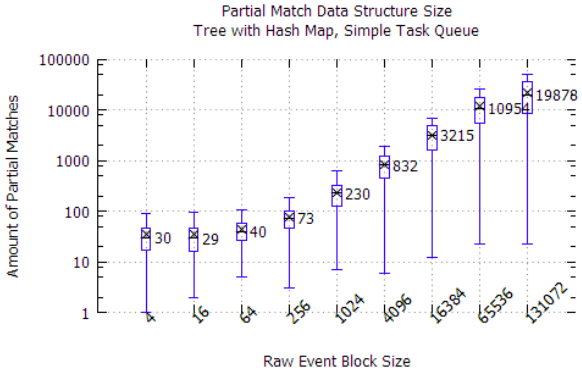
(c) Tree w/o Hash Map, Priority Task Queue



(d) Tree w/o Hash Map, Simple Task Queue

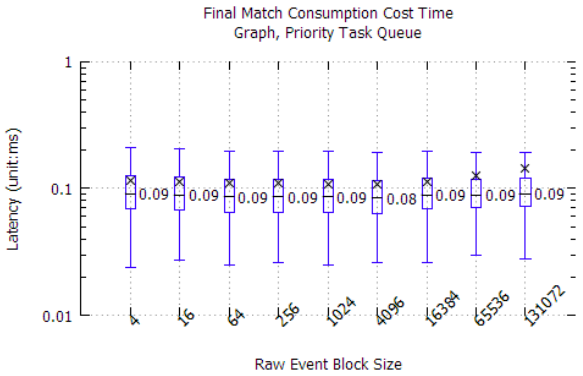


(e) Tree with Hash Map, Priority Task Queue

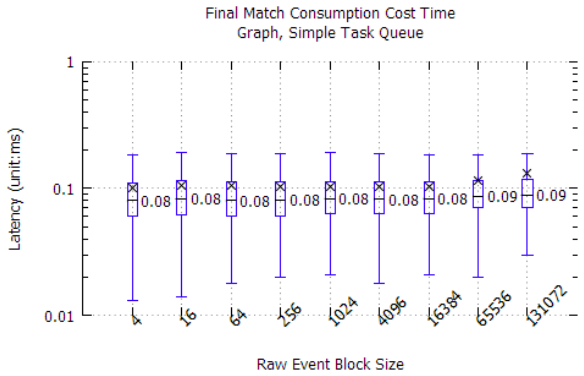


(f) Tree with Hash Map, Simple Task Queue

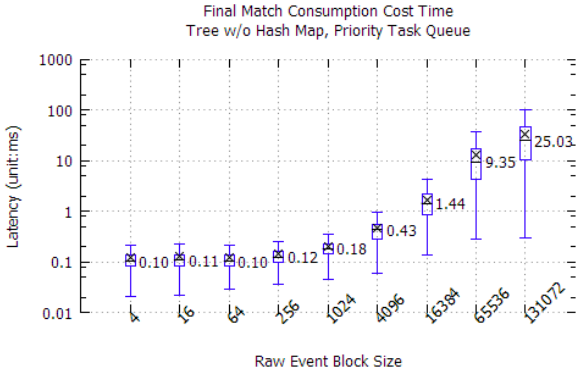
Figure 6.6: Partial Match Data Structure Size w.r.t. Raw Event Block Size



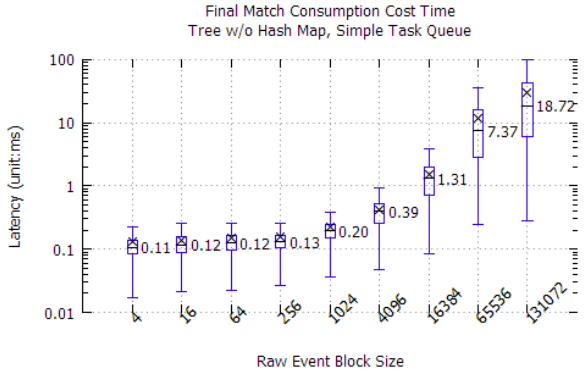
(a) Graph, Priority Task Queue



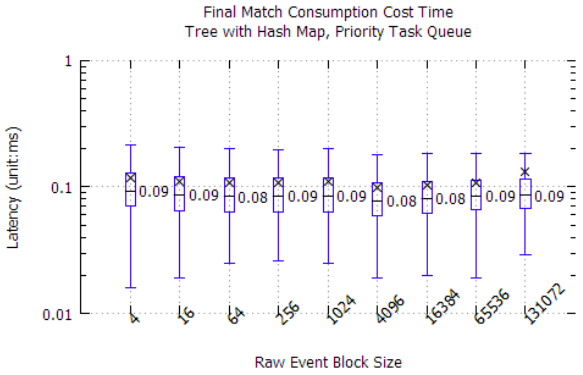
(b) Graph, Simple Task Queue



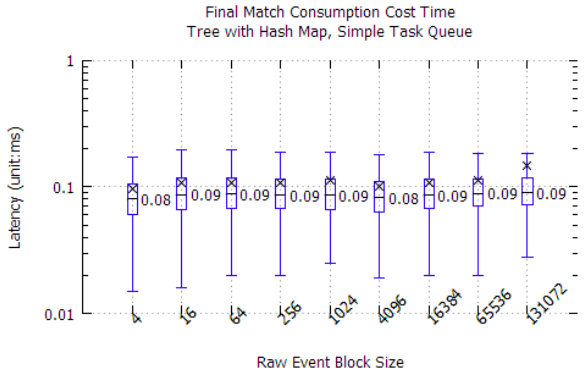
(c) Tree w/o Hash Map, Priority Task Queue



(d) Tree w/o Hash Map, Simple Task Queue

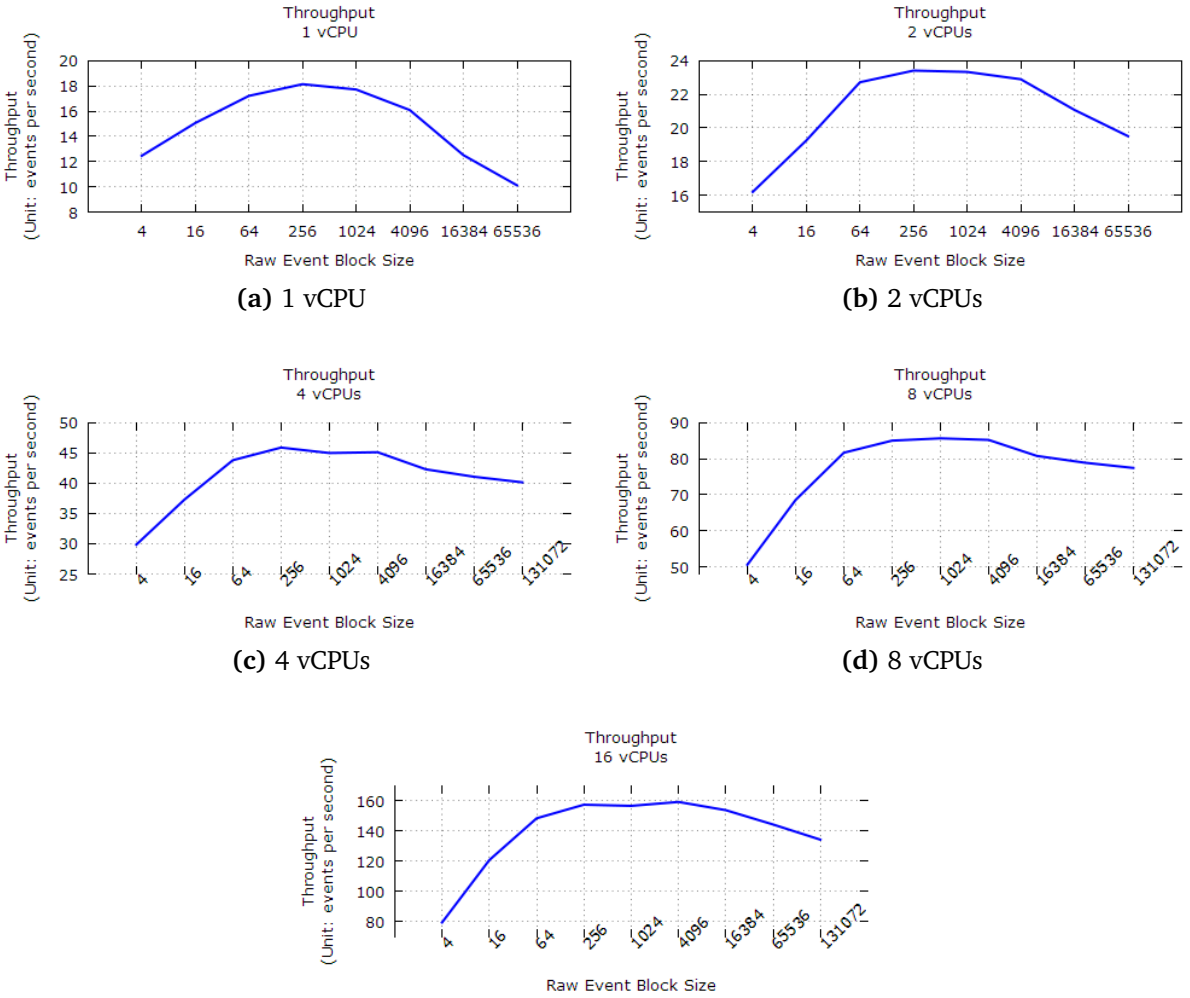


(e) Tree with Hash Map, Priority Task Queue



(f) Tree with Hash Map, Simple Task Queue

Figure 6.7: Final Match Consumption Cost Time w.r.t. Raw Event Block Size



(a) 1 vCPU

(b) 2 vCPUs

(c) 4 vCPUs

(d) 8 vCPUs

(e) 16 vCPUs

Figure 6.8: Throughput w.r.t number of cores

6.3.2 Scalability

Figure 6.8 shows the throughput of Tree with Hash Map, Priority Task Queue against different number of vCPU cores, and Raw Event block size. As discussed in previous subsection, there is an optimum point in Raw Event block size. The optimum throughput is obtained at block size 256, 256, 256, 1024, and 1024 w.r.t. 1, 2, 4, 8, 16 vCPUs. It is obvious that the optimum point is moving from small block size to large block size as the number of CPU cores increasing. This result matches to the mechanism of Raw Event block size since larger block size means higher parallel degree. A machine with more CPU cores should be more suitable for higher parallel degree.

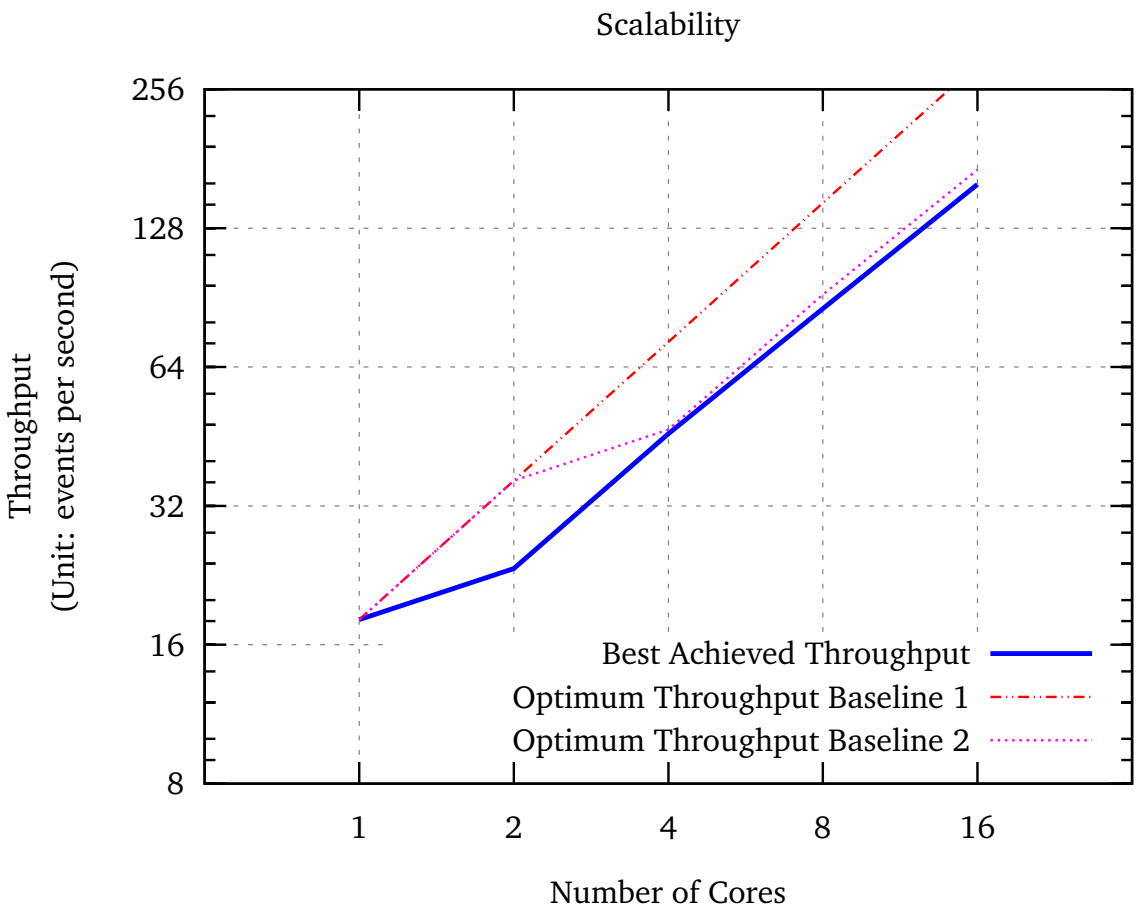


Figure 6.9: Optimum Throughput w.r.t. number of cores

Figure 6.9 is the best throughput can be achieved w.r.t. different number of vCPU cores. (i.e. The throughput at optimum Raw Event Block Size) For 1 vCPU machine, the optimum throughput is around 20 events per second. For 2 vCPUs, the optimum throughput becomes 23 events per second. Then the optimum throughput achieves 46, 85, 160 events per second for 4, 8, 16 vCPUs. In the figure, there are two baselines. Baseline 1 is based on the throughput of 1 vCPU, which means proportional scalability. I.e. the throughput of a multi-core machine should be equal to the product of throughput of 1 vCPU machine and the amount of vCPU cores. Baseline 2 shows the doubled throughput of half vCPU cores, i.e. if the number of vCPU cores are doubled, then the throughput should also be doubled. From the figure, obviously, if the number of vCPU cores is doubled, the throughput is also nearly doubled, except when vCPU cores increase from 1 to 2, the throughput only increases about 30%. There is a huge loss when vCPU cores amount increases from 1 to 2. The reason for the loss is still unclear. It may be caused by frequent context switch among four working threads (For 2 CPUs,

there are 2 operator instances, 1 Raw Event Receiver, and 1 Merger). Since there are always only 1 Raw Event Receiver and 1 Merger, the more CPU cores a machine have, the less influence these two thread will impact to the system.

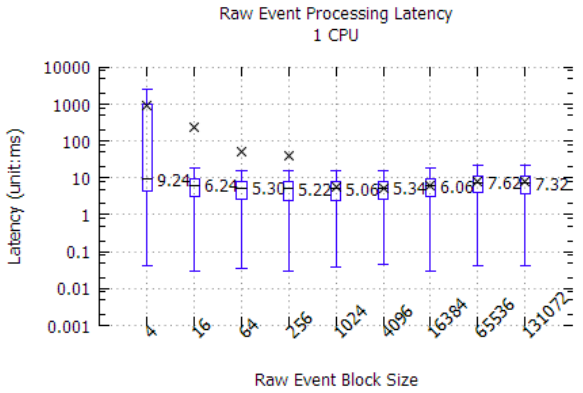
Figure 6.10 shows the Raw Event Processing Latency w.r.t. different number of vCPU cores. The Raw Event Processing Latency at block size 4 is still very high, the range of distribution is from 10ms to 1000ms. This is mainly because the input event rate is so close to its best throughput can be achieved at block size 4 that some Raw Events are inevitably queued up, which leads to very high Raw Event Queuing Time. While the Raw Event Processing Latency maintains 6ms around optimum block size. Compared with Figure 6.3, the Raw Event Processing Latency result remains same, which means the number of cores doesn't influence the Raw Event Processing Latency as long as the machine is not overloaded.

6.3.3 Performance about different complexity of operations

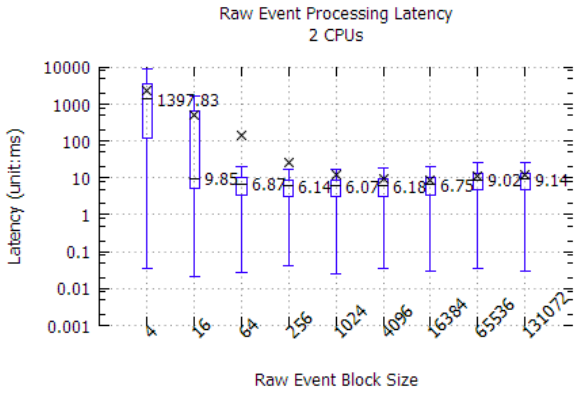
Figure 6.11 shows the average cost time of evaluation for each Task. Although the variation range is quite small, within $\pm$1ms, the cost time increases when the Raw Event Block Size increases. This is because when larger block size is used, more Tasks are created and it is more likely to occur context switch in CPU scheduling. In general, the Task Evaluation Cost Time is stable near a certain value. This value is related to the image size. An image with size 80x60, the average cost time is about 2ms. When the image size grows up to 160x120, the cost time rises to 5.5ms. The cost time becomes 29ms when the image size doubled again. There seems to be a relationship between image size and cost time, but that's not our emphasis. We will focus on the performance of the system w.r.t different Task Evaluation Cost Time.

First of all, Figure 6.12 is the throughput w.r.t. different complexity of operation. For 2ms operation, the optimum throughput can achieve 200 events per second. For 5.5ms operation, the optimum throughput is around 90 events per second. Compared with 2ms operation, 5ms operation is 2.75 times heavier than 2ms operation, the throughput drops 55% off. For 29ms operation, the optimum point is nearly 18 events per second. compared with 5.5ms operation, the operation is 5.27 times heavier, the throughput decreases 80% off. It seems if the complexity of transition is in high value range (e.g. more than 5ms), the throughput is proportional to the complexity of transition, but if the complexity of transition is in low value range (e.g. less than 5ms), the throughput will lose more as the complexity of transition increases.

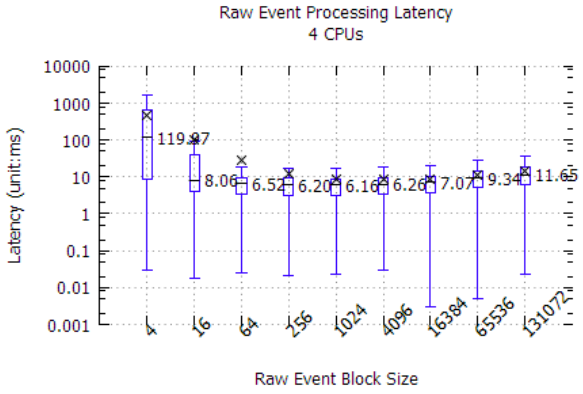
The result is the same as expected, since this architecture is designed for heavy weight operation, which is not suitable for dedicated splitter in Split-Merge-Process architecture. The manipulation to the Partial Match Data Structure is designed not to be influenced



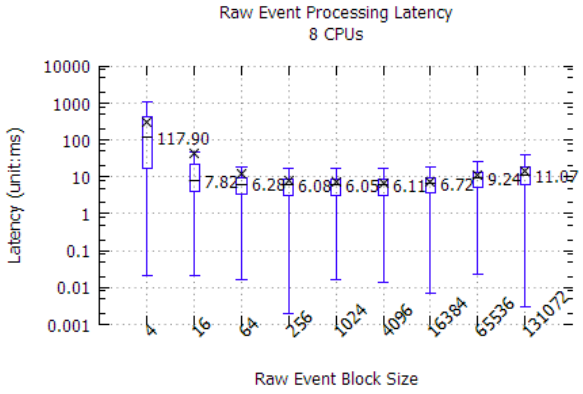
(a) 1 vCPU



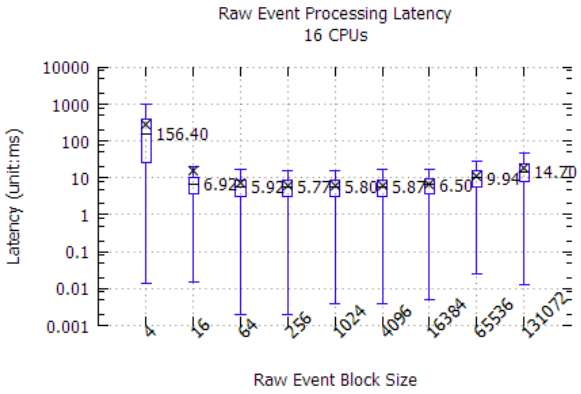
(b) 2 vCPUs



(c) 4 vCPUs

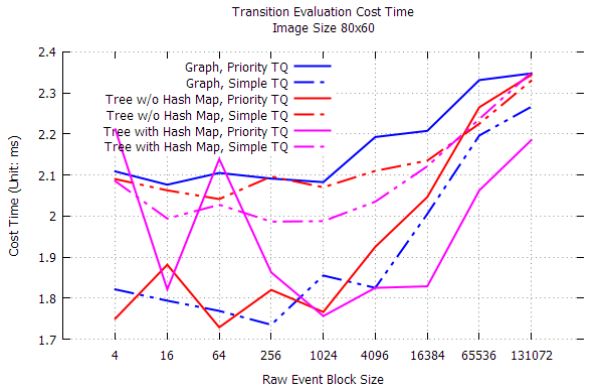


(d) 8 vCPUs

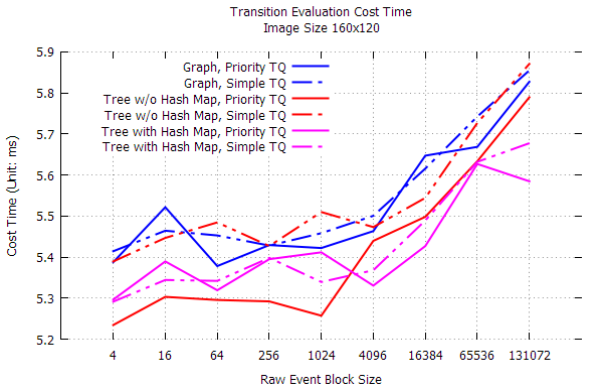


(e) 16 vCPUs

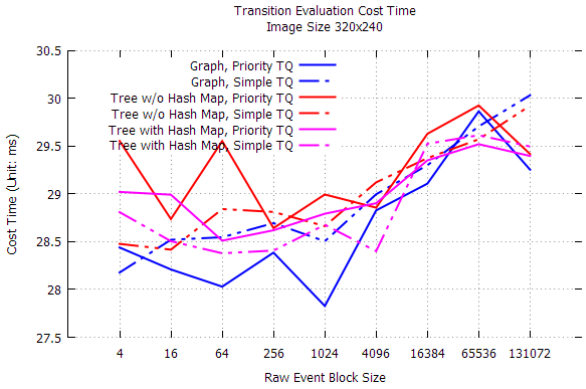
Figure 6.10: Raw Event Processing Latency w.r.t. number of cores



(a) Image Size 80x60



(b) Image Size 160x120

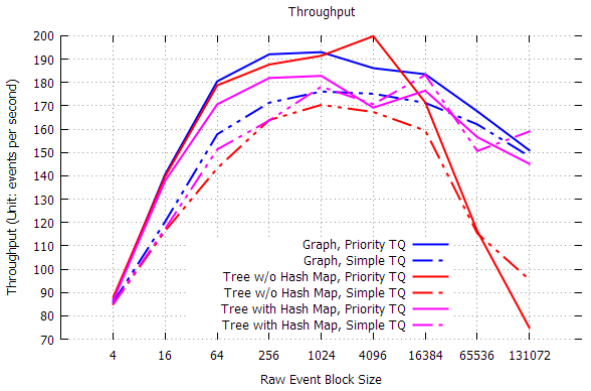


(c) Image Size 320x240

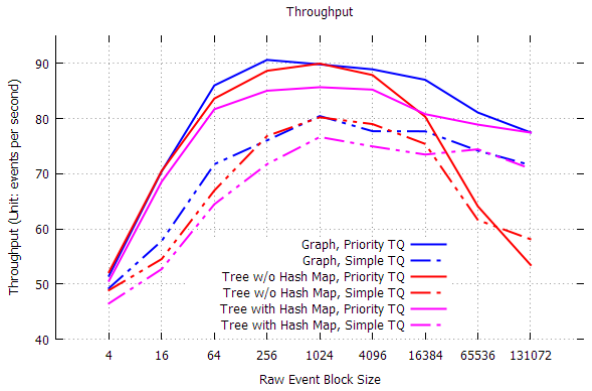
Figure 6.11: Task Evaluation Cost Time

by the complexity of operation, as shown in Figure 6.13. The Final Match Consumption Cost Time of Image Size 320x240 at block size 65526 and 131072 is much less than other two Image Sizes, because the Event Generation Rate for Image Size 320x240 is much less than other two Image Sizes, only 9 events per second, therefore there are not as many Final Matches for Image Size 320x240 as for other two Image sizes. Thus, the Final Match Cost Time is much less than other two Image Sizes. However, the Final Match Cost Time at optimum block size is quite similar.

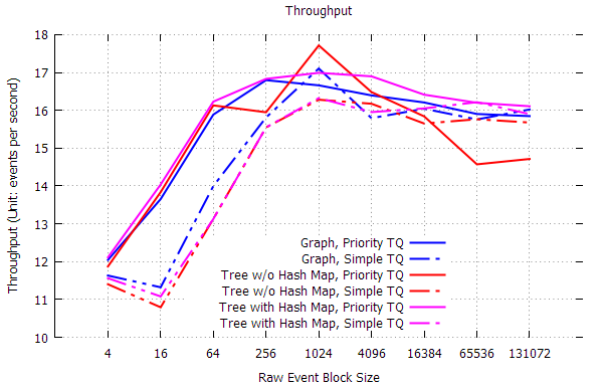
Figure 6.14 also shows the Raw Event Queuing Time is not influenced by complexity of operation. The Raw Event Queuing Time keeps around 4ms to 6ms if the system is not overloaded. The Raw Event Queuing Time at block size 4 is much higher is because the Event Generation Rate is very close to the maximum throughput at block size 4, which means the system is nearly overloaded.



(a) Image Size 80x60 (2ms)



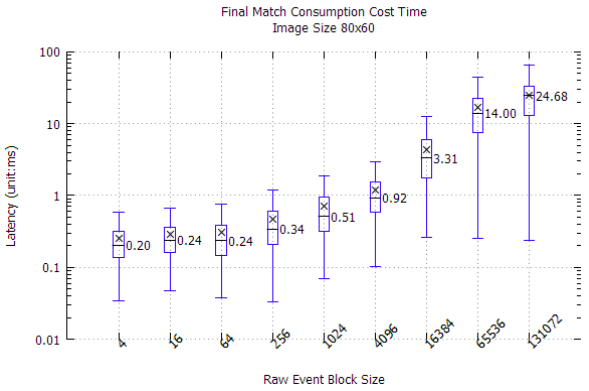
(b) Image Size 160x120 (5.5ms)



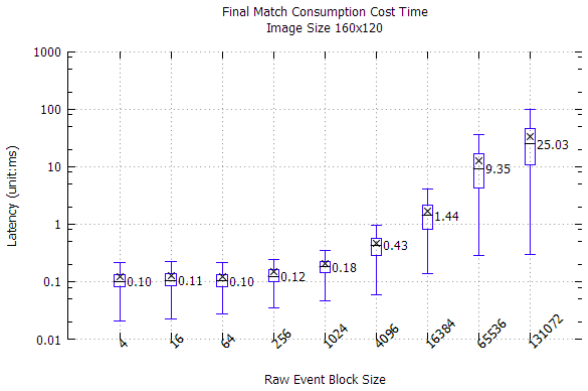
(c) Image Size 320x240 (29ms)

Figure 6.12: Throughput w.r.t. Different Complexity of Operation

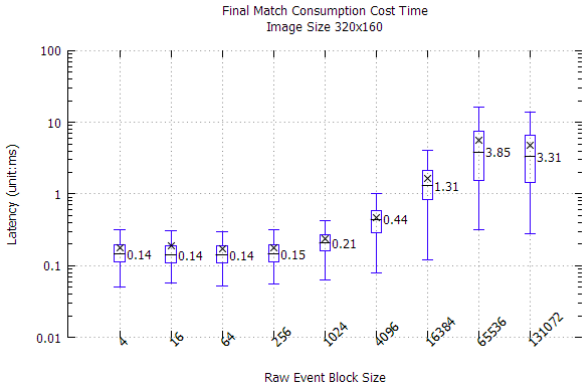
Since variables like Raw Event Queuing Time and Final Match Consumption Cost Time are not influenced by the complexity of operation, the heavier operation to execute, the less these variables impacts to the performance.



(a) Image Size 80x60 (2ms)



(b) Image Size 160x120 (5.5ms)



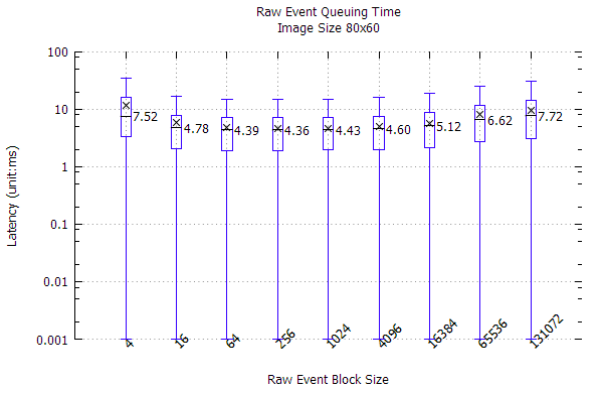
(c) Image Size 320x240 (29ms)

Figure 6.13: Final Match Consumption Cost Time of Tree w/o Hash Map, Priority Task Queue, w.r.t. Different Complexity of Operation

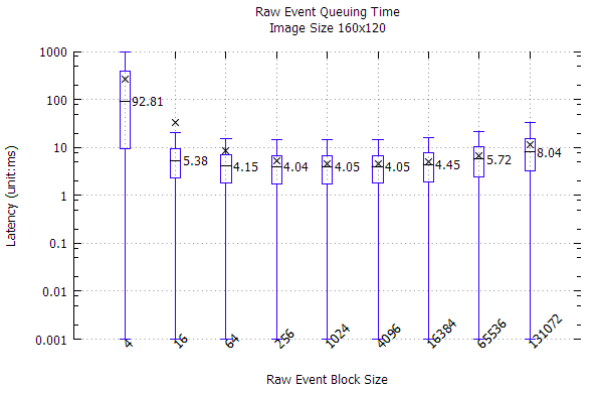
6.3.4 Evaluation Conclusion

In the evaluation, following configuration parameters are measured.

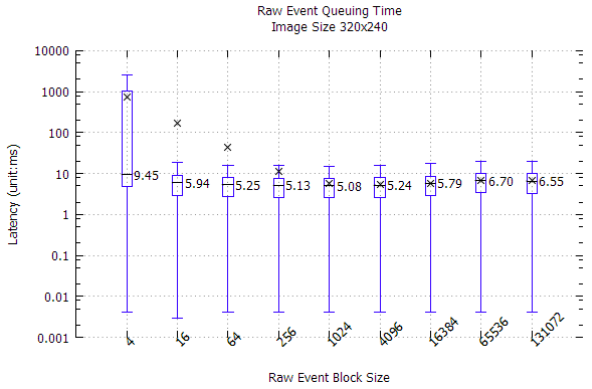
- Partial Match Data Structure
- Raw Event Block Size
- Task Scheduling Strategy (i.e. Task Queue)
- Scalability
- complexity of operation



(a) Image Size 80x60 (2ms)



(b) Image Size 160x120 (5.5ms)



(c) Image Size 320x240 (29ms)

Figure 6.14: Raw Event Queuing Time of Tree w/o Hash Map, Priority Task Queue, w.r.t. Different Complexity of Operation

The evaluation results indicate that there is an optimum Raw Event Block Size. A higher block size will lead to more expensive clean-up cost and more unnecessary Tasks created, which will neutralize the benefits of the large block size.

The results also show that for larger Raw Event Block Size, Partial Match Graph and Partial Match Tree with Hash Map is more suitable than Partial Match Tree without Hash Map.

The results proves as well that a good Task scheduling strategy will improve the performance. However, the Task scheduling strategy is query dependent. It is not easy to derive a strategy that suits every query.

The new architecture scales well if the number of CPU cores is more than 4. The improvement from 1 CPU to 2 CPUs is not so satisfied.

The new architecture also performs well as the complexity of operation increases, as long as the operation is heavy enough (e.g. more than 5ms).

7 Conclusion and Outlook

In this Thesis the limitation of typical Split-Process-Merge architecture has been put forward. According to the analysis, for computational expensive splitting, the single splitter will become the bottleneck of the system. To avoid the "heavy" splitter, a new architecture has been proposed. The new architecture doesn't split the event stream, but creates Tasks by combining every incoming events to all existing Partial Matches. Through this method, the "heavy" splitter is avoided since all Transition Conditions are evaluated only in operator instances. Meanwhile, the parallelism degree is still nearly unbounded, or bounded by a very high value, which will grow as the size of Partial Match Data Structure increasing.

In this new architecture, arbitrary number of operator instance are allowed to be added. However, this architecture is designed for shared-memory machine. There is no reason to add more operator instances than the number of CPU cores. Therefore, to actually achieve unbounded parallelism degree, it is necessary to adapt the architecture into distributed environment.

While, in the distributed environment, the component Centralized Data Structure could be a problem. As the number of operator instances increasing, the manipulation cost and traversing cost of Partial Match Data Structure in Centralized Data Structure will eventually grow beyond the capacity of single machine. Then, distributed Data Structure is needed. For the Task Creation Algorithm, there is no need to modify it. Multiple threads can run concurrently to traverse the data structure. either a single shared iterator or multiple iterators can be used. Of course, it is also possible to modify the traversing algorithm to adapt the case that multiple iterators are used. However, the data structure itself needs some adjustment. For Partial Match Tree, it is quite simple to implement. The whole tree can be chopped into several sub-trees from the Root node and put each sub-tree in a single machine. Then, the operator instances only need to communicate with one of sub-trees and there is no communication needed between sub-trees unless re-balance is needed. For Partial Match Graph, it is more complicated to divide a graph into several sub-graphs. Also, the sub-graphs need to synchronize between each other.

Another interesting direction is how does the system reacts to a unbalanced input event stream. Currently, the Raw Events in input event stream are assumed evenly distributed. What if all events come in as a pulse? Then how should the system react to this case? E.g.

adapting the Raw Event block size dynamically may or may not improve the performance in such situation.

Bibliography

[AE04] A. Adi, O. Etzion. “Amit-the situation manager.” In: *The VLDB Journal—The International Journal on Very Large Data Bases* 13.2 (2004), pp. 177–203 (cit. on p. 20).

[Aga12] A. Agarwal. “High-frequency trading: Evolution and the future.” In: *Capgemini, London, UK* (2012) (cit. on p. 20).

[ASF16] The Apache Software Foundation. *Apache Storm*. 2016. URL: http://storm.apache.org/ (cit. on p. 22).

[BDWT13] C. Balkesen, N. Dindar, M. Wetter, N. Tatbul. “RIP: run-based intra-query parallelism for scalable complex event processing.” In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM. 2013, pp. 3–14 (cit. on pp. 20, 22, 39, 41, 45).

[BEH+10] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, D. Warneke. “Nephele/-PACTs: a programming model and execution framework for web-scale analytical processing.” In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 119–130 (cit. on p. 22).

[CM10] G. Cugola, A. Margara. “TESLA: a formally defined event specification language.” In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM. 2010, pp. 50–61 (cit. on pp. 20, 23, 39, 41, 45).

[CM12a] G. Cugola, A. Margara. “Complex event processing with T-REX.” In: *Journal of Systems and Software* 85.8 (2012), pp. 1709–1728 (cit. on p. 20).

[CM12b] G. Cugola, A. Margara. “Processing flows of information: From data stream to complex event processing.” In: *ACM Computing Surveys (CSUR)* 44.3 (2012), p. 15 (cit. on p. 19).

[CM94] S. Chakravarthy, D. Mishra. “Snoop: An expressive event specification language for active databases.” In: *Data & Knowledge Engineering* 14.1 (1994), pp. 1–26 (cit. on pp. 19, 23).

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, et al. "The Hipac project: Combining active databases and timing constraints." In: *ACM Sigmod Record* 17.1 (1988), pp. 51–70 (cit. on p. 19).

[DGP+07] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. M. White, et al. "Cayuga: A General Purpose Event Monitoring System." In: *CIDR*. Vol. 7. 2007, pp. 412–422 (cit. on p. 32).

[Dun09] J. Dunkel. "On complex event processing for sensor networks." In: *2009 International Symposium on Autonomous Decentralized Systems*. IEEE. 2009, pp. 1–6 (cit. on p. 15).

[GGD91] S. Gatziu, A. Geppert, K. R. Dittrich. "Integrating active concepts into an object-oriented database system." In: *DBPL*. Citeseer. 1991, pp. 399–415 (cit. on p. 19).

[GJP+12] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, P. Valduriez. "Streamcloud: An elastic and scalable data streaming system." In: *IEEE Transactions on Parallel and Distributed Systems* 23.12 (2012), pp. 2351–2365 (cit. on p. 20).

[Goo16] Google. *Machine Types*. 2016. URL: https://cloud.google.com/compute/docs/machine-types#highcpu (cit. on p. 83).

[HLR+13] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, B. Koldehofe. "Mobile fog: A programming model for large-scale applications on the internet of things." In: *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*. ACM. 2013, pp. 15–20 (cit. on p. 16).

[Hyp16] Hyperic. *Sigar*. 2016. URL: https://support.hyperic.com/display/SIGAR/Home (cit. on p. 83).

[IBY+07] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks." In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 3. ACM. 2007, pp. 59–72 (cit. on p. 22).

[Its16] Itseez. *OpenCV*. 2016. URL: http://opencv.org/ (cit. on p. 83).

[Jam12] J. James. "How much data is created every minute." In: *Domo, Retrieved from https://www. domo. com/blog/2012/06/how-much-data-is-created-every-minute/(accessed 12 January 2016)* (2012) (cit. on p. 16).

[KKR10] G. G. Koch, B. Koldehofe, K. Rothermel. "Cordies: expressive event correlation in distributed systems." In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM. 2010, pp. 26–37 (cit. on p. 20).

[KMR+13] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, M. Völz. "Rollback-recovery without checkpoints in distributed event processing systems." In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM. 2013, pp. 27–38 (cit. on p. 21).

[LGA96] B. Lieuwen, N. Gehani, R. Arlein. "The Ode active database: Trigger semantics and implementation." In: *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*. IEEE. 1996, pp. 412–420 (cit. on p. 19).

[Luc02] D. Luckham. *The power of events*. Vol. 204. Addison-Wesley Reading, 2002 (cit. on p. 15).

[Lun06] A. Lundberg. "Leverage complex event processing to improve operational performance." In: *Business Intelligence Journal* 11.1 (2006), p. 55 (cit. on p. 15).

[MKR14] R. Mayer, B. Koldehofe, K. Rothermel. "Meeting predictable buffer limits in the parallel execution of event processing operators." In: *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE. 2014, pp. 402–411 (cit. on p. 22).

[MKR15] R. Mayer, B. Koldehofe, K. Rothermel. "Predictable Low-Latency Event Detection with Parallel Complex Event Processing." In: *IEEE Internet of Things Journal* 2.4 (2015), pp. 274–286 (cit. on p. 22).

[MMTR16] R. Mayer, C. Mayer, M. A. Tariq, K. Rothermel. "GraphCEP: real-time data analytics using parallel complex event and graph processing." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM. 2016, pp. 309–316 (cit. on p. 22).

[OKRR13] B. Ottenwälder, B. Koldehofe, K. Rothermel, U. Ramachandran. "MigCEP: operator migration for mobility driven distributed complex event processing." In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM. 2013, pp. 183–194 (cit. on p. 22).

[Ora16] Oracle. *ConcurrentSkipListMap Java API*. 2016. URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html (cit. on p. 68).

[WDR06] E. Wu, Y. Diao, S. Rizvi. "High-performance complex event processing over streams." In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pp. 407–418 (cit. on p. 20).

[WSK08] W. Wang, J. Sung, D. Kim. "Complex event processing in epc sensor network middleware for both rfid and wsn." In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE. 2008, pp. 165–169 (cit. on p. 15).

[YCL11] W. Yao, C.-H. Chu, Z. Li. “Leveraging complex event processing for smart hospitals using RFID.” In: *Journal of Network and Computer Applications* 34.3 (2011), pp. 799–810 (cit. on p. 15).

All links were last followed on July 18, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature