Scaling CEP

Using Distributed Stream Computing to Scale Complex Event Processing

Tom van Duist
tomvanduist@gmail.com

August 15, 2016, 62 pages

Supervisor: Prof. dr. Jurgen J. Vinju
Host organisation: Pegasystems, Inc., http://www.pega.com/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract 4

I Project Conception 5

1 Introduction 6
- 1.1 Problem Statement 6
  - 1.1.1 Research Goals 7
- 1.2 Document Outline 7

2 Background 8
- 2.1 Event Stream Processing 8
  - 2.1.1 Event Stream 8
  - 2.1.2 Event definition 8
  - 2.1.3 Complex event 9
- 2.2 Event Stream Processing Language Model 9
  - 2.2.1 Composition-Operator-Based Query Languages 10
  - 2.2.2 Data Stream Query Languages 12
  - 2.2.3 Hybrid 12
  - 2.2.4 Conclusion 12
- 2.3 Event Stream Processing Systems 13
  - 2.3.1 Active Databases 13
  - 2.3.2 Data Stream Management Systems 13
  - 2.3.3 Complex Event Processing 13
  - 2.3.4 Distributed Stream Computing Platforms 14
- 2.4 Conclusion 14

II Project Setup 15

3 Research Method 16
- 3.1 Experimental Setup 16
  - 3.1.1 Pega Decisioning 16
  - 3.1.2 Apache Storm 17
  - 3.1.3 Events 18
  - 3.1.4 Hardware and Architecture 18
  - 3.1.5 Benchmarking 19
  - 3.1.6 Profiling 19
- 3.2 Amdahl's Law 20
- 3.3 Running Example 21
  - 3.3.1 Trending Scenario 21
- 3.4 Method 22
- 3.5 Threats to Validity 22

4 Baseline 24

4.1 Motivation . . . 24
4.2 Method . . . 24
4.3 Result Analysis . . . 25
4.4 Threats to Validity . . . 26
4.5 Benchmark Noise . . . 26
4.6 Conclusion . . . 27

III Optimization Experiments **28**

5 Partitioned Parallelism **29**
5.1 Limitations . . . 29
5.2 Use Case . . . 30
5.3 Single Machine Parallelism . . . 30
5.3.1 Hypothesis . . . 30
5.3.2 Method . . . 31
5.3.3 Result Analysis . . . 31
5.3.4 Threats to Validity . . . 32
5.3.5 Conclusion . . . 32
5.4 Multi Machine Parallelism . . . 32
5.4.1 Hypothesis . . . 32
5.4.2 Method . . . 33
5.4.3 Result Analysis . . . 33
5.4.4 Threats to Validity . . . 35
5.4.5 Conclusion . . . 35

6 Shift Partition **36**
6.1 Use Case . . . 36
6.2 Problem . . . 36
6.3 Solution . . . 37
6.4 Hypothesis . . . 38
6.5 Method . . . 38
6.6 Result Analysis . . . 39
6.7 Threats to Validity . . . 39
6.8 Conclusion . . . 40
6.9 Related Work . . . 40

7 Pipelined Parallelism **41**
7.1 Problem . . . 41
7.2 Solution . . . 42
7.3 Hypothesis . . . 42
7.4 Method . . . 42
7.5 Result Analysis . . . 43
7.6 Threats to Validity . . . 43
7.7 Conclusion . . . 44

8 Flatten Split Join **45**
8.1 Limitations . . . 45
8.2 Hypothesis . . . 46
8.3 Method . . . 47
8.4 Result Analysis . . . 48
8.5 Threats to Validity . . . 48
8.6 Conclusion . . . 48
8.7 Related Work . . . 48

9 Pull Operator Up **49**

9.1 Hypothesis . . . 49
9.2 Method . . . 50
9.3 Result Analysis . . . 50
9.4 Threats to Validity . . . 51
9.5 Conclusion . . . 51
9.6 Related Work . . . 51

IV Evaluation **52**

10 Putting It Together **53**
10.1 Comparison . . . 53
10.2 Conclusion . . . 54

11 Evaluation **55**
11.1 Research Goals . . . 55
11.2 Future Work . . . 56
11.2.1 Additional Optimizations . . . 56
11.2.2 Increase Cluster Size . . . 57
11.2.3 Automate Parameter Optimization . . . 57
11.3 Recommendations . . . 57

12 Conclusion **59**

Bibliography **60**

Abstract

Complex event processing (CEP) is the act of extracting high level knowledge from unbounded and continuous streams of low level events. For example, to extract trends from social media or financial streams to detect changes in real time.

A wide range of commercial and academic CEP and data stream management systems exist, but there is a clear lack of horizontally scalable systems or distributed frameworks with native CEP capabilities. Because of the statefulness of a CEP engine, implementing it in a distributed stream computing platform (DSCP) introduces errors in the event strategy results. In this project we identify the challenges that arise when performing CEP in a distributed environment – it is important to overcome these challenges in order to truly scale CEP processing.

In this effort we implement a CEP engine developed by the host organization Pegasystems in the Apache Storm DSCP. We show how the challenges can be overcome and compare the performance increase achieved by different types of horizontal scalability through parallel execution. We also present other optimizations that are enabled through the introduction of a distributed architecture. Using these optimizations, and parallel execution on a two node cluster, we achieve a performance that is several orders greater than the regular sequential execution.

Part I

Project Conception

Chapter 1

Introduction

An increasing number of IT systems generate a continuous stream of data, such as RFID sensor networks, financial transactions, stock trades and so forth. Changes in these systems can be viewed as events that are communicated through the data stream. For many of these systems it is crucial to recognize (complex) patterns within these streams of data. Such as inventory and supply chain management [WL05][GR06], (digital) surveillance [Hin03], health-care and record keeping [GR06][HH05] and financial services [DGH+06].

Different classes of systems exist that deal exclusively with processing unbounded streams of data and extracting higher level abstractions from this data, these can be seen as Event Stream Processing (ESP) engines [CM12]. See Chapter 2 for a detailed description of these classes and their corresponding commercial and academic systems. Looking at the current ESP systems there is a clear lack of horizontally scalable solutions that provide full fledged Complex Event Processing (CEP) capabilities. Mature, commercial, systems are either general purpose Distributed Stream Computing Platforms (DSCP) such as Apache Storm [Foub] and Apache Spark [Foua] or centralized CEP systems such as Esper [Teca].

CEP systems that are designed to run on a single, centralized, system can be implemented distributively by running it on multiple nodes in a distributed computing network. However, this does not provide true horizontal scaling when implemented naively as described above. Because CEP systems are stateful, each node has to be aware of the existence of every single event that is being processed by the system. Unless knowledge about the queries running on each node is pulled up to a higher level to distribute the events more intelligently.

In this thesis we present the results of our research and experimental efforts towards enabling horizontal scaling for a stateful CEP engine using a distributed computer cluster. Other optimizations to maximize the event processing efficiency are explored as well.

1.1 Problem Statement

The problem that we study concerns the correct implementation of event strategies by a sequential CEP engine deployed in a distributed environment to achieve horizontal scaling. A sequential CEP engine is inherently not aware of its own parallelization when deployed distributively, which can invalidate the event strategy and result in unexpected results.

Problems occur because CEP engines are stateful. Correct results are only ensured when the engine has knowledge of (i.e. receives) the events that their event strategy depends upon. This is trivially true when a single instance of the engine receives all events within the event stream. Conversely this is not the case when multiple engine are spawned in parallel.

We explore both the enabling of horizontal scalability and other optimizations, possibly enabled by the introduction of a distributed architecture.

1.1.1 Research Goals

The host organization, Pegasystems, is interested in significantly increasing the throughput and size of queries that their CEP engine can handle. Preferably through the introduction of a distributed architecture in order to flexibly scale up. Together with the problem statement the following research goal can be derived:

Increase the processing performance of the Pegasystems CEP engine by utilizing a distributed architecture.

This is subdivided into the following subgoals:

- Achieve higher throughput by utilizing horizontal scalability for the Pegasystems CEP engine.
  - Under what conditions and scenarios is horizontal scalability possible for a stateful CEP engine?
  - What benefits can be expected when horizontally scaling a CEP engine?
- Increase the Pegasystems CEP engine capabilities by rewriting the topology and/or event strategy.
  - What are the biggest bottlenecks introduced by the distributed topology or the CEP event strategy?
  - How can these bottlenecks be optimized?

Note that the research goals scope the project explicitly. We are looking for optimizations that are enabled by the capabilities that become available by implementing a CEP engine in a distributed environment. This means optimizations on a high level by enabling horizontal scalability, distributing computations and restructuring the topology or event strategy. These are optimizations that can potentially be generalized to other distributed environments and CEP engines with similar properties as those used in this research. For example, conditions that inhibit partitioning of the event stream can hold for other systems as well. As do the solutions.

1.2 Document Outline

This thesis is structured in four parts. The first part introduces the problem statement and the ESP domain, touching upon the different classes of query language and processing systems developed by commercial and research organizations.

The second part presents our research method and experimental setup. It also establishes a baseline for the architecture and CEP engine, this is used as a reference and guides our research towards specific bottlenecks.

The third part contains the experiments that we performed to evaluate the optimizations we discovered. Each chapter of this part is roughly structured the same: introducing the experiment, outlining the problem and proposed solution where applicable, stating the hypothesis and research method and evaluating the results and threats to the validity of the results. Where applicable, related work is described in closing.

The fourth part of this thesis concludes our research findings by combining our discoveries, evaluating the results and recommendations to the host organization and closing with the conclusion, where we revisit the research goals and questions.

Chapter 2

Background

The act of processing continuous and unbounded streams of data has a large application area, from RFID-based inventory systems and network intrusion detection to financial and environmental monitoring [CM12]. Because the number of domains that use ESP is so extensive, different research communities focused on problems within the domain before this was broadly recognized as a domain in its own right. Recognition of which was mainly spurred in 2002 by David Luckham's book *The Power of Events* [Luc02].

The following sections will highlight these differences by first defining event processing and explaining the nature of events, then describe the query and pattern language models and finally the different classes of systems that implement these languages.

2.1 Event Stream Processing

As a result of the dispersed research effort there is no definitive terminology [EB09], there are different definitions for processing streams of information. Such as Information Flow Processing [CM12], Data Stream Processing [CM12], Continuous Dataflow Processing [CCD+03] or Event Stream Processing [BGHJ09]. An ongoing effort is being made by the Event Processing Technical Society (EPTC) to standardize the glossary that is being used [LS11]. We will use the terms defined by the EPTC and thus will use the term event stream processing (ESP):

> **Event Stream Processing**: Computing on inputs that are event streams.

This means any systems which performs any sort of computations on (unbounded) event streams. Event Stream Processing is the overarching definition that encompasses narrower defined definitions of systems that will be described further on such as Data Stream Management Systems (DSMS) and Complex Event Processing (CEP).

2.1.1 Event Stream

Event streams are "*linearly ordered sequences of events*" [LS11]. Event streams can either be homogeneous (contain events of the same type and data fields) or heterogeneous (contain different types of events) and are usually unbounded. This means that the event stream is conceptionally infinite and queries that are performed on the stream run indefinitely, continuously altering the stream or returning results when conditions satisfy.

As a result, dealing with event streams requires a different approach than for example a database, because you cannot query the stream for the non-occurrence of an event without defining an interval, otherwise the query will run indefinitely without satisfying.

2.1.2 Event definition

Events are defined and modeled differently in different systems. As Chandy and Schulte outline in their book, *Event processing: designing IT systems for agile companies [CS09] p.111.*, there are three

major schools of thought:

State-change view An event object is a report of a state change. The object being reported on can be anything in the physical, or digital, world and the event reports a change of state of this object.

Happening view The event is "*anything that happens, or is contemplated as happening*" and the event object is "*an object that represents, encodes, or records an event, generally for the purpose of computer processing*" [LS11]. This is the definition by David Luckham from his book *The Power of Events* [Luc02]. An event object signifies, is a record of, an activity that happened.

Detectable-condition view An event is a "*detectable condition that can trigger a notification*". Where a notification is "*an event-triggered signal sent to a runtime-defined recipient*". This definition can be applied by software engineers and reactive programming to distinguishes between a conventional procedure call to a server and a more dynamic procedure call that conveys an event signal.

Any particular definition of an event has a couple of consequences for the modeling of the event object. For example, if an event is instantaneous, the *state-change view* says an event happened at a particular moment in time. When an event has a specific duration, the *happening view* says an event happened during a specific time interval.

Practically speaking, events have at least a time stamp, or position, which determines the temporal relationship in the event stream. This can either be the occurrence time or detection time. Events can also model the duration as per the *happening view*. Next to the temporal information an event also carries data, this can be arranged in several ways; the type of system generally determines the way the event is modeled. Data Stream Management Systems (DSMS), see Section 2.3.2, generally represent events as a tuple of data fields, apart from the user defined fields the temporal data is only known to the internal system. Complex Event Processing (CEP) systems represent events as first class typed objects where the temporal data is embedded in the object and thus accessible for user defined rules, see Section 2.3.3.

The system that we evaluate in this thesis exhibits the *happening* view.

2.1.3 Complex event

Complex events – sometimes called composite or aggregate events but we will again stick with the definition of complex event which is preferred by the EPTC [LS11] – are defined as follows:

> **Complex Event**: An event that summarizes, represents, or denotes a set of other events.

Events can enter the event stream in two ways: *i.* from event sources as simple events, and *ii.* from Event Processing Agents (EPA) as complex events. Detecting the occurrence of simple events such as a single temperature reading or a stock price is trivial and of limited value. Complex events on the other hand form a higher level abstraction and are detected from patterns of simple (and possibly other complex) events. Examples are the average temperature during a day or a rising price of a single stock during a specific time period. Because a complex event is derived from a user defined pattern of simple events, they are inherently more valuable, and more complex.

The process of detecting complex events from patterns in (simple) event streams is called Complex Event Processing (CEP). CEP systems either immediately react on the detection of a complex event or broadcast the occurrence of a complex event, the latter allows the complex event to be used in different pattern detection rules. This opens up the possibility of chaining pattern rules and create event hierarchies.

2.2 Event Stream Processing Language Model

Just like there is a multitude of terminology used within the ESP domain, there are different language models developed to create event stream queries or define complex event detection rules [EB09].

Virtually every DSMS and CEP system defines a different query language to detect complex events, with a different combination of language constructs, as can be seen in Figure 2.1 – composed by Cugola and Margara in [CM12].

Distinctive clusters of language constructs can be observed at specific groups of systems within Figure 2.1. Three common categories of language models within ESP can be identified [EB09][EBB+11]: *i.* composition-operator-based; *ii.* data stream query languages; *iii.* hybrid languages.

The composition-operator-based and data stream query language models are the most commonly used by academic systems. Most commercial systems use a hybrid approach. These will be explored in more detail in the following sections.

Running example

To demonstrate some of the strengths and weaknesses of the different language models a running example will be used, inspired by [CM10]. For each language model a rule expressing the following pattern will be presented:

> *Fire occurs when a temperature higher than 45 degrees is detected and it did not rain in the last hour in the same area. The fire notification has to embed the measured temperature.*

This example is chosen because it expresses different language constructs: *i. selection:* the measured temperature must be selected; *ii. parameterization:* only events from the same area must be considered; *iii. windowing:* pattern must match within a 1 hour window; *iv. negation:* the non occurrence of an event. These constructs will highlight the constructive strengths and weaknesses of each language model.

2.2.1 Composition-Operator-Based Query Languages

Composition-operator-based query languages define complex event queries by composing simple event queries with logical operators such as conjunction, disjunction, sequence and negation. They originate from active databases [EB09] (the first four systems in Figure 2.1) such as ODE [GJ91] and SnoopIB [AC06].

Most modern CEP systems that exclusively use a composition-operator-based language model originate from the academic domain. Some examples are: Cayuga [DGP+07], Raced [CM09], SASE+ [WDR06][DIG07] and Tesla [CM10] (see also Section 2.3.3).

```
1 PATTERN   SEQ(!(RAIN r), TEMPERATURE t)
2 WHERE     [area] ∧ t.val > 45
3 WITHIN    1 hour
4 RETURN    t.val AS FIRE(val)
```

Listing 2.1: Running example in SASE+

		Single-item			Logic						Windows						Flow Management											
Name	Type	Selection	Projection	Renaming	Conjunction	Disjunction	Repetition	Negation	Sequence	Iteration	Fixed	Landmark	Sliding	Pane	Tumble	User Defined	Join	Union	Except	Intersect	Remove Dup	Duplicate	Group By	Order By	Parameterization	Flow Creation	Detection Aggr	Production Aggr
HiPac	Det	X				X			X																X			X
Ode	Det	X			X	X	X	X	X	X															X			X
Samos	Det	X			X	X	X	X	X	X	X														X			X
Snoop	Det	X			X	X	X		X	X															X			X
TelegrahCQ	Decl	X	X	X							X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X
NiagaraCQ	Decl	X	X	X													X						X	X	X		X	X
OpenCQ	Decl	X	X	X	X	X		X									X	X	X	X	X	X	X	X	X		X	X
Tribeca	Imp	X	X										X		X			X				X	X		X		X	X
CQL/Stream	Decl	X	X	X									X	X	X		X	X	X	X	X	X	X	X	X	X	X	X
Aurora/Borealis	Imp	X	X										X		X		X	X				X	X		X		X	X
Gigascope	Decl	X															X	X					X		X		X	X
Stream Mill	Decl	X	X	X							X	X	X	X	X	X	X	X	X	X			X	X	X	X	X	X
Traditional Pub-Sub	Det	X																										
Rapide	Det	X			X	X	X	X	X	X	X		X				X	X							X		X	
GEM	Det	X			X	X		X	X		X	X	X												X			
Padres	Det	X			X	X			X	X															X			
DistCED	Det	X			X	X		X	X	X			X												X			
CEDR	Det	X	X	X	X	X		X	X								X	X	X	X			X		X	X		X
Cayuga	Det	X	X	X	X	X			X	X							X								X		X	X
NextCEP	Det	X	X	X	X	X			X	X			X														X	X
PB-CED	Det	X			X	X		X	X																X			
Raced	Det	X			X	X	X	X			X	X	X												X			
Amit	Det	X			X	X	X	X	X	X			X												X		X	X
Sase	Det	X			X	X		X	X				X												X			
Sase+	Det	X			X	X		X	X	X			X												X		X	X
Peex	Det	X			X	X		X	X																X			
TESLA/T-Rex	Det	X						X	X		X		X													X	X	X
Aleri SP	Imp+Det	X	X	X	X	X		X	X				X	X	X		X	X				X	X	X	X	X	X	X
Coral8 CEP	Decl+Imp+Det	X	X	X	X	X		X	X		X	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X
StreamBase	Decl+Det+Imp	X	X	X	X	X		X	X		X		X	X	X	X	X	X	X	X			X	X	X	X	X	X
Oracle CEP	Decl+Det+Imp	X	X	X	X		X		X	X			X				X	X	X	X	X	X	X	X	X	X	X	X
Esper	Decl+Det	X	X	X	X	X	X	X	X	X			X	X	X	X	X	X					X	X	X	X	X	X
Tibco BE	Det	X			X	X	X	X	X	X	X		X				X	X							X		X	
IBM System S	Decl+Imp	X	X	X							X	X	X	X	X	X	X	X					X	X	X	X	X	X

Figure 2.1: ESP Language Model by Cugola et. al. [CM12]

2.2.2 Data Stream Query Languages

Data stream query languages are derived from the Relational Database Management System (RDBMS) query language SQL. It was first used for Relational Data Stream Management Systems (RDSMS) and is based on the following concept: data streams (which are unbounded) are converted into relations on which regular SQL is performed. Note however that this transformation is mostly conceptual and only happens at the language implementation level [EBB+11].

With the addition of windowing operators it becomes possible to convert the infinite, unbounded, stream of events into a finite, bounded, relation [SMMP09]. This allows the execution of bounded operators such as aggregations to compute averages.

Examples of data stream query languages are STREAM's Continuous Query Language (CQL) [ABW03], not to be confused with the Cassandra Query Language [Fouc]. Esper's Event Processing Language (EPL) [Tecb], and more recently Apache Spark's SparkSQL [AXL+15]. The advantage of a language based on SQL is that it is widely used and engineers are likely to be familiar with it. The downside is that some constructs which are very natural to event stream queries, such as negation and aggregation, are not naturally expressed in SQL, see Listing 2.2.

```
1 SELECT      T.val
2 FROM        Temperature [Now] as T
3 WHERE       T.val > 45
4             AND NOT EXISTS (
5                SELECT *
6                FROM Rain [Range 1 Hour] as R
7                WHERE R.area = T.area
8             )
```

Listing 2.2: Running example in STREAM's CQL

2.2.3 Hybrid

Many commercial CEP systems adopt a hybrid approach – they combine the best of both worlds from the composition-operator-based and data stream query language models [EB09]. This can also be seen in by the last seven systems in Figure 2.1 which are all commercial systems. Most hybrid systems are data stream query languages by nature and include certain, or all, aspects of the composition-operator-based language model.

It is important to note that by including composition and logic operators into data stream query languages the expressiveness of the language does not increase. It can however improve the conciseness and readability of the queries. This is evident when comparing Listing 2.1 and 2.2. An example of a hybrid approach is the Event Processing Language of Esper [Tecb]. This is a data stream query language with support for composition and logic operators which makes it more natural to express the running example, see Listing 2.3.

```
1 select      temp.val
2 from        pattern [ every t:Temperature  → not r:Rain(area = t.area)
3                where timer:within(1 hour) ]
4 where       t.val > 45
```

Listing 2.3: Running example in Esper's EPL

2.2.4 Conclusion

The composition-operator-based language model more naturally expresses streaming event queries because they are specifically designed for this purpose. While data stream query languages piggyback on SQL to express streaming queries, the conversion from stream to relational table can seem odd at first.

When enabling and optimizing an event strategy to run in a distributed environment, the composition-operator-based language model might be the best fit. The execution more closely resembles the composition of the query, which opens up possibilities to optimize or enable horizontal scaling by modifying the query alone. It will also be possible to split and distribute a single query.

2.3 Event Stream Processing Systems

ESP systems come in many different flavours, and often they are categorized in the following categories: active databases, DSMS's, CEP systems and commercial systems [CM12]. The distinction between the different categories is not always clear and can become quite fuzzy, but we will still use these categories to highlight the generalized differences of their approaches.

Because commercial systems are also part of one of the former categories, and we want to highlight the differences between those categories, the commercial category has been omitted. Also a new category has been added: distributed stream computing platforms – general purpose, distributed, stream processing platforms. The significance of this category will become clear in Section 2.3.4.

2.3.1 Active Databases

Active databases are, arguably, the first timely CEP systems. Active databases use event-condition-action (ECA) rules [CKAK94] [ZU99]. Where the *event* is the execution of a query, either an insertion, deletion or update. After each event the condition, or trigger, is evaluated. This is a query without side-effects. When the trigger evaluates to true, the action is performed. The action is the execution of a new query, or even procedural code [RG00].

When a condition is satisfied over simple events, complex events can be inserted through the action. Because the insertion of complex events also trigger the evaluation of ECA rules, event hierarchies can be established. This is also one of the major drawbacks of active databases when considering performance. Because for every event all the triggers will be evaluated – recursive triggers amplify this effect.

2.3.2 Data Stream Management Systems

Data Stream Management Systems (DSMS) perform continuous queries over homogeneous data streams from different sources [CM12]. Queries manipulate data streams, produce new data streams as output or trigger production rules. Because DSMSs have their roots in traditional DBMSs they adopt the data stream query language model (see Section 2.2.2) which extends on common SQL.

Often they are integrated, or work in unison, with a traditional DBMS. As is the case with Esper [Teca]. This allows the use of static persisted data from a traditional database for use in streaming queries and supplement new streaming data. Internally, DSMS generally use a centralized query plan based approach. Because of the use of SQL these systems are very expressive, but also slower than finite automaton based approaches.

2.3.3 Complex Event Processing

Complex event processing systems do not view event streams as homogeneous data streams but as a flow of event notifications, where each event object models an event happening in the external world [CM12]. As such, event streams in complex event processing systems are generally heterogeneous, the stream contains different types of events.

Instead of manipulating the event streams, CEP systems focus on detecting patterns that represent valuable, higher level, and more complex, events than the lower level events that make up the pattern. Upon detecting a complex event the subscribers will be notified, or the complex events will simply be added to the event stream upon which clients can subscribe. The latter allows for a hierarchy of events.

CEP systems commonly use the composition-operator-based language model. This usually means a novel language specifically designed for the system at hand, by borrowing features from both temporal logic and regular languages. This means their detection algorithms are automaton based.

Automaton based systems have some interesting advantages, and disadvantages, compared to their DSMS counterparts: automatons are very fast yet comprehensible and their stateful approach gives opportunities for failover and availability optimizations without having to persist the whole event stream. The downside is the introduction of a new and novel language. Also the stateful approach introduces challenges regarding memory allocation.

2.3.4 Distributed Stream Computing Platforms

Distributed stream computing platforms (DSCP) are known as the MapReduce for data stream processing. DSCP's provide a general purpose platform to process continuous flows of data in a distributed manner. They provide an easy way to achieve horizontal scalability and fault tolerance.

Because of their generality, DSCP's provide no out of the box CEP functionality. They merely offer a framework to extend upon to achieve the data flow processing that is desired in a scalable way. This is achieved by making it simple to add nodes to, or remove nodes from, the cluster.

Often fault tolerance functionality is also provided. Ensuring either *exactly once*, *at-least once* or *at-most once* processing.

CEP engines such as Esper are designed to be implementable in a DSCP architecture, such as Apache Storm [Teca]. However, special care has to be taken in this case when setting up the Storm topology (data flow architecture) to ensure that the Esper EPL queries provide the desired outcome, since the stream cannot simply be partitioned.

2.4 Conclusion

Because of the different research efforts, from different domains, that focused on the ESP domain, a multitude of query language models and ESP systems emerged. Some of which can supplement each other, such as a DSCP and a CEP engine, while others have the same purpose but make different trade-offs. It is notable that there is no freely available CEP or DSMS engine that natively deploys on a clustered or distributed architecture without invalidating the results of the running queries. Vice versa all DSCPs provide rudimentary ESP logic, or the architecture to do so, but no full CEP functionality.

Looking at the systems described in this chapter, we believe the following is the best approach to (horizontally) scaling complex event stream processing: a combination of a DSCP with a CEP engine that incorporates the composition-operator-based language model.

Part II

Project Setup

Chapter 3

Research Method

Before we can start our research and perform any experiments we need a framework to execute event strategies in a distributed cluster. This chapter describes our experimental setup, outlines real world examples and explains the research method used.

3.1 Experimental Setup

To run a CEP engine concurrently in a distributed architecture we have to set up a DSCP (see Section 2.3.4). This platform will incorporate the CEP engine capable of detecting complex patterns. Enabling it to run on multiple nodes within a cluster.

The following sections describe and motivate our decisions for the CEP engine, DSCP, event stream and hardware architecture used in our experimental setup.

3.1.1 Pega Decisioning

The *decisioning engine*, the CEP engine used by the host organization's Pega 7.2 software platform, will be used as the CEP engine to run the complex event strategies. This engine is chosen instead of for example the more expressive and open source Esper [Teca] because internally the *decisioning* engine closely resembles a composition-operator-based CEP engine which we deem preferable over a DSMS (see Chapter 2). Also this has the most value for the host organization.

An event strategy in the decision engine is modeled by a Directed Acyclic Graph (DAG), where each operator is represented by a different vertex – or shape – in the graph. However, the event strategy graph is more restrictive than a conventional DAG; a stream can be split and joined up again, but has always exactly one source and one emit shape. See Figure 3.1 for an example strategy which aggregates dropped calls and performs an action when the amount exceeds 2 during a one week time window.

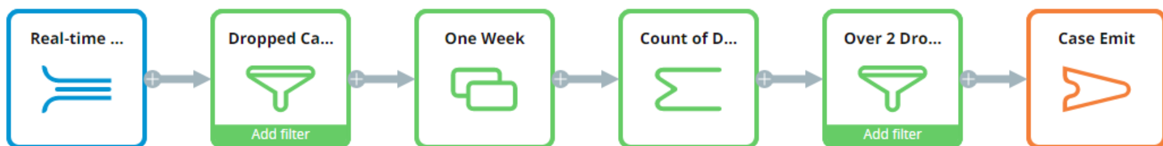


Figure 3.1: Example event strategy: the first shape is the source and emits events as they arrive in real time; the second shape is a filter that selects *dropped call* events; the next two shapes combine a sliding window of one week with a count aggregate; the fifth shape is a filter that selects the third (or more) dropped call; the last shape is the emit, in this case the emit strategy only emits the first event within the window and triggers an action.

3.1.2 Apache Storm

We will use the Apache Storm [Foub] engine (version 1.0.0) to distribute the decisioning engine. The data processing in a Storm application is done by a topology. A topology consists of one or more data generators called spouts, and one or more processing nodes called bolts connected as a DAG, see Figure 3.2. Each component – spout or bolt – of a topology can be run in parallel; potentially on different nodes that are added to the cluster.

Storm is chosen as the DSCP because of the native real time stream processing properties and clear processing structure, or topology, expressed as a DAG. The topology actually resembles the possible hardware topology, where each Storm component within the topology potentially resides on a dedicated processing node.

Another option is Apache Spark [Foua], but this is less suitable for our research because it works with batches through resilient distributed datasets on which processing can be done in parallel, this introduces the assumption that the dataset is known when a processing cycle begins.

By implementing the CEP engine in a Storm bolt it becomes almost trivial to run the decisioning event strategy in parallel on different machines. The clear processing structure, which mimics the structure of a physical computer topology, also presents possibilities for optimizations. For example by pulling constraints up to earlier components within the topology.

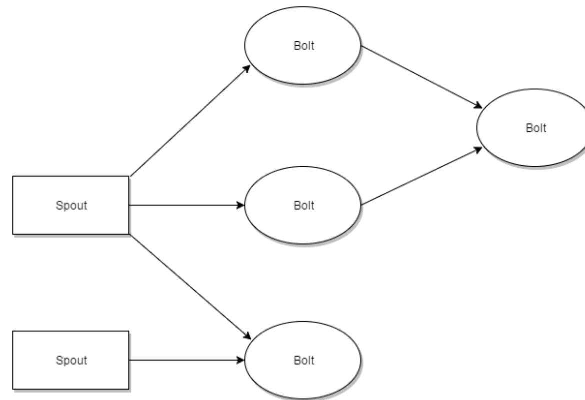


Figure 3.2: Example Storm topology.

Internally, a topology is executed by one or more workers, each worker process runs in a dedicated JVM to enable distribution of the topology over multiple nodes. A worker executes one or more specific (parts of) topologies through its executors. Each executor runs in a single thread and executes one or more tasks that are associated with the components of a topology.

Storm Configuration

Unless stated otherwise, the following settings and configuration options are used when running the experiments:

Workers: one worker per node in the cluster. Each worker is assigned a maximum heap size of 4096MB. A cluster of size one is used unless stated otherwise.

Executors: each component is run by a single executor by default. When a higher parallelism than one is stated this means multiple executors are deployed for the event strategy decisioning bolt(s) only.

Tasks: the Storm default of one task per executor.

Ackers: *acking* can be used to ensure that events have properly arrived. When enabled, a specified number of *acker* bolts will be spawned by Storm that resent events that are not acknowledged as received by the receiving bolt within the time-out period. However, acking introduces a significant overhead and our tests have shown that it can easily become the bottleneck of a

topology. This can be alleviated to some extend by spawning multiple acker bolts, but this requires testing to tune according to each topology.

To remove the variance in performance that acking can cause, this option will be disabled when running benchmarks. We conjecture that because all systems in the cluster will be on the same local network the events that get lost in transit will be negligible, also the damage caused is insignificant.

Spoutpending: the *maxspoutpending* option limits the number of events that can be in *flight* – events that have *not* been acked or failed – and requires acking to be enabled. The objective of limiting event in flight is to prevent long queues resulting in time-outs causing fails and replays.

We will disable this feature for the same reasons (and as a result of) disabling *acking*.

Buffers: Storm uses configurable buffers at the topology, worker, and executor level. These buffers determine the size of incoming and outgoing queues. Tuples are not processed immediately but are written to (in memory) queues of which they are read in small configurable batch sizes either by the worker or executor specific receive or send threads. The purpose of the buffers is three fold: *i.* use internal batching to reduce resource contention; *ii.* batch network usage to reduce overhead; *iii.* use buffered queues to handle throughput fluctuations.

We use the default buffer values.

Watermarks: Storm provides the *automatic backpressure* feature as an alternative to throttling using *maxspoutpending*. A low and high watermark are used which are expressed as a ratio of a task's used buffer size. Whenever the high watermark is reached, the spout(s) are throttled to slow down the topology until the low watermark is hit.

We did not change this setting and use the default values of 0.4 and 0.9 for the low and high watermark respectively.

3.1.3 Events

The events that are send through the engine for benchmarking are pulled from the GitHub Events API [Git]. Every (user) action such as pushing to, watching and forking a repository, create an event which can be read from the events API in JSON format.

We choose these events because it is an easily accessible real world heterogeneous event stream with many different types of events, carrying many unique attributes. A lot of usable real-world real-time example queries can be imagined with this data such as activity trends but also suspicious behaviour detection. This results in interesting and relatable use cases [DRKK+15].

In order to remove variations in the event stream and web request latency ∼360.00 events[1] have been pulled from the stream and will be read from disk and loaded in memory. To create an unbounded stream these will be cycled through repeatedly.

3.1.4 Hardware and Architecture

The hardware that we use to perform the main experiments for this research consists of the following two machines:

Server:

CPU: 4x CPU with 10 cores and 20 threads (Intel Xeon CPU E5-2650 v3 2.30GHz [Int14]).

Memory: 16GB of DDR4 RAM (per VM).

Disk: 90GB Solid-State Disk.

Workstation:

CPU: 4 cores and 8 threads (Intel Core I7-4700MQ CPU 2.40GHz [Int13]).

[1] http://tomvanduist.com/gh_data/gh_data.zip

Memory: 32GB of DDR3 RAM.

Disk: 500GB Solid-State Disk.

On the server we run two Virtual Machines (VMs), both with the Ubuntu (version 14.04.3 LTS) operating system (OS). The first VM, hereafter referred to as the *master*, runs Apache Zookeeper[2] to manage the inter-cluster connection discovery and setup. The master node also runs the Storm Nimbus daemon which delegates and distributes the work and topology code to the workers.

The second VM, hereafter referred to as *slave two* or the *second slave node*, exclusively runs te Storm Supervisor daemon. The supervisor(s) execute the topology. It gets his work assigned from the master and starts/stops worker processes accordingly. The Supervisor daemon connects to the Zookeeper cluster – in our case this is simply the *master* node, but for failover resilience this can be a cluster in its own right – which in turn subscribes the slave with the Nimbus master.

The workstation, which runs Windows 7 (Enterprise 64-bit), serves as the main slave node and is hereafter referred to as *slave one* or simply *the slave node* within a single node context. It is setup to connect to the master the same way as the other slave node. This node is used for the majority of the benchmarks.

Even though both slave nodes run on a different OS and consist of very different hardware, their performances when executing the topologies are comparable. Our tests have shown that their performance differs by around 5%.

Unfortunately, because of resource limitations, we could not employ a larger computer cluster. However, we achieved significant results using the resources at hand.

3.1.5 Benchmarking

To gather the experiment results we developed a benchmarking framework[3]. The framework runs on the master node and given a set of benchmarks, consisting of storm topologies, each topology is submitted to the cluster one by one and the results are written to disk in CSV format.

The data on the number of events processed by the topology, events processed per second by individual components and latency are gathered through the Storm API[4]. This is the same data that can be viewed through the Storm UI when running the *storm ui* daemon. Metrics on the memory usage are gathered through the JVM management interface *MemoryMXBean*[5]. The number of events within the event strategy windows are retrieved from each decisioning bolt window operators.

Unless stated otherwise, each topology within a benchmark is ran for ten minutes. Counting starts whenever each and every component within the topology is processing data for at least 25 seconds. The latter is chosen quite arbitrarily, but our data shows that this warm-up period is sufficient.

3.1.6 Profiling

For certain experiments we use a profiling tool to predict or quantify results. To this end we use the YourKit Java profiler (version 2016.02-b36). We use the sampling mode to measure the CPU time. Primarily we look at entire processing threads to determine the amount of time that a specific processing bolt, such as the decisioning bolt, uses. We also look at smaller execution areas, such as the transformation of event objects.

We only use the profiling data to acquire ballpark estimates. This is because profiling tools are inaccurate (see Section 5.3.4), and the profiling happens under different circumstances than when we run the benchmarks: running benchmarks happens in clustered mode, whereas the profiling tool can only be used in local mode, consequently it cannot be used to profile execution of a topology on multiple nodes.

---

[2] `https://zookeeper.apache.org/`

[3] A version removed of all metrics that depend on the Pegasystems decisioning engine can be found here: `https://github.com/tomvanduist/storm-perf-test`

[4] `https://storm.apache.org/releases/1.0.0/javadocs/org/apache/storm/generated/ClusterSummary.html`

[5] `https://docs.oracle.com/javase/7/docs/api/java/lang/management/MemoryMXBean.html`

3.2 Amdahl's Law

Amdahl's law [Amd67], presented in 1967, states the theoretical speedup that can be obtained by improving part of a system by increasing its resources. Amdahl introduced it in the context of multiprocessor computer systems to parallelize program execution and used it to argue that improving the sequential parts of the program cannot be neglected.

The law has since been generalized and can be used to determine the effectiveness of improving part of the program execution. Let S be the theoretical speedup, and f be the portion of the program that is affected by S, then the following formula calculates the total speedup [HM08]:

$$Speedup^{(f,S)} = \frac{1}{(1-f) + \frac{f}{S}} \tag{3.1}$$

For example, $f = 0.5$ and $S = 2$ gives a total *speedup* of 1.333. Any additional increase in S will yield diminishing returns. As a result, one can never double the speedup when $f = 0.5$, i.e. when 50% of the execution is being optimized, see Figure 3.3.

Amdahl's law can be used to provide an *upper* bound for the expected total speedup of the program execution time. As it overestimates the possible gains by not taking into account any overhead introduced by the speedup through parallelization. In reality every thread causes some overhead in the form of load distribution, memory stack allocation and in our case partitioning of the event stream.

However, Amdahl's law can easily be modified to account for the overhead [Qui03]:

$$Speedup^{(f,S)} = \frac{1}{(1-f) + \frac{f}{S} + k(S)} \tag{3.2}$$

Where $k(S)$ denotes the overhead. If k remains constant with increasing S, then the overhead may be negligible and can be accounted to the sequential portion of the program. If k increases with respect to S, then the overhead will predominate when it becomes larger than the speed gained. At this point increasing S will decrease the overall *speedup* as illustrated in Figure 3.3.

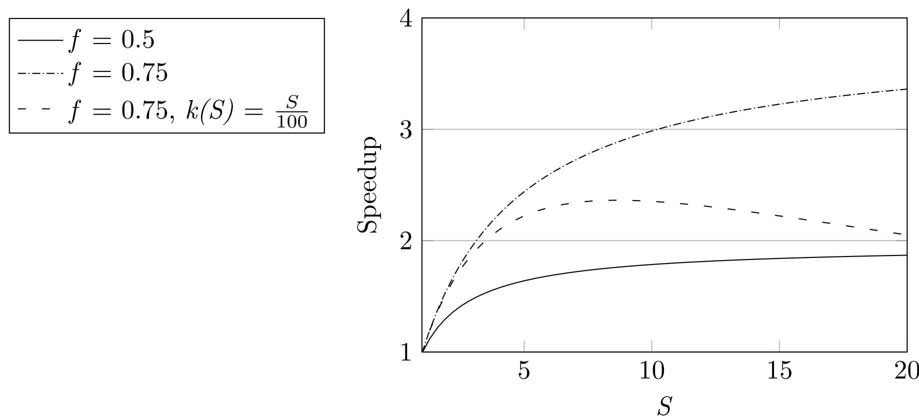


Figure 3.3: Amdahl's law plotted with and without increasing overhead.

Using the enhanced Amdahl's law (Equation 3.2) we can derive an equation to calculate the serial fraction *sFrac* when the *speedup* is known [Qui03]. Where *speedup* is the actual total speedup measured, and S is like before the speedup of the improved program fraction:

$$sFrac = \frac{\frac{1}{speedup} - \frac{1}{S}}{1 - \frac{1}{S}} \tag{3.3}$$

This can be used to reason about the actual fraction of the program execution that is affected by an optimization after performing an experiment. Which is useful to see if previous assumptions hold true.

In the context of this thesis, the fractional speedup (S) will originate from parallelizing the event strategy logic or optimizing part of the topology. Using the equations introduced in this section we can hypothesize about the speedup achievable by an optimization, or quantify the achieved results.

3.3 Running Example

The following running example will be used throughout the succeeding chapters. This example describes a real world use case in which a specific event strategy helps achieve a business goal.

3.3.1 Trending Scenario

A trending scenario measures the trend (change) of an event or event attribute over a time window. A famous example from technical analysis of stock prices is the MACD (moving average convergence/-divergence) indicator. Where two exponential moving averages of different sizes are used to detect a trend change, Figure 3.4. When the short moving average (blue) crosses the larger moving average – it goes from being lower to being higher or vice versa – this is interpreted as signalling a long or short term trend change. This is a sign to either buy or sell.

For example, consider the following simplified example: at T0 the 10-day and 2-day average prices are $10 and $5 respectively, this means the short term trend has been down since it is much lower than the long term average. When at T2 the 10-day and 2-day average prices have changed to $8 and $10 respectively, the short term *crossed* the long term trend in an upward motion, which signals a possible long upward trend and can be interpreted as a sign to buy.

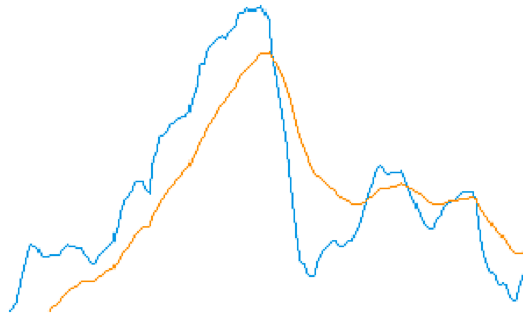


Figure 3.4: MACD exponential moving averages.

Trending scenario's are excellent candidates to be encoded in an ESP strategy to enable near real time detection of trend changes. Social media trends, stock prices, road congestion, usage of network enabled services come to mind as possible use cases. Trending scenario's are also heavily employed by Pegasystems, and her customers, on their platform. This scenario is also interesting because it employs *cooperation* between multiple commonly used operators in real time event processing, namely time windows and aggregations, possibly combined with filters and the join operator.

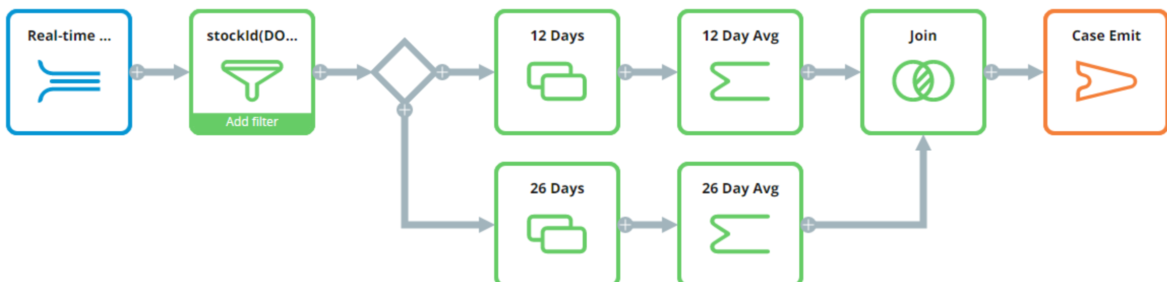


Figure 3.5: A stock price trend event strategy modeled using the Pegasystems decisioning engine.

3.4 Method

Before exploring the effects of horizontal scalability and other optimizations – using the experimental setup described in the preceding sections – we will establish a baseline for the normalized scenario, Storm plus the decisioning engine on a single machine. This helps us determine the maximum possible gains along certain vectors and steer direction of our research. See Chapter 4 for details.

The subsequent chapters each report on a single optimization. They can be categorized in two distinct groups. Namely those that achieve or enable parallel execution of the event strategy and thus increase the performance through horizontal scaling, and those that increase performance through an efficiency increase.

Optimizations regarding horizontal scaling are implemented to try and achieve similar benefits as the tried and tested solutions such as *MapReduce* have proven to be very effective [EHHB14]. Efficiency optimizations are discovered by performing experiments. Easily set up exploratory experiments are used to narrow the search field and find bottlenecks. The chapters themself are all organized similarly and each reflect the research method used along the following steps:

Exploratory experiments: are executed to find possible avenues along which optimizations might be discovered.

Vector: choose either a limitation (of a previous optimization) to overcome or a bottleneck vector to attack.

Hypothesis: observations, experiments or literature will be used to present a possible solution.

Benchmarks: are gathered for both the baseline – regular scenario – and the enabled/optimized scenario.

Comparison: the hypothesis is used to compare the results with the baseline.

Evaluation: the significance of the results are evaluated.

3.5 Threats to Validity

The biggest threat to the validity of this research is bugs in, or misinterpretations of, the benchmarking tool. We believe we have covered this sufficiently by only using the metrics provided by either the Storm client or the CEP engine. The validity is checked by running well understood and predictable topologies to calibrate the research method and discover possible bugs.

Furthermore the accuracy of benchmarks largely depends on two factors [RCCB16]: executing the benchmark in conditions that faithfully resemble the real execution conditions, and a framework that executes the benchmark a large number of times, or for a large amount of time, to get a statistical estimate of the execution time.

We are confident that we covered the first factor by running examples of real world event strategies on an actual cluster.

The second factor – which could be attributed to the fact that the performance of a topology depends on a lot of variables, including but not limited to network latency, background services, garbage collection and hardware caches – is covered by running long benchmarks of up to ten minutes. Depending on the strategy, this pushes more than 300 million events through the engine, and our tests show that this smooths out any significant variance, see Section 4.5.

Measuring and gathering metrics by benchmarking software could potentially slow down the software that is being benchmarked. Similar to how profiling an application introduces overhead. However, because of the way that we set up our architecture this does not affect our benchmarks when measuring throughput. The metrics that are used are already gathered by the Storm engine by default and are accessible by the master node, which does not directly execute the topology. Therefore the metrics are already gathered, our benchmark framework only polls them regularly, introducing minimal overhead.

Cycling through a limited set of events, as explained in Section 3.1.3, creates an unrealistic continuous stream of realistic events. The distribution of events within the stream is realistic, but because we

have to cycle them this creates an unrealistic repeating pattern. It does however create consistently the same stream for each benchmark. Being able to compare benchmarks reliably is paramount for the research in this thesis.

The replicability of the results presented in this thesis can largely be dependant on the make up of the event stream. For instance, the effect of partitioning the stream depends on the number of partitions and data transfer and transformation on the size of events. Therefore we make available the events that we used, see Section 3.1.3.

We disabled *acking* and the *maxspoutpending* option (see Section 3.1.2) which could result in packet loss, and therefore loss of events. This is not a significant risk because we use continuously satisfying queries that are not dependent on the (non-) occurrence of a particular event. In the worst case scenario it takes an extra cycle for a window to become *full*.

Because of limited resources available we are not able to experiment with a cluster larger than two comparable nodes. As a result we cannot test if our results are generalizable to larger clusters. When adding more nodes diminishing returns are expected, in the same manner as parallelizing on a single machine, and additional bottlenecks might be discovered. However, even on a two node cluster we achieve significant results.

Chapter 4

Baseline

We created a benchmark framework (see Section 3.1.5) to measure the performance of an event strategy along a set of possible bottleneck vectors – namely throughput, memory load, intermediate results, latency and bandwidth measured by number of events. To get a sense of the load that our architecture can handle, we benchmarked the Storm topology and event strategies with each of the core operators. These are used to establish a baseline for the architecture and the different event strategy operators.

4.1 Motivation

Any improvement – either enabling or optimizing – will either have to bring the performance of an event strategy (operator) closer to the architectural maximum, or increase the performance of the architecture as a whole.

With the knowledge of the relative cost of the different operators – such as filter and aggregation – we can determine where the most performance can be gained by improvements at the topology or event strategy level.

These baselines will also define what the maximum improvement can be of an optimization along a specific vector. If for example a bare event strategy – one with no logic and only a source and a sink – can consume at most 100 events/s. Any strategy that adds logic, i.e. operators, to a strategy will be slower. Optimizing on operator level will not make it faster than the bare event strategy.

The baseline forms an upper bound that can be approached by optimizing at a lower level, or that can be moved by optimizing at a higher level.

4.2 Method

To determine these baselines, eight different topologies are created. We start by measuring the throughput of the Storm engine itself without the decisioning engine. A stub bolt will be used instead, doing no processing but immediately emitting the events to the sink, Figure 4.1.

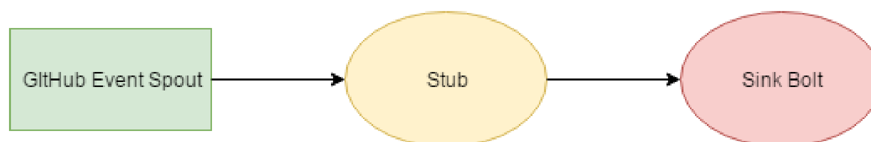


Figure 4.1: Storm topology without the decisioning bolt, the stub merely forwards the events to the next component.

The subsequent topologies all incorporate a decisioning bolt that executes an event strategy. Starting with an empty strategy, simply consisting of the source and emit operator, Figure 4.2. Next, the

event strategies introduce the different operators; first the filter, then (windowed) aggregation and so on.

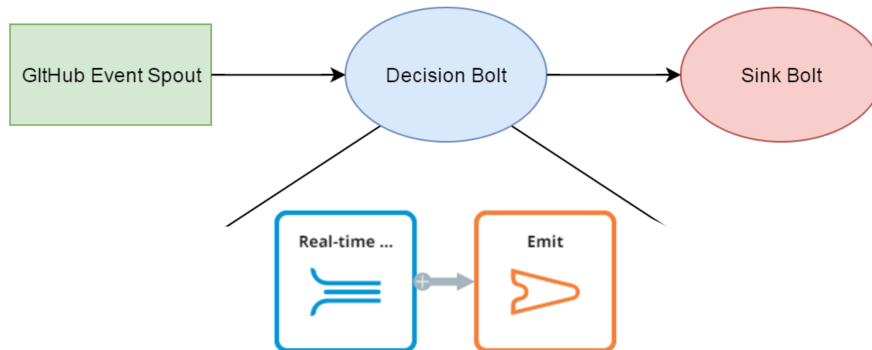


Figure 4.2: Decisioning benchmark topology. The *Decision Bolt* incorporates the Pega decisioning engine running a single strategy. Any events emitted by the strategy are sent to the *Sink Bolt*.

The cluster layout for measuring these baseline benchmarks is the same as will be used for most of the other experiments that we conduct during this research. Specifically, it consists of a master node and a single slave node. The master coordinates communication between the slave nodes in the cluster, and distributes the work (and accompanying code). Essentially the master delegates work to the slaves which perform the actual processing.

Thus we will be using a single node to process the topology, see preceding Section 3.1 for more details about the architecture and hardware that is being used.

4.3 Result Analysis

The results clearly show which operators are more expensive. Furthermore it also shows that there is an inherent cost to transferring data between individual Storm components. Because the decisioning bolt is assigned only one executor (see Section 3.1.2). This also means that the thread that is responsible for performing the decisioning logic also transfers the data to the next Storm component. Strategy 4 in Table 4.1 applies a filter that reduces the amount of events that are emitted by the event strategy by 50%, this in turn increases the total intake of the decisioning engine by almost 15%. Each component' task is responsible for emitting its data to the next component. This transfer of data introduces noticeable overhead for the topology as a whole.

The latency is the amount of time it takes for one event tuple to travel through the topology; from the spout to the sink. It denotes the average time it takes for an event to be processed. Because events are processed sequentially, a higher latency results in a lower throughput.

#	Input	Output	Latency	Heap Size	Decision Strategy
1	574	574	5 ms	1380 mb	No decisioning engine
2	375	375	99.6 ms	1851 mb	Source and emit shape
3	374	374	98.4 ms	1747 mb	Filter 0%
4	442	221	22.5 ms	1847 mb	Filter 50%
5	332	207	105.9 ms	1993 mb	Filter 50%, aggregate with winSize(100)
6	303	303	129.8 ms	2001 mb	Aggregate with winSize(100)
7	192	192	212.6 ms	2240 mb	Split and Join
8	140	140	279.5 ms	2427 mb	Split to aggregates with winSize(100)

Table 4.1: Baseline result table, input and output is measured in k events/s.

The input number is paramount when comparing the individual decisioning shapes. This determines the number of events per second that the decisioning engine can process. The event strategy itself

determines the amount of inserted events that are emitted by the decisioning bolt. This makes the total not a good metric when measuring the effectiveness of the decisioning bolt.

Looking at the table it is clear that the filter shape itself imposes a negligible performance hit of less than 1%, but can actually increases the throughput of the decisioning bolt by decreasing the data that has to be transferred to the next component in the topology.

All other shapes introduce a very noticeable decrease in overall performance. Up to 50% in the case of the Split and Join operator combination.

4.4 Threats to Validity

Changing the parameters used by the event strategies – filtering 50% and window size of 100 – can potentially change the results of the benchmarks in a non-significant manner. However, both parameters give significant results while being quite low. Because the stream is by default partitioned on the 24 distinct types, the window size of 100 results in a total intermediate result set of 2400 events. Our tests have shown that a split join strategy with two aggregates only begins to slow down significantly when there is an intermediate result set of ∼20 million events. Therefore the changes in performance can safely be attributed to the windowed aggregate shape and not the particular size of the window.

4.5 Benchmark Noise

Inherent with running benchmarks where the result of the measurements are dependent on many variable and undecidable influences, there will be a certain amount of noise. Running the same benchmark will not result in exactly the same result.

To find the error margin, or noise, that can be expected when running benchmarks on our experimental setup we gathered a set of results for the same topology, calculated the standard deviation (SD) and generated a boxplot. We used an *expensive* event strategy with a split-join operator and on each split stream an aggregator.

Running 60 benchmarks on the experimental setup (see Section 3.1 for details) with a single slave node resulted in the following results: a mean of 131.6k e/s and a SD of 3657.3 e/s. That is a deviation of 2.78% from the mean.

Running the same experiment on a two node cluster resulted in a mean of 148.9k e/s and a SD of 4217.3 e/s. That is a deviation of 2.84% from the mean.

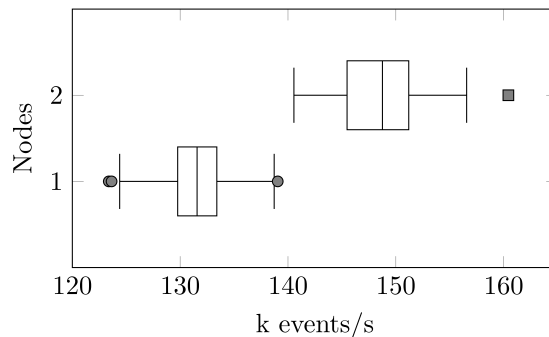
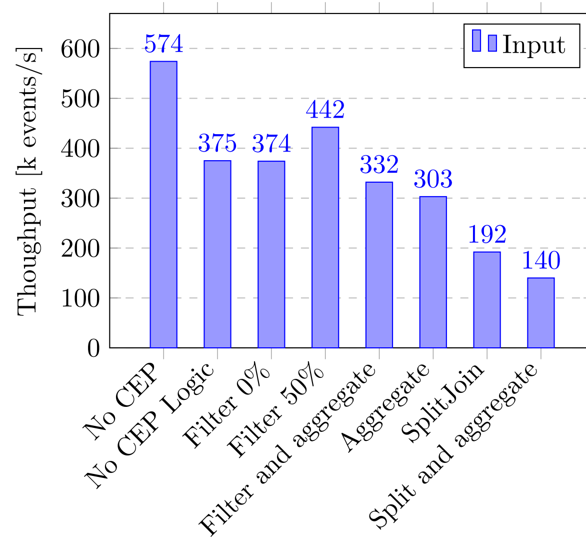


Figure 4.3: Benchmark noise boxplot for both the one node and two node setup.

4.6 Conclusion

In conclusion we can state that the CEP engine introduces a lot of overhead even when doing no real processing; it is 35% slower than a topology without CEP engine, even without any logical operators. The filter operator does not introduce a noteworthy bottleneck. It can even increase performance by reducing the number of events that are transferred.

Furthermore the windowed aggregation significantly reduces performance, and the split join combination even more so.



Part III

Optimization Experiments

Chapter 5

Partitioned Parallelism

Every Storm component can be run in parallel, on one node within the cluster or on multiple. Storm will automatically choose the best layout for a topology and the available resources. Even a cluster of just a single node can potentially benefit from parallelism because this will allow the components to be run concurrently by multiple threads.

For each data stream between components within a Storm topology – for example the *GitHub Event Spout* and *Decision Bolt* in Figure 5.1 – a specific streaming strategy can be selected. This defines how the events are divided among different instances of the receiving component when parallelized. *Shuffling* randomly distributes events to the bolt with the lowest load, whereas *partitioning* ensures that events with the same attribute always go to the same bolt. The latter can ensure correct results when the event strategy that runs on the decisioning bolt is partitioned as well.

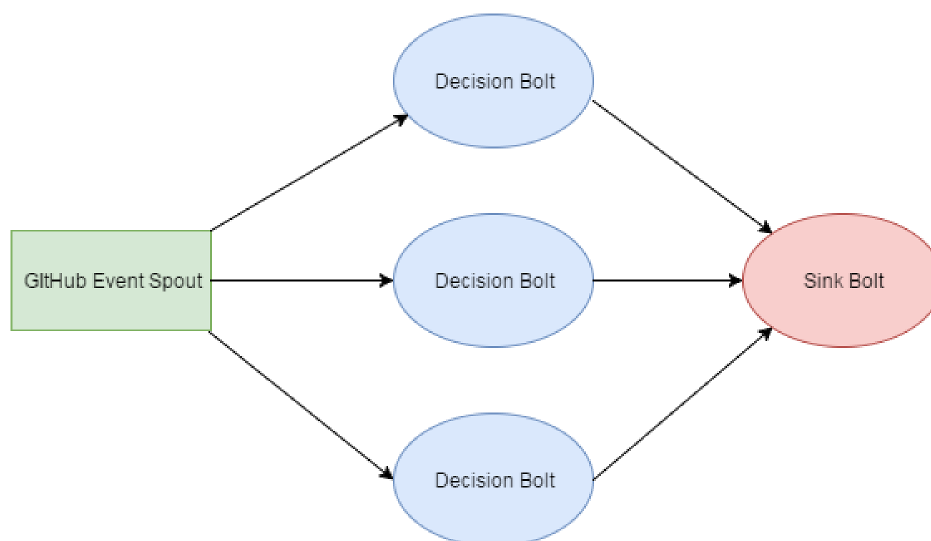


Figure 5.1: Storm topology with Decision bolt partitioned parallelism level of three.

5.1 Limitations

The possibility of applying partitioned parallelism without invalidating the results depends on the event strategy in question. The event strategy must be partitionable, this ensures that events that fall within a partition will always arrive at the same instance of the engine. This is important because a CEP engine, in this case the Pega decisioning engine, is stateful – unlike most DSMS [CM12]. In many cases partitioning the stream makes sense as well: a stock market strategy partitions a stream containing data on multiple stocks on *stock_id* to aggregate data of the same stock index.

There are however cases in which partitioned parallelism cannot be applied without invalidating the results of the event strategy:

Shifting partition: When an event strategy changes its partitioning attribute in the middle of the strategy, for instance to aggregate a different attribute. Events with the same partition key will be distributed to different instances of the engine. See Chapter 6 for a detailed description of the problem and a proposed solution.

Global knowledge: Event strategies that only satisfy on the (non-)occurrence of a global event, or an event outside of the current partition. For example detecting when a wind turbine decreases in performance while the wind power within the park remains the same. See Chapter 7 for a detailed description of the problem and a proposed solution.

These scenarios however can be enabled to run in parallel by remodelling the event strategy and Storm topology. It can also be said that they are too complex to start with. When comparing data, or trends, of different stocks it makes sense to use multiple queries. One layer of queries responsible for extracting the desired attribute or trend from simple events and emitting a complex event, this can run in parallel. A top level, more complex, event strategy can subscribe to these events to extract higher level knowledge.

5.2 Use Case

A standard trending scenario over GitHub user events, derived from the running example (see Section 3.3), will serve as the use case for this chapter. See Figure 5.2 for the accompanying event strategy. The strategy is partitioned on the GitHub user attribute and filters the type attribute on the *PushEvent* key, this removes 50.5% of the events. Next the stream is split, this means all events after this point are duplicated over two streams. Both streams apply a count aggregate with a sliding window, but of a different size. The join shape joins these events back together, the resulting complex events are push action events with two appended attributes – namely the two different count aggregates. This knowledge signifies an up or down trend in distinct user push activity and can be used by other strategies.

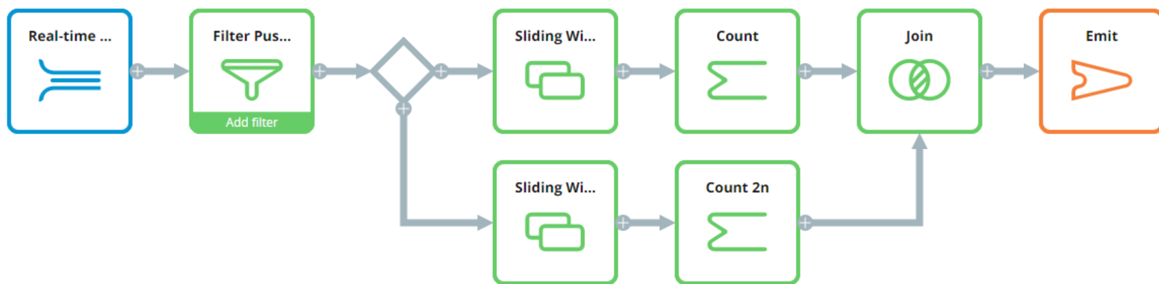


Figure 5.2: This event strategy uses two count aggregates to append trending knowledge to user push events.

5.3 Single Machine Parallelism

Storm topology parallelism can be used to run a topology in parallel in a multi core environment on a single machine. When running a component in parallel, we configure Storm to create a dedicated executor for this component, this means it will run in a dedicated thread (See Section 3.1.2).

5.3.1 Hypothesis

When parallelizing the decisioning bolt on a single node cluster, multiple instances of the CEP engine will run in parallel. The total available CPU time for the decisioning event strategy will increase but

the memory consumption will increase as well because more components, namely decisioning bolts, are added to the topology.

Because only one portion of the program can be parallelized, and this gives diminishing returns as per Amdahl's law, at a certain parallelism level a tipping point is met. Using a profiler we determined that the decisioning bolt accounts for 47% of the CPU time. Recall Amdahl's law as Equation 3.2 and let $f = 0.47$.

We cannot determine the overhead introduced by parallelism beforehand without running the experiment, so we use the regular version of Amdahl's law to determine an upper bound for the speedup. This results in the following table:

PARALLELIZATION	1	2	3	4	5	6
Speedup	1	1.31	1.46	1.54	1.60	1.64

With a parallelism of two, this gives a speedup of 1.31. The tipping point will likely be at a parallelism level of four to five, where the increase of performance per parallelism level has decreased to about 5%.

5.3.2 Method

The setup of this experiment consists of a Storm topology with a single spout, partitionable decisioning bolt, and a sink bolt as in Figure 5.2. The baseline is created with a parallelism of one. This is a regular topology where the decisioning engine is executed sequentially.

By partitioning the event stream on the *actor_id* attribute the decisioning engine can be partitioned and run in parallel. This means that correctness of the results is ensured by sending events with the same partitioning value to the same instance of the engine. Note that only the decisioning bolt will be parallelized.

The architecture of the cluster consists of two nodes, a master node and a slave node. The master initiates the experiment and distributes the topology – and code – over the cluster, in this case the single slave node. The slave node is solely responsible for executing the topology, the master monitors and logs system metrics. For each parallelism level the master submits a new topology.

5.3.3 Result Analysis

Results show that the throughput of the trending scenario is indeed increased, see Figure 5.3. The best results are achieved with a parallelism of four with a speedup of 2.77 compared to serial execution. This is however more than the upper bound that we hypothesized using Amdahl's law. Meaning that the program portion that benefits from the parallelization is higher than hypothesized.

Note also that the creation of extra components, as we hypothesized, causes memory use to increase with each extra level of parallelism.

We can calculate the sequential program fraction *sFrac* using Equation 3.2. For the results presented in Figure 5.3 this gives the following table:

PARALLELIZATION	1	2	3	4	5	6
Speedup	1	1.73	2.34	2.77	2.74	2.64
sFrac	-	0.15	0.14	0.15	0.21	0.26

The serial portion denoted by *sFrac* is next to constant for the first three levels of parallelization, and corresponds with an f value of $\sim$0.85 instead of 0.47. From a parallelism level of five and onwards the overhead increases steadily, decreasing the overall performance.

Profiling the topology with a parallelism level of five shows that the most busy single thread is now the source spout, with 27% of the CPU time. It also shows that all five decisioning bolt threads are regularly *waiting*, whereas the spout is always busy. This means the source spout has become the new bottleneck and any additional decisioning bolts add nothing but overhead

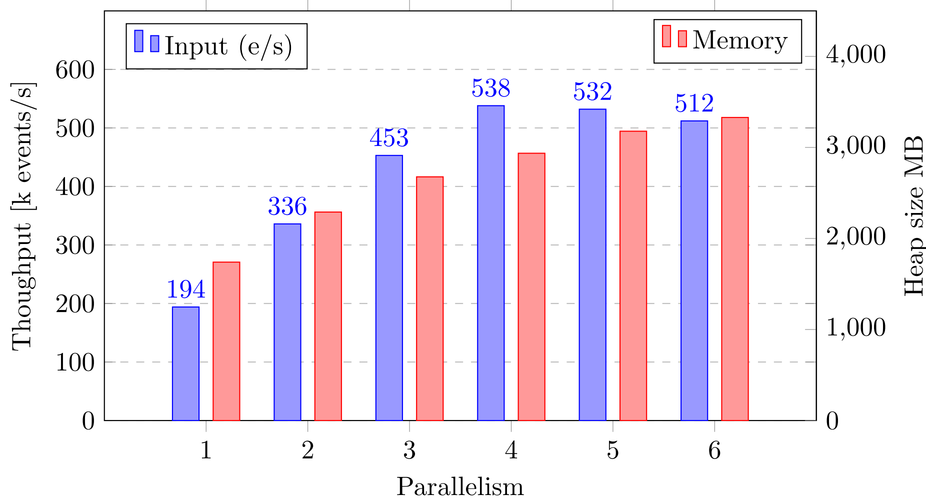


Figure 5.3: Trend strategy parallelized on a single machine.

5.3.4 Threats to Validity

Research has shown that (Java) profilers can be inaccurate and give inconsistent results, even between different runs of the same profiling tool [Kli14]. This happens especially when instrumenting, but also with the less intrusive *sampling* mode. To reliably detect the *hottest* method, and its respective CPU time, it is advised to use multiple profiling tools. However we only use the YourKit tool, see Section 3.1.6. Because the profiling data we gathered is not 100% accurate, we only use it as a *ball park* estimator to find *hot areas*. With regards to this chapter we only had to determine the estimated CPU time used by the decisioning engine executor thread and it does not matter which low level methods are actually responsible for this load.

The results we presented are specific for the event strategy that the decisioning bolt implements. It does however implement a real world scenario, showing that these results are within expectations. Our tests with other event strategies from subsequent chapters show similar results, however, for each topology experimentation is required to find the optimal settings regarding parallelism.

5.3.5 Conclusion

Based on the results shown in Figure 5.3 we can state that in our experiment the hypothesis holds. Parallelizing CPU intensive event strategies can increase total throughput until a certain tipping point. However, using a profiling tool does not accurately predict the program portion that will benefit from the parallelization.

Experimentation is needed, for each distinct topology, to discover the actual performance gains by using parallelization on a single node and determine the optimal level of parallelism.

5.4 Multi Machine Parallelism

Storm topologies that are run on a cluster will distribute their component execution over multiple workers. Ideally, if there are enough nodes with idle workers in the cluster, each worker will run on a dedicated node. In our setup we use a cluster with two workers and two slave nodes.

5.4.1 Hypothesis

When increasing the size of a computer cluster, the available CPU and memory increases linearly with every node that is added. Compared to executing a topology on a single node, deploying it on a cluster increases its available resources proportional to the number of nodes, but parallelizing on a cluster also introduces overhead. The stream has to be partitioned – just as with single machine

parallelism – and the events have to be transformed to different data types and transferred over a network from one node to the other.

In the previous section we discovered that it was not the diminishing returns of Amdahl's law that caused the performance to slow down, but that the spout had become the bottleneck of the topology. The topology would not benefit from parallelizing the decisioning bolt further. We therefore hypothesize that adding a node to the cluster and only increasing the decisioning bolt parallelism *decreases* performance compared to single machine parallelism.

However, when the number of spouts can be increased, we effectively increase the parallel part of the program. If the events are received through web sockets each spout can listen to a different socket. Looking at the performance gained by single node parallelism, and the derived f value of 0.85, we can again use Amdahl's law to predict the speedup: let $f = 0.85$ and $S = 2$ results in a *speedup* of 1.74 as the upper limit. Although overhead introduced by the network communication between nodes lowers the expected speedup.

A cluster of multiple nodes can also be used to increase the total memory that is available to the topology or decrease the intermediate results per node. We hypothesize that using a multi-node cluster for a strategy with a very large window size will run unhindered for a longer period. Having more events reside in the windows of the engines before slowing down.

5.4.2 Method

The same topology will be used as the preceding Section 5.3 experiment. The event stream will again be partitioned on the *actor_id* attribute. By running the topology on a cluster, and parallelizing the decisioning bolt, the event strategy will run in parallel on two different nodes.

The parallelism of the decisioning bolt must always be a multiple of the number of nodes. Otherwise one node will process events much quicker than the other one and request more events from the event spout. This would not be a problem if it did not matter which node got an event, and load balancing can be applied, but because the event strategy is stateful and the stream is partitioned, the load cannot be dynamically balanced.

This means that a parallelism of one in Figure 5.4 means that there is one decisioning bolt per worker, for the two node setup a parallelism of one results in two decisioning bolts, one per node. This ensures that the topology is balanced.

Three experiments will be performed:

1. With just a single spout, this will test our hypothesis that this will not increase performance, as per the diminishing returns predicted by Amdahl's law.

2. For the second experiment the spout is parallelized equal to the number of nodes in the cluster, in this case we hypothesize that the performance increase corresponds to an f value of 0.85.

3. The third experiment is aimed at increasing the number of events that reside in the windows as fast as possible. Unlike the other strategies it is partitioned on the repository id, which is also incremented at each cycle. In combination with window sizes of 50k events the number of events in both windows will rise indefinitely.

The cluster setup is the same as with the Chapter 5 experiment. The only difference being that there are now two slave nodes instead of just a single slave.

5.4.3 Result Analysis

The results in Figure 5.4 show that the two node cluster, which uses a single spout, does not increase performance compared to a single node. This is in line with what we hypothesized. Note that the parallelism of the decisioning is defined per node. This means that both the single node cluster and the two node cluster reach maximum throughput at a total decisioning parallelism of four, irrespective of the node count.

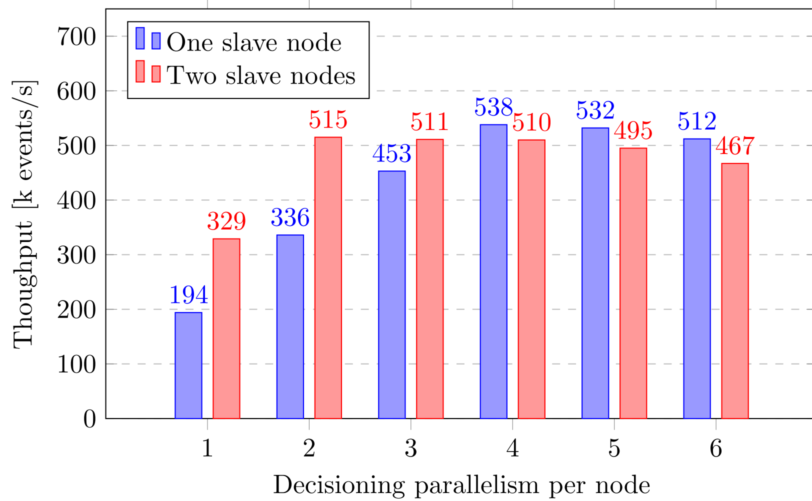


Figure 5.4: Trend strategy parallelized on a two node cluster with a single source spout.

The results in Figure 5.5 show that the two node cluster which uses multiple spout nodes can, as we hypothesized, reach a throughput that higher than the single node cluster. Particularly, it reaches a speedup of 1.46 at a parallelism of five compared to the single node cluster. By determining the *sFrac* using Equation 3.2, this corresponds to an f value of 0.63. This means the two node cluster, at a parallelism of five, increased the serial fraction through added overhead from 0.15 to 0.37.

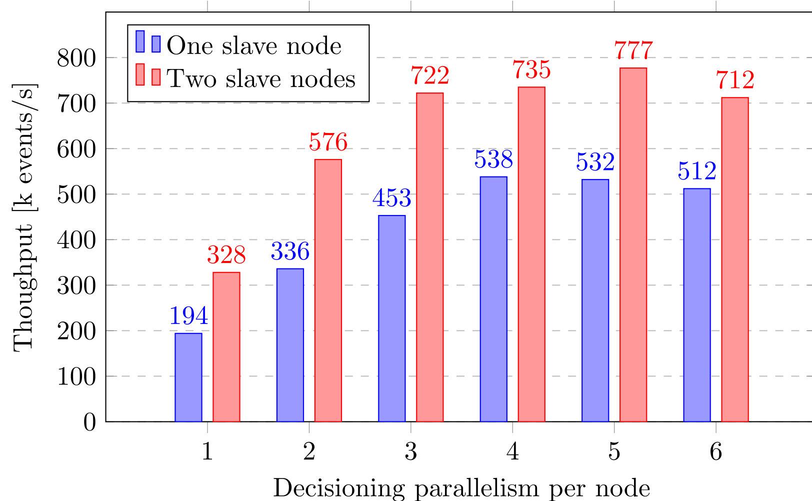


Figure 5.5: Trend strategy parallelized on a two node cluster with two source spouts. Note that the small difference between two nodes parallelism level three and four falls within the standard deviation from the mean, see Section 4.5.

The results in Figure 5.6 compare, both for a single node and two node cluster, the amount of events in the windows and the total throughput of the decisioning bolt. The single node cluster begins to drastically decline in performance at around 35 million events.

Comparing the single node and two node cluster it is clear that the two node cluster runs relatively stable for a longer period of time than the single node cluster. Only at around 75 million events it begins to significantly degrade in performance.

Using a profiler we can see that the CPU time that is needed to insert events in the window and calculate the aggregate result increases with the number of events. At around one million events this takes around 25% of the decisioning bolt executor thread's CPU time. At around twenty million

events this has increased to a total of 40%. Also the time spend in garbage collection (GC) increases from an average of 246ms to 539ms, with a maximum GC pause of more than 11 seconds. By using multiple nodes this load is distributed and GC time on individual nodes reduced by reducing the heap size.

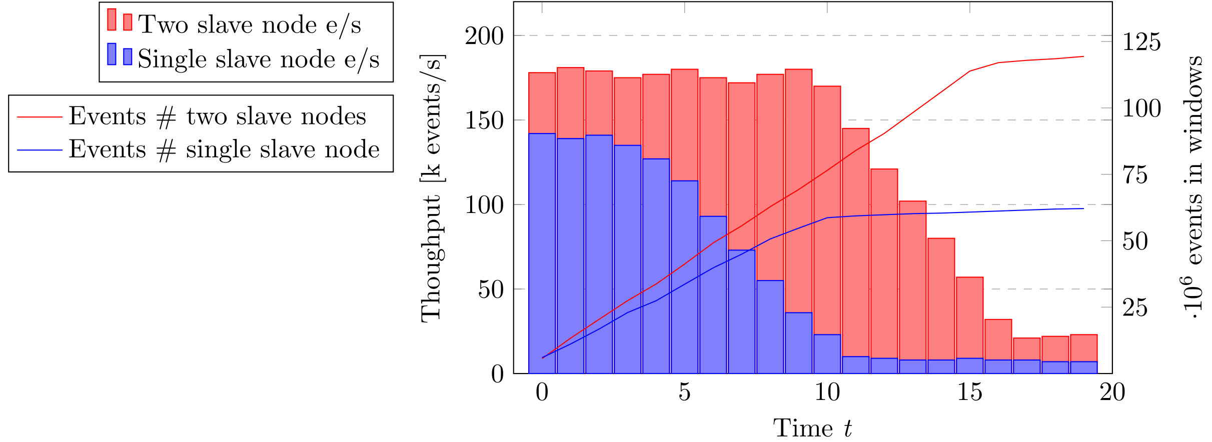


Figure 5.6: Trend strategy with windows of size 50k events. Compares the moment at which the events in memory become the bottleneck of a single node cluster and a two node cluster.

5.4.4 Threats to Validity

Unless the values of the partition attribute are evenly distributed, partitioning an unbounded stream of events will never yield a completely even load distribution. For example, within our dataset the *type* attribute contains 24 distinct types, of which ~49.5% are of type *PushEvent*. The node that gets this slice assigned, which is selected through a randomized round-robin process, gets at least 49.5% of the load plus any other slices. However, profiling shows that partitioning on GitHub users results in an even load distribution among the decisioning executors.

Unfortunately we only have a limited set of nodes available to perform benchmarks, which is quite small compared to computer clusters usually used for continues stream processing. Because our node count is quite small we cannot measure if the results are generalizable to for example a ten or twenty node cluster. According to Amdahl's law this of course depends on the portion of the topology that would benefit from the horizontal scaling. However, having such a large cluster has another benefit beyond mere horizontal scaling. The total memory increases as well, and if not all nodes are busy executing a single topology, it could be used to deploy and change multiple topologies on the fly without having to worry about spinning up new servers first.

The results as presented in this chapter do only apply to event strategies where the available CPU time – or available memory – is the bottleneck. If the bottleneck of the event strategy is for example I/O operations through network calls or to a shared data source this could easily become the bottleneck. In those scenario's parallelizing could even lead to race conditions.

Other threats to the validity are the same as mentioned in Section 5.3.4.

5.4.5 Conclusion

Based on the results as discussed in Section 5.4.3 we can state that our hypotheses hold regarding the use of a multi-node cluster to increase performance of our experiment using partitioned parallelism. However, Amdahl's law could not be reliable used to predict the speedup using knowledge about the CPU time of the decisioning bolt. Where possible, the spout can be partitioned to parallelize an otherwise sequential part of the topology to take benefit of the multi node setup.

Similar results hold when it is the window size – or intermediate result size – that is the bottleneck of a topology. Using multiple nodes this load can effectively be distributed to significantly delay the moment that performance degradation kicks in as a result of large intermediate result sizes.

Chapter 6

Shift Partition

An event strategy that performs aggregations over two or more different attributes has to change the event strategy partition attribute. Either by splitting the event strategy into differently partitioned streams, or changing the partition attribute before each aggregation.

Recall from Chapter 5 that when a single event strategy has to partition on multiple attributes, the results will be invalidated when parallelized, thus disabling the possibility of parallelizing such event strategies. This can be fixed by dividing the event strategy over two distinct Storm bolts. Storm has a higher level knowledge of the topology and can change the partition attribute before each component, ensuring each decisioning engine gets the correct events.

6.1 Use Case

Since the running example cannot be used to describe this problematic edge case the following use case will be used as a running example throughout this chapter:

We want to know, for each push event in the event stream, both the number of occurrences of the GitHub user and the repository within a given window. This means a strategy that adds knowledge to each event about both the number of GitHub push events for users and for repositories, Figure 6.1. The resulting events can for example be used to identify high profile contributors – people who contribute extensively to popular repositories.

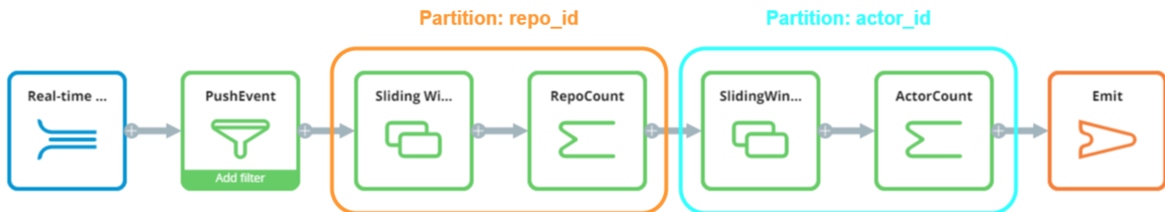


Figure 6.1: Event strategy that adds both repository count and actor count attributes to push events. Both windows partition the stream on a different attribute to aggregate the correct events.

6.2 Problem

The above use case, when implemented as Figure 6.1, will not give correct results when parallelized either locally or in a cluster. The problem occurs when the decisioning bolt is parallelized and Storm partitions the stream on an attribute, in this case *repo_id*. See Figure 6.2.

Events with the same repository id will be send to the same instance of the engine: T1 and T3 to the first instance because they both have *repo_id: 1*, T2 and T4 to the second. When the partition attribute in the event strategy is changed to the *actor_id*, the events that should be in the same partition are now in different instances of the decisioning bolt and by extension in different partitions.

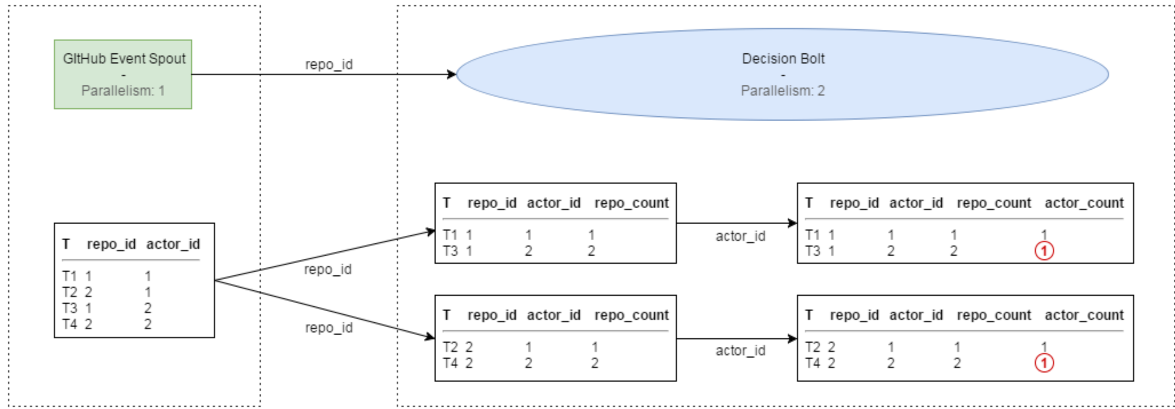


Figure 6.2: The parallelized strategy from Figure 6.1 returns invalid results because after changing the partition, events are aggregated in different instances of the decisioning engine while the aggregation should be over all events. The circled numbers should be 2.

The aggregation over *actor_id: 1* is happening in different instances of the engine, resulting in multiple different aggregations that should actually be combined.

The result is that the *actor_count* attribute in all resulting events equals *one*, whereas for both T2 and T4 it should actually be *two* because it is the second event from the particular actors.

Given the following function that retrieves the partition attribute keys:

$$PartitionKeys(chain) = \{partition \mid partition \ \in \ chain\} \tag{6.1}$$

We can summarize by stating the following:

$$\begin{aligned} & stormStream \ \rightarrow \ eventStrategy \\ & \quad \Longrightarrow \\ & PartitionKeys(stormStream) \ \Leftrightarrow \ PartitionKeys(eventStrategy) \end{aligned} \tag{6.2}$$

The partitions of the stream that connects the preceding component to the decisioning bolt must be equal to the partitions used by the event strategy. In the example above this is not the case because:

$$\{repo_id\} \ \not\Leftrightarrow \ \{repo_id, \ actor_id\} \tag{6.3}$$

6.3 Solution

The solution is to divide the event strategy over different decisioning bolts. One distinct bolt for each distinct partitioning. This gives the responsibility of changing the partition of the stream to Storm instead of the decisioning engine. Storm has a higher level knowledge, governing the topology level, whereas the decisioning bolt only has knowledge about his particular event strategy.

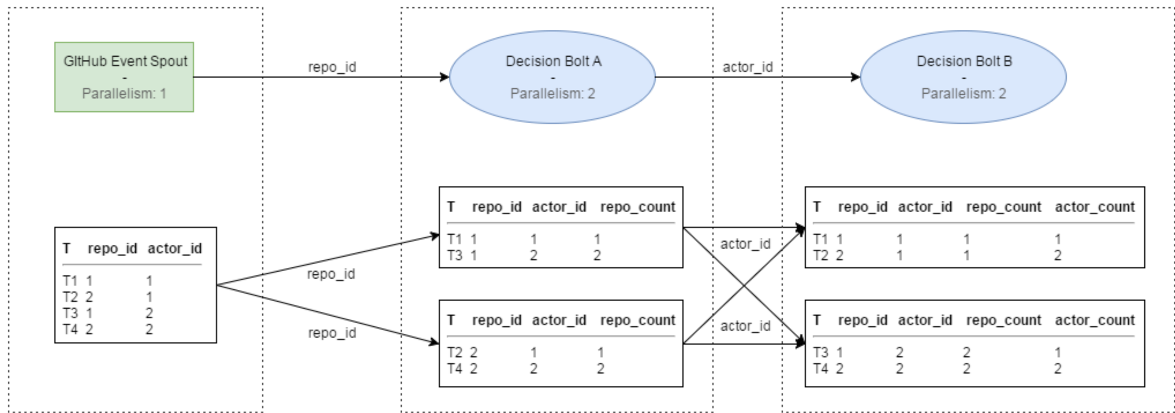


Figure 6.3: By splitting the strategy from Figure 6.1 and create a distinct bolt for each distinct partitioning, the results when parallelized are now correct.

This gives:

$$\begin{aligned} & stormStreamA \rightarrow eventStrategyA \wedge stormStreamB \rightarrow eventStrategyB \\ & \quad \Longrightarrow \\ & \{repo_id\} \Leftrightarrow \{repo_id\} \wedge \{actor_id\} \Leftrightarrow \{actor_id\} \end{aligned} \tag{6.4}$$

This enables the topology to take advantage of partitioned parallelization, but because the event strategy is now executed by two or more decisioning bolts instead of one it also has a cost. Because there are more components, the total memory used by the topology increases. As does the total amount of events that are transferred between the topology components, potentially increasing the bandwidth.

6.4 Hypothesis

As we show in Chapter 9, transferring events between topology components has an inherent, and significant, cost. By implementing the solution proposed in the previous section, we are adding an extra bolt for each distinct partitioning, increasing the components in the topology and the events that have to be transferred. This can introduce significant overhead.

But each executor thread is responsible for transforming and inserting the data in the sent buffer. Also for each executor an additional thread is spawned that is responsible for taking the data from the buffer and sent it to the next component. This means a lot of the overhead is not sequential but parallelized as well, doubling the decisioning bolts also doubles this overhead.

We hypothesize that despite the overhead introduced by the extra decisioning bolt, the enabling of parallelization will result in a net increase of the total achievable throughput. Specifically, the decisioning executor spends 23% of its CPU time on different tasks than executing the event strategy, we call this its overhead. Introducing an additional decisioning bolt reduces performance by around ~23% compared to the problematic scenario.

6.5 Method

Two different topologies will be compared with each other. The first topology exhibits the problematic scenario, a single decisioning bolt implementing the topology from Figure 6.2. Shifting the partition to a different attribute after the first aggregation.

The second topology exhibits the solution, implementing the topology from Figure 6.3. It consists of two decisioning bolts placed one after the other, where each decisioning bolt is responsible for one of the aggregations.

Dividing the decisioning logic into two distinct bolts instead of just one introduces overhead and potentially decreases performance. By comparing both topologies we can determine if the decrease in performance justifies enabling parallelism.

The cluster setup used for this experiment consists of a single slave node.

6.6 Result Analysis

The results in Figure 6.4 show that the topology that exhibits the solution and enables correct execution under parallelization is, as hypothesized, less efficient than the regular version. A possible reason for the fact that the enabled version displays a higher performance at a single parallelism is due to the fact that the decisioning logic is spread over two bolts, it is actually executed by two threads instead of one thread in the regular topology.

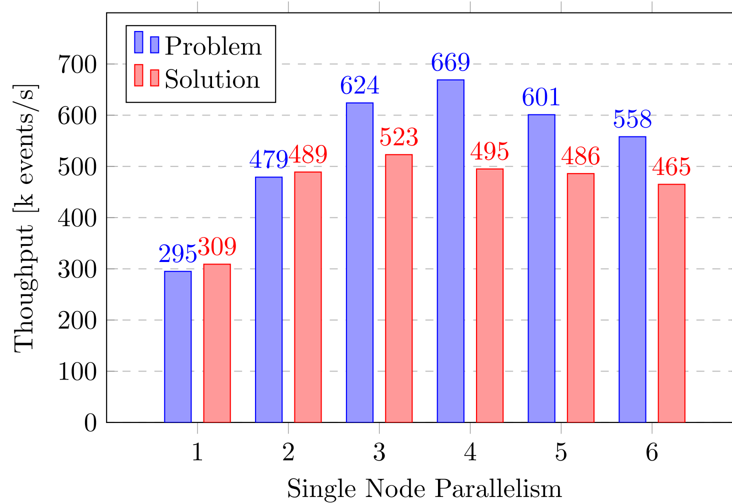


Figure 6.4: The problematic shift partition strategy versus the solution version that enables parallelization. Recall that the problematic strategy gives invalid results when parallelized and is only used as comparison.

At the maximized parallelism of four the enabled version is ∼25% slower than the regular version, this is almost the same as our hypothesized value. Because the solution can benefit from parallelism we achieved an effective total speedup of 1.69. This can be increased further by adding more nodes to the cluster, see Section 5.4.

Comparing the profiling data of both topologies at a parallelism level of three, we see that the combined CPU use of the spout and decisioning bolts is 5% lower for the solution topology. Because there are more components in the solution topology, Storm spawned more *disruptor executors*, seven and ten respectively, which use more CPU time. The disruptor executors are the threads that are responsible for handling the intra node communication.

The additional CPU time used by the disruptor executors in the solution topology is the overhead added by introducing more components to the topology. Together with the overhead within the decisioning bolts mentioned in the hypothesis it results in an overall lower throughput than the problematic topology.

6.7 Threats to Validity

A strategy that changes the partitioning attribute aggregates different higher level attributes from the event stream. One could argue that each aggregation to a higher level attribute justifies the creation of a distinct strategy to extract this knowledge exclusively. Resulting in smaller event strategies that

can be combined, and removing the need for the solution proposed in this chapter because a single strategy does not aggregate different attributes any more.

With the example use case introduced in this chapter, this would result in one event strategy that aggregates the GitHub user data, and a different event strategy that aggregates over GitHub repositories. Both strategies would be combined by a final strategy to give in the desired behaviour, resulting in three different topologies.

This would also allow the low level strategies, that perform a general aggregation over the GitHub users for example, to be reused by different higher level strategies. Resulting in a strategy hierarchy where data streams are reused.

6.8 Conclusion

From the results we can conclude that our hypothesis holds. The overhead introduced by splitting the decisioning strategy over two distinct bolts reduces overall performance, but by enabling parallelism the total achievable performance is increased.

6.9 Related Work

The Next CEP system developed by Schultz-Møller et al. [SMMP09] is a distributed automaton based CEP engine. Event queries are written in a high level language and are rewritten to more efficient forms by changing operator order or composition before being translated to event automata. Operators are distributed and can be reused by multiple queries, the layout is chosen using a greedy algorithm with a cost model. Rewriting patterns to achieve an optimization goal, e.g. CPU efficiency, is possible because of the provable theorems associated with the language operators.

For example, the union operator is commutative and associative and the following equivalences hold: $E1|E2 = E2|E1$ and $E1|(E2|E3) = (E1|E2)|E3$ [SMMP09]. Depending on the cost model, rewriting one form for the other can be more efficient.

The greedy algorithm and cost model can potentially be used to split complex queries into reusable sub-queries.

Chapter 7

Pipelined Parallelism

To overcome a limitation discussed in Section 5.1 that makes it impossible for particular event strategies to benefit from horizontal scaling through partitioned parallelism, pipelined parallelism can be used. With pipelined parallelism the event strategy is divided into constituent parts that are placed one after the other. Theoretically, each part can be executed in parallel, parallelizing the event strategy.

However, pipelined parallelism has a few disadvantages compared to partitioned parallelism [DG92], namely:

- The pipeline is only as long as its constituent parts – operators in the event strategy – which potentially limits the maximum parallelizability.
- Operators only execute in parallel when they both have inputs to consume. If an operator (e.g. the tumbling window) first consumes a number of inputs before emitting their output they *hold the line* and potentially reduce the benefits of parallelism.
- Not all operators have the same footprint, which leads to *skew*. Pipelining a cheap and expensive operator will yield very limited benefits as the expensive operator will still be the bottleneck. The pipeline is only as fast as its slowest component.

Considering these disadvantages, pipelined parallelism can nevertheless enable event strategies to benefit from horizontal scaling that otherwise could not.

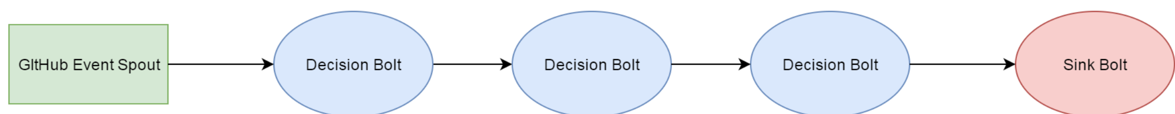


Figure 7.1: Storm topology with an event strategy pipeline of size three.

7.1 Problem

The problem, which was already briefly touched upon in Section 5.1, is that certain event strategies cannot be partitioned while keeping the results correct. The first case is with *shifting partition* and was handled in Chapter 6. Which was actually a form of pipelining that enabled partitioned parallelism.

The second case is where some global knowledge, or knowledge from a different partition, is needed to satisfy the event strategy. The problem is the same as elaborated in Section 6.2. When partitioning on an attribute, other events that the strategy depends upon can end up in different instances of the engine, invalidating the results.

7.2 Solution

The solution is similar to that of Chapter 6, but unlike that solution, the topology can be broken down into constituent parts all the way to event strategy operator level. Each constituent part can potentially run in parallel – as long there are inputs to consume.

This form of horizontal scaling effectively cuts the event strategy in smaller pieces that are attached one after the other using decisioning bolts, see Figure 7.2. Potentially this could enable some parts of the topology to become partition parallelizable. For example when the filter cannot be partitioned, because of the problem stated in the preceding section, but the succeeding aggregate operator can be partitioned.

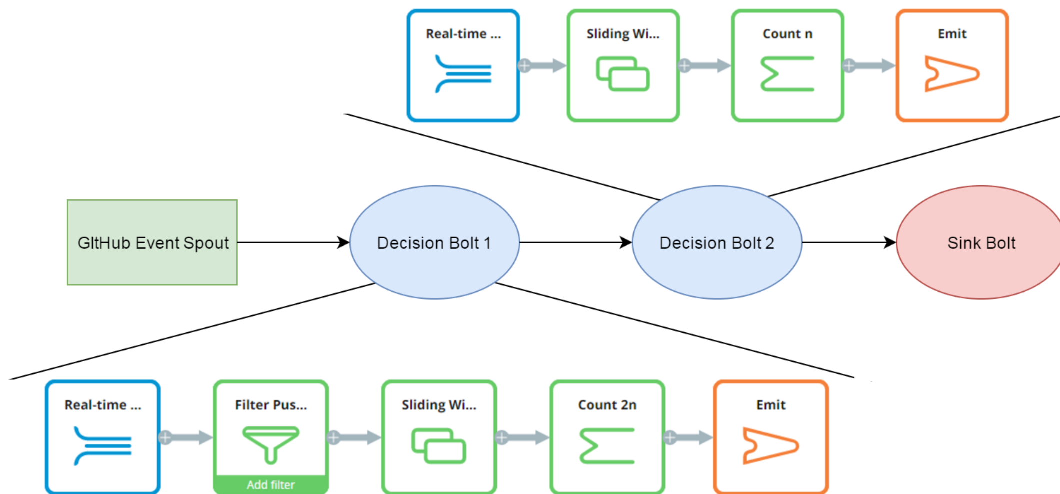


Figure 7.2: Flattened trend strategy (Figure 8.2) implemented in a pipelined topology of size 2.

7.3 Hypothesis

By pipelining an event strategy, multiple decisioning bolts potentially execute the strategy in parallel. Because of internal buffering between workers and executors, as long as the buffers are filled – e.g. the bolts have input to consume – they can be executed in parallel. Still, a pipeline is only as fast as the slowest component.

However, when pipelining identical components, we hypothesize to see the same speedup achieved by partitioned parallelization. Partitioned parallelization of the baseline topology – which is described in the next section – resulted in $sFrac = 0.26$, or in other words $f = 0.74$. Using Amdahl's law this results in the following speedup table:

PARALLELIZATION	1	2	3	4
Speedup	1	1.59	1.97	2.25

7.4 Method

To test our hypothesis, which depends upon pipelined components of the same complexity, we use an event strategy that concatenates four windowed aggregates. Equal complexity is achieved by using windows and aggregators with the same properties – a sliding window of 100 events large and a count aggregate – four times in a row. Each aggregate adds the count as a new attribute to the event.

As baseline we use a single decisioning bolt where all four aggregates are incorporated in a single event strategy. Then we use two different pipeline sizes, namely two and four. The pipeline of size

two divides the event strategy into two distinct decisioning bolts that both contain two aggregations. Likewise, the pipeline of size four implements the event strategy in four distinct decisioning bolts that each contain a single aggregation.

The baseline is also parallelized using partitioning as per Chapter 5. This is used to compare the speedup achieved by partitioning and pipelining, but also to determine the serial fraction of the topology.

All topologies are executed on a cluster consisting of a single slave node.

7.5 Result Analysis

The results show that the hypothesis holds for a pipeline of size two. Both the partitioned parallelization of level two and the pipeline of size two achieve a speedup of ∼1.60.

However, the hypothesis does not hold when the pipeline size is increased. At a partitioned parallelism level of four, the throughput increases as expected and in line with results from Chapter 5, but a pipeline of size four does not increase the throughput.

Using a profiler we compared both topologies. The combined CPU time used by the decisioning executors is 69% and 79% for the partitioned and pipelined topologies respectively. This means that the partitioned topology is more efficient, leaving more CPU time available to the spout – 16% compared to 10% – which spends less time *waiting*.

This shows that the pipelined parallelism incurs a lot more overhead on the decisioning executors. This has likely to do with the fact that for each additional pipelined component the event has to be transformed and transported. With partitioned parallelism there are the same number of decisioning bolts spawned, but partitioning ensures that each event only visits one of them. In the pipelined scenario each event has to traverse through all components of the pipeline, where each event has to be transformed and transferred from one component to the next.

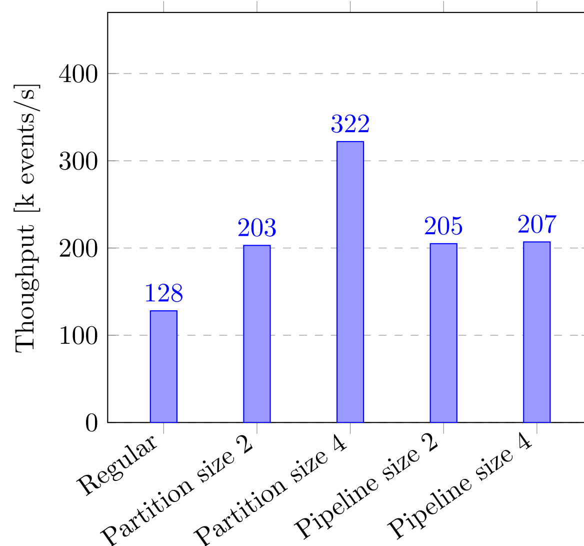


Figure 7.3: Graph comparing the pipeline efficiency.

7.6 Threats to Validity

The effect of a pipelined architecture can be highly influenced by the buffering regime. Since a Storm topology is effectively a pipeline of components that are possibly parallelized through partitioning, Storm already uses optimized buffers at critical stages, see Section 3.1.2.

The buffer sizes are configurable and we tried different settings to increase the performance of the pipeline. However, we are confident that the default buffer configuration is optimal for the topologies

presented in this chapter. Other configurations, both higher and lower than the default, resulted in either lower or roughly equal but unstable performance.

7.7 Conclusion

In conclusion we can state that our main hypothesis, which stated that the performance of pipelined parallelism should be comparable to partitioned parallelism, is not correct. Because of the event transfer between components, and possibly the reliance on buffers, a pipelined topology incurs much more overhead than partitioned parallelism.

However, in scenarios where partitioned parallelism is not possible, a pipeline can still increase the overall performance. We nevertheless achieved a speedup of 1.60 compared to the regular sequential implementation.

Chapter 8

Flatten Split Join

For event strategies that are modeled by a DAG that connects the operators, it often feels natural to split the stream and join them again. As is the case in the GitHub *PushEvent* trend strategy, Figure 8.1, because conceptually two concepts are being combined or compared. However, many such strategies can also be expressed by a flat strategy, which may be faster.

Because sliding windows collect specific attributes of events that fall within its constraints and immediately emit them to the next shape, the stream does not have to be split in order to aggregate results for two different window times. They can also be placed one after the other with exactly the same semantics, see Figure 8.2.

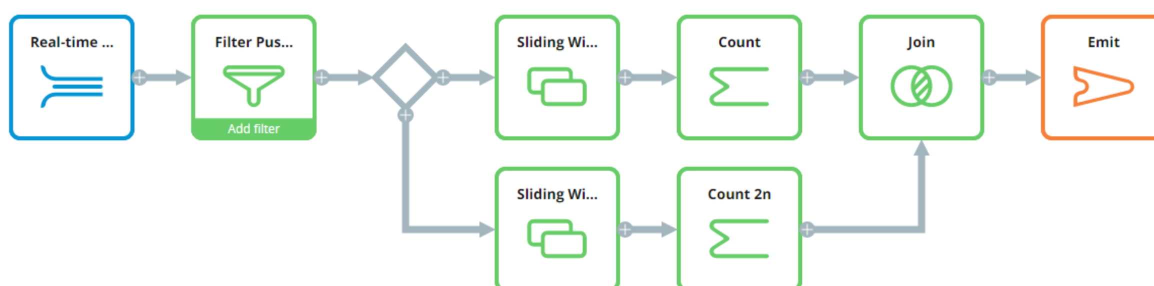


Figure 8.1: GitHub *PushEvent* trend strategy modeled with a split and join operator.

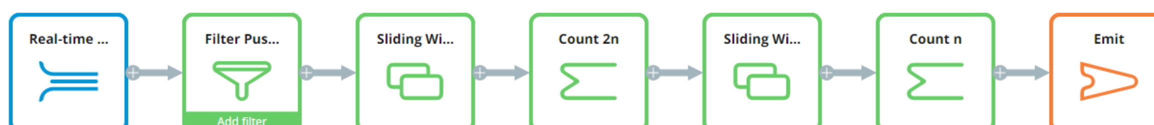


Figure 8.2: GitHub *PushEvent* flattened trend strategy. Semantically this is the same as the figure above, but the intent of this strategy is arguably harder to understand without splitting the stream.

8.1 Limitations

There are however scenario's in which an event strategy that uses a split cannot be flattened because this would change the semantics. If the split streams deviate from each other by changing the events that are respectively emitted, they cannot be placed one after the other on a single stream without changing the semantics and thus the event strategy results. The first part of the stream can potentially remove events on which logic in the second part depends. In the case of the decisioning engine this can happen in two cases:

Filter: Stream A applies a filter that stream B does not, or applies it differently.

Tumbling aggregate: A tumbling window aggregates all events until the window fills up. At which point all events are released at once and the window *tumbles* to the next interval.

Applied to an aggregate this results in special behaviour where the aggregated events are reduced to a single event. A tumbling aggregate of size 10 will thus decrease the number of events in the stream tenfold. Furthermore, if the window never satisfies, the events within the window are never emitted.

Given the definitions for the *FilterOp* function:

$$FilterOp(o) = \begin{cases} 1 & \text{If } o \text{ is a filter operator} \\ 0 & \text{Otherwise} \end{cases} \tag{8.1}$$

The *TumblingOp* function:

$$TumblingOp(o) = \begin{cases} 1 & \text{If } o \text{ is a tumbling window operator} \\ 0 & \text{Otherwise} \end{cases} \tag{8.2}$$

We can retrieve the stream modifying operators from an operator chain:

$$StreamModifyingOps(chain) = \{o \mid o \in chain \wedge (FilterOp\ o \vee TumblingOp\ o)\} \tag{8.3}$$

And finally the definitions for *Split* and *Join* functions:

```
1 Split :: (OpChain -> e) -> (OpChain -> e) -> e -> (e, e)
2 Split chainA chainB event = (chainA event, chainB event)
3
4 Join :: (e, e) -> e
5 Join (event1, event2) = event1 ∪ event2
```

We can now create an equivalence model. Given two operator chains, A and B, that originate from a split operator and are joined by a join operator. If the filter and tumbling window operators that are present in both chains are equivalent, then the split-join combination on the chains is equivalent to a simple concatenation of the chains:

$$\begin{aligned} &StreamModifyingOps\ chainA \ \equiv\ StreamModifyingOps\ chainB \\ &\quad\implies \\ &Split(chainA, chainB) \rightarrow Join(chainA, chainB) \ \equiv\ chainA \rightarrow chainB \end{aligned} \tag{8.4}$$

8.2 Hypothesis

A minimal event strategy that uses a split join operator, as depicted in Figure 8.3, already imposes a narrow bottleneck for the topology.

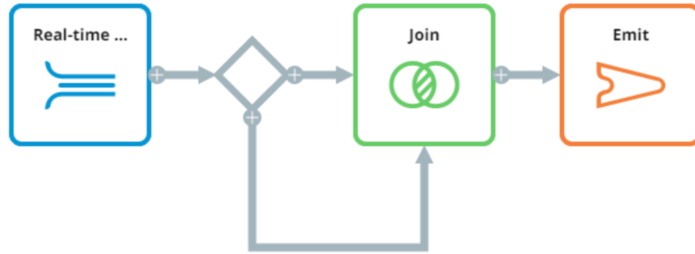


Figure 8.3: Minimal split and join event strategy.

The throughput of an empty event strategy, a windowed aggregation and the split-join operator combination are reiterated in Figure 8.4. This clearly shows how expensive the split and join combination is compared to the aggregation operator. See Chapter 4 for more details on the cost of different operators.

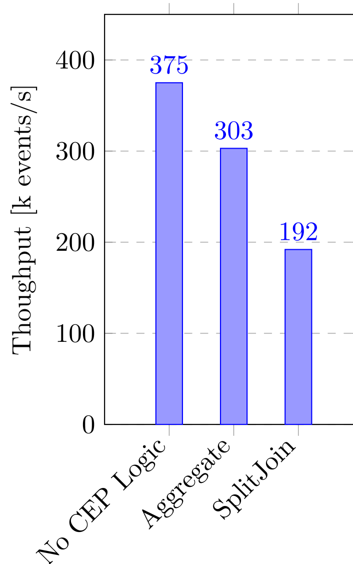


Figure 8.4: Graph comparing the aggregate and split join operators.

Upon profiling the trending strategy (Figure 8.1) we observed that the decisioning bolt spends a total of 6% of its CPU time duplicating the events, and another 16% joining the streams back together. When the strategy is flattened we remove both the split and join operator, without adding any new logic. Using the derived f value of 0.85 from Chapter 5 and $S = 1.22$ derived by profiling gives $speedup = 1.18$.

8.3 Method

Two different topologies will be compared with each other. The regular trending strategy modeled with the split and join operators that is used in Chapter 5 and repeated in Figure 8.1, and a semantically equivalent flattened version that does not use the split or join operators that is depicted in Figure 8.2

The cluster architecture for this experiment consists of a single slave node. The benchmarks will be repeated for different levels of parallelism to compare how the performance of both strategies is affected by this.

8.4 Result Analysis

The results show that the trending event strategy modeled with a flat strategy achieves a speedup of 1.51 at a parallelism of one. This is much larger than our hypothesized value of 1.18.

Using the profiler we see that the CPU time used by the decisioning bolt increased by 6% to a total of 53%, instead of decreasing the CPU time because computations are removed. This shows that the effects of changing the efficiency of a single component within the topology has far reaching consequences that cannot be accurately predicted using a profiler.

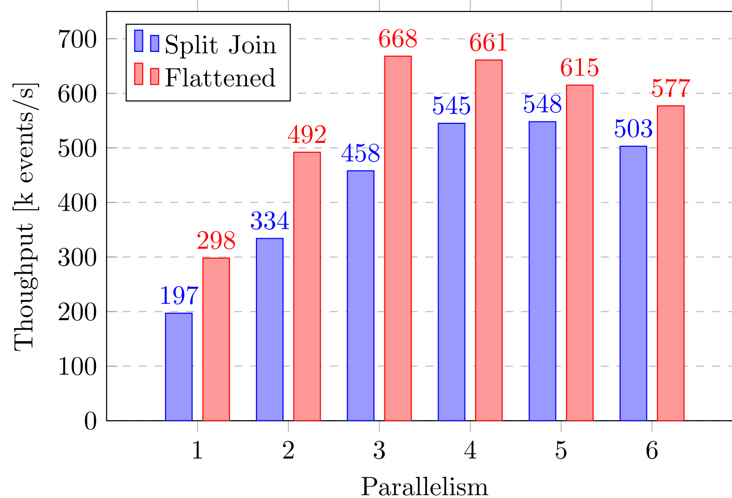


Figure 8.5: Trend strategy modeled with a split and join versus a flat strategy.

8.5 Threats to Validity

Many of the same threats to validity hold here as mentioned in Chapter 5. Profilers are not an accurate means of predicting the gained benefit of flattening the strategy. It can however be used to determine if splitting and joining significantly reduces performance. If the split-join operator is not the most significant bottleneck but I/O operations are, then large results are not to be expected. It could however never hurt to remove operators if the semantics of the event strategy stay intact.

8.6 Conclusion

We can conclude that our hypothesis is indeed correct regarding increased throughput, but we were not able to accurately predict the actual speedup.

In conclusion we can state that flattening the event strategy increases the performance. As long as the semantic correctness can be assured, it is beneficial to always flatten the event strategy. Because the amount of operators in the event strategy is reduced while ensuring semantic correctness, the worst result is that it does not impact performance. This could for instance happen when not the available CPU cycles is the bottleneck of the topology but I/O operations.

8.7 Related Work

In their research towards highly parallel database systems [SD89][DG92][PD00][SAD+10], DeWitt et al. introduced and evaluated parallel join algorithms. They implemented parallel versions of sequential join algorithms such as *Simple hash* and *Hybrid hash-join* [DKO+84] and achieved noteworthy performance increases (at the cost of memory usage). Implementing a parallel join algorithm might significantly increase the throughput for event strategies that cannot be flattened.

Chapter 9

Pull Operator Up

As was discovered by running the initial baselines, see Chapter 4, there is an inherent cost of transferring events between the Storm topology components and inserting them in the event strategy. For many strategies the efficiency of the topology as a whole can be improved by pulling some of the knowledge about the strategy up to an earlier component.

Pulling up one or multiple operators could have either, or both, of the following goals:

Reduce data transfer: This can be achieved by pulling a filter, or tumbling widow, up to an earlier component in the topology, for instance to the source spout. Less events have to be transformed and transported, freeing up resources for other activities.

Reduce memory pressure: By pulling up parts of the event strategy that are memory intensive, such as large sliding windows, the load can be shared by a larger part of the cluster without (further) partitioning the stream.

Both approaches trade added complexity and reduced modularity for the aspired goal.

9.1 Hypothesis

There is an inherent cost associated with transferring events between components in a Storm topology. Especially with acking enabled, see Section 3.1.2. The same holds true for inserting events into the decisioning engine, in part because the receiving Storm data tuple is converted to an event object.

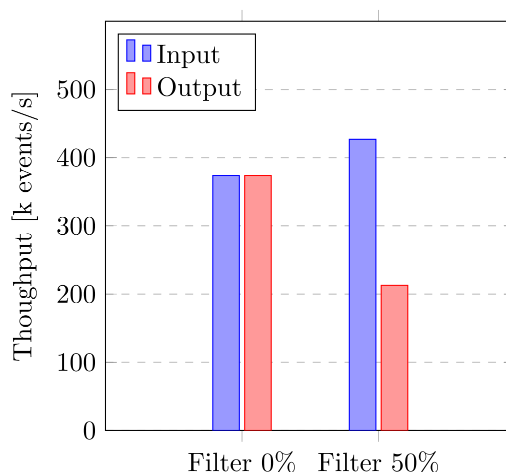


Figure 9.1: Graph comparing event transfer load.

Figure 9.1 exemplifies this point by depicting results for two different filter policies (see Chapter 4 for more details). The strategy that filters half of the events decreases the amount of events that are

transferred by reducing the output of the decisioning bolt, and in the process frees up some resources. This increases the number of events processed per second by the event strategy, depicted in the figure by the *Input*.

By profiling the application we discovered that the spout thread dedicates 45% of its CPU time to transferring events and the decision bolt thread spends 7% converting the data tuples to event objects. Depending on the event strategy, much of this processing is wasted because the event strategy simply drops these events.

By pulling knowledge about the event strategy up, we can drop these events at an earlier stage. This reduces the number of events that have to be transferred, transformed and inserted into the event strategy, only to be dropped again by the filter operator. We hypothesize that this increases the efficiency of the topology and thus the total throughput. The necessary bandwidth will also decrease. The filter reduces the number of events by 50.5%, and because the events are transferred two times pulling up the filter before the first transfer instead of the second reduces total transfers by ∼33%.

9.2 Method

We will focus on reducing the data transfer by pulling up the type filter to the source spout. The event strategy used is again the trending scenario, Figure 5.2, but instead of the event strategy filtering on the *PushEvent* type, this knowledge is pulled up to the source spout. The spout will filter and only send events of the desired type to the decisioning bolt as depicted in Figure 9.2. This reduces the amount of events that are transferred to the decisioning bolt by 50.5%.

The cluster architecture consists of a single slave node, each benchmark will be repeated for different levels of parallelism.

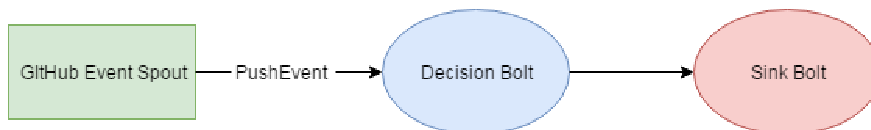


Figure 9.2: Storm topology for the push event trending scenario where the knowledge of the event filter is pulled up to te spout. Only events of the type *PushEvent* are transferred to the decisioning bolt.

9.3 Result Analysis

The data shows that pulling up the filter has an immediate effect. At a parallelism of one, the total speedup achieved is 1.37 compared to the regular topology. Also because the number of events emitted by the spout are decreased by 50.5%, the total events transferred, and bandwidth used, by the topology is reduced by around 33%.

However, the most interesting result is the maximum throughput when the parallelism is increased. The regular topology plateaus, and even degrades, from a parallelism of three. The optimized topology increases in efficiency up to a parallelism of four. Plateauing at a total throughput that is ∼55% higher than the regular topology.

This can again be explained by Amdahl's law: the data transfer at the spout does not benefit from parallelization because it is not parallelized. So by reducing the non-parallelized portion of the program we have effectively increased the parallelized proportion of the program. Therefore the diminishing returns are decreased.

This is supported by our profiling data. The decisioning bolt consumes 29% more CPU time than the spout during execution of the regular topology. Whereas the decisioning bolt consumes 68% more than the spout during execution of the optimized topology. This means that freeing up resources at the spout increased the parallelizable portion of the topology, namely execution of the decisioning bolt, and thus the total efficiency.

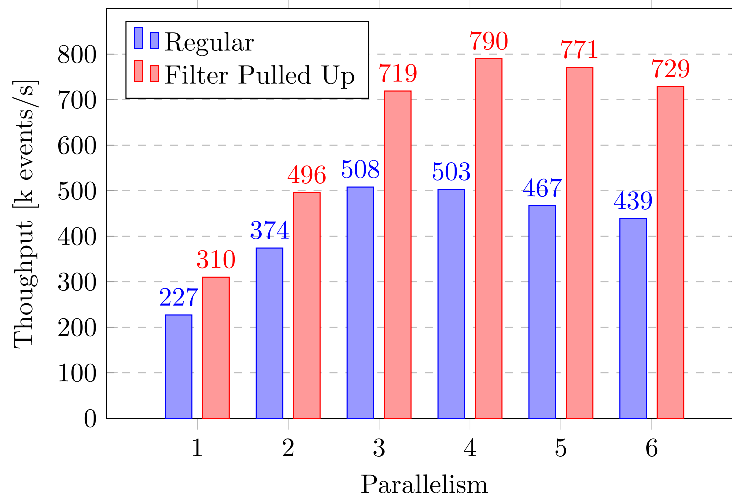


Figure 9.3: Regular trend strategy versus a strategy that filters at the spout.

9.4 Threats to Validity

Depending on the CEP engine used, removing events from the stream before they enter the engine might be undesirable and produce side effects. Because events that have happened are not *seen* by the engine. This could influence logging, statistics gathering or performing of diagnostics. In this case logic has to be pulled up as well to the level of the DSCP, entangling their dependencies which might not be desirable from a maintenance point of view.

9.5 Conclusion

From the results we can conclude that our hypothesis is correct. Pulling up knowledge from the event strategy to reduce data transfer can indeed increase the total performance of the topology.

Note however the trade-off that was already briefly mentioned at the onset of this chapter: pulling up knowledge from the event strategy to other components of the topology significantly increases the complexity and decreases the modularity of the code involved. When the business logic changes in this new scenario we no longer only have to change the event strategy. Because knowledge of the event strategy has been pulled up to other components, they have to change as well. This is an important consideration when moving to implement such a scenario. Our findings indicate however that the gains can be significant and well worth the consideration.

9.6 Related Work

Diao at al. developed the SASE and SASE+ composition-operator-based CEP engine [WDR06] [DIG07] [ADGI08]. The SASE engine uses a query plan based approach to detect complex events. In [WDR06] optimizations are presented that involve *pushing predicates down* to be evaluated earlier in the query plan. The goal is to reduce the size of the intermediate result sets. To this end, both predicates and window operators are pushed down.

The main difference with their approach is that it is applied to the internals of a centralized system with the sole purpose of reducing in-memory intermediate result sets. We pull operators, and thus knowledge about the event strategy, up from the CEP engine to a higher level, namely the DSCP. This approach is more far-reaching in terms of its consequences, but has the benefit of also reducing the required bandwidth.

Part IV

Evaluation

Chapter 10

Putting It Together

To evaluate the compounding effect of the optimizations presented in this thesis we combine them into a single topology. Taking the running example, the partitionable trending strategy introduced in Section 3.3, we can choose between partitioned or pipelined parallelism. Because our experiments have shown that partitioned parallelism is preferable in terms of achievable throughput – with an achieved throughput of 538k versus 207k e/s – we can apply the following performance increasing optimizations:

1. Partitioned parallelism on a multi machine cluster (Chapter 5).
2. Flatten split-join operator (Chapter 8).
3. Pull operator up (Chapter 9).

We will compare the compounded optimizations with the regular sequential execution of the event strategy by the decisioning engine both with and without Storm. So far we have only examined the performance of the engine while implemented in a Storm topology, which introduces overhead that is only justified if it is possible to benefit from the topology layout or parallelism.

10.1 Comparison

Looking at the graph in Figure 10.1 we see that the *regular* implementation in Storm is around 10% slower than without using Storm. This exemplifies the need to use the capabilities provided by Storm to justify its use.

Reiterating the speedup achieved through partitioned parallelism in Chapter 5, we can see the compounding speedup. Using partitioned parallelism together with the other optimizations that we discovered in this thesis, we achieve a speedup of 4.86 compared to running the event strategy without Storm. This is while using a cluster of two slave nodes and a parallelism level of four. Note that this is an additional speedup of 1.35 on top of partitioned parallelism, which can be attributed to the other optimizations discovered instead of using horizontal scaling.

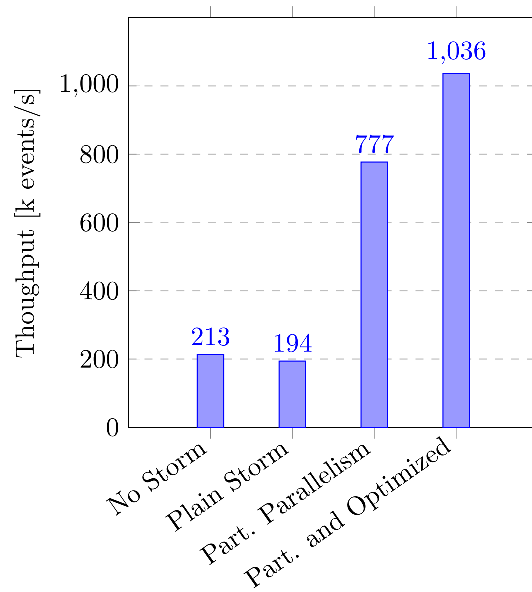


Figure 10.1: Graph comparing the throughput of the running example strategy without storm, regularly implemented in storm, and with optimizations introduced in this thesis.

10.2 Conclusion

We have effectively increased the throughput of the running example trending strategy by more than 480%. This is achieved by flattening the split-join operator combinations. Pulling up the event filter to the spout and by applying partitioned parallelism on a two node cluster.

Chapter 11

Evaluation

11.1 Research Goals

In Chapter 1 we stated our research goals and accompanying sub-questions. In the preceding chapters we achieved these goals to the degree permittable within the scope and time limitations of our project. Below we will revisit each goal and question and summarize our findings.

Achieve higher throughput by utilizing horizontal scalability for the Pegasystems CEP engine.

By utilizing a cluster of two nodes we increased throughput significantly. We also introduced the means by which an upper limit of the expected results can be determined.

- **Under what conditions and scenarios is horizontal scalability possible for a stateful CEP engine?** We defined the conditions under which two types of horizontal scalability can be enabled, namely partitioned and pipelined parallelism. We showed that the former is preferable and that it can be applied while ensuring that each instance of the engine receives the events that its event strategy depends upon.

  In case of the Pega decisioning engine this can, in all but one case, be achieved by simply partitioning the event stream to multiple instances of the engine. We also provide solutions for the counter example.

- **What benefits can be expected when horizontally scaling a CEP engine?** Horizontal scaling is used to increase the available resources for the event strategy, either CPU time or memory. The former increases total throughput for CPU intensive strategies whereas the latter can increase throughput and/or maximum window sizes for memory intensive strategies.

  The effectiveness of horizontal scaling largely depends upon the event strategy used and the CEP engine in question. In our scenario we were able to increase the throughput of the event strategy by several orders by using partitioned parallelism. Pipelined parallelism yields smaller results because it adds more overhead when using Storm.

  The upper limit of the expected performance increase can be predicted using Amdahl's law, but the actual increase is not predictable. There are too many variables and inter-connected moving parts that influence each other. For example when increasing the efficiency of the decisioning bolt it requests events at a faster pace, increasing transfer rates and bandwidth, but also increasing the load on the spout which again affects the decisioning bolt. The high watermark can also be hit earlier or more often.

  Experimentation is needed to figure out the actual benefits of partitioned or pipelined parallelism on a per case basis.

Increase the Pegasystems CEP engine capabilities by rewriting the topology and/or event strategy.

In addition to applying horizontal scaling to increase performance, we discovered two other optimizations that can be applied to either the distributed execution topology or to the event strategy. This further increased the maximum throughput achieved for the running example. Additional optimizations are proposed in the future work section.

- **What are the biggest bottlenecks introduced by the distributed topology or the CEP event strategy?** The biggest bottlenecks are introduced by expensive operators within the event strategy, namely the windowed aggregators and the split-join combination.

  A different bottleneck is introduced by the Storm architecture when transforming and transferring huge numbers of events. The same holds when transforming data to be inserted into the Pega decisioning engine.

- **How can these bottlenecks be optimized?** For the split-join operator combination we provide a formal definition of the scenarios in which this operator combination can be omitted without changing the semantics of the event strategy. Entirely removing the computations associated with the operators.

  The event data transformation and transportation overhead can be decreased by pulling knowledge about the event strategy up to earlier components. Discarding events as early as possible to decrease the data transfer.

11.2 Future Work

Many analysis and experiments have been performed during this project. In our experiments we scaled CEP processing throughput through horizontally scalable parallelism and other optimizations. However, there is still more work that can be done towards this effort. In the succeeding sections we will touch upon different approaches to extend upon our work.

11.2.1 Additional Optimizations

There are more optimizations that could be applied than what we covered in this thesis. Below we will list the options that we considered, but that we could not address because of scope or time limitations.

- **Operator reordering:** Operators in an event strategy may be sub-optimal ordered [HSS+14]. A scenario can be imagined where an expensive operator is succeeded by an operator that reduces the number of events in the stream but does not rely on the previous operator. In this case the event strategy can be rewritten to a more efficient form by first performing the operators that reduce the number of events in the stream.

- **Pull tumbling window up:** In the same manner as pulling the filter up as in Chapter 9, the tumbling window can be pulled up as well. The trade-off involved is more pronounced as the tumbling window is a more complicated operator and it would need the implementation of the CEP engine at the spout level. However, the gains are also more significant: *i.* The tumbling window reduces the number of events in the stream by the factor of its window size; *ii.* Since the tumbling window is a memory intensive operator it can also help distribute the load to multiple nodes.

- **Parallel joining:** As mentioned in the related work section of Chapter 8, DeWitt et al. [PD00] describe in their research several algorithms to parallelize join operations. Since our optimization to flatten a split-join operator combination can only be applied in certain scenarios, a parallel join operation could be used to optimize joining. This can yield additional benefits on top of the parallelization achieved through Storm because the parallel join algorithms have different trade-offs than Storm.

The parallel join algorithms described by DeWitt et al. increase the performance of CPU intensive join operations at the cost of increased memory usage. This is different from parallelization achieved through Storm that increases available resources for increased overhead in the form of data transformations and bandwidth. Furthermore, increasing the event strategy efficiency can also benefit parallelization as seen in Chapter 8.

11.2.2 Increase Cluster Size

Because of limited resources available we were not able to experiment with a cluster larger than two comparable nodes. It would be interesting to see how generalizable our results are when scaling to ten or fifteen nodes. When adding more nodes diminishing returns are expected, in the same manner as parallelizing on a single machine, and additional bottlenecks might be discovered.

11.2.3 Automate Parameter Optimization

When using Storm, especially to perform CEP, many parameters have to be configured to achieve an optimization goal. Generally, this is throughput or latency. In this thesis we disabled *acking* and as a consequence the *maxspoutpending* option which can be used to throttle the topology to prevent failures as a result of buffer overflows. One of the main reasons for disabling these features is because they have to be optimized on a case by case basis per topology to find the best configuration. Other options such as parallelization and worker count have this caveat as well.

We believe this process can largely be automated by a tool that, given a topology or an event strategy, automatically discovers the optimal parameter options by trial using greedy algorithms [SMMP09]. For organizations such as Pegasystems such a tool is paramount to being able to implement the discoveries presented in this thesis. Because the clients define the event strategy, the entire process from definition to deployment must be automatable. This includes rewriting the event strategy to include optimizations, enabling parallelization and discover the optimal deployment parameter settings.

11.3 Recommendations

For my recommendations to Pegasystems it is important to see that the optimizations presented in this thesis can be grouped in four categories. Below I will touch upon each category, ordered from relatively easy to realize, with small consequences to the system as a whole, to changes with large implications:

1. **Event strategy rewriting.** This includes changes such as flattening the split-join operator combination as described in Chapter 8 and operator reordering described in the preceding section. The rewriting rules are relatively easy to implement and the consequences are contained locally within the decisioning engine, unit tests can cover the added code and no previous functionality will be effected.

2. **Local parallelism.** This would entail multi-threading the decisioning engine. This can be realized in the same manner as Storm but simpler because the parallelization is not realized by spawning multiple instances of the engine but can happen locally. This change has more implications because the development effort is greater and introduces the potential for concurrency bugs. Our research shows the gains can be significant, however, this can be limited when event strategies regularly accesses shared resources.

3. **Distributed parallelism.** Introducing distributed parallelism can increase throughput of CPU intensive event strategies, but also memory intensive strategies. For maximum gains local parallelism should be applied as well. An open-source, off the shelf, or in-house developed DSCP can be used to this end, but this has big implications to the current event strategy data flow. Because execution must be delegated to a separate cluster, which results feed back into the flow, this adds a complex layer to the architecture.

4. **Distributed topology rewriting.** This includes changes that have to pull up knowledge of the particular event strategies to the distributed topology level. Such as pulling up operators to earlier components or nodes, as described in Chapter 9, but potentially also reusing and combining other topologies and results. This has high implications at an architectural level because the DSCP must have very detailed knowledge about the CEP engine. This breaks the separation of concerns and decreases modularity since both engines become intertwined.

Much more can be said about the trade-offs and implications associated with the changes introduced by each category. However, because each category is dependent on, or more valuable, when the previous is implemented, the recommendation for scaling CEP within Pegasystems becomes quite straightforward.

Event strategy rewriting should be tackled first as it is the low hanging fruit from the list above. Implementing an intermediate language can help in this regard to make it easy to define multiple rewrite rules to flatten the strategy and possibly perform operator reordering and other optimizations.

Multi-threading the application should be evaluated in depth. The gains can be significant but the development effort and associated risks with this change can be large as well. Distributed execution is needed to truly scale the engine, however, consequences of this change have to be evaluated in detail and only be considered if the associated scaling is truly necessary.

Chapter 12

Conclusion

The research goal of this thesis is to scale and optimize the Pegasystems CEP engine through the introduction of a DSCP. In this effort we explored both horizontal scalability and optimizations that attack a specific bottleneck.

Specifically, we defined the conditions under which two forms of horizontal scaling, partitioned and pipelined parallelism, can be used with a stateful CEP engine. We showed that partitioned parallelism is preferable because it introduces less overhead than pipelined parallelism. Partitioning can be used in all but one scenario, and we provide a solution for the counter example where results are invalidated. Using partitioned parallelism we increased throughput by 170% on a single node and by 300% on a two node cluster.

We also investigated the bottlenecks of the system. Both at the level of the DSCP topology and the CEP engine event strategy, on both levels we introduce an optimization to reduce a specific bottleneck:

- At the event strategy level we discovered an invariant where the expensive split-join operator combination can be removed entirely without changing the semantics of the strategy, increasing performance by around 20-50%.
- At the topology level we reduced the events that are transferred by pulling a filter up to the spout, decreasing event transfers by 33% and increasing total throughput by 35-60%.

Using Amdahl's law, and an amended version that incorporates overhead, we use a profiler to predict performance increases. However, because too many variables are in play that directly influence each other, it is not feasible to predict the outcome. Actual experimentation is needed to discover the results and find the best parameters.

By combining the optimizations explored in this thesis we increase throughput by 385% compared to regular execution without the Storm DSCP. This is achieved by combining both discovered optimizations and using partitioned parallelism on a two node cluster. The running example consists of a trend detection event strategy. Similar strategies are often employed by the host organization and its clients and used for internal testing.

Bibliography

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. Cql: A language for continuous queries over streams and relations. In *Database Programming Languages*, pages 1–19. Springer, 2003.

[AC06] Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. *Data & Knowledge Engineering*, 59(1):139–165, 2006.

[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 147–160. ACM, 2008.

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[AXL+15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[BGHJ09] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 3. ACM, 2009.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[CKAK94] Sharma Chakravarthy, Vidhya Krishnaprasad, Eman Anwar, and Seung-Kyum Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, volume 94, pages 606–617, 1994.

[CM09] Gianpaolo Cugola and Alessandro Margara. Raced: an adaptive middleware for complex event detection. In *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware*, page 5. ACM, 2009.

[CM10] Gianpaolo Cugola and Alessandro Margara. Tesla: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 50–61. ACM, 2010.

[CM12] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[CS09] K Chandy and W Schulte. *Event processing: designing IT systems for agile companies.* McGraw-Hill, Inc., 2009.

[DG92] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[DGH+06] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *Advances in Database Technology-EDBT 2006*, pages 627–644. Springer, 2006.

[DGP+07] Alan J Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M White, et al. Cayuga: A general purpose event monitoring system. In *CIDR*, volume 7, pages 412–422, 2007.

[DIG07] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. Sase+: An agile language for kleene closure over event streams. *University of Massachusetts Amherst*, 2007.

[DKO+84] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984.

[DRKK+15] Davide Di Ruscio, Dimitris S Kolovos, Ioannis Korkontzelos, Nicholas Matragkas, and Jurgen J Vinju. Ossmeter: A software measurement platform for automatically analysing open source software projects. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 970–973. ACM, 2015.

[EB09] Michael Eckert and François Bry. Complex event processing (cep). *Informatik-Spektrum*, 32(2):163–167, Springer, 2009.

[EBB+11] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. A cep babelfish: languages for complex event processing and querying surveyed. In *Reasoning in Event-Based Distributed Systems*, pages 47–70. Springer, 2011.

[EHHB14] Ismail El-Helw, Rutger Hofman, and Henri E Bal. Scaling mapreduce vertically and horizontally. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 525–535. IEEE, 2014.

[Foua] Apache Software Foundation. Apache spark. Last accessed: 2016-04-24. URL: http://spark.apache.org/.

[Foub] Apache Software Foundation. Apache storm. Last accessed: 2016-04-24. URL: http://storm.apache.org/.

[Fouc] Apache Software Foundation. Cassandra query language (cql). Last accessed: 2016-04-29. URL: https://cassandra.apache.org/doc/cql3/CQL-2.2.html.

[Git] GitHub. Event API. https://developer.github.com/v3/activity/events/. Last accessed: 2016-05-28.

[GJ91] Narain H Gehani and Hosagrahar Visvesvaraya Jagadish. Ode as an active database: Constraints and triggers. In *VLDB*, volume 91, pages 327–336. Citeseer, 1991.

[GR06] Simon Garfinkel and Beth Rosenberg. *RFID: Applications, security, and privacy*. Pearson Education India, 2006.

[HH05] Lilian Harada and Yuuji Hotta. Order checking in a cpoe using event analyzer. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 549–555. ACM, 2005.

[Hin03] Annika Hinze. Efficient filtering of composite events. In *New Horizons in Information Management*, pages 207–225. Springer, 2003.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41:33–38, 2008.

[HSS+14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[Int13] Intel. Core i7-4700mq, 2013. Last accessed: 2016-06-17. URL: http://ark.intel.com/products/75117/Intel-Core-i7-4700MQ-Processor-6M-Cache-up-to-3_40-GHz.

[Int14] Intel. Xeon 2650v3, 2014. Last accessed: 2016-06-17. URL: http://ark.intel.com/products/81705/Intel-Xeon-Processor-E5-2650-v3-25M-Cache-2_30-GHz.

[Kli14] P Klijn. How accurately do java profilers predict runtime performance bottlenecks? Universiteit van Amsterdam, 2014.

[LS11] David Luckham and W. Roy Schulte. Event processing glossary version 2.0. Technical report, Event Processing Technical Society, 2011. URL: http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf.

[Luc02] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[PD00] Jignesh M Patel and David J DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *Proceedings of the 8th ACM international symposium on Advances in geographic information systems*, pages 54–61. ACM, 2000.

[Qui03] MJ Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003. University of Notre Dame.

[RCCB16] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. Automatic microbenchmark generation to prevent dead code elimination and constant folding. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, 2016.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[SAD+10] Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[SD89] Donovan A Schneider and David J DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *ACM SIGMOD Record*, volume 18, pages 110–121. ACM, 1989.

[SMMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, page 4. ACM, 2009.

[Teca] Esper Tech. Esper. Last accessed: 2016-04-24. URL: http://www.espertech.com/esper/.

[Tecb] Esper Tech. Event processing language. Last accessed: 2016-04-29. URL: http://www.espertech.com/esper/release-5.2.0/esper-reference/html/epl_clauses.html.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2006.

[WL05] Fusheng Wang and Peiya Liu. Temporal management of rfid data. In *Proceedings of the 31st international conference on Very large data bases*, pages 1128–1139. VLDB Endowment, 2005.

[ZU99] Detlef Zimmer and Rainer Unland. On the semantics of complex events in active database management systems. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 392–399. IEEE, 1999.