Imperial College of Science, Technology and Medicine
Department of Computing

Distributing Complex Event Detection

Kristian A. Nagy

MEng Individual project report
Supervised by Dr. Peter Pietzuch
June 2012

Abstract

In recent years there is a huge increase in real-time data, which cannot be stored in its whole entirety. Nevertheless, its timely processing in a form of events exhibits enormous potential for business intelligence. To allow for inferring high-level informations from vast amounts of continuously arriving data, complex event detection systems, capable of discovering user-defined event patterns, were developed.

The latest developments in these systems includes distribution of event detection by query partitioning and their execution on multiple nodes. However, some applications, such as stock monitoring or mobile fraud detection, need to deal with extreme input event rates, which may be beyond the capabilities of the existing solutions.

In this report we present the Step complex event detection system, which goes further than distributing queries and achieves better scalability by parallelising event detection, and also higher efficiency through the use of many optimizations. Event queries specified in a high-level language are compiled into data-flow graphs, parts of which may be replicated, and run on a novel distributed computing platform Storm. The degrees of parallelism, as well as other system parameters, are estimated using a performance model. Our evaluation shows that event detection is fast and for some event patterns it scales linearly, however for others logarithmically.

Acknowledgements

First and foremost, I want to thank my parents Bozena and Imrich for their financial support, motivation and encouragement that kept me focused for the duration of my studies.

I would also like to express my gratitude to my supervisor Dr. Peter Pietzuch for his continuous guidance, patience, and the time that he dedicated to this project. Have it not been for his prompt replies and suggestions, the project would not advance.

Many thanks go also to Clemens Lutz, for his insightful discussions about the Storm framework, and to the residents of the William Temple House, who provided me with a pleasant environment during the last two years of my studies.

Furthermore, I want to thank my close friends Grace Chen and Samira Barzin, who took on the difficult task of proofreading several chapters of this report.

Contents

Chapter 1

Introduction

1.1 Complex event detection

In recent years there has been a huge increase in the data produced on the Internet, effectively doubling in size every year [25]. In many cases, the data is continuously produced by software applications in quantities that are not examinable manually. Only a small percentage of these is of interest. However, the vast amounts of data exhibit an enormous potential for business intelligence, possibly generating more revenues and better fitting customer needs. Examples of this in practice include performing click-stream analytics, inferring information from market stock data, detecting financial frauds, managing networks, and monitoring pervasive environments, telecommunication systems or electricity grids.

Historically, the common approach for such applications was to store the generated data in databases or logs, and process it afterwards in batch processing jobs, leveraging distributed frameworks such as Hadoop. However, it is becoming increasingly inefficient to store all the data in its whole entirety. Many businesses also require answers as soon as the data becomes available. Furthermore, the businesses are not primarily interested in raw data, but rather in the high-level intelligence that can be extracted from it. As a response, systems were developed that can filter, aggregate and correlate data, and notify interested parties about its results, abnormalities, or interesting facts.

The latest advance in such systems is the development of high performance *complex event processing* (CEP) engines (a summary in [15]) that are capable of detecting patterns of activity from continuously arriving data. As illustrated in figure 1.1, CEP systems receive continuous streams of events from multiple data sources over underlying network, they discover patterns of interest among the events, and notify the users.

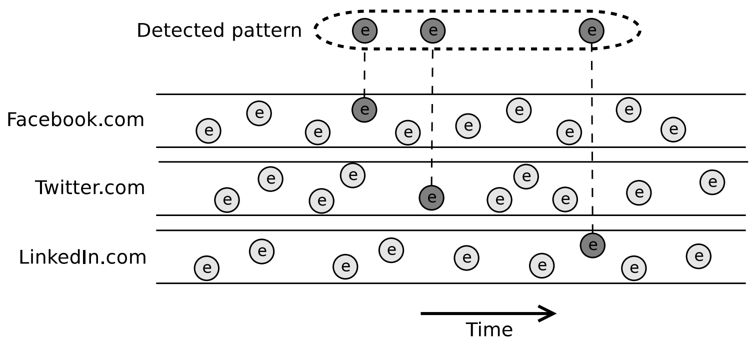


Figure 1.1: Example of event pattern detection from three different streams.

An applicability of such systems is for example fraud detection, often performed by financial institutions [32] or mobile operators [31]. Patterns of unusual transactions or calls can be encoded in a CEP language, and vast amounts of data can be effectively filtered in real-time, yielding suspicions that can be confirmed or processed further. An example of such pattern are a series of small transactions to multiple accounts, followed by a large transaction within a short time period[1]. On the other hand, cloning type of mobile fraud could be discovered by detecting a series of calls from distant areas within a short time, employing a so called *velocity trap*.

CEP systems also have a huge applicability to financial stock monitoring (one system that targets this is Cayuga [5]). By continuously analysing stock quotes, possible arbitrage opportunities could be detected. For example, given that the CEP system receives all orders made on a market, it could detect multiple orders on some equity with increasing prices within a short time, indicating that there is a big demand for such equity. It could also detect a series of price quotes for a company with an upward trend, indicating an interesting investment opportunity.

In many cases CEP systems are not used as standalone, but are rather coupled with components that perform further processing. For example, in the case of fraud detection, the filtered and aggregated data may later be processed by a slower machine learning system for confirmation of unusual user behaviour.

1.2 Motivation

In recent years, research has focused on distributing complex event detection to achieve better scalability. This is based on dividing event queries and running their parts on separate machines. Most systems are designed to run a large number of queries simultaneously, and therefore simply distribute event detection on query or operator granularity. Limitations of these systems arise when there is abundance of resources, only a few submitted queries, and high event throughput is required. In this case, queries can be distributed only up to a certain level, utilising only a certain fraction of the available machines. However, there are applications that require extreme throughput, fraud detection and stock market monitoring for example (e.g. New York options exchange is capable of disseminating 1.5 million stock quotes per second [26]).

We can go beyond distributing queries and try to replicate detection, such that patterns for the same query are detected in parallel at multiple nodes. This would improve scalability and achieve higher event detection throughput. To our knowledge, such work has only been done marginally in distributed Cayuga [7], which is capable of parallelising detection at operator level. However, Cayuga does not solve the scalability problem for all queries, but only for those that can be split on predicates. Thus, it is important to design an alternative approach, which could scale complex event detection beyond the simple distribution of query parts.

In addition, most of the current CEP systems were designed to optimize for low network usage, such that they fit into the common 100 Mbps network bandwidth range. However, there is a trend towards cloud computing, where resources are offered on demand, can scale to hundreds of machines, and 1 Gbps network connections are readily available. Thus, a novel system is needed, which could exploit this environment, while also examining possibilities of automatically scaling event detection depending on required throughput and available hardware.

Recently a *Storm* [33] stream processing platform was released that allows for a development of applications that process streams of data in a cloud environment. Programmers in Storm

[1]Interestingly, a fraud of 540£ with this pattern was committed on our bank account, but was detected by a similar system, thus preventing further damage.

compose computation graphs from processing elements, which can be arbitrarily parallelised depending on required throughput. However, coding Storm components is tedious and error prone. This offers an opportunity for a development of a scalable complex event detection engine, where event patterns would be expressed as computation graphs and query operators would be implemented as replicable processing elements. Since Storm is a novel framework, we are additionally looking for interesting performance measurements.

1.3 Project aims

The aim of this project was to design a dedicated CEP system that explores the possibilities of parallelising complex event detection in cloud environments, and also achieves higher throughput, while retaining low detection latency. Users of such system would define event patterns in a domain specific language, which then would be compiled into a network of processing elements and run in the cloud on the Storm computing platform. Given the available resources and the desired detection throughput, the CEP system would utilise a cost model to determine the optimal parallelism of event detection. Furthermore, the implementation would try to obtain a balance between CPU, memory and available network usage, and also explore optimizations, which could be applied to event detection for higher performance.

The main goals and contributions of this project are:

- **Event pattern language.** We designed a high-level language which allows for definition of event patterns from the combination of a unique set of well-defined operators. The designed language is very concise, contains a small number of operators, and is still very expressive and simple to parallelise.
- **Event detection system.** We designed a framework that allows for complex event detection at the Storm platform. We also implemented a compiler from the event pattern language into a set of constructs that can run on this framework, and detect specified complex event patterns. Additionally, we created a comprehensive GUI for specification, compilation and submission of complex event queries to a Storm cluster, as well as remote monitoring of their performance.
- **Parallelism cost model.** We developed a performance model that can infer optimal parallelism of complex event queries depending on event arrival rates and available resources, combined with other runtime parameters.
- **Optimizations** We introduced a number of effective optimization heuristics that improve cluster utilisation. In particular, these are expression indexing for fast predicate evaluation, reuse of common event patterns, event batching for reduced communication overhead, garbage collection of events that cannot be matched any more, two-phase event stabilization, and efficient serialization through generation of external event payloads.
- **Evaluation** We evaluated how performance of individual language operators improves with increased parallelism, and determined the applicability of our performance model on full queries. We also examined some performance characteristics of the Storm framework and its applicability to complex event detection.

1.4 Report outline

We will start with the background chapter and describe the characteristics of complex event processing systems and existing work in the field. Chapter 3 will then introduce a language for the detection of event patterns, and its event and temporal model. We will continue in Chapter 4 with the design of Step (STorm complex Event Processing) CEP system and its approach to complex event detection. Particular Step algorithms and implementation issues will further be described in Chapter 5. The parallelism cost model and our approach for estimating various runtime metrics will be explained in Chapter 6. Chapter 7 then evaluates the scalability and throughput of the Step CEP system on a granularity of individual operators, as well as complete queries, and comments on some performance aspects of the Storm framework. Chapter 8 concludes findings of this project, and explains possible future directions that similar projects could take.

Chapter 2

Background

2.1 Introduction to complex event processing

2.1.1 Complex event processing

CEP - Complex event processing (as defined in [14]), is a technology for extracting higher level knowledge from simple events received over messaging infrastructure from different sources. An event is any happening of interest and contains business-sensory data, for example a temperature sensor measurement or a stock quote. A combination of such atomic (also referred to as *primitive*) events is called a *complex, or composite event.* Complex events are specified using *event patterns*, which are primitive events combined with *event composition operators.* An example of an event pattern is $A; (B|C)$, detecting event A followed by either event B or C. Here composition operators are sequence ; and union |.

An event in a CEP system typically carries *event attributes* and one or more *timestamps*. The timestamps can refer to its occurrence time, its arrival time to the system, or can specify its duration. Different CEP systems implement different time models, as we will see later. The event attributes specify the properties of an event (i.e. its carried payload), and must adhere to an *event schema*. The schema usually specifies the types and names of the attributes, as well as their format. Some systems use XML format, in which an event carries both its schema and its properties, others rely on attribute ordering. It is common that a CEP system implements *adapters* that transform incoming events from different sources into a unified internal schema, which is later used during complex event detection.

The role of a CEP system is to continuously receive events from event sources on so called *event streams*, detect user-specified event patterns and disseminate complex events to *event sinks*. To specify complex event queries (i.e. event patterns), it is popular to use verbose, SQL-like languages. Such high level language then gets translated into a number of CEP operators, which will be used for the detection. Each CEP system uses a different detection method, tailored to its own set of operators.

The design requirements of a CEP system can be summarised into three points:

- *Expressiveness* The event pattern language must contain enough powerful operators, to be able to express many queries applicable to different scenarios. When designing a CEP system, it is therefore important to carefully define operator semantics.
- *Usability* Complex event queries have to be expressible in a simple way. This affects the design of the high level language, as well as operator semantics.

- *Efficiency* CEP system efficiency could be measured in network bandwidth usage, in event detection latency, or in the number of events the system is able to detect per unit of time (throughput). Often trade-offs have to be made here, as for example both low detection latency and low network usage may not be achievable. Each detection system implements its own set of optimizations that are suited for the given operator semantics. We will discuss this later on concrete systems. In many applications, a CEP system will have to handle thousands of queries at the same time. As a result, most systems try to implement *multi-query optimizations*, e.g. operator reuse. Other optimizations are query rewriting using cost models, reordering or merging operators, or implementation-specific optimizations like custom garbage collection or custom memory management. To deal with event overload and resource shortage situations, it is also common to implement approximation techniques, such as random dropping of events during processing.

2.1.2 Characteristics of CEP systems

We can summarize the characteristics of CEP systems in the following points (as explained in [15, 22]):

- Input to CEP systems is continuous, possibly infinite stream of events.
- Event streams require real-time processing, low latency event detection and are usually too big to be stored in their whole entirety.
- The input event streams are volatile. I.e. the arrival rate can vary, events can arrive in bursts and out of order, be lost or intentionally omitted, and timestamps may be imprecise.
- Input events usually exhibit strong temporal relationships, and come from external sources and not from a central database or permanent store.
- CEP systems must cope with large number of submitted queries in real-time and must process a large number of events, out of which only a small percentage is of interest. As such, they are often used for monitoring.
- CEP systems are usually concerned with relationships between events and their patterns, rather than with individual events. They combine data from multiple sources and infer from it more high-level and useful information.
- The processing in CEP systems is directed by newly arrived events rather than historical data (as compared to databases).
- CEP systems follow DAHP (database active, human passive) model, in which a system does continuous processing and notifies user. In comparison, traditional database systems employ HADP model (human active, database passive), meaning that data is simply stored and users query it manually.

2.2 Event-based systems

Complex event detection became popular with introduction of *event-based* communication model in *event-based* enterprise systems. *Event based systems* (as explained in [11, 12]) are systems consisting of distributed and loosely coupled components, generating and receiving *event notifications*, where an *event* is any happening of interest. The service responsible for disseminating event notifications is known as *publish/subscribe middleware system.* The publish/subscribe system consists of *event consumers* and *event producers.* Event consumers express their interests in events in a form of subscriptions, where event notifications published by producers will be delivered. The communication between subscribers and publishers is anonymous (a subscriber does not name the publisher and vice versa), asynchronous and similar to multicasting (a publisher publishes events to many subscribers).

When comparing a publish/subscribe system to a database, events can be seen as data and subscriptions can be seen as queries over it. However, instead of the data being stored, the system stores only queries. The data continuously arrives from publishers, is matched against the queries, and is delivered to subscribers.

To register subscriber's interest in published events, there are different types of subscriptions:

- **Topic (subject) based**. Publishers always annotate events with a subject string. The subscribers can express interest and receive events based on their subject.
- **Type based**. Type based subscriptions allow events to be received depending on their type attribute. The event types can form a hierarchy and subscription can refer to any of its type sub-trees.
- **Content based**. Content based subscriptions allow events to be received depending on their attribute values and can include predicates (e.g. subscribe to all temperature readings smaller than 20C).

An examples of publish/subscribe systems are Siena, Hermes and Gryphon (comparison can be found in [13]). Gryphon provides an implementation of JMS (Java Message Service) topic based subscription API, as well as content based filtering on predicate conjunctions. Content based subscriptions are also provided by Siena; Hermes implements only type based subscriptions.

Publish/subscribe systems can typically scale to very high event rates with lots of publishers and subscribers. By using subscriptions, events of interest can be sent from event sources to event sinks. In some systems filtering on event attributes can be done. This can be seen as a very simple form of complex event detection. For many applications though, the expressiveness of subscriptions is not enough. To correlate multiple events, applications can be built on top of publish/subscribe systems (e.g. DistCED [10] that will be described later), to enable for *complex event detection*. These systems use the advantage of scalability and performance of a publish/subscribe middleware, but are also capable of detecting variety of complex event patterns.

2.3 Active databases

In this section we will briefly explain active databases (aDBS), where the early work on complex event detection was done. The most influential works in this area were the object-oriented database management system SAMOS [20] and the model independent event detection language SNOOP [21].

2.3.1 ECA rules

Active databases differ from conventional databases in that they are able to automatically perform operations in response to a certain event occurring and a certain condition being satisfied. The detection of events in a database and consequent action is specified by *Event-Condition-Action* (ECA) rules. Event part of the rule specifies an event pattern, which consists of *primitive events* composed with a set of *operators*. A primitive event can be any change of the database state (e.g. insertion, update or delete of a row). The *condition* part of an ECA rule is a predicate that is evaluated after an event pattern is detected. On satisfied condition an action will be invoked (also called *event signalling*). The action can execute an external program or perform a database operation, which may in turn cause other ECA rules to be triggered. An example of an ECA rule is: On insertion of two stock quotes (event) where second has smaller price (condition), compute their price difference and save it into a separate table (action).

To implement ECA rules (i.e. triggers), active databases have introduced trigger specification languages similar in style to the SQL language. These languages specify trigger rules in form similar to:

```
<rule>: ON <event pattern> CONDITION <condition> ACTION <action>
```

For us the most relevant part of active databases is how event patterns are specified and what detection techniques are used for them, which we will examine now closer.

2.3.2 Event pattern languages

SAMOS builds event patterns from primitive events, which describe a point in time of some occurrence in the DBMS. Primitive events can be *transaction events* (e.g. signalled when inserting an object to a database), time events (e.g. periodically signalling timers), and method events (signalled when a method is executed on an object in a database). To compose these, SAMOS offers six event operators. Composite events can be formed as a *conjunction* (E_1, E_2) (requires both events to occur), *disjunction* $(E_1|E_2)$ (at least one of the events needs to occur) or *sequence* of events $(E_1; E_2)$ (an event E_2 occurs after an event E_1 occurred). The other three operators are *history*, *negation* and *star*. History operator $(TIMES(n, E)\ IN\ I)$ signals when an event E occurs n times within an interval I. Negated operator $(NOT\ E\ IN\ I)$ signals when E does not occur within an interval I. Here, intervals are specified as a pair of discrete timestamps. Finally, star operator $(*E\ IN\ I)$ can be used to limit the signalling of multiple event occurrences in an interval to at most one.

SNOOP is a model independent language, and as so it distinguishes between two types of events: *logical* (conceptual) and *physical* (implementation specific). Logical events are atomic (e.g. a point in time after row insertion), whereas physical can have duration (e.g. the insertion procedure). The mapping between the two is specified using event modifiers. Event patterns in SNOOP are composed only from logical events. Similarly to SAMOS, primitive events in SNOOP include transaction events and time events. In addition, explicit (user defined) events can be added. With regards to event patters, SNOOP also can express disjunction and event sequence. The conjunction $Any(m, E_1, E_2, ..., E_n)$ is more general and signals only when at least m out of n declared events occur. A new operator is aperiodic operator $A(E_1, E_2, E_3)$, which can be used to detect an occurrence of E_2 during an interval bounded by events E_1 and E_3. Last operator can detect a periodic event $P(E_1, [t], E_2)$, which will periodically signal time events in interval between E_1 and E_2. Similarly to SAMOS, SNOOP also has star variants of periodic and aperiodic operators, which makes them signal at most once within an interval.

2.3.3 Event detection strategies

SNOOP and SAMOS use different algorithms for complex event detection. SAMOS is a centralised system based on Petri nets. A Petri net is a collection of *places* and *transitions*. Places can be connected to transitions and vice versa via *arcs*, but an arc cannot connect two places. Each place can store a number of *tokens*, in which case we say that it is *marked*. Each transition connects a set of input places and a set of output places. Whenever all input places of a transition are marked, the transition will be applied, removing one token from each of its input places and adding one token to each of its output places.

Every operator in SAMOS language can be translated into a Petri net. Given a set of event patterns, SAMOS will translate all their operators into Petri nets, which will be composed together. The resulting single Petri net can be used to detect complex events, which is done as follows: Every time a primitive event is received, a place in the Petri net is marked and a game of tokens played. I.e. some transitions can be applied and token can traverse the whole net. The token that moves across the Petri net can be seen as an event detected so far. Tokens that reach an accepting place correspond to a fully detected complex event. An example of a Petri net detecting events $(E_1|E_2)$ *and* $((E_1|E_2), E_3)$ can be seen in Figure 2.1.

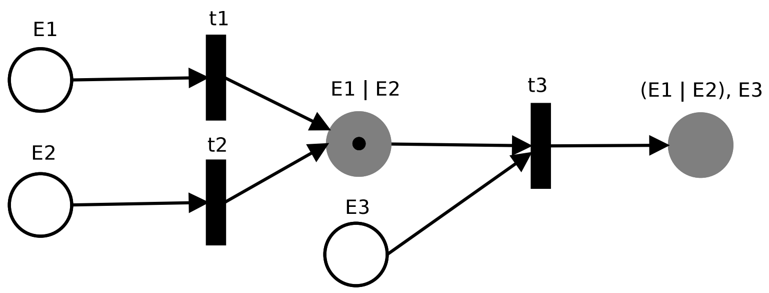


Figure 2.1: Petri net for detecting events $(E_1|E_2)$ and $((E_1|E_2), E_3)$ after event $E1$ was received (transition $t1$ was applied). Grey places are accepting, black dots represent tokens.

Petri net representation and algorithm are quite simple. The disadvantage is that composed Petri nets can become quite big with many queries injected, and the algorithm becomes slow. For our project, it is interesting to see that we could distribute the net across many machines. This could be done on operator granularity, in which case each node would execute one partition of the Petri net, or even on transition granularity, in which case each node would represent a place or a transition. In both cases, the nodes would cooperate with each other by receiving tokens and possibly sending tokens further. Accepting nodes would output complex events.

SNOOP chose a different approach and uses event graphs for complex event detection. Each leaf in the graph (node with no ingoing edge) can receive primitive events and each internal node is an operator processing them. When a primitive event occurs, its corresponding leaf node is activated, which will in turn activate all nodes attached to it via outgoing edges. Activating a node means notifying it of an event. When a node is activated, it will process received event and may activate nodes attached to it. A complex event is signalled when an accepting node in the graph is activated. Figure 2.2 shows a detection graph for event queries $(E_1|E_2)$ *and* $((E_1|E_2), E_3)$. Arrival of the event E_1 causes node E_1 to be activated, which will activate $E_1|E_2$, which will signal a detected complex event and activate node $(E_1|E_2), E_3$. The last node will store the event. When E_3 arrives, node E_3 is activated, which activates node $(E_1|E_2), E_3$. This node will signal a conjunction event, since it previously stored an event $E_1|E_2$.

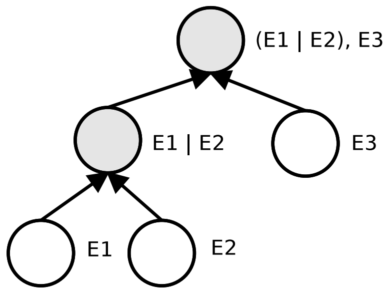


Figure 2.2: Event graph for detecting events $(E_1|E_2)$ and $((E_1|E_2), E_3)$. Grey nodes are accepting.

All queries in SNOOP are compiled into a single detection graph. The advantage of event graphs is that even with many queries, they will stay relatively simple and small. Also, as we will see later, detection graphs can be easily distributed. The disadvantage is that processing is not as uniform as in Petri nets; each node in the graph behaves differently when activated, depending upon operator semantics that it implements (e.g. union just forwards events, whereas conjunction needs to store and process them).

2.3.4 Conclusion

Primary function of database systems is to persist large number of data such that it can be retrieved effectively. As such, they are primarily not concerned with continuous data processing and hence do not store temporal relationships between data. Also, the events and their timestamps correspond to internal database state changes (e.g. insertion), rather than to external sources from which they arrive. It should be noted that triggers are an addition to database systems and as such are limited by their architecture. Typically, the functionality is centralised and difficult to distribute. As a result, active databases do not scale to a large number of triggers and high rate of arriving events. Active databases also do not have the mentioned characteristics of a general CEP system, but their ideas are used by CEP and data stream management systems (DSMSs), which we will discuss next.

2.4 Data stream management systems

Data Stream Management Systems - DSMSs (an excellent introduction in [23]) are systems somewhat similar to databases, but better equipped to process large amounts of data continuously arriving from input streams. DSMSs are not only capable of detecting complex events among their input data, but also can compute analytics on it, manipulate with it as it was relational table, save it, or replay it. They are popular in business intelligence, as they can handle large sets of data and apply many general queries on it. However, because of their generality, they are not able to accomplish such performance as dedicated CEP systems.

The most important characteristic of DSMSs is that they can handle both permanent relational data and continuous stream data. Thus, they support either standard SQL-like database queries or continuous stream processing queries. Database queries are executed only once, whereas continuous queries will execute forever on streaming data. A DSMS cannot store all streamed data to answer continuous queries due to limited number of resources. Common technique to deal with this is to evaluate the queries against a size-limited window of data that slides across the stream. The outputs of the queries can be stored in permanent relational tables, or streamed to external applications. An example use of DSMS query is: “Each day compute electric power consumption of every customer segment, where measurements come on streams from customer power meters”. Here, the DSMS needs to aggregate measurements over streams and correlate them with customer type stored in a relational database.

In this section we will look at two examples of data stream management systems, STREAM [1, 2] and Borealis [24]. In particular, we will examine closely their data model, event detection language, event detection strategies, and implemented optimizations, as these design choices are highly relevant to a successful CEP system.

2.4.1 STREAM

Data model and query language

STREAM [1, 2] models an input as a continuous stream S of pairs $\langle s, \tau \rangle$, where s is an input data tuple (a row of attribute values) and τ is a discrete timestamp of tuple arrival time on stream S. A relation in STREAM is modelled as a bag of tuples, corresponding to a window of received data on stream S. Streams can be partitioned into windows in terms of the number of tuples (*tuple-based sliding window*), the window time duration (*time-based sliding window*), or by attribute values (*partitioned sliding window*).

Complex event patterns can be specified by *continues query language* (CQL), which is an extension to the SQL language. Internally, STREAM executes queries only on relational tables, and not directly on streams. To implement querying continuous input streams, STREAM transforms them into temporary relational tables and executes queries against the tables instead. CQL language therefore contains three types of operators: *relation-to-relation*, *stream-to-relation* and *relation-to-stream.*

Relation-to-relation operators can be used to execute complex data queries over relational tables. These are specified using the SQL language, and support standard database operators such as select, project, union, intersect, except, aggregate, duplicate-eliminate and different types of joins. Stream-to-relation operators can be used to specify how input streams will be transformed into temporary relational tables that can be queried. This can be defined in terms of three different sliding windows, as mentioned earlier. Relation-to-stream operators are used to specify translations of relational tables into output streams, i.e. when data from a relational table should be output to a stream. These are *Istream* operator (outputs all data that is inserted into a relation), *Dstream* operator (outputs all data that is deleted in a relation) and *Rstream* operator (outputs all data present in a relation all the time).

The usage of different operator types is best visible on an example from [2]:

```
Select Istream(*) From S [Rows Unbounded] Where S.A > 10
```

Here an unbounded-length window is taken from stream S and is transformed into a temporary relation (the relation continuously updates as the window slides). SQL algebra is then applied on this relation to filter out tuples of interest. As specified by Istream operator, the newly filtered tuples are then sent to an output stream.

Operation and optimizations

Submitted CQL query is compiled into a query plan, similar to event graph in SNOOP. The query plan consists of processing *operators* connected to each other via input and output *queues*. Each operator can also have an associated *synapse*, which stores its run-time state, typically a materialized view of a relation. For example, a join operator needs to keep a window of tuples for both of its input streams; hence, it will keep two synopses, each storing tuples from one stream.

When data arrives on a stream, it is timestamped and put on a queue of some stream-to-relation operator. Each operator in query plan consumes an input from its input queue, does some processing (join will for example update its synopses, and join them), and outputs result onto its output queue. STREAM is a centralised system and all operators run in the same thread together with a scheduler. Scheduler assigns each operator a time slot in which it will run.

A critical part of STREAM system is efficient memory management, since synopses can grow quickly. Therefore, synopses are shared across operators, thus not replicating data in the system. Also, an optimizing scheduler is implemented, choosing operators in a way such that input from queues is consumed as fast as possible. Furthermore, two approximation techniques are implemented - *load shedding* and *synopses shrinking.* Load shedding (detailed description in [3]) is employed when running out of CPU resources, and is implemented as a sampling operator, which probabilistically drops tuples. If required, it can be inserted into any part of the event query. Shrinking synopses is a way to handle low memory conditions by limiting their size. This is done by making their windows smaller, or introducing new windows. Finally, STREAM utilizes operator reuse. Whenever a new CQL query is submitted, the resulting query plan is merged with existing queries in the system and common operators reused, sharing memory and computation load.

2.4.2 Borealis and Aurora

Borealis [24] is a DSMS based on Aurora [22]. It is a complex system due to its ability to deal with *dynamic query revisions* (streams of data that correct errors in already received data), *dynamic query modifications* (deployed query plans can be changed at run-time), and implemented *optimizations.*

Data model and operators

The data arrives to Borealis as a sequence of append-only tuples of the form $(k_1, .., k_n, a_1, .., s_n)$, where $k_1, .., k_n$ specify unique key of the tuple (to support data revisions) and $a_1, .., s_n$ carry its attribute values. Each arriving tuple can be an insertion, deletion or revision of already inserted tuple. The tuples are timestamped on arrival and can carry quality of service metrics, such as total resources consumed by its processing. These metrics are then used by load shedders and optimizers to distribute load.

Borealis implements two types of operators - *order-agnostic* and *order-sensitive.* Order-agnostic operators are filter (i.e. select), map (i.e. projection), and union. Order-sensitive operators are aggregation (application of function to a stream window), sort (approximate sorting of tuples), join (joining of two stream windows on predicate), and resample (aligning streams and computing a function over them). Each order sensitive operator needs to specify assumed order of input streams and a window size, over which they are applied. The operators in Borealis are quite sophisticated, for example, filter is able to filter multiple streams on multiple predicates and map can output multiple projections of one tuple.

Operation and optimizations

Queries in Borealis are represented as data-flow graphs. They consist of boxes (operators) connected by arrows representing data flow. Input data flows into boxes, where it is processed and sent further downstream. The graphs can also contain connection points, where new boxes can be added and data-flow changed. They can also persist data (analogous to materializing a view), which can be pushed downstream, or pulled by downstream boxes (e.g. join box pulls data for efficiency).

Borealis is a distributed DSMS. Each node runs a component called *query processor*, which executes one part of the data-flow graph. Query processor receives data from an input queue, processes it by a number of boxes and outputs it on another queue. Each query processor has its own persistent storage, load shedder, scheduler and local optimizer.

Optimization in both Aurora and Borealis is done dynamically. Aurora gathers statistics such as the average cost of box execution and box selectivity, which are then fed into a cost model. An optimizer uses the cost model to optimize a sub-network of boxes. It can insert projections to eliminate unneeded data attributes, it can reduce box execution overhead by combining boxes, and it can reorder boxes to achieve higher selectivity early in the query plan (e.g. filters can be sometimes pushed upstream).

Similar ideas are used in Borealis. During query processing, *monitors* gather quality of service statistics on each site, such as CPU and bandwidth usage, and compare them to other sites. The monitors then continuously refine box distribution by triggering *local*, *neighbourhood* and *global* optimizers. Local optimizers are able to shed load with regards to event priority and quality of service. This information is also used when scheduling boxes. Neighbourhood optimizers can improve resource utilisation by migrating a box to a neighbourhood site, and can also initiate distributed load shedding (tuples will be dropped in collaboration with upstream sites). Finally, global optimizer has information about all sites and can try to improve throughput, latency, lifetime of sensor network, or its coverage. It does this by identifying bottlenecks and migrating operators to different sites (details in [24]).

2.4.3 Conclusion

DSMSs are complicated systems that provide general stream processing capabilities for any kind of application. This includes complex event detection, aggregation of data, computing of statistics, and data mining. Thus, DSMSs support similar functions as databases, but on real-time streams. As such, they often try to implement SQL-like languages with operators and expressiveness similar to the SQL. As a result, they support a broad range of queries, but expressing complex event patterns can be cumbersome (e.g. event sequences have to be expressed as joins). Even though DSMSs implement sophisticated optimizations such as moving operators, combining them, and shedding load, they are not able to achieve the performance of dedicated CEP systems. The data-flow graphs used by DSMSs are a convenient abstraction, on top of which a scalable dedicated CEP system could be built. This is because data-flow graphs closely match the streaming paradigm, support optimizations, and can be easily partitioned for distribution.

2.5 Dedicated CEP systems

Complex event processing differs from data stream management in two major ways (as explained in [5]): Firstly, it is dedicated to detecting event patterns, instead of providing general capabilities to process streams. As a result, CEP systems provide languages that are more suitable for expressing non-occurrence of events (e.g. safety conditions), order-related constraints and to correlate long sequences of events. We could express such relationships also in DSMSs, but it would be cumbersome and it would result in long queries that are difficult to optimize. Secondly, dedicated CEP systems are similar in scalability and performance to publish/subscribe systems. As such, they are able to process large number of concurrent queries, which outperforms the scalability of stream management systems.

In this section we will describe four CEP systems - *Next* [4], *Cayuga* [5, 6], *SASE* [9, 8] and *DistCED* [10]. In particular, we will address event representation, event pattern languages, detection strategies and optimizations. Some of these systems are also able to *distribute* event detection and we will discuss how these ideas could be used to also *parallelise* event detection. We will use the term distribution to refer to partitioning of query plans into smaller units, which can be run on separate nodes. Under parallelism[1] we will refer to replicating query plans across many nodes, such that each query plan processes only some fraction of input streams, thus detecting event patterns in parallel.

2.5.1 Cayuga

Cayuga [5, 6] is a centralised general purpose event processing system that allows event detection through a small number of well-defined composable operators.

Data model

Event streams in Cayuga have a fixed schema and are (possibly infinite) sets of event tuples $\langle a, t_0, t_1 \rangle$, where $a = (a_0, ..., a_n)$ are the attribute values of an event (payload), t_0 is the start timestamp and t_1 is the end timestamp of the event. The timestamps are discrete and for instantaneous events identical. Duration events, overlapping events and simultaneous events are also allowed. Events arrive and are processed in time order: event e_1 comes before event e_2 if $e_1.t_1 < e_2.t_0$. Thus, overlapping events are not assumed to arrive in order.

Query language

Cayuga expression language consists of four unary and three binary operators. The operators are explained in the following list:

- **Selection** (σ_θ) filters out all events not satisfying θ predicate.
- **Projection** (π_{ATTR}) selects those attributes of events, as provided in the $ATTR$ set.
- **Renaming** (ρ_f) renames attributes and changes the schema of a stream according to function f.
- **Aggregation** (α_g) applies an aggregation function g over a stream, adding a new attribute-value pair to the event schema. This enables for example computation of an average over last stock prices.
- **Union** ($S1 \cup S2$) detects all events from stream $S1$ and stream $S2$.
- **Conditional sequence** ($S1;_\theta S2$) detects an event from $S1$, followed by event from stream $S2$, satisfying θ predicate. The predicate can refer to events from both streams.
- **Iteration** ($\mu_{\tau,\theta}(S1, S2)$). Similarly to semantics of Kleene star operator, this detects event $S1$ followed by any number of events $S2$ satisfying predicate θ. Informally, this is equal to $S1 \cup (S1;_\theta S2) \cup (S1;_\theta S2;_\theta S2) \cup ....$ Cayuga avoids unbounded memory by storing just $S1$ values and most recent $S2$ values in iteration. The τ expression enables to modify the result on on each iteration, and can contain projection, selection and renaming operators.

To detect for example stock quotes of the same stock falling for at least 30 minutes, we could express the query as: $\sigma_{\theta 3}(\mu_{\sigma_{\theta_2},\theta_1}(S1, S2))$, where $S1$ and $S2$ are streams of quotes, and $\theta_1 = S1.name = S2.name, \theta_2 = S1.price >= S2.price.last, \theta_3 = DUR \geq 30min$

[1]Later it will be seen that Cayuga refers to distribution/parallelism, as column/row scaling.

To submit queries, Cayuga provides an SQL-like language, in a form of:

```
SELECT attributes FROM stream-expression PUBLISH output-stream
```

Note that unlike other systems, both event patterns and predicates are specified in the stream-expression part. The stream-expression contains constructs like *FOLD, NEXT and FILTER*, which are translated into mentioned Cayuga operators. Also note that input and output of an event query is a stream, enabling queries to be arbitrarily nested inside other queries.

Operation

Cayuga is based on non-deterministic finite state automatons, where an input event can cause an automaton instance to non-deterministically explore all outgoing edges. The input alphabet to automaton is all possible events, and every edge between states P and Q is annotated with a pair $\langle \theta, f \rangle$. θ is a predicate over events from P, and f is a transformation function over the events (e.g. renaming function). In each state, the automaton also stores so called *bindings*, which are events that contributed to a state transition of an instance.

Cayuga translates a query expression into an NFA. Each node in the NFA has a forward edge (apart from last one), filter edge (self-edge) and possibly a rebind edge. Forward edge corresponds to an operator detecting an event, filter edge to a predicate filtering (e.g. θ predicate in the iteration operator) and a rebind edge to a function that changes the binding stored in a state (e.g. τ function in the iteration operator). The automaton works as follows: given it is in a state P, storing binding x, it will transition into state Q only if an event e arrives that satisfies the edge predicate $\theta(x, e)$. The new binding stored at state Q will be $f(x, e)$, where f is the above-mentioned transformation function.

To explain the formal definition, we will use an example from [5]. Suppose we want to output consecutive stock quotes with the same name. The event algebra and corresponding NFA are shown in Figure 2.3. Here, P is the starting state, S is the accepting state, and the renaming operator gets translated into the edge $e1$. Since the predicate on $e1$ is true, any input event e will cause a new automaton instance to be spawned and transitioned into state Q. During the transition, the renaming $e.Name \rightarrow N$ will take place. At state Q, any event not satisfying the next predicate will cause self-edge transition. A sequence of events is detected when the Q state's forward edge is traversed. By definition, this will happen when the edge predicate is satisfied (true) and predicate associated with state P is satisfied ($Q1.name = e.Name$). If this holds, automaton will apply the projection function on edge $e3$ and transit to the accepting state S.

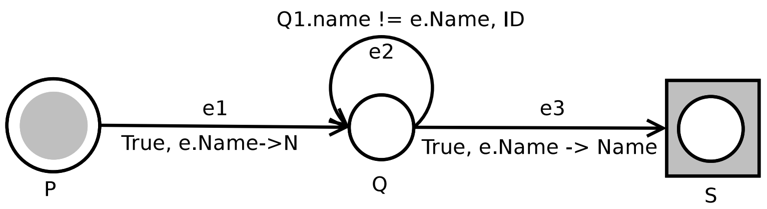


Figure 2.3: Detection of $\pi_{Name}((\rho_{Name \rightarrow N} Stock);_{N=Name} Stock)$ in Cayuga (no rebind edges)

Distributed Cayuga

Cayuga was implemented as a monolithic centralised system. Input events were stored in a central priority queue that was consumed by a *Query Engine*. Query Engine managed instances and transitions of the NFA. However, a work has been done in [7] on distributing Cayuga by using techniques called *row/column scaling* and *pipelining*.

Row/column scaling is distribution of event queries across an $n \times m$ matrix of available nodes (see Figure 2.4). The number of queries in the system is divided between n machines, forming one row in the matrix. The rows are then replicated m times. A received event is then uniformly routed to all machines in one of the rows, for example in a round robin fashion. All queries in a row will process the event. By adding a column to the matrix, we reduce the number of queries processed at each node (i.e. increase query distribution). By adding a row to the matrix, we reduce the number of events being processed at each row (i.e. increase parallelism). Both ways, we increase throughput. The smallest unit of distribution is a single query. A caveat is that a simple round robin event dispatching will not work because to be able to correlate events, they have to be dispatched to the same row. Thus, a dispatcher is employed that depending on query selectiveness, dispatches related events to the same row (stock example: if each query operates only on one stock, all events from stock exchange can be partitioned on stock name). Dispatcher might be also parallelised, to avoid it being a bottleneck.

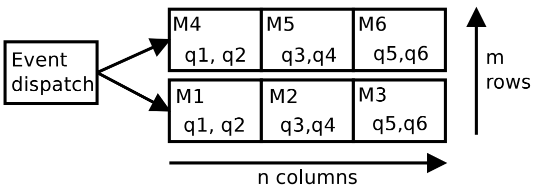


Figure 2.4: Distributing 6 queries across 3x2 matrix of machines.

Pipelining makes the unit of distribution smaller. That is, a single query is split into sub-queries to be executed at separate machines. Each machine executes a part of Cayuga automaton and its output is routed to another machine in the pipeline. The last machine contains an accepting state and detects a complex event.

Conclusions

Cayuga has a very powerful language with well-defined semantics and high throughput. It also implements many optimizations, such as custom garbage collection and query indexing. Query indexing keeps track of which events affect which transitions. Consequently, only the predicates on affected edges are evaluated, thus greatly reducing CPU load. Also, multi-query optimization is performed by merging automaton instances into one instance (each state in Cayuga automaton will represent multiple instances).

Techniques implemented in distributed Cayuga are important in our project. Pipelining is a common technique, and it was implemented, such that the smallest unit of distribution is an operator. Row/column scaling is a nice idea for parallelising operators on predicates. The problem arises when an operator does not contain a split-able predicate and thus cannot be parallelised, e.g. $A;_{true} B$. In this case different technique should be used, e.g. replicating A stream and splitting B stream across different nodes.

2.5.2 Next

Next [4] is a distributed CEP system implemented in Erlang that specifies event patterns in an SQL-like language and detects them using finite state automaton (FSA).

Data model

Input streams in Next are infinite sequences of events that arrive from event sources with the same schema. An event is a pair $\langle s, \mathbf{t} \rangle$, where s is a set of fields as defined by a schema S, and $\mathbf{t}$ is a sequence of timestamps. The timestamps are discrete and include the start and end time of an event, and for composite events also all primitive event times. There are four types of events - *instantaneous*, *duration*, *empty* and *failed-detection*. To enable for operator associativity and optimization, an event E1 with timestamps $(t_{s1}, ..., t_{e1})$ is considered to occur before another event E2 with timestamps $(t_{s2}, ..., t_{e2})$ when $t_{e1} < t_{s2}$.

Query language

Queries in Next can be defined using a *high-level* SQL-like event query language, which consists of two parts: the *Stream Definition Language*, which is used to define schemas of data sources, and the *Event Query Language*, which is used to specify the complex event queries. The latter has the form of:

```
SELECT selection FROM eventpat WHERE qualifications
```

Here, *selection* specifies the output fields of the detected event, event pattern is built by using next, exception, union and iteration operators, and *qualifications* contain filtering predicates and time constraints. Queries specified in this way are then translated into *core language*, which is a set of event composition operators with well-defined semantics. The Core Language has six operators:

- **Filter** (E_θ) detects all events satisfying θ predicate
- **Union** ($E1|E2$) detects occurrence of event $E1$, or event $E2$, or both
- **Next** ($E1; E2_{\phi,\theta}$) detects event $E2$ satisfying predicate θ, which follows event $E1$, but skipping any intermediate events $E1$ not satisfying predicate ϕ.
- **Iteration** ($E+_{\phi,\theta}$) detects events E satisfying θ, but skipping intermediate events E not satisfying ϕ
- **Exception** ($E1 \setminus E2_\lambda$) detects event $E1$, but fails if $E2$ satisfying λ occurs
- **Time** ($E@_{time}$) detects time point after specified time from occurrence of an event E

Operation

Each operator can be translated into a finite state automaton. Given a complex event query, Next will translate all of its operators and compose them into one single FSA. The states in the FSA correspond to partially detected events; the start state represents an empty event, and accepting states correspond to fully detected complex events. The transitions in the FSA are labelled with an event, a predicate, and a transformation function (e.g. aggregation function). At any time, there will be a number of existing automaton instances detecting different event patterns. Whenever a new event arrives that causes a transition in any of the instances, a new automaton instance will be spawned, extending the transitioned instance. When a transition is made into an accepting state, a complex event is output. When a failed-detection state is reached, the automaton instance will be discarded. Instance will be also discarded if received events do not cause any transitions.

To distribute the detection, Next partitions the FSA across multiple nodes. Each node has a source and a sink proxy to receive and publish events, and runs a generic event automaton corresponding to a single event operator. For example, a node detecting $A; B_{\phi,\theta}$ will receive events A and B on its source proxy and publish complex event $A; B_{\phi,\theta}$ to its sink proxy. To detect the event $A; B_{\phi,\theta}$, it will spawn and discard automaton instances corresponding to the next operator FSA.

Optimizations

Next tries to optimize for low CPU usage and low event detection latency. This is justified by high computational demand of event pattern detection. The cost model used by Next estimates costs of operators as a function of arrival rates and input streams' burstiness, by considering the worst case scenarios. Next uses the cost model to implement query rewriting techniques that utilise operator associativity to bracket query for minimal cost execution. For example, if in the event pattern $E1; (E2; E3)$ event $E1$ occurs rarely and event $E2; E3$ occurs often, it is more cost effective to rewrite this pattern to equivalent $(E1; E2); E3$. Union and exception operators are optimized in a similar way.

Next also implements a cost model for optimal placement of operators on nodes, which takes into account CPU utilisation of different nodes, as well as communication overhead. Operators are distributed such that deployment cost is minimized. Furthermore, Next also implements operator reuse optimization, which means that common operators across multiple queries are deployed only once.

Conclusion

The finite state automaton that Next uses is a common technique for event detection. Since the FSA can be arbitrarily partitioned, it is also suitable for distribution and same ideas for parallelism as in distributed Cayuga apply. However, all operators have to be treated uniformly and mapped into an FSA. This increases complexity and could also lower performance, since for example next and iteration operators consist of many states. Therefore, we consider data-flow graphs where each operator can have its own implementation more superior performance-wise.

Furthermore, the data model in Next is designed to allow query rewriting optimization. As a result, it does not allow detection of a sequence of overlapping events, which could be a disadvantage for some applications. Also, complex events in Next can carry potentially a lot of timestamps (two for each primitive event they contain), which may pose significant communication overhead on large event patterns.

Finally, the query language is much simpler than in Cayuga, but yet quite expressive and has well-defined semantics. Because of a small number of simple operators, the language could serve as basis for our project.

2.5.3 SASE

SASE [9, 8] is another complex event processing system, designed to cope with high rate and variability of RFID chip readings. SASE is interesting because it introduces operators and optimization techniques that have not been yet mentioned, and it uses both data-flow graph and a non-deterministic finite state automaton for simple complex event detection.

Event model and language

SASE receives infinite sequences of events on input streams, each event having a *type* and a set of attributes. A received event is assumed to be timestamped by its originator. Events do not have duration, and are assumed to be totally ordered. The SASE language has the form:

```
 EVENT <event-pattern>
[WHERE <qualification>]
[WITHIN <window>]
```

The language will get translated into 5 operators, which have the following semantics:

- $ANY(A_1, A_2, ..., A_n)$ is a union operator over multiple streams.
- $SEQ(A_1, A_2, ..., A_n)$ detects a sequence of events (equivalent to applying Cayuga's next operator multiple times, but without a filtering predicate).
- $SEQ\_WITHOUT(S_1, B, S_2)$ specifies that no event of type B can occur between two sequences of events S_1 and S_2. The negative event B can also be located at the start of a sequence or at its end, in which case the *WITHIN* clause should be used to bound computation.
- $SELECTION(pattern, P)$ filters event pattern on predicate P, where P refers to event attributes contained in the pattern.
- $WITHIN(pattern, T)$ - specifies a time window T in which pattern should be detected.

An example of an event query constructed using such operators is (from [9]):

```
EVENT SEQ(SHELF_READING x, !(COUNTER_READING y), EXIT_READING z)
WHERE x.TagId = y.TagId && x.TagId = z.TagId
WITHIN 12 hours
```

Here, SHELF_READING, COUNTER_READING and EXIT_READING are primitive events. The event pattern will get translated into the SEQ_WITHOUT operator because it contains a negative event, and the within clause into the *WITHIN* operator. One neat feature of SASE is that events are named (e.g. SHELF_READING x), which aids readability when referring to them in filter expressions.

Operation

SASE translates each query into a query plan, which consists of a subset of five operators connected in a pipeline. Each stage of the pipeline does some processing on events, with only the last one detecting a complex event. Each of the stages work on their own data structure and can be implemented differently. The query plan pipeline contains the following stages:

1. **Sequence Scan and Construction (SSC).** Firstly, all arrived events are processed by the SSC stage, which detects sequences of positive events. Any negative events are left for the the negation stage.
2. **Selection.** Selection stage filters detected sequences of events on predicates.
3. **Windowing.** At this stage, the sequences are filtered according to their duration, as specified by within operators.
4. **Negation.** Here, sequences containing specified negative events are filtered out.

5. **Transformation.** Finally, a complex event is output by concatenating all attributes of events in the detected sequence.

Most of these stages are trivial, apart from SSC, which does the actual detection of event sequences. The operation of the SSC is illustrated in Figure 2.5 and is based on an NFA. A runtime stack is used to keep track of active states, which will grow after an event contained in a sequence is received. Each active state instance in the stack also keeps track of predecessor instance, from which it was constructed (in the figure this is displayed with arrows). Every time an accepting state is reached (here state 3), sequence construction (SC) will take place. SC will output an event sequence by scanning runtime stack backwards; it will start at the accepting state and follow its predecessors, until a starting state is reached. Figure 2.5 displays three event sequences detected this way for the last accepting state.

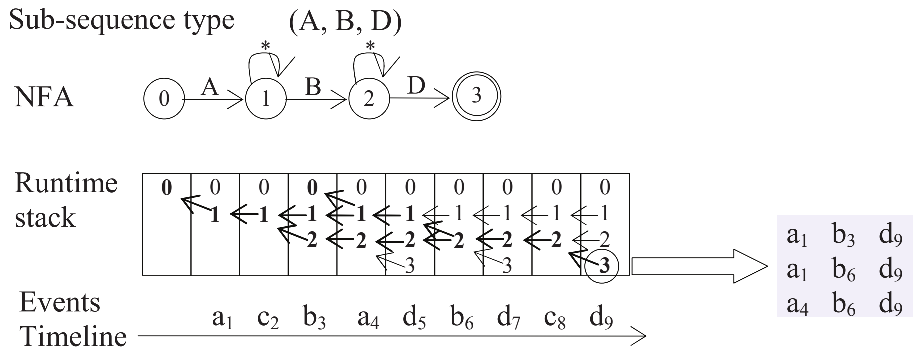


Figure 2.5: NFA-based Sequence Scan and Construction (figure from [8])

Optimizations

Firstly, since each stage is implemented differently, SASE can apply optimizations on a stage level. Such optimization is compressing the number of predecessor links in the SSC NFA by using Active Instance Stack. Secondly, SASE also optimizes on pipeline level. It tries to move selection predicates and windows down the operator pipeline to the SSC stage, to increase SSC stage selectivity and reduce the number and length of detected sequences.

Evaluation of predicates in the SSC stage is implemented by partitioning the NFA runtime stack on attribute values (Partitioned Active Instance Stack). Thus, SASE is able to push down only equivalence predicates. Windows can also be pushed down and applied during sequence construction, in which case they will limit the backward predecessor search for a sequence. They can be also be applied during sequence scan, in which case the events that are outside of a window will be dynamically pruned from the runtime NFA stack.

Conclusions

The SASE language is not as expressive as languages described earlier - it does not support aggregates or detection of event sequences of arbitrary length, as it focuses mainly on RFID applications. Also, it allows only for composition of primitive events into complex events, but not arbitrary composition of complex events. One issue that prevents this is assumption that events are totally ordered.

However, the language uses a concept of negated events, which are a neat way to easily express safety conditions (i.e. that no event occurs). Secondly, even though SASE is centralised, the pipeline concept enables to distribute each processing stage onto a separate machine. Furthermore, selection, transformation, negation and window stages could be easily parallelised, since they are stateless. Parallelising the SSC stage is more difficult, as it needs to see all incoming events. To achieve this, we could replicate the SSC stage and deliver a sliding window of events to each replica. This would be similar to partitioning of input streams in STREAM.

Finally, moving predicates and windows in a query plan is an interesting optimization, since it definitely reduces the load on later stages of the detection pipeline. However, it also introduces higher complexity and makes computation more centralised, a trade-off that has to be considered.

2.5.4 DistCED

DistCED [10] is a complex event detection framework that works on top of JMS publish/subscribe system, detecting events in a distributed environment.

Data model

Unlike SASE, events in DistCED are treated uniformly and the event model allows for composition of complex events into other complex events. Events arrive on subscriptions by underlying publish/subscribe system and carry a pair of timestamps, thus being able to represent both instantaneous and duration events. Such interval timestamp can also include clock uncertainty on event sources, which may be significant in distributed setting when using time synchronization protocols, such as NTP. DistCED implements two event orderings - *partial* and *total*. Events are partially ordered by a single timestamp, and total order is imposed by taking both event timestamps into account. Based on ordering, an event can follow another event *weakly* (partial order) or *strongly* (total order). Different orderings are implemented by different operators.

Query language

The event queries in DistCED are specified by composition of seven core operators. *Atoms* are able to filter out arriving primitive events according to their type. Similarly to Next, the language also contains *sequence* $A;B$ (event A strongly followed by event B), *iteration* $A*$ (detects any number of A events) and *alternation* $A|B$ (union) operators. Additionally, it introduces *concatenation* AB (event A weakly followed by event B), parallelisation $A \parallel B$ (similar to alternation, but any order and overlapping of events is allowed), and timing $(A,B)_{time}$ (detection of an event pattern within a time interval) operators. The language is rich in considering different variation of a single operator. However, the operators are not parametrized with predicates, or aggregation and transformation functions, which makes them less powerful. In this paradigm, incoming events could be filtered on predicates using the underlying publish-subscribe system.

Operation and optimizations

The event patterns in DistCED are split into sub-patterns, which are detected using a Composite Event Detector (CED). Each CED can be deployed on a separate node, thus distributing event detection. Since all operators are at most binary, CED consumes events from at most two subscriptions and can publish one detected composite event. The smallest unit of distribution is an operator, which is implemented using a standard finite state automaton. For each state, the automaton processes only the events associated with the state's input domain Σ. The automaton will transition from state s_0 to state s_1 when an event corresponding to the transition is received. Composite event is published when the automaton reaches an accepting state. An example automaton for sequence $A; B$ is shown in Figure 2.6.

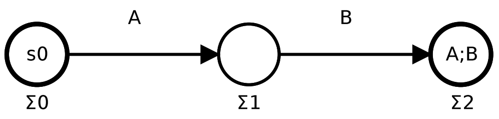


Figure 2.6: DistCED FSA for sequence $A; B$. s_0 is initial state, $A; B$ is accepting state. Σ are alphabets associated with each state.

In distributing event detection, it is important to consider optimal distribution of CEDs across network. Optimality here means finding a balance between low bandwidth usage and low latency. Composite event detectors in DistCED are mobile, and can migrate to a different location depending on what is optimal. Secondly, already deployed CEDs could be reused to minimize network bandwidth. However, the reuse of CEDs might not be ideal when trying to achieve low latency, in which case replication is better. DistCED specifies the behaviour of a mobile detector using distribution policies. These define the locations of detectors, the degree of their decomposition and the reuse of CEDs. It would be interesting to replace the policy with a cost model, which takes into account the available network bandwidth, deployment costs and event arrival rates.

2.6 Stream processing frameworks

To implement a distributed complex event detection system that works over streams, we use the Storm stream processing framework. This section describes the Storm framework, the reasons for choosing it, and other similar frameworks that also enable distributed applications to be made out of arbitrarily composed stream processing elements.

2.6.1 Storm

Storm [33] is a distributed real-time stream processing platform that can be used to assemble and execute stream processing elements. The applications that run on top of Storm cluster are called *topologies*.

Storm topologies

A topology in Storm is a data flow graph of computation, which consists of elements called *Bolts* and *Spouts*, connected together with streams. Streams are unbounded sequences of tuples, and a tuple is a list of values of any type. The sources of streams are Spouts, which read data from an external source, for example stock exchange, sensors or program logs. The streams are consumed by Bolts, which do some processing and possibly emit new streams that can be consumed by further Bolts. The processing done at Bolt can be anything, from filtering or aggregation, to saving tuples to a database. An example of a topology can be seen in Figure 2.7.

Topology can be seen as a blueprint for a runtime computation graph. At runtime, each component of a topology will run within a number of *tasks*, which are specified by a parallelism of that component. Every task for the same component executes the same blueprint code, but is a different instance and runs in a separate thread of execution. A task receives a tuple on its

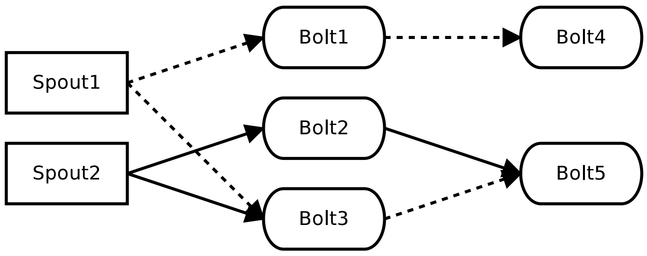


Figure 2.7: An example of a Storm topology.

input queue, processes it and may emit new tuples to its output streams. Streams are divided between multiple tasks depending on specified *stream grouping*. Available stream groupings include shuffling stream in round robin fashion, replicating stream to all tasks, or shuffling stream depending on tuple attributes. An illustration of component tasks communicating over streams can be seen in Figure 2.8.

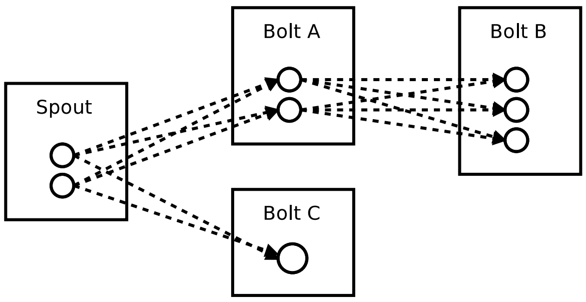


Figure 2.8: Each topology component runs in multiple tasks. The tasks exchange tuples on streams. A stream grouping specifies which tuple is delivered to which task.

To sum up, topology is a computation graph, where one specifies Bolt and Spout components, the stream connections between them with stream groupings, and parallelism for each component. A Storm cluster can run multiple topologies at the same time. From the time it is submitted, a topology will run forever, or until it is killed.

A submitted topology is run on a cluster by specified number of *worker* processes. Every worker runs an equal share of topology tasks (as seen in Figure 2.9) and the number of workers or tasks cannot change during topology run. Each Bolt task receives tuples from an input queue and can emit tuples to other Bolt tasks. Spout tasks only emit tuples. This interaction is implemented using an open-source messaging middleware, ZeroMQ [35]. ZeroMQ is a native transport layer implementation of asynchronous messaging, supporting publish/subscribe, request/reply, N to N and pipeline communication. Storm uses ZeroMQ through native Java binding library JZQM.

Components of a Storm cluster

A Storm cluster consists of three components: a distinguished *Nimbus* node, a number of *Supervisor* nodes, and a number of *Zookeeper* nodes, as illustrated in Figure 2.10.

The Nimbus node runs a daemon, which is responsible for receiving submitted topologies, distributing their code around the cluster, assigning topology tasks to workers, and monitoring for failures. Every *Supervisor* node runs a daemon, which stops and starts worker processes at its node. The maximum number of worker processes is configurable per node, and workers are spawned only after a topology is submitted (a topology specifies how many workers it

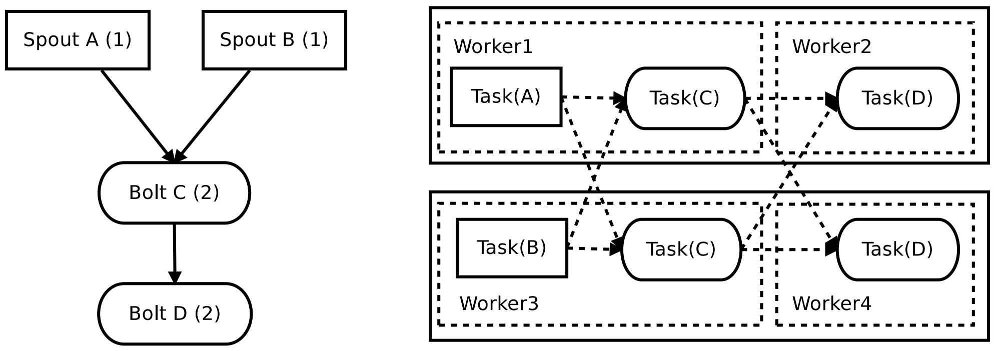


Figure 2.9: Storm topology (count in brackets means parallelism) and its organization at run-time, when two physical nodes run two workers each. A total number of 6 tasks is divided between the workers.

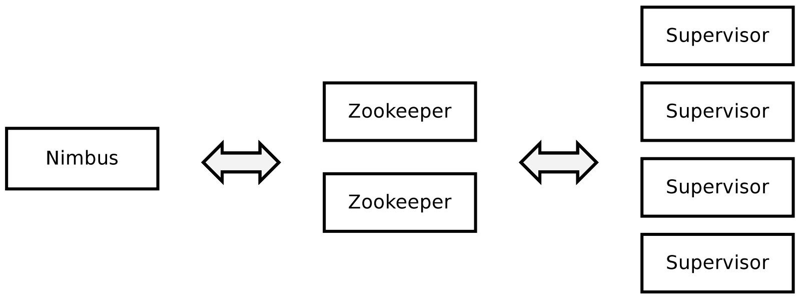


Figure 2.10: Storm cluster consists of nimbus, zookeeper and supervisor nodes.

requires). The spawned workers will be exclusively reserved for running only tasks of the submitted topology, which do the actual stream computation.

Apart from Nimbus and Supervisors, a Storm cluster needs to run a number of Zookeeper servers. Zookeeper [36] is an open source centralized coordination service, which maintains configurations, naming and synchronization across distributed processes. Since Nimbus and Supervisors are stateless and fail-fast, Storm keeps their configuration and state in a Zookeeper cluster. In addition to Zookeepers, a fourth component, *Storm UI* can be run, which is a graphical web frontend for basic monitoring of topologies.

Other features of Storm

One feature of Storm is high resiliency through usage of reliability API. If Nimbus, Supervisors, or Workers are restarted at any time, Storm will continue to correctly execute topologies without loosing any tuples. Furthermore, Storm implements guaranteed tuple processing by keeping track of all tuples in the topology by special acknowledger tasks and resending undelivered tuples.

Although Storm is written in Clojure and runs on JVM, its topologies can be implemented in any language (though most supported are Java, Python and Ruby). This is because components in Storm are implemented as services in the Thrift framework. Thrift [37] is a framework for cross-language services development, where user defines a service interface, from which client and server implementation stubs can be generated. Using the Thrift interface, users can write applications that interact with Nimbus and Supervisors. We use this capability to connect to the UI service and hence are able to monitor topology performance.

Conclusion

The Storm programming model enables developers to specify application logic without the need to care about component distribution, their communication and the underlying network. This makes it very simple to write topologies and enables developers to focus on the application logic. However, even for simple topologies, it still requires a lot of code to be written, and there are many caveats to watch for. It is also worth noting that the Storm programming abstractions map closely to the complex event processing: the tuples in Storm can be thought of as events, Spouts can represent event sources and input adapters, and Bolts can represent complex query operators that filter, aggregate and join events.

A big disadvantage of Storm is that it does not support migration of tasks across machines to improve utilisation of resources. Once a topology is submitted, a task will be fixed to one worker (i.e. one machine) until a manual rebalance command is issued. However, rebalance command just reassigns workers in a round-robin fashion, not taking into account resource use at each node. Another disadvantage of Storm is that as a relatively young framework, it still changes a lot, and its community is relatively small[2].

2.6.2 InfoSphere Streams

InfoSphere Streams (also known as System S) [16, 17] is a commercial large-scale distributed data stream processing middleware developed at IBM Research. Similarly to Storm, user can define data flow graphs that consist of a set of *processing elements* connected together with data streams. The processing elements implement data operations and their connection is specified in a configuration file as pairs of input/output ports. However, System S does not have the capability to parallelise processing elements and does not implement stream groupings, all of which would have to be done manually.

The data flow graphs in InfoSphere can be specified in three ways: using Java or C++ generic API, using SQL-like query language called SPADE [17], or by simply selecting from a number of predefined domain-specific enquiries. SPADE can be used to detect complex events by combining many available operators, such as aggregation, join and functors. Queries specified in SPADE, as well as predefined enquiries will be first compiled into data-flow graphs and then deployed. As a result, the system can be considered to be both a DSMS (because of SPADE) and a general stream processing framework (because of generic processing API). Its disadvantage is that it is more heavyweight than Storm, lacks parallelism of processing units, and most importantly is not open source.

2.6.3 S4

S4 [18, 19] is a free stream processing platform developed at Yahoo and is considered the predecessor of Storm and InfoSphere systems. Stream applications in S4 are graphs of connected processing elements (PEs), each consuming input from a queue, processing it and possibly sending it to another PE. As in Storm, this graph specifies only the logical computation, and runtime organization is determined by the platform. The events in S4 arrive on named streams and contain a key and attributes. Each PE processes only events from one stream with one specified key. As a result, the S4 runtime will create one PE for each possible combination of stream/event key pairs. The PEs will be then equally distributed across available machines and events with the same key will be routed to the same PE for processing.

[2]However, recently there are many ongoing related projects - e.g connecting Storm to Kestrel, RabbitMQ or using it together with Hadoop.

S4 is less powerful than Storm, since it does not support guaranteed message processing, and computation graphs are inconveniently specified in Spring XML files, which have to be submitted manually (in Storm topologies are submitted programmatically). Also, S4 supports routing events only on keys, which is implemented in Storm through field groupings.

2.7 Cloud platforms for evaluation

Once developed, it is necessary to test and evaluate a CEP system in a target cloud environment. In this section we describe Emulab and Amazon EC2 clouds as candidates for the system's evaluation.

2.7.1 Emulab

Emulab [39] is a free network testbed in University of Utah containing around 500 PC nodes. The available hardware is very variable, ranging from computers with 2.4GHz Quad Core Xeon to older Pentium III processors. Emulab is a public facility that allows researchers to run networking and distributed systems experiments typically on a scale of tens of nodes. The experiment sizes are usually limited to the availability of nodes, shared across many research groups. When creating an experiment, users specify a set of machines with required OS images and a network topology between them. Custom OS images can be created, though they will be visible and available to other users. The nodes in Emulab cluster run emulation software, which can throttle system resources as required by an experiment (e.g. it can emulate link loss rate and network bandwidth). Once experiment is swapped in (nodes are allocated), users can use SSH to interact with each physical machine. Also, there is no graphical interface for monitoring resource use. The disadvantage of Emulab is a big demand for high-end PCs, sometimes resulting in a relatively small experiment not being able to swap in, and common downtimes[3].

2.7.2 Amazon EC2

Amazon EC2 [38] is a paid service that provides resizable compute capacity in the cloud. When using EC2, a user configures an image and chooses an instance type depending on required number of computational units, memory and storage size. The service is charged accordingly. EC2 can automatically scale an instance depending on resource utilisation and also provides a resource monitoring service (Amazon CloudWatch). The advantage of EC2 is its reliability (guaranteed 99.95% uptime) and availability. The disadvantage of EC2 is that it provides fully virtual environment for users - users receive virtual CPU cores and not a whole physical PC, as it is in Emulab. Also, as compared to Emulab, users cannot specify and emulate network bandwidth, topology and packet loss.

2.8 Conclusions

In this chapter we introduced many systems that implement some form of complex event detection: High throughput publish/subscribe systems enable for event selection and filtering. Active databases can handle a small number of very expressive event queries, but work mainly

[3]In short time we encountered more of these than expected - nodes not being able to boot, nodes do not react to SSH, there are power outages and most commonly, the web interface is down

on historical data. Distributed stream management systems provide a generic method to work with streams, support many expressive queries and also are equipped to process complex events. Expressing these with generic languages could however be cumbersome. Finally, dedicated CEP systems are optimized for high throughput and a large number of queries, and have languages tailored for this task. All of these systems implement different data models, detection strategies and optimizations.

The data model of events is usually based around a fixed schema, specifying the count and type of event attributes. The events also carry timestamps and sometimes keys that uniquely represent them (e.g. in Borealis). Some systems timestamp events upon arrival (e.g. active databases), while others assume arriving events to be timestamped by event sources (e.g. dedicated CEP systems). Some implementations only support instantaneous events with one timestamp (e.g. SASE), some can cater for duration events by including two timestamps (e.g. DistCED), and others include even more timestamps (e.g. Next).

Techniques for event detection vary greatly. These include finite state automaton (e.g. in Cayuga and Next), Petri nets (e.g. in SAMOS), detection graphs (e.g. in SNOOP), and data-flow graphs (e.g. boxes and arrows in Borealis or pipelines in SASE). Some systems are also able to distribute event detection across many nodes. This can be done on a query level (e.g. in Cayuga), operator level (e.g. in Next and DistCED), or a combination of the two (e.g. Borealis). Distribution techniques depend on how the underlying detection model can be partitioned into independent parts. Parallelism is implemented by distributed Cayuga, and also in a limited form by DistCED.

Some optimizations introduced in presented CEP systems can be applied in general: pushing window and predicates up or down in a query tree, reusing already deployed operators, moving operators to different nodes depending on utilization, rewriting queries into a more efficient form, and combining, distributing or reordering operators. Furthermore, some systems use cost models for optimization (e.g. Borealis and Next) and also develop important approximation techniques, such as load shedding.

Finally, we have looked at available frameworks that could be used to build a CEP application and also cloud environments to evaluate it on. Here, due to its advantages we have chosen to use the Storm stream processing framework and the Emulab cloud. Note, that we only presented a small fraction of all the available CEP systems. In particular, there are other commercially available solutions. Out of these, the most known are Coral8, StreamBase, Oracle CEP, Sybase CEP or Esper (free). A list of many others, whether ECA-based or rule-based engines can be found in [30].

Chapter 3

The Step event query language

In this chapter we will describe the Step (STorm complex Event Processing) query language for specifying complex event patterns. We will explain the its goals and motivation, then specify its event and temporal models, and finally describe syntax and semantics of individual operators. We will finish with a brief discussion on design choices and possible extensions.

3.1 Goals and motivation

Our goal was to develop an efficient, scalable and fast system for complex event detection. Many CEP system implementations exist, each applicable to different scenarios. Some process only historical data (active database), some process only streaming data (dedicated CEP systems) and others are combinations of both (data stream management systems). We discovered that in general the more expressive a system, the slower and less scalable it is. In the extremes, advanced messaging middleware is the fastest system for filtering of events, whereas expressive data stream management systems are much slower. Some solutions (e.g. dedicated CEP systems) design their event detection languages to enable query optimizations and query distribution. As such, they sacrifice some expressiveness for higher throughput (e.g. Cayuga and Next) or better operator placement near the event sources (DistCED).

Ideally, an event detection language should be expressible with regards to many scenarios, be simple enough to use, and be structured in a way that enables easy implementation, optimization, distribution and detection of events in parallel. It should be noted that all of the required attributes are not achievable and that the design of a query language is a difficult process of finding a balance between them. For example, expressibility does not favour distribution and parallelism. Also, design for convenient usage often results in a language, of which syntax does not resemble the runtime structure of event queries, leading to complicated compilers and optimization techniques.

We wanted to explore complex event detection in environments where there is abundance of computational resources (e.g. cluster environments), and very high event throughput is required. We did not necessarily aim for being able to process high number of event queries (e.g. Cayuga is optimized for this), since, given enough queries, we can always saturate available resources. Our focus was rather on how single queries can be distributed and parallelised, thus enabling extreme throughput and scalability to all of the available hardware. As such, the design goals of the Step event detection language were:

- *Good expressiveness* with regards to a fixed set of scenarios. We did not attempt to create a language applicable to all complex event detection scenarios, but rather implemented a small number of expressive operators, which can be used with regards to some scenarios.

- *Rigid structure.* We wanted to design the language to be well structured with respect to its runtime query representation. As a result, the implementation of its compiler, parallelism and optimization techniques would be simpler.
- *Ability to distribute and parallelise event detection.* We aimed for operator semantics that allows for detection of events in parallel.

The expressiveness of semantics was chosen according to two CEP-applicable domains: *mobile/financial fraud detection* and *financial stock monitoring*. Both of these scenarios exhibit characteristics of CEP systems. They deal with continuous real time data and require fast event detection due to high input event rates. These domains require the inferring of high-level event knowledge in a form of event patterns. Also, the input events show strong temporal relationships (e.g. the order of financial transactions made). In general, scenarios of fraud detection, network intrusion detection, and monitoring in their various forms are common applications of CEP systems. They are often discussed at CEP blogs (e.g. [14]) and various commercial systems include these scenarios in their sample use cases (e.g. use cases of StreamBase in [27]).

The most common type of mobile fraud is cloning. The attacker steals identity of user's phone and programs his device to identify itself as the user. As a result any expensive calls made by the attacker are billed to the victim. Companies try to deal with these frauds by usage profiling (keeping history of calls for a user and spotting deviations), by detection of duplicate calls (multiple calls from the same number at the same time), or by velocity traps (calls from the same number within short time but very different locations). Other detection techniques and types of mobile frauds are explained in [31]. Credit cards and financial transactions are also subject to cloning and identity thefts, and are detected by similar tools. Here, attackers often follow the same patterns, specifiable in a form of rules. For example, when attacker clones a card, he will make a number of small online transactions to test it, followed by one or two big amount transactions, possibly in different currencies. If such pattern is spotted, user will be required to confirm these, for example by a text message. Examples of credit card fraud and its detection can be found in [32].

In general, detection of fraud happens using different tools at once. For example, usage profiling is often done by using neural networks and other kinds of machine learning systems, whereas spotting patterns of particular attacker behaviour is done by CEP systems. In these scenarios, the CEP systems are usually required to detect sequences or overlapping of events within some time constraints and satisfaction of some predicates. On the other hand, financial stock monitoring requires specification of trends (for example that the stock was rising for the last hour), and filtering on long-run statistics. We will now describe the design of our language for the detection of some of these patterns.

3.2 Event and temporal model

Events in Step are triples $\langle p, t_0, t_1 \rangle$ that continuously arrive from external sources at some specified mean event arrival rates. Throughout the report we refer to these events as to *external* or *primitive* events. Here p corresponds to event's *payload*, which contains values that correspond to event's attributes, as specified in an event schema. The values are populated by an input source (e.g. by stock market or messaging system adapters), and their types and order of occurrence must be the same as declared in the event schema. For efficiency and low communication overhead, the payload does not carry attribute names or their types. As we will see later, these can be effectively inferred.

Apart from the payload, each event carries a pair of timestamps t_0 and t_1 that are populated by event's originator. These are arbitrary integer values representing respectively the start timestamp and the end timestamp of the event, where timestamps follow logical time (i.e. they do not need to correspond to system time). We require that start timestamp is smaller or equal to the end timestamp, i.e. $t_0 \leq t_1$. The pair of timestamps allows us to handle not only instantaneous events (when $t_0 = t_1$), but also duration events (when $t_0 < t_1$). This approach is very similar to Cayuga or DistCED solutions. When we compare this to systems with only one timestamp, this allows for better expressibility and more interesting operator semantics. However, it does not incur high overhead, as compared to systems that include multiple timestamps (e.g. Next CEP).

Furthermore, we require all events on the same input stream to be ordered by their end timestamps. Once an event was received with timestamp t, only events with timestamps t', such that $t' \geq t$, can be received on the same stream. The start timestamp of an event can be arbitrary even in the cases when end timestamps are equal. We can afford these semantics since the key algorithms proceed only if the end timestamps are strictly bigger. The choice of ordering input events on end timestamp, instead of start timestamp was not arbitrary. When using end timestamps, operators detecting sequences of events can be optimized for faster detection, but require more memory to store pending events. When using start timestamps, the detection at these operators takes much longer, but we can be more efficient at event garbage collection that cannot be matched any more. We have decided to use end timestamps, since we require high-throughput and memory is usually available.

Since events include two timestamps, some care has to be taken when defining their ordering. That is, answering the question "What does it mean for an event to occur after another event?". To specify the semantics of various operators, we need to make this clear. First, let us specify the set of all possible external events as:

$$\mathbb{E} = \{\langle p, t_0, t_1 \rangle \mid p \ is\ a\ valid payload \ \wedge \ t_0, t_1 \in \mathbb{N} \ \wedge \ t_0 \leq t_1\}$$

Then we can define an ordering on events $\prec$ as a tuple $(\mathbb{E}, <)$, where $<$ is the usual ordering on natural numbers and:

$$\forall e, e' \in \mathbb{E}.\ e \prec e' \ \Leftrightarrow \ t_1 < t'_0, \ \ where \ \ e = \langle p, t_0, t_1 \rangle \ \ and \ \ e' = \langle p', t'_0, t'_1 \rangle$$

This means that an event e precedes another event e' only if the end timestamp of e is smaller than the start timestamp of e'. This is a partial ordering since overlapping events or events with the same end timestamp are incomparable. Take an example from Figure 3.1. Here $E1 \prec E3$ and also $E2 \prec E3$, but $E1 \nprec E2$ and $E2 \nprec E1$.

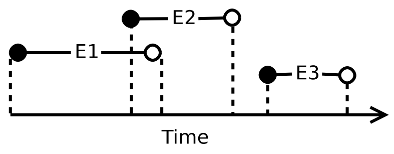


Figure 3.1: Illustration of interval timestamps for events.

Some operators in the Step language receive input from multiple streams and output a complex event, which contains values from all composed events. To be able to specify what is the result of these operators, we define the notion of event composition. A composition $\sqcup$ of two events $e^x = \langle p^x, t^x_0, t^x_1 \rangle$ and $e^y = \langle p^y, t^y_0, t^y_1 \rangle$ is an event $e^{xy} = \langle p^{xy}, t^{xy}_0, t^{xy}_1 \rangle$, such that:

$$t^{xy}_0 = min(t^x_0, t^y_0) \ \wedge \ t^{xy}_1 = max(t^x_1, t^y_1) \ \wedge \ p^{xy} = [p^x, p^y]$$

Here $[p^x, p^y]$ is a composite payload, which contains both payloads from composed events. Also we will use the following shorthand:

$$e_1 \sqcup e_2 \sqcup e_3 \sqcup ...e_n \equiv (...((e_1 \sqcup e_2) \sqcup e_3) \sqcup ...e_n)$$

3.3 High-level event pattern language

The first step in specifying an event query is to create a file containing a sequence of declarations. We will call this file a *Step program*. The program has to start with a declaration of its unique name, also referred to as the *topology name*, and contains two sections - *event schema definitions* and *event query definitions*. The former specifies the schemas of events that continuously arrive on streams from source adapters and the latter describes the complex event queries that should be run against continuous streams of input events. An example of the simplest Step program is shown here:

```
topology "Stock Quote Filter"

external Stock(name: string, price: int) < "SourceAdapter" [25000.0]

Filter(S.name, S.price): Stock/S[S.name = ’MSFT’ && S.price > 100] > "OutputAdapter"
```

This program is known under the unique name “Stock Quote Filter” and outputs all stock quotes of Microsoft that have value above 100. On the second line we see a schema definition of an input stream called “Stock”. The events that arrive on this stream are provided by the “SourceAdapter” and carry a name value of type string and a price value of type integer. Also, we see that the mean event arrival rate for this stream is 25,000 events per second. The last line of the example contains an event query called “Filter”. The query outputs the event attributes S.name and S.price into “OutputAdapter”, which is a component that collects detected complex events. The event pattern renames the Stock stream to S, and the filter specified in square brackets filters out events with name “MSFT” and price above 100.

The first thing to notice is that we do not use an SQL-like syntax. Our first implemented prototype featured an SQL-like language, however it proved itself to be very verbose. We wrote a large amount of event queries for the purpose of acceptance testing, which would be too tedious with an SQL-like syntax. Instead, our intention in the final implementation was to have a very concise language, where simple queries fit into one line. It should also be noted that the structure of the query does not resemble common *select - from - where* paradigm, but rather omits the *where* clause and is similar to the *select - from - publish* style used by Cayuga. In fact, since Cayuga had a very well structured queries that were easy to compile into abstract concepts, its syntax was the main source of inspiration for the Step language.

3.3.1 Event schema definitions

Event schemas are specified prior to execution and stay fixed during the complex event detection. The main reason for this restriction is that we later perform optimizations depending on attribute ordering within an event. Each input source has exactly one event schema, which declares the types and names of event attributes, as well as their unique name. We refer to event schemas also as *input stream schemas*, as they fully describe the system input streams. The syntax for event schemas is following:

```
external Event-Name(Attr1 : TYP, Attr2 : TYP, ...) < "Adapter-Class(params)" [EventRate]
```

The external keyword is used to indicate that events come from an external event source and thus are primitive. Event-Name identifies an input stream and the attribute values carried by an input event are specified in round braces. This is a list of attribute - type pairs, corresponding in the order of the values that an event will carry. The attributes can be of type *string, integer, boolean* and *real.* Additionally, the event schema specifies an *input adapter*, which is a component implementing an interface for event retrieval from input streams (the '<' and later '>' notations were adopted from Unix pipes). The input adapter can also take parameters, which are provided to it at its initialization. Finally, an event schema also contains an estimated input event rate, which is the mean number of events that are estimated to arrive to the CEP system in a time window.

The event rate must be specified in events per second and is a real number. Since event timestamps and query time constraints are arbitrary numbers, the rates could theoretically be in arbitrary units. In practice, however, a specific time window is required to correctly implement event throttling and stream flushing (described in implementation chapter). Furthermore, the ideal implementation would infer the event rates at runtime by counting incoming events. However, since Storm framework does not allow for dynamic topology changes and thus parallelism changes, this is not possible and we have to determine most of the topology properties at compile time.

An interesting question is whether each adapter should be used only once in the event schema definitions. Using two streams from the same adapter under different names has only the effect of renaming them, but this can also be accomplished with the renaming operator. As such, we require one to one correspondence between streams and input adapters.

3.3.2 Event query definitions

After event schemas are defined, an arbitrary number of event queries can follow. The general syntax for an event query is:

```
Complex-Event-Name( Projection ): Event-Pattern > "Event-Sink-Adapter(params)"
```

This syntax follows the *select* - *from* - *publish* paradigm of Cayuga. The complex event query needs to specify a *unique identifier* for the event, a *projection pattern* (the *select* part), an *event pattern* of interest (the *from* part) and a *sink adapter* for receiving of detected events (the *publish* part).

The *sink adapter* is specified by its class name, should be located in a specific directory, and can be parameterized. This introduces more flexibility and allows different queries to use the same sink adapter class, but instantiated differently. We do not require one to one correspondence between a complex query and an event sink adapter.

The *projection pattern* describes the set of fields that should be output upon detection of a complex event, and is implemented by a *projection operator.*

The *event pattern* specifies a complex event, which the user is interested in detecting. The pattern consists of *external operators* composed with *event detection operators.* External operators are the basic building blocks of event patterns, and can be seen as streams on which external events arrive.

Apart from projection and external operators, the Step language contains five additional event detection operators: *union*, *next*, *conjunction*, *exception* and *iteration.* Each operator takes input events from one or multiple input streams and outputs some detected events on its output stream. A detected event can be either one of the input events, or their composition. Union, next, conjunction and exception operators are binary, and iteration is a unary operator. The syntax of the event patterns formed by operators is best described using a BNF grammar form (the complete grammar for Step can be found in Appendix B):

pattern	::=	union | conjunction | next | iteration | exception | external
union	::=	pattern | pattern
conjunction	::=	pattern , [expr]? pattern
next	::=	pattern (; | $;_?$) [expr]? pattern
iteration	::=	pattern (+ | $+_?$) [expr]? pattern
exception	::=	!pattern ; pattern | pattern ; !pattern ; pattern
external	::=	event_schema_ref (/ alias)? [expr]?

As we can see, pattern can be composed from external operators by using characters '\|', ',', ';', '$;_?$', '+', '$+_?$', and '!'. An external operator contains a reference to an event schema, and provides input events for this schema. It also may be aliased or renamed by the '/' operator. Some of the event detection operators may also be annotated with an expression, which further specifies its semantics. The left-factored BNF grammar for expressions is shown below:

expr	::=	comparison ((|| | &&) expr)?
comparison	::=	arithmetic (Op_c comparison)?
arithmetic	::=	basic (Op_a arithmetic)?
basic	::=	field_ref | prev(field_ref) | dur Op_c Int | len Op_c Int | !basic | −basic | Int | Float | String | true | false
field_ref	::=	external_ref . field_name
Op_c	::=	! = | *contains* | = | <= | >= | < | >
Op_a	::=	+ | − | ∗ | /

In short, this is a standard expression grammar as seen in most programming languages, but enriched with extra constructs. Top-level expressions are built from comparisons by using logical operators && (and) and || (or). A comparison is built from arithmetic expressions using standard comparison operators: '! =', '=', '<=', '>=', '<', '>'. We have added an extra operator 'contains' for checking if a string is contained within another string (e.g. S.name contains 'MSFT'). Arithmetic expressions are built from basic expressions using arithmetic operations '+', '-', '*' and '/'. A basic expression is any number, string or boolean, as well as boolean negation (!) and unary integer negation (-).

Basic expressions can contain additional constructs that are specific to the Step language. First of these are *field references*, which retrieve value of a particular field from an event. Other constructs are specific to individual operators, which are previous event field selection (prev), event duration specification (dur), and maximum or minimum iteration length specification (len). We will explain these shortly on individual operator semantics. The following example demonstrates how field referencing works:

```
external Stock(name: string, price: int) < "SourceAapter" [25000.0]

// S1.name and S2.name are field references
Example(*): Stock/S1 ;[S1.name = S2.name] Stock/S2 > "OutputAdapter"
```

This example detects a stock quote followed by another quote from the same stream and with the same name. Here, Stock/S1 and Stock/S2 are external operators that provide external events from the Stock input stream. In order to disambiguate between the two operators we have to rename them to S1 and S2 respectively. Now it is clear that using the field references $S1.name$ and $S2.name$ we access field name of events from S1 stream and S2 stream respectively.

External operator

The syntax of external operator is following:

$$InputStreamSchema\ (/\ alias)?\ [expr]?$$

External operators are the sources of primitive events. The inputs of the operator are primitive events that arrive from input adapters, as declared in corresponding $InputStreamSchema$. The operator may rename the events using the '/' operator, or filter them on associated predicate $expr$. Only primitive events satisfying the predicate are output. Renaming and filtering is illustrated in the following event queries:

```
Renaming(S.name): Stock/S > "OutputAdapter"

Filtering(Stock.price): Stock[Stock.name = 'MSFT'] > "OutputAdapter"
```

The first query renames Stock events to S events. The second query shows how events are filtered on a predicate - in this case only events with attribute name equal to 'MSFT' are detected. The external operators in the example are $Stock/S$ and $Stock[Stock.name =' MSFT']$ respectively.

Projection operator

The syntax of the projection operator is:

$$QueryName(E1.field1, E2.field2, ...)\ \ or\ \ QueryName(*)$$

The operator is used to select a set of fields from an event and report them to the associated output adapter. There are two types of projection:

- *All projection* ($QueryName(*)$) outputs the values of all fields of all payloads that a complex event contains. These are the values of individual external events that the detected complex event contains.
- *Field projection* ($QueryName(E1.field1, E2.field2, ...)$) specifies in brackets an exact set of fields that should be output to an event sink. The selection of fields has the same syntax as field referencing, i.e. $external\_ref.field\_ref$.

Union operator

Syntax of this operator is:

$$P_1\ \mid\ P_2$$

The union operator detects the event pattern P_1 or the event pattern P_2. I.e., it outputs all events detected by at least one of these event patterns. This can be seen as the merging of output streams of operators detecting P_1 and P_2 patterns, where the resulting events are output in increasing order of their end timestamps. The operator is also commutative, i.e. $P_1 \mid P_2$ is equivalent to $P_2 \mid P_1$

Consider the scenario shown in Figure 3.2. The query $S1 \mid S2$ will output events b, c and the query $S1 \mid S3$ will output events a, b, d, e (assuming a is received on input stream before b).

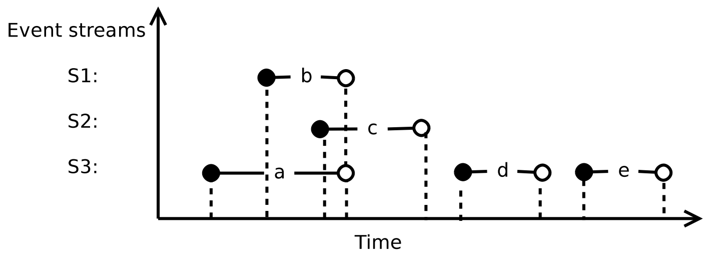


Figure 3.2: A scenario of five event arrivals from three different streams.

Next operator

The syntax of the next operator is:

$$P_1 \ ; [expr] \ P_2 \ \ or \ \ P_1 \ ;_? [expr] \ P_2$$

The operator detects an event e_1 of the pattern P_1 followed by an event e_2 of the pattern P_2, such that $e_1 \prec e_2$ and predicate *expr* is satisfied. A detected event is a composition $e_1 \sqcup e_2$ of both events. Depending on different selection policies, we define two different types of next operator:

- *Greedy next (P_1 ; [expr] P_2)* implements single selection policy, which specifies that for each event of the pattern P_1 at most one event of the pattern P_2 can be matched, and hence at most one complex event output. In other words, the operator will detect the first event e_2 of P_2 that occurred after event e_1 of P_1 such that predicate *expr* was satisfied. Similar semantics are implemented by Cayuga and Next CEP.
- *Any next (P_1 $;_?$ [expr] P_2)* implements multiple selection policy, which specifies that for each event e_1 of pattern P_1, infinitely many events e_2 of pattern P_2 can be matched, as long as *expr* predicate is satisfied and e_2 occurred after e_1. This means that for each event e_1 multiple complex events can be output. These semantics are for example implemented by detection language TESLA [28], which also includes both versions of next operator.

The difference between the two types of next operators is best understood through examples. Consider Figure 3.2, queries $S1; S3$ and $S1;_? S3$. Both events d and e occur after the event b. The any next query will output both complex events $b \sqcup d$ and $b \sqcup e$, whereas the greedy next will only output the event $b \sqcup d$.

The different semantics are applicable to different scenarios. Consider a scenario where we are interested in detecting a sequence of transactions, in which the first transaction is for a small amount and the second is for a large amount. For a transaction stream T we can define the patterns:

$$T/T1 \ \ ; [T1.amount < 50 \ \&\& \ T2.amount \geq 50] \ \ T/T2$$

$$T/T1 \ \ ;_? [T1.amount < 50 \ \&\& \ T2.amount \geq 50] \ \ T/T2$$

The first pattern can be used to detect the first fraudulent transaction per stream, whereas the second pattern can detect all the fraudulent transactions.

The predicate of the next operator can refer to fields of any events from pattern P_1 or P_2, and can contain a duration predicate. The duration predicate can be used to limit the duration of a detected complex event. For event $e = \langle p, t_0, t_1 \rangle$ the duration is calculated as $t_1 - t_0$. This allows us to detect events with some minimum duration, and bound detection for events up to some maximum duration. For example the query:

$$S1 \ \ ; [S1.name = S2.name \ \&\& \ dur > 10 \ \&\& \ dur < 100] \ \ S2$$

will detect an event from stream $S1$ followed by event from $S2$, such that both events have the same value for field *name* and the duration for the detected event is between 10 and 100 time units.

Conjunction operator

The syntax for conjunction operator is:

$$P_1 \ ,[expr] \ P_2$$

The operator detects an event e_1 of the pattern P_1 that occurs at the same time as event e_2 of the pattern P_2 while predicate *expr* is satisfied. The output complex event is the composition $e_1 \sqcup e_2$. An event e_1 occurs at the same time as event e_2 only if the events overlap, that is:

$$e_1 = \langle p^1, t_0^1, t_1^1 \rangle \ \ overlaps \ \ e_2 = \langle p^2, t_0^2, t_1^2 \rangle \ \ iff \ \ t_0^1 \leq t_1^2 \ \wedge \ t_1^1 \geq t_0^2$$

Consider the events from Figure 3.2. The query $S1, S2$ will detect the complex event $b \sqcup c$ whereas the query $S2, S3$ will output the event $c \sqcup a$. A more complex query $S1, S2, S3$ would first detect the conjunction of $S1, S2$ and then conjunct the result with events from $S3$, thus detecting event $b \sqcup c \sqcup a$.

The condition of conjunction operator can refer to fields of any event from patterns P_1 or P_2, and can also contain the duration predicate. Similarly to the next operator, the duration predicate specifies the bounds on duration of a detected complex event. For example the query:

$$Call/C1 \ \ ,[C1.number = C2.number \ \&\& \ dur < 100] \ \ Call/C2$$

could detect two calls from the same number at the same time, given that their complete duration shorter than 100 time units.

Iteration operator

The syntax of iteration operator is:

$$P+[expr] \quad or \quad P+_? \, [expr]$$

The operator detects one or more consecutive events of pattern P in $\prec$ ordering, satisfying the predicate *expr*. The operator is similar to Kleene Plus operator for regular expressions and is very expressive when correlating long sequences of events. It should be noted that for efficiency reasons iteration operator does not emit the whole detected sequence of events, but simply the first and last events (Cayuga uses this approach as well). We argue that usually only these events are of interest: for example in detecting an increasing stock price sequence, only the smallest and highest price are important (length of the sequence can be computed from timestamps).

Similarly to the semantics of next, Step also implements two variations of the iteration operator:

- *Any iteration* can be used to detect any sub-sequence of consecutive events satisfying predicate *expr*. These semantics correspond to the semantics of Kleene Plus operator or the iteration operator in Next CEP. Although any iteration results in many detected complex events, the iteration sub-sequences are detected independently, thus enabling distribution and parallelism.

- *Greedy iteration* can be used to detect the longest sub-sequence of consecutive events satisfying some predicate. In other words, the greedy iteration will detect the longest sequence of events from e_1 to e_n such that $e_1 \prec e_2 \prec ...e_n$, where e_n is the last event satisfying the iteration predicate. Once the longest sequence is output, the matching is restarted at event e_{n+1}. Greedy iteration cannot be easily distributed, since the longest sequence would have to be known at all times by all nodes.

The difference between these operators is best understood by examining Figure 3.3. The query $S3 + [len < 4]$ will detect only the sequence a, b, c and hence output the event $a \sqcup c$. It should be noted that the event will not be detected until the predicate evaluates to false (i.e. when event d arrives). After the complex event is detected, new matching will be started, at this point containing only the sequence d, e. On the other hand, the query $S3 +_? [len < 3]$ will detect all sub-sequences of events up to the length of 3, i.e. events a, b, c, d, e, $a \sqcup b$, $a \sqcup c$, $a \sqcup d$, $a \sqcup e$, $b \sqcup c$, $b \sqcup d$, $b \sqcup e$, $c \sqcup d$, $c \sqcup e$, $d \sqcup e$.

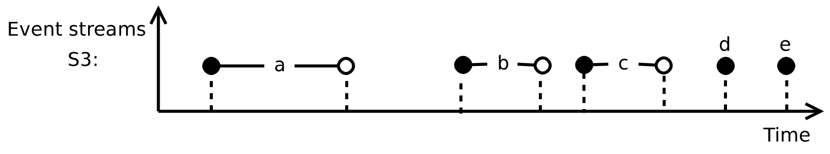


Figure 3.3: A scenario of five event arrivals from stream $S3$. d and e are instantaneous events.

The iteration predicate can be built from the same constructs as conjunction and next predicates, with addition of two special operators:

- *len* can be used to specify the minimum and maximum length of iteration sequence, which is output. For example the query $S +_? [len > 2 \ \&\& \ len <= 3]$ will output all sub-sequences from stream S of length 3. This is not equivalent to $S + [len > 2 \ \&\& \ len <= 3]$, which outputs one sequence of length 3 at a time.

- *prev(field_ref)* can be used to access a field of a previous event in the iteration sequence. This is very useful for specifying properties that must hold between two consecutive events. Because of this operator, the predicate for the first event in the iteration sequence always evaluates to true [1].

Note that the behaviour of *dur* predicate is different from the next and conjunction cases and cannot be used to limit iteration sequence, but rather refers to durations of individual events in the sequence. This decision was made for implementation simplicity and can be changed if required.

For example to detect the longest stock rises (price going up) which are at least 10 quotes long for the Microsoft company we could use the query:

$$Stock/S[S.name =' MSFT'] + \ [prev(S.price) <= S.price \ \&\& \ len >= 10]$$

Here, we first filter out all stock quotes with name ‘MSFT’ and then detect their longest price-increasing sequences. The iteration is not bounded by length and it might lead to infinite sequence (e.g. if the stock price never goes down). To illustrate the use of any iteration, we could detect all possible combinations of transactions with the same currency using the query:

$$Transactions/T +_? \ [prev(T.currency) = T.currency]$$

[1] We assume that the iteration predicate is mainly used to specify constraints between two consecutive events and will always contain previous field selection. Therefore, it is inevitable to evaluate the predicate for the first event to true.

Exception operator

The syntax of exception operator is very similar to $SEQ\_WITHOUT(S1, B, S2)$ found in SASE, which detects sequences of events $S1; S2$, but fails to detect them if an event B occurs in between. Step allows using exception only in combination with the next sequence and comes in two variations:

$$!P\ ;\ P_1 \quad or \quad P_1\ ;!P\ ;\ P_2$$

Step has two types of exception operator:

- *Exception not before ($P_1 \setminus_b P_2$)* detects an event pattern P_1 only if an event pattern P_2 did not occur earlier by the $\prec$ ordering. If an event of pattern P_2 already occurred, no complex event would be output.
- *Exception not during ($P_1 \setminus_d P_2$)* detects an event pattern P_1 only if an event pattern P_2 did not occur in between. We say that an event $e_2 = \langle p^2, t_0^2, t_1^2 \rangle$ occurs in between an event $e_1 = \langle p^1, t_0^1, t_1^1 \rangle$ iff $t_0^1 \leq t_0^2 \ \wedge \ t_1^2 \leq t_1^1$.

It should be noted that exception is the only operator where we denote syntax differently from its semantics. The syntax of exception is unary, whereas the semantics are binary. To be precise, the syntax $!P_2\ ;\ P_1$ corresponds to the semantics $P_1 \setminus_b P_2$ and the syntax $P_1\ ;!P_2\ ;\ P_3$ corresponds to the semantics $(P_1\ ;\ P_3) \setminus_d P_2$. Since we find it more natural to use a unary operator to specify a position of undesired event in an event sequence, the exception syntax is different from the semantics. The transformation of syntactic representation into semantic will be described in the implementation chapter.

The different semantics of exception operator can be illustrated by Figure 3.2. The query $(S_3\ ;\ S_3) \setminus_d S_2$ will detect only the event $d \sqcup e$. It will fail to detect events $a \sqcup d$ and $a \sqcup e$, as event c from stream S_2 occurs in between. Similarly, the $S_3 \setminus_b S_2$ will detect only the event a, but omit events d, e, as event c from stream S_2 happens before them.

The semantics of exception can be used when a detection of sequence must be cancelled because of the occurrence of some event. For example, a detection of a sequence of fraudulent transactions may be cancelled if an event occurs that they were deleted.

3.4 Conclusion

We described the semantics and syntax of each individual operator. To see how different operators can be composed to detect more complicated patterns, we present some applicable stock and mobile fraud detection queries. Suppose we are receiving stock quotes and want to detect a peak price for each company within some time window, for which the company price was initially rising and then falling for specified duration. This could be done by the following query:

```
Spike(S1.name, S1.price):
  (Stock/S1+[prev(S1.price) < S1.price && prev(S1.name) = S1.name && len > 75])
    ;[S1.name = S2.name && dur < 7200]
  (Stock/S2+[prev(S2.price) > S2.price && prev(S2.name) = S2.name && len > 100])
```

The following pattern could detect mobile cloning fraud by detecting two calls happening at the same time (duplicate trap), or two calls happening from different areas within short time period (velocity trap):

```
CloningFraud(*):
  (Call/C1 ,[C1.number=C2.number && dur<600] Call/C2) |
  (Call/C3 ;[C3.number=C4.number && dur<600 && C3.area!=C4.area] Call/C4)
```

It also calls for suspicion when a sequence of international calls happens within a short period of time, where the subsequent calls are fairly long and no national calls happen in between (the adversary first tries whether a number works and then uses it). This can be detected by:

```
Suspicious(C1.number):
  Call/C1[C1.international && dur<300]
    ; !Call/C0[!C0.international]
    ;[C2.number=C3.number && dur<7200] Call/C3[C3.international && dur>600]
```

Another common mobile fraud is when Eve (adversary) has a premium account with hidden number and performs many missed calls to victims within a short time period. The curious victims cannot see the caller number and often call Eve back at a premium price; Eve then earns money. This could be detected with the following pattern:

```
Fraud(Eve.accNo):
  (Call/Eve[dur=0 && Eve.userTyp="hidden"]
    ;[Eve.fromNo=Alice.toNo && dur<3600]
  Call/Alice[Alice.unitprice > 10])
    +[prev(Eve.accNo)=Eve.accNo && len > 10]
```

We designed the syntax of Step to be concise and well structured, such that it does not require much transformation during its compilation process. In order to have a more concise syntax of the exception operator, we however deviated a bit from the second requirement.

We argue that the semantics of Step are much richer than that of the dedicated CEP systems mentioned in the background chapter. Even though count and types of event detection operators are roughly the same, Step adapts some new ideas from recent development of the TESLA [28] language. Thus, the semantics of different Step operators can be fine-tuned by specifying different detection policies (e.g. exception before vs. exception during, greedy next vs. any next, or greedy iteration vs. any iteration) and by specializing operator behaviour by the use of predicates. As a result, the Step language is applicable to more complex event detection scenarios.

The expressiveness of Step could by further extended by adding an aggregation operator capable of performing user-defined data analytics on sequences of events. As such, it could be implemented on top of greedy iteration (an approach similar to fold operator in Cayuga), and could detect for example the mean or standard deviations of a window of recently seen events. Clearly, the data analytics functions would have to be user-definable and we would require extra syntax for this purpose. It should also be noted that aggregation operator would be difficult to parallelise and its implementation is beyond the scope of this project. Thus, we leave this idea for future work.

Chapter 4

The Step CEP design

This chapter will describe the design of the Step CEP system. First, we will introduce the general structure of the system and explain its input, internal and output streams. Then we will describe the runtime organization for event detection and how Storm is used for this purpose. We conclude with explanations of how detection is parallelised and how Step programs are compiled. This chapter intends to give a general idea of how Step CEP works. Individual aspects of the design will be later detailed in Chapter 5.

4.1 Overview

We have chosen the Storm stream processing framework as the basis for implementing complex event detection. The reason for this decision was that it seemed to be relatively powerful, simple to use, and novel [1]. As it is a novel framework, we wanted to determine its properties and applicability to CEP detection. This also proved to be the most problematic part of the project because of a lack of documentation and ongoing change in APIs. Nevertheless, we believe that we have succeeded in creating an interesting system worth studying further.

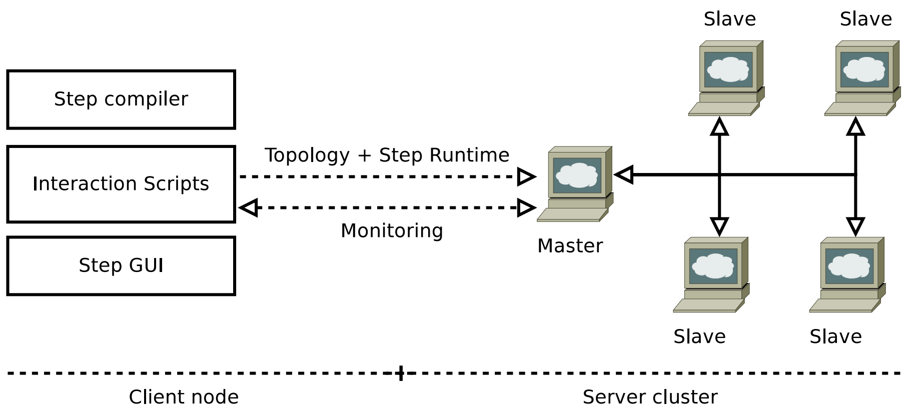


Figure 4.1: Overview of Storm topology submission and cluster structure.

Apart from Storm, technologies were chosen such that the development proceeds fast, by reusing many existing tools, choosing appropriate development languages, and reusing our industrial experience. The majority of our design decisions were governed by the Storm design and the Step language design. The most significant trait of Storm is that after topologies are submitted

[1]Version 0.5 of Storm was released only in September 2011

to a cluster for execution, they are fixed. This means that the parallelism of components, the component code, the number of workers and the overall configuration for a topology always stays the same. Due to this lack of capabilities, many sophisticated runtime optimizations or features could not be designed. However, we were able to explore many compilation optimizations and tricks, which result in highly efficient topologies.

The design at a very high level can be seen in Figure 4.1. It consists of a *client side* which interacts with a *cluster side*. We will now briefly explain how these form a CEP system together.

4.1.1 The client side

The components of the client can be divided into three categories:

- *Query compiler*
- *Cluster interaction scripts*
- *Graphical user interface*

Once a set of complex event queries were specified (the so called *Step program*), the *query compiler* is used to produce a set of *topology fragments*, which are constructs that form a base for a Storm topology. The topology fragments will then be packaged together with the *Step runtime framework* and submitted to the cluster where they will run on Storm. The Step runtime framework implements general functionality for event detection, and will be described shortly.

The client also contains cluster interaction scripts. They are used to manage cluster processes and interact with them. The scripts are for example capable of starting or stopping desired processes at the cluster, packaging topology fragments with runtime framework, submitting or killing topologies, downloading logs, or rebooting the cluster. These scripts also contain cluster-specific configuration of IP addresses, ports, directory and logs locations.

Finally, the functionality of writing Step programs and compiling, submitting or killing them is provided by the Step GUI. The Step GUI also provides ability to change configuration files and manage the cluster through interaction scripts. We have built the GUI on top of the Eclipse Rich Client Platform and will describe its architecture in Chapter 5.

4.1.2 The cluster side

The cluster contains a standard installation of the Storm framework[2]. This means that our Storm cluster needs to run a Nimbus daemon, a number of Zookeeper and Supervisor daemons, and must have the messaging middleware of ZeroMQ and JZMQ installed. Optionally, Storm UI daemons can be run to enable monitoring of topologies over http protocol.

We divided the cluster into two types of nodes: *the master nodes* and *the slave nodes*. The master nodes run replicated Zookeeper daemons, a Nimbus daemon and a Storm UI daemon. These nodes are not used to perform topology computations, but are only used for their management and storage of the Storm global state. On the other hand, the slave nodes are dedicated to running actual topologies and perform distributed complex event detection. They only run Supervisor daemons, which spawn worker processes as topologies are submitted. Experimentally, we observed that using one master node (no resiliency here) has enough resources to manage at

[2]Because of the constant API changes we have fixed to use the stable branch 0.6.2

least 15 slave nodes. Hence, for better resource utilization it is also possible to run additional Supervisor daemons on the master nodes[3].

The cluster also contains a number of management scripts, called by the client side, and a number of configuration files for Zookeeper and Storm. Furthermore, we also included some standard utilities on all nodes to enable monitoring and profiling. In particular, we use a scalable distributed monitoring system *Ganglia* [40], which runs monitoring daemons at all nodes, and statistics collecting processes at the master node. We also included *Jstad*, a daemon that enables remote profiling of JVM processes, which was helpful when determining performance bottlenecks.

4.1.3 Topology structure

Specified complex event queries are compiled into a set of topology fragments and packaged with the runtime framework into a full Storm topology. The structure of the submitted topology can be seen in Figure 4.2. In summary, the runtime framework encapsulates the key data structures with event handling and detection algorithms. The generated topology fragments then provide the framework with query-specific information, the names of event streams for example, the implementation of predicate conditions, the maximum and minimum event matching window sizes, or the stream specifications between different topology components. The distinction between the abstract runtime framework and the topology fragments simplifies the Step compiler, enables it to unit test key algorithms and minimizes the topology code size.

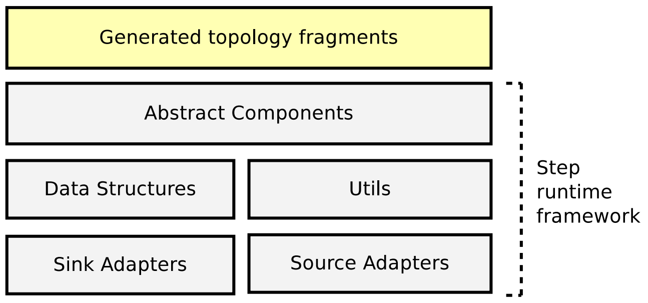


Figure 4.2: The components of a Step topology: topology fragments packaged with the Step runtime framework.

The runtime framework contains the following elements:

- *Abstract components* specify a set of abstract topology classes (abstract Bolts and Spouts) that are the building blocks of a topology and are capable of complex event detection. The topology fragments generated for concrete event queries are subclasses of these components.
- *Source and sink adapters* are used to receive external events from input sources and publish detected events to output sinks.
- *Data structures* specify the interfaces and formats of events that are exchanged between different topology tasks.
- *Utilities* contain tools to initialize topology at runtime and also provide common features, like event throttling capabilities.

[3]We did not experience any global state management issues with running topology computations on the master node.

The submission of the topology as shown in Figure 4.1 is done by a Storm client utility. The Storm client will connect to the Nimbus daemon running at a master node and upload the topology together with its configuration (e.g. its required number of workers or component parallelisms). Once Nimbus receives a topology it will distribute its complete code to all Supervisors running at slave nodes and instruct some of those Supervisors to spawn worker processes, each with an equal share of topology tasks. After topology tasks are created, the topology starts running.

4.1.4 Failure model

Before we describe how topologies detect complex events, we have to consider the type of failures the system can cope with. The Storm framework provides us with three types of reliability API:

- *Unreliable delivery* - a tuple (in our case an event or a batch of events) may not be delivered to its destination in the occurrence of crashes. Additionally, if Spouts produce tuples faster than Bolts can consume them, Bolt input queues may overflow leading to low memory conditions and kernel panic. We say that Storm does *not throttle* event queues in this case.
- *At least once delivery* - enables a tuple to be re-sent from a Spout, if it was not acknowledged as processed within some time-out. Note, that a tuple t can cause a Bolt to emit multiple tuples, and thus t can be considered as processed only if all emitted tuples were processed. Storm takes care of this situations by keeping a tree of processing dependencies for each tuple. Using this approach, it is also possible to avoid buffer overflows, as tuples are not emitted if destination queues are full. Unfortunately, this type of reliability requires twice the number of messages as compared to unreliable delivery, since every tuple must be acknowledged by every Bolt. Using time-outs for detection of lost tuples may also cause reception of duplicates, thus making it more difficult to establish correctness.
- *Exactly once delivery* - the newer versions of Storm also introduce transactional processing of tuples that guarantee that every Bolt will process a tuple exactly one time. However, this requires more messages and additional global state, as compared to at least once delivery.

It should be noted that the underlying ZeroMQ message middleware delivers messages in the order sent and implements a quasi-reliable channel, i.e. it does not loose messages if nodes do not crash. Also, note that even if we use the exactly once reliability API, a correctness of complex event detection is difficult to establish in the presence of node failures. This is because operators need to keep some internal state (e.g. a window of events seen so far), that is wiped out in the case of a crash. As a result, we would have to store this state at a permanent storage, or share it through Zookeeper, thus leading to severe performance slowdown.

Instead, Step tries to implement safe best effort semantics by using at least once delivery API, without tuple replaying in the case of crashes. The safety semantics are specified by the following axioms:

- If a complex event is detected, its pattern of events is guaranteed to have occurred even in the presence of crashes.
- Without the presence of node crashes and load shedding (system overload), if a desired pattern of events occurs, a complex event will be detected.
- Network buffers never overflow or lead to unexpected node crashes.

Since tuples are never replayed leading to duplicates, we can guarantee that when a complex event is detected, the desired pattern must have occurred (i.e. the first axiom). The second axiom justifies that the implementation of complex event detection is correct. The third axiom is guaranteed by Storm API, since tuples are never sent if destination input queues are full. However, the message overhead of at least once delivery API was experimentally determined to halve event throughput if each tuple corresponds to only one event. We have solved this problem by introducing batching optimization, making the overhead of this API insignificant.

4.2 Event streams

Events to Step CEP arrive on continuous input streams from *input adapters* and detected events are consumed by *output adapters*. We will now briefly discuss the Step input, output and internal streams.

4.2.1 Input streams

The input to the system is handled by Storm components called Spouts. As seen in Figure 4.3, Spouts receive events from external sources by means of input adapters, and distribute them to different Bolts corresponding to operators detecting events. A single Spout may emit input events to multiple Bolts.

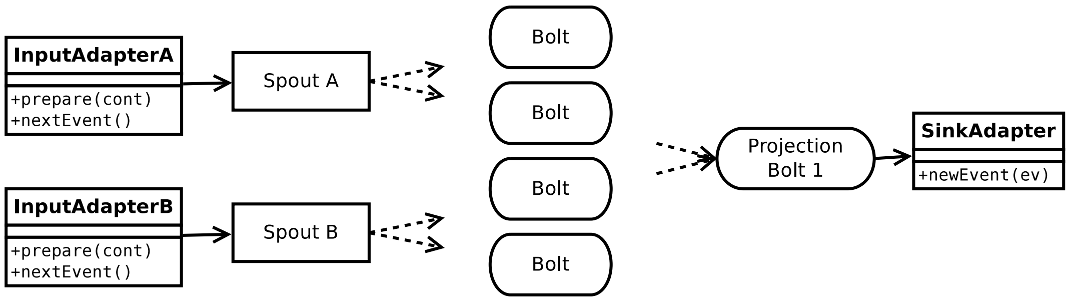


Figure 4.3: The input and output of Step runtime framework.

Input adapters are user defined, enabling data to arrive from a variety of sources, for example sensors supplying measurements, databases providing historical data, or event queues providing data from stock exchanges. A structure of an external event provided by an input adapter can be seen in Figure 4.4. As explained in the previous chapter, the event carries start and end timestamps, and a list of values corresponding to some schema as defined in the Step program.

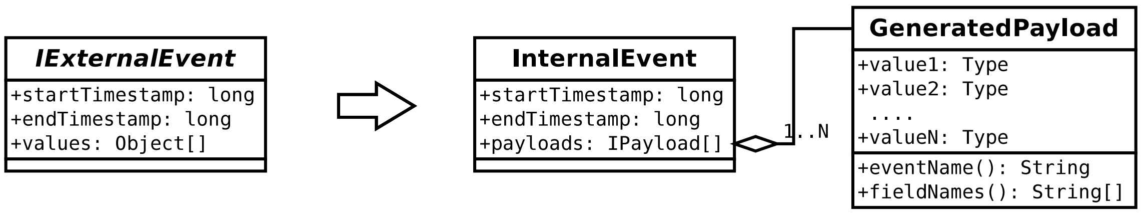


Figure 4.4: Transformation of external event into an internal Step event.

Since detection in Step is based on Storm framework, which implements a *pull-based* data input, input adapters are required to be pull-based. That is, the adapters are being called to request new events, instead of them calling the Step framework. This approach is justified by the fact that events will typically already reside in some system queue (for example JMS, Kestrel or Kafka queues), from which they can be pulled. If this is not the case, a push-based adapter can be turned into a pull-based by introducing buffering of events. Thus, our choice is purely an implementation simplification.

Recall that we require one to one correspondence between input streams and input adapters. Each input adapter is therefore handled by a single Spout. As seen in Figure 4.3, Spouts are responsible for instantiating input adapters, pulling external events from them and feeding them into a topology. Upon creation, a Spout will instantiate the corresponding input adapter by calling the prepare method on it. The input adapter should connect to any external source of events and be ready for their consumption. Once prepared, the Spout will only request new events from the input adapter if it is capable of emitting more events to Bolts without their input queues overflowing. Events must be provided by input adapters in the order of increasing (recall our event model) end timestamps and must contain values that correspond to a schema as defined in the Step program. The schemas will be used by Spouts to translate external events into an internal representation. If an input adapter is not capable of providing a new event (e.g. the event did not arrive yet or connection to source failed), it should return null events.

Note that to handle high event rates (circa above $8x10^5$ events per second) we require input adapters to be parallelisable. This is because for high input event rates more than one task for a Spout will be spawned, such that input can be consumed in parallel. Each task will instantiate one input adapter and will request events from it in the order of increasing end timestamps. It may be the case that different instances of the same adapter will run on different nodes. Therefore, upon instantiation we provide the adapter with information about topology parallelism (in particular, the adapter parallelism and the index for this adapter among its other instances). A simple strategy for an adapter handling parallelism n is each adapter instance only providing events where end timestamp modulo n is the same as adapter index[4].

4.2.2 Internal streams

After an external event is received, a Spout will transform it into an internal event as shown in Figure 4.4. All events in Step are uniformly handled as internal events and have the structure illustrated in Figure 4.5.

An internal event can be seen as a composition $\sqcup$ of one or more external events. It contains a start timestamp, end timestamp and payloads. In Figure 4.4 we also see that internal payloads are capable of reporting the name of the event stream from which they originate and the attribute names for the values they carry. This information is needed when reporting detected events to event sinks, but for network efficiency we do not include it in payloads. Instead, since event schemas do not change at runtime, Step is able to compile a new payload class for each declared external event. The payload class contains fields with types as declared in the event schema, and also contains hard-coded event and field names. Thus, when an event is sent over the wire, only value fields are sent, but not their names. When a Spout transforms an external event into an internal one, it first instantiates the corresponding payload and then fills its fields with values from the received external event.

Each operator in Step receives a number of internal events on internal streams and can output a number of internal events to all of its output streams. Thus, internal events are also composable by using the $\sqcup$ operator - the composed event $e_1 \sqcup e_2$ will contain all the payloads from event e_1 followed by all the payloads from event e_2.

[4]For example, at parallelism 2, first adapter instance would provide events with odd end timestamps, whereas second with even end timestamps.

4.2.3 Output streams

After a complex event was detected, it will be reported by a projection operator to the corresponding event sink adapter, as seen in Figure 4.3. Similarly to input adapters, event sinks cannot be singleton objects, but should be designed to run in multiple instances. This is because for high event rates projection operators will be parallelised, and each projection task will instantiate its own sink adapter. This will also be the case when different projection operators (queries) report to the same sink adapter class. Thus, it may happen that different adapter instances will run on different physical nodes.

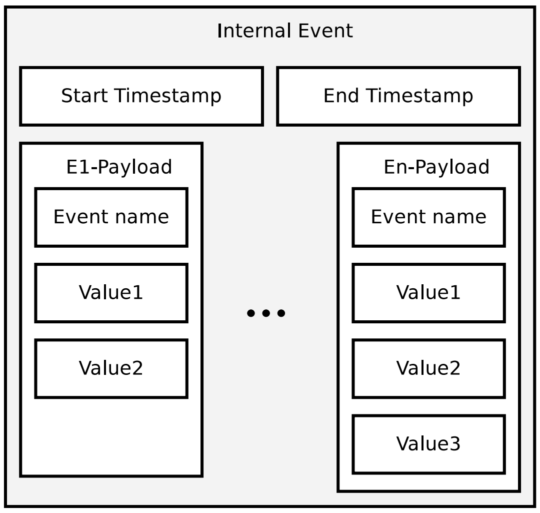


Figure 4.5: The structure of internal events in Step runtime.

The structure of reported complex events can be seen in Figure 4.6. Since detected events consist of multiple external events, each having their own fields, the reporting format is different from external event format. An output complex event contains start and end timestamps, and a list of fields. A field is described by its name, value, and a name of an external event stream, from which the value originates.

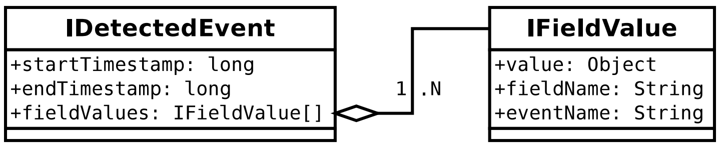


Figure 4.6: The format of complex events reported to event sinks.

4.2.4 Event ordering

The assumptions of our event model specify that events on input streams are received in the order of increasing end timestamps and also detected events are emitted to output streams in the same order. The guarantees given by Storm also ensure that events are received in the order sent (i.e. streams do not reorder events). To simplify the implementation of event detection, we strengthen these guarantees and maintain the following three event ordering invariants:

Ordering invariant 1. *Events that are emitted by any topology component are emitted in the order of increasing end timestamps. If an event was already sent with end timestamp t, no event with timestamp $t_0 < t$ can be sent.*

Ordering invariant 2. *Events that are received on each stream (input, internal or output streams) are received in the order of increasing end timestamps. If an event was received with end timestamp t, no event with timestamp $t_0 < t$ can be received from the same stream.*

Ordering invariant 3. *The event matching algorithms implemented by individual Step operators process the events in the order of increasing end timestamps.*

The first invariant extends the input stream ordering assumption to internal and output streams. This is maintained by the implementations of individual Spouts and Bolts. Given the first invariant holds, the second invariant is guaranteed by Storm and ZeroMQ. The third invariant simplifies event detection algorithms and is maintained by an event stabilization algorithm, which will be described later.

4.3 Event detection

4.3.1 General idea

The event detection in Step happens on data-flow graphs, which is a method also used by some DSMS systems (e.g. Borealis). Data-flow graphs consist of boxes connected by streams, where each box performs some computation over its input streams and may emit some events to its output streams. It can be seen that data-flow graphs exactly resemble the Storm topology paradigm, and hence are the best design choice when using Storm as an underlying framework. From now on we will refer to event detection graphs as *Step topologies*.

Step topologies consist of Spouts and Bolts connected by streams with a specified grouping[5]. We already know that one of the responsibilities of Spouts is to handle event input and projection Bolts handle event output. In fact, each topology component (whether Bolt or Spout) corresponds to some operator in the Step language and performs its function. For example, Spouts are implementations of the external operator and projection Bolts of the projection operator. An illustration of a general topology built from operators can be seen in Figure 4.7.

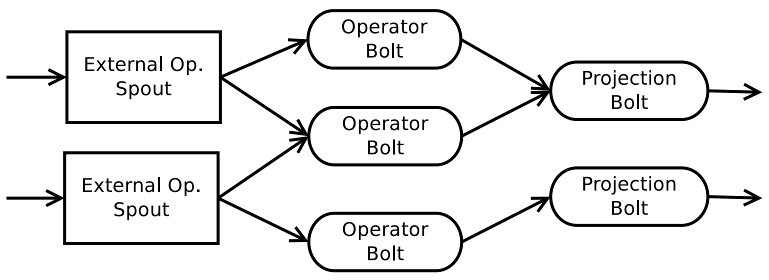


Figure 4.7: A Step topology containing components that correspond to operators in the Step language.

Each topology component acts upon a number of input streams and may emit some events to its output streams. The events from input streams are received on a single input queue. Each component works as follows: it receives an event on its input queue, possibly decides from which input stream it originated from and then does some processing on it - for example matches the event to already received events or filters the event on a predicate. Afterwards, the component may decide to emit one or more events that will be sent to all of its output streams and delivered to other components. Before we describe how individual components implement functionality of Step language operators, let us consider an example of Step queries and describe some peculiarities of their translation into a topology:

```
Query1(*): (A | C) ; B > "..."
Query2(*): B > "..."
```

The topology for the above queries can be seen in Figure 4.8. Firstly, note that even though we have specified multiple event queries, for efficiency they are translated into a single topology.

[5]Stream groupings will become only relevant when we start parallelising individual components. This will be described at the end of this chapter.

The topology contains only Spouts for the external operators that occur in the event pattern (here event streams A, B and C). The event schema could have specified more external event sources (e.g. E and F), but we do not need these to be included in the topology. Secondly, note that the queries contain two external operators for events from stream B (one in $Query1$ and another in $Query2$), but the topology contains only one Spout. Recall that we required one to one correspondence between Spouts and input adapters, and thus we cannot create two different Spouts for events of type B. Instead, we merge them into one Spout, which will then emit events to two different streams.

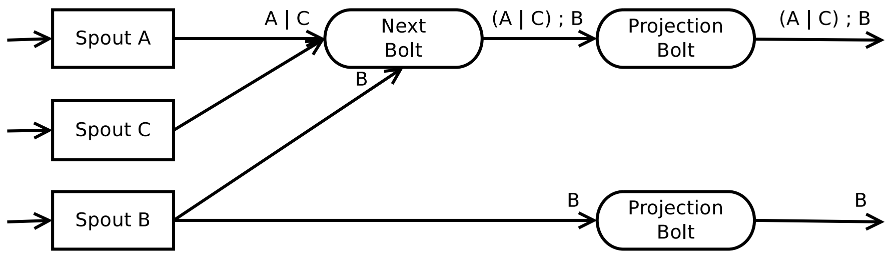


Figure 4.8: An example topology for two event queries.

Even though some operators are binary or unary, their corresponding generated topology components may consume input from more streams. This can be seen on the next Bolt, which receives events from Spouts A, C and B, but is a binary operator. The reason for this are union operators, which are implemented for efficiency as a merging of two streams. Therefore the next operator can still be seen as binary, but acting on streams $A|C$ (merged stream of A and C) and B. Also note that operators in the Step language always output only events on one stream, whereas corresponding Spouts and Bolts may emit events to multiple streams. For example, this is the case for $SpoutB$.

Please also note that the projections for $Query1$ and $Query2$ are translated into two different projection Bolts, as they correspond to different operators. The fact that they may use the same output adapter does not affect this - the projected fields may be completely different and thus it is not possible to handle such cases with only one projection Bolt.

4.3.2 Topology components

We can now explore how individual topology components implement Step operators.

External operators

External operators ($EventName$ ($/alias$)? [$expr$]?) are implemented by Spouts and their structure can be seen in Figure 4.9. A Spout has an associated input adapter, which is polled for new events. When a new external event is received, it will be transformed into an internal event. External operators sometimes need to be merged, since all events from the same stream must originate from the same component. The merging of external operators is simple, since each operator has only one output stream and only one predicate to filter events for that stream. To merge external operators, the Spout will contain multiple output streams, each with an associated filtering predicate. An internal event will be emitted to an output stream only if the filtering predicate is satisfied. This approach is highly efficient, since events are sent only after filtering was applied, minimizing the number of events in the system already at its input. The time of emitting event is linear in the number of contained external operators.

Consider for example the following queries which require external operators to be merged:

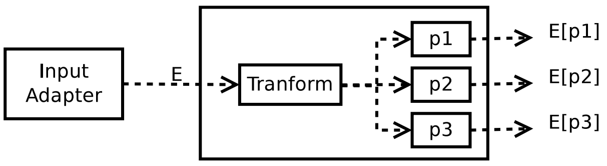


Figure 4.9: The structure of a Spout in Step CEP.

```
Query1(*): Stock/S[S.Name = 'MSFT'] > "..."
Query2(*): Stock/S[S.Price > 100] > "..."
```

First query contains an external operator that filters events on the name attribute and an external operator in the second query filters events on the price attribute. The topology merging these operators is shown in Figure 4.10.

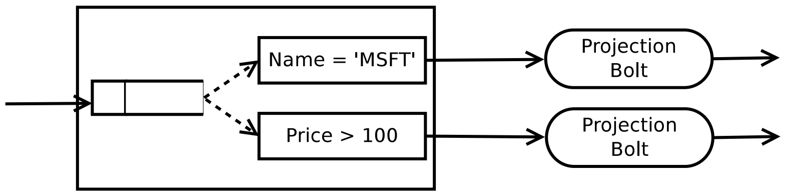


Figure 4.10: An example of a Spout, for external operators $Stock/S[S.Name = `MSFT']$ and $Stock/S[S.Price > 100]$.

Also note that the Spout does not need to handle aliases or event names, since internal events do not carry them. This information is only used during compilation to implement correct referencing and distinguish between different external operators.

Projection operator

A projection operator ($QueryName(projection)$) is implemented by a projection Bolt, as shown in Figure 4.14. The projection Bolt takes an internal event from its input queue and transforms it into a detected event, which it reports to the associated output adapter. Recall that event payloads are capable of reporting the field names of values they carry, as well as the external event to which they belong. A field projection Bolt needs to go through all payloads and their fields, and only report those that were specified in the projection. All projection reports all values to the output adapter. Thus, the time to create a detected event from an internal event is linear in the number of fields of all of its payloads.

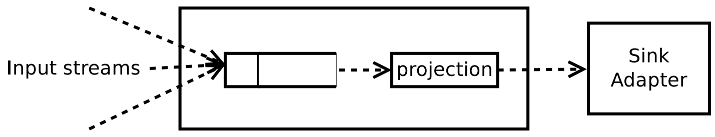


Figure 4.11: The structure of Step projection Bolt.

Union operator

Union operators are implemented by stream merging. Consider for example the union query $A \mid B$. Its topology will not contain any additional Bolts besides the projection Bolt, which will take as its input all the events from Spout A and all the events from Spout B. Because of the event stabilization described later, the events will be received in the order of increasing end timestamp, as required, effectively merging both streams.

Binary operators

All binary operator Bolts - next, conjunction and exception have the same structure, which can be seen in Figure 4.12. Events from all input streams are received on a single input queue. Depending on the stream name, a binary Bolt first decides to which stream an event belongs - whether to the *left* (or the first) or to the *right* (or the second) stream. Note that because of the union operator multiple streams can be merged into the left stream or the right stream.

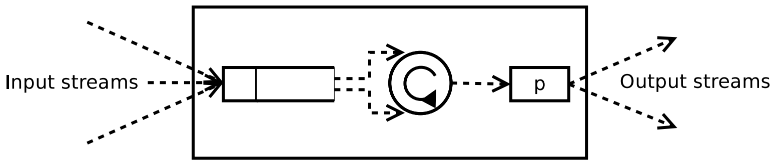


Figure 4.12: The structure of binary Step operators.

The input event will be processed according to the stream on which it arrived (left or right) and the semantics of a particular operator. An event that may be detected will be first filtered by a predicate condition. If the condition is satisfied, the detected event will be emitted to all Bolt output streams. An example of a topology with a binary Bolt for the query $!A\ ;\ B$ can be seen in Figure 4.13.

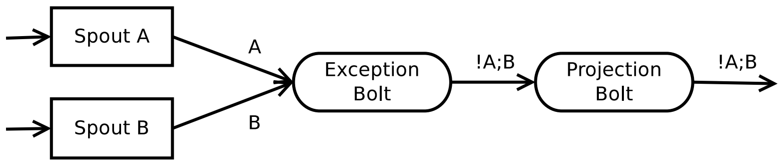


Figure 4.13: An example of exception Bolt topology for the query $!A\ ;\ B$.

The matching of events performed by each operator will be detailed in the next chapter. A Bolt will typically keep a history of events that it has already seen and new events will be matched against that history, subject to satisfaction of some time constraints. Sometimes input events will be stored for future matching and some newly detected events may be output. In the case of an exception operator, an event from the left stream will be output only if an event from the right stream was not received subject to some time constraints. In the case of next and conjunction operators, pairs of matched events may be composed into a new internal event (the detected complex event) that will be output. Recall from the event model that the internal event resulting from composition of events e_1 and e_2 will have the start timestamp $min(t_0^1, t_0^2)$, the end timestamp $max(t_1^1, t_1^2)$, and will contain all payloads from event e_1 followed by the payloads from event e_2.

It should also be noted that every operator tries to garbage collect the events in its history, which can no longer match any new events. This is usually done by exploiting time constraints specified for each operator, and leads to lower memory footprint and faster event detection.

Iteration

An iteration operator is implemented by a Bolt as seen in Figure 4.14. It receives events on an input queue and matches them against event sequences that it has seen so far. New sequences may be added, existing sequences extended, or a sequence emitted depending on the iteration predicate. The exact algorithm of this will be described in the next chapter. The output of the iteration operator is a composition of the first and last event in the iteration sequence. That is, the resulting internal event will contain the payloads of the first event followed by the payloads of the last event.

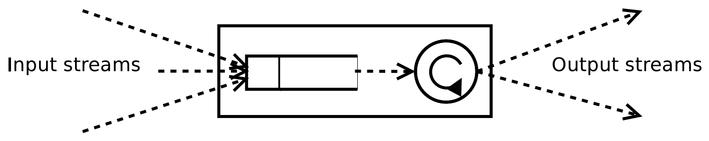


Figure 4.14: The structure of iteration Bolt.

4.3.3 Evaluation of predicates

We have seen how complex events are detected, but did not specify how predicates are evaluated against internal events. In particular, we are interested in evaluation of field accesses, event duration, and accesses of previous event in an iteration sequence.

The simplest way to access fields is by searching for them within an internal event. This is slow if events contain many payloads with many fields each. However, we can do better and access fields in constant time, by knowing an offset to a desired payload within an internal event and an offset to the desired field within the payload. This is possible, since we can determine the order of events and fields for any operator at compile time. The algorithm that achieves this is called *expression (or predicate) indexing* and will be described in the next chapter.

Event duration can simply be calculated for any internal event by subtracting the event start timestamp from the event's end timestamp. Also, accessing a previous event field in an iteration sequence is simple, since we just need to perform normal field access (using the same offsets) on the previous event in a sequence, which is kept by the iteration operator. The evaluation of other arithmetic, boolean or comparison predicates is straightforward.

4.3.4 Component hierarchy

We have described how event detection works in Step. Most of this functionality is implemented by the Step runtime framework, in particular by the abstract components part. The hierarchy of abstract components is illustrated in Figure 4.15. We can see that it contains an implementation for each variation of operators found in the Step language. These implementations can easily be replaced by alternative ones, depending on optimizations. We can also see that all Bolts are subclasses of AbsBatchedBolt and AbsSimpleBolt, and all Spouts are subclasses of AbsSpout. These abstract components implement a couple of essential algorithms, which we will briefly introduce, with their implementation being explained in the next chapter.

The first important algorithm is *event stabilization*, which ensures that events are processed by Bolts in the order of increasing end timestamps. We require this to satisfy event ordering invariants, which will lead to more efficient implementations of operator matching algorithms. Since we have no assumptions on the ordering of events between different streams, cases can occur when events will be received out of order. Consider for example a binary Bolt, which takes input from two streams, as shown in Figure 4.16. If event $B(3)$ arrives first, it cannot be processed until all events from stream A with smaller timestamps are processed. In general, an event can be processed if events from all streams with higher timestamps were received prior.

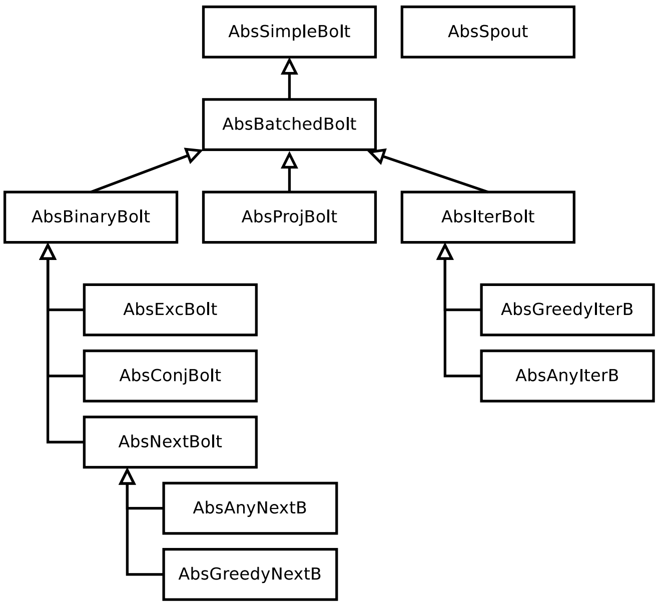


Figure 4.15: Hierarchy of abstract topology components provided by the runtime framework.

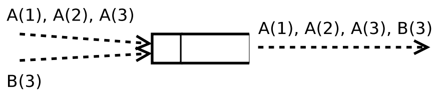


Figure 4.16: The need for the stabilization of events. Events are instantaneous and their timestamp is shown in brackets.

Another important part of event detection is the usage of *punctuations* to flush stabilization queues. Event streams might be finite, and an event with a higher timestamp may never be received. This is particularly true for an exception operator, which guards against undesired events that occur very rarely. If a desired event occurs, it must be processed by an exception Bolt as soon as possible. Thus, stream punctuations are used occasionally to flush queues in cases when the next event on a stream does not arrive within a time-out.

We shall also introduce techniques of *load shedding*, which are used to handle low memory conditions. By using Storm reliability API we already guard against network buffer overflows and having too many events in the system. However, low memory conditions can still be achieved when stabilization queues or operator matching queues grow too long. For this purpose, and also for evaluation purposes we shall additionally consider *event throttling*, which is a way of limiting of input event rates.

4.4 Parallelising event detection

In this section we will explain how Step parallelises individual components to achieve higher throughput, and also consider an alternative approach.

Through profiling we detected that the most time and CPU-consuming operation is the matching of events against events that were already seen, the evaluation of predicates and the stabilization of events. Thus, when parallelising event detection, we want each component task to process only a fraction of all events, such that CPU-usage is divided among all parallel replicas. When partitioning events between parallel operator replicas, we require the following:

- An equal share of events should be delivered to each replica. This ensures equal workload.
- Each parallel replica should handle only a fraction of all events. This allows for increased detection throughput with higher parallelism and requires a mechanism that divides events between replicas.
- No two parallel replicas can detect the same event. This avoids detection of duplicate events.
- Correctness with regard to non-parallelism. If an event is detected by a single component, then the same event must be detected also when the component is parallelised.

Parallelising complex event detection at operator level is simple, as each operator corresponds to a topology component. Each component in Storm runs in a number of tasks, which are its parallel threads of computation. That is, each task is a parallel replica of that component, detecting a fraction of all events. Storm divides input events between the tasks of the same component according to stream groupings. We will mostly be concerned with two of them - *shuffle grouping*, which divides events equally between tasks in round robin fashion, and *all grouping*, which replicates an event and sends it to all tasks. For these cases we also use the expression that events on operator input streams are shuffled or replicated. It can be seen that achieving parallelism in Storm is simply a question of dividing events between operators using stream groupings such that the above correctness criteria hold.

4.4.1 Parallelism through stream replication

Step implements parallelism through stream replication. Some event streams are replicated such that parallel tasks see the same events for correct functioning, whereas other streams are shuffled between different replicas, effectively distributing the workload.

Consider for example the binary exception operator that receives desired events and undesired events that result in the cancellation of desired events. The undesired events have to be sent to all replicas of the exception operator by using the all stream grouping, such that each operator correctly cancels desired events. However, the desired events can be shuffled between the replicas, dividing the workload. Each replica can then detect only those desired events that were received (i.e. no duplicates), but still correctly cancels them because of undesired event replication (correctness). Also, each replica handles an equal share of events, and no replica handles all events, as required. An illustration of this approach is shown in Figure 4.17.

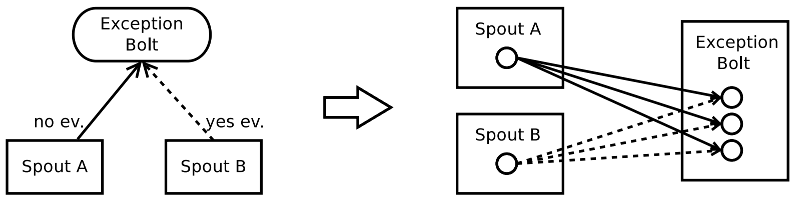


Figure 4.17: The replication of exception operator for the pattern $!A\,;\,B$. On the left side we see part of the topology and on the right its runtime organization, where exception Bolt got parallelised. The full arrows show replicated streams and dotted arrows show shuffled streams.

Similarly to the exception operator, other operators can also be parallelised, which is shown in similar fashion in Figure 4.18.

The operation of a projection operator does not depend on previously seen events, but only on the current event. Thus, its input stream can be shuffled. Each projection replica will receive an equal share of events and will work correctly, since processing is independent.

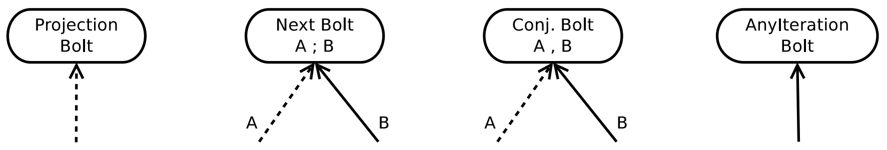


Figure 4.18: Replication and shuffling of streams for different operators. Dotted lines represent shuffled streams and full lines represent replicated streams.

In the case of the next Bolt, one stream needs to be replicated such that one replica will detect a complex event if it occurs (correctness). For a greedy next Bolt we have no choice, but to replicate its right stream, since for each event from its left stream at most one complex event may be detected (i.e. an event from left stream can be delivered only to one replica). In the case of any next Bolt the choice of which stream to replicate can be arbitrary and will depend on estimated event rates (we want to replicate streams with lower event rates).

Similarly, at least one stream of a conjunction Bolt must be replicated, so one replica will be able to spot a conjunction of events. However, since semantics of the conjunction Bolt do not depend on the event order, we can choose the stream to replicate arbitrarily. This will depend on the estimated event rates, as we will replicate the stream with the lower estimated event rate.

Recall that it is problematic to parallelise greedy iteration Bolts because of the need to share information about the longest event sub-sequence between parallel tasks[6]. However, any iteration Bolts ca be parallelised in an interesting way. First, note that the whole input stream has to be replicated among all iteration tasks, since each replica requires to see all events in order to correctly extend iteration sequences that it already holds. However, creation of new iteration sequences can be divided among different replicas. In other words, each replica will detect a sequence of events starting from different events. Consider for example having two replicas. The first replica will start matching a new sequence of events only if it receives an event with an odd end timestamp, and the second replica will start a new sequence from events with an even timestamp. As a result, the first replica will detect all sub-sequences starting with odd timestamps and second with even timestamps. It might be surprising, but evaluating shows that even though this approach results in significant increase in messaging overhead, it still improves performance of the iteration Bolt up to a certain level.

It should also be noted that Spouts can be parallelised if higher throughput is required. Since we required input adapters to be parallelisable, this constitutes no problem.

4.4.2 Alternative: parallelism through windowing

Replicating some events and shuffling others is not the only solution to achieving parallelism. This could also be achieved by using *windowing*.

Windowing is a simple technique, where events are sent on streams as part of larger groups called windows. Each window contains a fixed number of events and overlaps with consequent windows. Each detection operator waits until it receives event windows from all input streams, which form a view used for complex event detection. Any event matching algorithms will be run only on the view and then the view will be discarded. The detected events will be sent to further operators, which will go through the same windowing process. The overlap between

[6]Sharing of state between replicas is not acceptable, as it requires synchronization and leads to performance bottleneck.

windows guarantees that events at the end of the first window will be correctly matched with the events that are at the start of the following window. Windowing is illustrated in Figure 4.19.

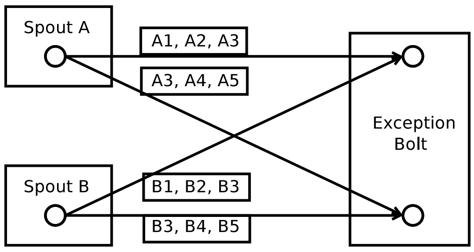


Figure 4.19: Using windowing to achieve parallelism: each parallel replica acts on a different window of events.

Widowing could be implemented by a centralised manager that would divide an input stream of events into windows. However, the manager could become a bottleneck. Instead, we suggest that windowing on Storm could be implemented using field groupings. A field grouping divides events between replicas according to a value of some field. Through the use of hashing, tuples that have the same value for that field will be sent to the same replica. We could use this mechanism to bucket events into windows according to their end timestamp (e.g. timestamps 0 - 100 will be in the first bucket, timestamps 50 - 150 in the second, 100 - 200 in the third, etc.). This can be achieved by simple modulo arithmetic on timestamps. As a result, we would eliminate the central manager bottleneck, since replicated Spouts would still correctly categorize events into buckets, and Storm would deliver the same buckets to the same operator replicas.

4.4.3 Comparison

We chose to implement the replication technique because of the following:

- Replication was not explored before and we are looking for interesting performance measurements.
- Replication results in a lower detection latency, as events can be sent to operator Bolts for detection as soon as they are available. Windowing instead requires sending events in big windows (some cases). If operator matching windows are large, the windowing technique is infeasible.
- Windowing also requires replication of events, since events in overlaps will be sent twice. Thus, for windowing to be effective, the windows must be at least twice or three times larger than the operator matching windows. In general, if matching windows are kept small, then windowing would replicate less events. If operators require big matching windows, there will be more replication.
- Windowing requires detection of duplicates since events in window overlaps may be detected twice. This may cause additional overhead.

Ideally both approaches should be considered and compared, but due to project time limitations this was not possible. We leave windowing thus for future work.

4.5 Query compilation process

We have already described how complex event detection is implemented and how Step CEP is structured. Now we will conclude the Step design by describing a high-level process of transforming Step program into a Storm topology. The compilation process, as performed by the Step compiler is visualised in Figure 4.20.

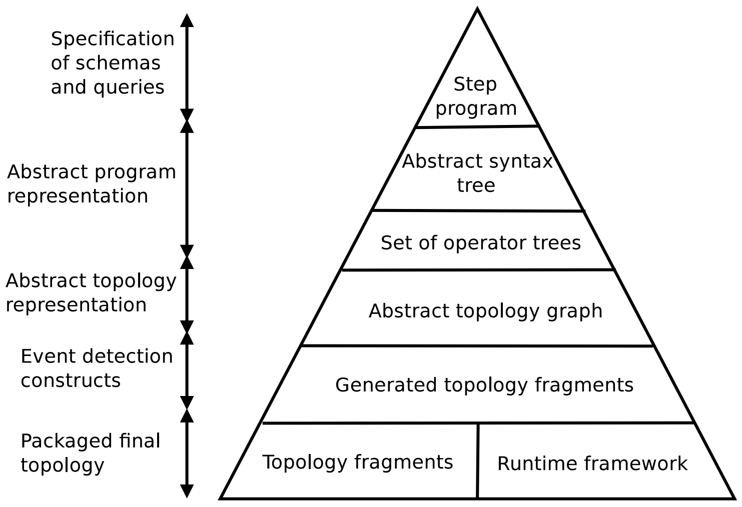

Specification of schemas and queries
Abstract program representation
Abstract topology representation
Event detection constructs
Packaged final topology
Step program
Abstract syntax tree
Set of operator trees
Abstract topology graph
Generated topology fragments
Topology fragments
Runtime framework

Figure 4.20: The compilation process of Step program.

The input to the Step compiler is a Step program, which is a collection of event schemas and event queries. The first phase of the compilation translates the program into an abstract syntax tree (AST), a tree representation of program syntax. This is done using standard parsing tools and in the process also validation of input and linking is performed. Validation of input does not only check that program is syntactically correctly structured, but also performs many semantic checks. The linking process establishes references between nodes in the abstract syntax tree.

The second compilation phase constructs a set of operator trees from the AST. An operator tree is the most abstract representation of a single Step query (levels of compilation abstractness are illustrated in Figure 4.21), which contains its operators as described in the Step language. Also, predicates are transformed into a shape that is later suitable for code generation. An operator tree is a perfect representation for rewriting queries and estimating of operator event rates.

Multiple operator trees are afterwards merged into a single topology graph, which contains Bolts and Spouts that are connected by streams. Each stream has a defined grouping, name and endpoints. Each graph node has associated names (unique file name and Bolt name), estimated event rates and parallelism, corresponding predicates, input and output streams, and source or sink adapters. Here, external operators are merged into single Spouts and common sub-queries are reused. The graph also contains structures of concrete payloads, which will be generated for each input stream.

An abstract topology graph is used to generate topology fragments. These are concrete Spout and Bolt implementations that subclass abstract components from the runtime framework. Additionally, we generate a topology configuration with a main set-up method, which defines stream wirings between different components and a big number of topology parameters.

Generated topology fragments are then joined together with the runtime framework and packaged into a topology that can be run by Storm. Each phase of the compilation process will be described in more detail in the next chapter.

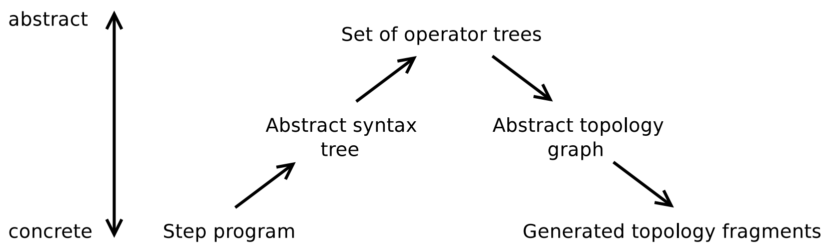


Figure 4.21: The compilation process visualised with regards to abstractness. Elements that are higher are more abstract representations.

4.6 Summary

In this chapter we outlined our method of performing complex event detection on Storm. We explained how the Step client is structured and how we organized and communicated with the cluster. Then, we described how event detection topologies look, including their input, output and internal streams, the failure model we use, and the required orderings of events. Then, we explained individual components of a topology, how they correspond to operators in the Step language, and which common event detection algorithms are provided by the runtime framework. Finally, we described how parallelism of event detection is achieved, comparing this to an alternative approach, and also outlined how Step programs are compiled into Storm topologies.

Chapter 5

The Step CEP implementation

This chapter will detail on the implementation of some aspects of Step CEP, that were already mentioned in the previous chapter. In particular, we will describe technologies that we used, explain some algorithms of the runtime framework and detail more on the compiler design. Addition of some optimizations and the structure of the Step GUI will be also described. We conclude with our approach to profiling and testing the system.

5.1 Technology choices

We used two development languages - *Scala* and *Java*. Even though Storm is capable of running topologies in other languages, at the start of the project using Java was the only documented approach. Thus, the generated topology fragments, the runtime framework and the GUI are written in Java. However we found Java too bulky for some purposes, in particular we used much more powerful and concise Scala for implementation of the Step compiler. Scala [41] is a functional object oriented language and was a good technology choice because of the following:

- Pattern matching is a powerful way to work with trees, which are common representation in compilers. It yields a powerful tool when also combined Scala extractors (used to retrieve only relevant informations from any data structure).
- Implicit Scala methods are an easy way to enrich existing behaviour (e.g. to define methods on existing AST). Also Scala case classes, singleton objects, type definitions and functional language capabilities (e.g. map, yield, fold) make development much faster.
- Code written in Scala is much more intuitive and concise as compared to Java. Triple quoted strings are ideal for code generation.
- Scala seamlessly integrates with Java. Java features like annotations, dependency injection or reflection work with Scala as well.

The Step language was developed using the *Xtext* [42] framework for development of domain specific languages[1]. Apart from lexing and parsing capabilities (provided by standard tools like ANTLR) Xtext also has:

- Validation and linking capabilities - it provides a framework where semantics of language constructs can be validated and also automatically resolves references.

[1]Xtext claims on their website that IBM InfoSphere Streams uses Xtext to also define a complex event detection language.

- Can generate an editor GUI for the specified language, which could be used to write programs in that language.
- The generated editor for the language has many advanced features, for example syntax highlighting, auto-completion, continuous compilation, outline or error markers.

Xtext is a technology based on Eclipse Rich Client platform (RCP) [46]. The compiler and the editor that it generates have a form of Eclipse plug-ins. The decision of using Xtext in combination with Eclipse RCP spared us a lot of development time with regards to client GUI (most of it was generated) and implementation of low level compiler details. The detailed structure of Step client resulting from usage of these technologies can be seen in Figure 5.13.

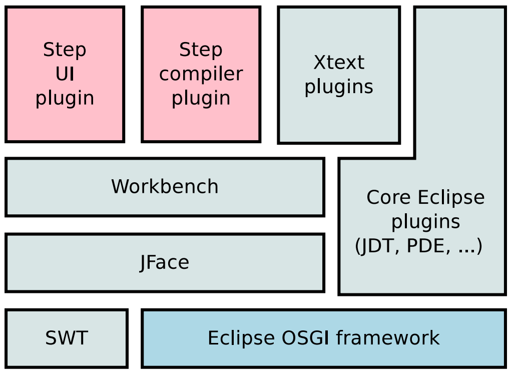


Figure 5.1: Step CEP client is built using Xtext on top of Eclipse RCP platform and consists of two plug-ins - compiler and UI. Eclipse RCP platform is based on the OSGI framework, contains standard Eclipse plug-ins, UI widgets (SWT) and the standard Eclipse UI (Workbench and JFace).

Other technologies that we used were *Bash* for scripting, Apache Maven [44] for library dependency management and building of Storm topologies, and Apache Ant [43] for automation of minor build tasks (mainly packaging during deployment). Analysis of data from evaluation and curve-fitting for parallelism cost model were done using the *Matlab* [47] tool. Last, but not least, testing was done using the JUnit [45] framework and source code was versioned in an SVN repository.

5.2 Runtime event detection

In this section we will describe some runtime concepts of event detection into more detail. In particular, we will explain event matching algorithms, event stabilization, punctuations, event serialization, throttling and load shedding.

5.2.1 Event matching

Event matching is a process of detecting event patterns among events arriving from multiple streams. We will now briefly explain event matching techniques for binary operators next, conjunction and exception, and for the unary iteration operator. The external operator and the projection operator do not belong here, since they do only simple filtering or post-processing of events. Also the union operator does not require any matching algorithm, since it is implemented as a simple merging of streams.

Binary operators

Recall that each binary operator Bolt processes an event according to the operator semantics and the stream from which the event arrived, whether left or right[2]. Each binary operator Bolt thus needs to declare which streams are left and which right.

The next Bolt keeps a queue of events received from the left stream, which is ordered in increasing end timestamps (i.e. in the order of event processing). If a new event is received from the right stream, it is matched against every enqueued left event. If the operator is greedy, then the first left event e_1 occurring before the new event e_2 will be used to create the composite event $e_1 \sqcup e_2$. If the composite event satisfies the next predicate, then it will be output and event e_1 will be removed from the enqueued events (so that it cannot match more events). In the case of the any next operator, event e_1 will not be removed and the matching will continue. Note that we do not need to store events received from the right stream[3].

The conjunction Bolt keeps two queues for events received from left and right streams. When a left event is received, it will be added to the left queue and matched against all enqueued right events. If a right event is received, it will be enqueued to the right queue and matched against all left events. Matching consists of iterating over all events and checking whether their timestamps overlap with the timestamps of the new event. In case of a match, composite event is created and predicate evaluated against it. Only composite events that satisfy predicate are output.

The exception operator needs to enqueue all undesired events (no events). When a desired event e arrives, it will be matched against all enqueued events e'. The event e will be output only if exception condition is not satisfied, otherwise it will be discarded. That is, for not before exception semantics, e will be output if no e' happens before it. For not during exception semantics, e will be emitted if no e' happens in between e.

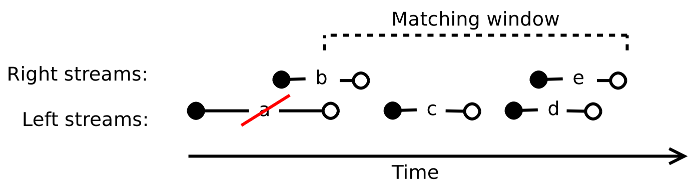


Figure 5.2: Event garbage collection at the next Bolt. Events from left stream that are outside matching window will be removed from the matching queue (here a).

The next and conjunction operators also implement garbage collection of events, which means that they remove events from their queues that cannot be matched any more. This can be done if matching window is specified by the duration predicate. The next Bolt (see Figure 5.2) will for each right event e_R remove all enqueued left events e_L such that the difference between the end timestamp of e_R and the start timestamp of e_L is bigger than maximum event duration (i.e. $t_1^R - t_0^L > maxDur$). Similarly, when a right event e_R is received at the conjunction operator, all left events e_L such that $t_1^R - t_0^L > maxDur$ can be discarded, as they cannot ever satisfy the conjunction predicate. Also if left event is received, all right events such that $t_1^L - t_0^R > maxDur$ will be removed.

Note that event garbage collection cannot be applied to the exception operator as enqueued

[2]For an event pattern P_1 *op* P_2 we say that events of pattern P_1 arrive from the *left stream* and events of pattern of P_2 arrive from the *right stream*

[3]Any received left event will always have a higher end timestamp than any stored right event because of event stabilization. Thus, received left event will never match any right event and hence right events do not need to be stored.

events may always match desired events[4]. Thus, the exception queues may grow infinite. In practice we deal with this through load shedding. If the queue of undesired events becomes too long, we will start dropping the oldest enqueued events.

Iteration operator

The iteration operator is unary and it handles events received from different streams uniformly. When a new event is received, the operator might create a new event matching sequence, or it might extend sequences that it already stores. Recall that if an iteration Bolt is parallelised, then a new event sequence will be started only by one parallel replica. Assume the component parallelism is P, the index of a replica among all parallel replicas is I, and an event e_2 was received. The replica will start a new event matching sequence only if $t_1^2 \% P = I$. The event sequences that are already stored by an iteration operator will be always matched against the new event.

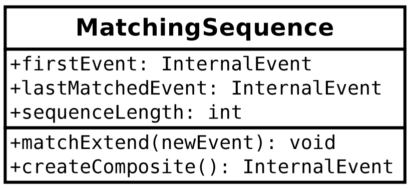

MatchingSequence
+firstEvent: InternalEvent
+lastMatchedEvent: InternalEvent
+sequenceLength: int
+matchExtend(newEvent): void
+createComposite(): InternalEvent

Figure 5.3: Data-structure used to keep sequences of matched events.

Event sequences are stored in a data-structure pictured in Figure 5.3. For efficiency we store only the first e_0 and the last e_1 events in the sequence, and only the event sequences that satisfy the iteration predicate are stored. A sequence e_0 to e_1 can be extended by a new event e_2 only if the predicate for events e_2 and e_1 evaluates to true[5]. In this case, the sequence e_0 to e_1 will be copied and the copy will be extended with the new event e_2. Thus, the operator will now store two matching sequences: e_0 to e_1 and e_0 to e_2 (i.e. all sub-sequences of observed events satisfying the predicate will be stored).

Any iteration operator will always output the extended sequence. The output event will be the composition of the first event in the iteration sequence and the last event. In the case when iteration predicate is not satisfied, the matching sequence e_0 to e_1 will be simply discarded.

On the other hand, greedy iteration will never output the extended sequences. A sequence can be output only if the predicate evaluates to false and the failed sequence is the longest one among all stored sequences[6]. Otherwise, the failed sequence will be discarded. Also note that after the longest matching sequence is output, all stored sequences will be discarded, thus starting a new matching process[7].

5.2.2 Event stabilization

Recall event ordering invariants specified in the previous chapter. We required that each operator processes events in the order of increasing end timestamps. This is established by the event stabilization algorithm. Because of the other invariants, we know that events are received on

[4]For example, each desired event may have start timestamp 0, and thus any received undesired event can match it.

[5]Previous field accesses in iteration predicate are evaluated against the last event in the sequence, i.e. e_1.

[6]If there are multiple sequences with the same longest length, all will be output.

[7]Recall that the semantics of greedy iteration state that after a longest sequence is output, the matching of new longest sequence starts from scratch.

each stream in the order of increasing end timestamps. Stabilization algorithm merges these streams, such that the resulting events are also ordered by increasing end timestamps.

To implement this, we note that an event e can only be processed by an operator after events with higher end timestamps were received from all operator input streams. Because otherwise, an event from some stream may still be received with smaller end timestamp, which would have to be processed before e. This formulation of the problem forms a basis for the event stabilization algorithm.

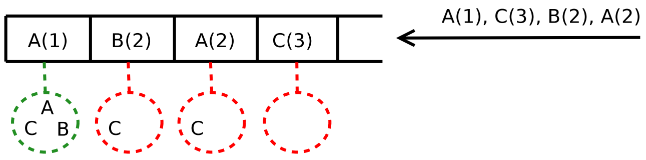


Figure 5.4: Illustration of the stabilization process. Events are put on a queue, each being marked by events with higher timestamps (displayed in circles). An event fully marked (green circle) can be released.

The algorithm is illustrated in Figure 5.4. It maintains a queue of events received from all input streams, in which events are ordered by their end timestamps. Each event has a set of marks. A mark is a stream identifier indicating that an event with a higher timestamp than the event marked was already received from that stream. The algorithm proceeds as follows: When an event is received from stream s with end timestamp t, all events with end timestamps $t_0 < t$ will be marked by stream s. The event is then inserted into stabilization queue and inherits marks from all following events. Note that inheriting marks from all following events is equivalent to just inheriting marks from the events with the same end timestamp plus all the marks from the first event with strictly higher timestamp. This is a bit faster, since it does not require always iterating till the end of the queue. Events that have been marked by all streams are removed from the queue in order of end timestamps and can be processed. The pseudo code for the algorithm is shown here:

```
stabilize(stream, newEvent):
  streamMark = mkMark(stream)

  foreach event in queue:
    if (event.endtime < newEvent.endtime)
        // mark all events with smaller timestamp
        event.mark(streamMark)
        if (event.markCount == inStreamCount)
          releaseEvent(newEvent)
          remove event from queue
    else // found a place to add new event. Following events should not be marked.
        add newEvent to queue
        break

  // the new event will inherit all marking from few following events
  foreach event after newEvent:
      newEvent.mark(event.allMarks)
      if (event.endtime > newEvent.endtime)
        newEvent.mark(event.streamMark)
        break

  // if the new event was marked by all streams, process it directly
  if (newEvent.markCount == inStreamCount)
    releaseEvent(newEvent)
    remove newEvent from queue
```

Assume N is the size of the stabilization queue. First for-loop iterates until a place is found to insert a new event. Second for-loop iterates from that point until the first event with strictly higher end timestamp is found (worst case till the end). Thus, the algorithm takes $O(N)$ steps in the worst case. We consider marking an event a constant time operation, since it is an insertion of a string into a set[8]. It should be noted that if all events have different timestamps and events arrive from each stream at the same rate, then the stabilization queue will have size on average the number of streams S and the complexity will be only $O(S)$. If the streams have different event rates, the average length of the queue will be the number of events that arrive during the time between receiving two events from the slowest stream are received.

5.2.3 Punctuations

Consider the case when an input stream is finite or when events do not arrive to the system with expected rate. This will result in events being blocked in the stabilization queue either indefinitely or for a long time, making the Step system stuck[9]. Consider for example the exception operator, for which exception events do not occur very often, but events to be cancelled arrive on high rates. The exception operator might not proceed unless some indication is received that an exception event was not received. To handle these cases, Step implements flushing of streams by using punctuations (further explained in [29]).

A punctuation is just a place holder for an event that was expected to arrive from an input stream (given the expected event arrival rate), but did not. Punctuations originate at Spouts, which monitor the external event arrival rates. If an event was not received from the corresponding input adapter for some time[10], punctuations will be sent to all output streams. Since output streams may be shuffled, we need to make sure that each destination task receives a punctuation. Thus, for each output stream the Spout will emit n punctuations, where n is the destination component parallelism.

When a component receives a punctuation from stream s, all events in the stabilization queue will be marked by s. The is because punctuation indicates that events that were blocked waiting for an event from stream s should not wait any more, which enables progress to be made. Also note that if a component receives punctuations from all of its input streams (i.e. all input streams are stalled), it will propagate the punctuations further, to all of its output streams. In the case when there are no more input events, the punctuations will eventually reach the last projection Bolts.

Punctuations are very useful for acceptance testing which will be described later. Here, we have a finite number of input events and we want to check whether all event patterns get detected. Thus, punctuations are needed to flush streams.

[8]Initially we have used constant time hash sets. Profiling with 20 input streams has shown though that tree sets perform better. This is because hash sets require computation of a hash code (and thus iteration over each character in stream identifier), whereas tree set just needs to compare the first few characters of the inserted mark.

[9] All following operators waiting for output from this operator will be stuck and might start dropping events that arrive from fast streams. This will result in high detection latency and some event patterns might not be detected.

[10]The time by which punctuations will be sent is configurable. By default we set it as $1/mean\_event\_rate$ seconds, which corresponds to expected event arrival rate.

Type of serializer	Size (bytes)	Time to serialize (ms*10^{-4})
Custom implementation	30	2.512
Public array Kryo	33	15.13
Private array Kryo	33	17.06
Explicit fields Kryo	28	6.289
Java fallback	374	296.53

Table 5.1: Comparison of different serializers performance with respect to the size of serialized data and the time it takes to serialize one event.

5.2.4 Event serialization

Before events can be sent on the wire they have to be efficiently serialized to an array of bytes. The Storm framework offers for this purpose the fast Kryo serialization framework[11] [48]. We designed the internal event data structures with respect to efficient stabilization, such that communication overhead between different components is minimized. The structure of internal events can be seen in Figure 5.5.

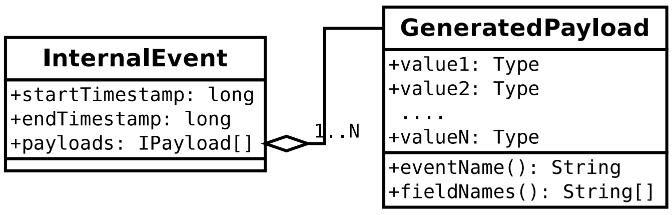


Figure 5.5: Recall the structure of internal events and payloads.

Recall that we decided to generate payloads for each type of input event, instead of using one generic data structure. The main reason for this was that we needed a payload to contain event and field names, but we did not want to send them over the wire. However, another reason was that it provided an opportunity for optimized serialization.

To see this, first let us have a look at a small benchmark that we tried for different Kryo and Java serializers. We were serializing an internal event as seen in Figure 5.5, containing two long timestamps and one payload. The payload contained following values: a five-character string, a one-character string, an integer and a byte value. We tried different types of payload with regards to storage of these values: first payload carried the values in a private array, second in a public array, and the third had a declared field for each value (this is the case in Figure 5.5). The results of the benchmark are shown in Table 5.1.

The fastest serializer was a custom one that serialized payloads with explicitly declared fields. When using an array of values, the Kryo serializer performed better if the array was declared public, as compared to private[12]. The best performance of Kryo was achieved by explicitly listing declared fields and making them public. Java serialization was proved to be very slow.

For simplicity we did not want to generate custom serializers for each payload. Rather, as a result of these measurements, we decided to use the Kryo field serializer for classes with explicitly declared fields. Thus, we have achieved payload serialization two or three times faster as compared to using a generic data-structure. This performance improvement is significant, since it allows for serialization of 1.6 million internal events per second.

[11]Storm does though not use Kryo by default, but rather defaults to slow Java serialization. It should be noted that to use Kryo, we need to explicitly register data-structures with it.

[12]The documentation explains that for classes with public fields a faster bytecode generation approach is used, instead of a bit slower reflection with setting access permission flag.

5.2.5 Event throttling

Event throttling is a process of limiting event rates on input streams, to avoid system overload. This can be for example caused by overflow of Storm messaging queues, leading to low memory conditions and possible crashes. Event throttling is also useful for evaluation of the CEP system, as we sometimes need to receive external events at some constant event rate. Additionally, throttling can be also seen as a feature used to bound resource usage of a CEP system. This might be useful when resources are limited and some topologies should be given priority over others by setting their throttled event rates higher.

We will briefly describe three types of event throttling that we have implemented, each with its own features:

- *Using Storm reliability API*
- *Local throttling at components*
- *Centralized throttling*

We solved the problem of overflowing messaging queues through usage of Storm reliability API. The API requires each topology Spout and Bolt to acknowledge events that it received. Acknowledgements are collected by special acknowledger tasks, which keep track of tuples that are in the system. If tuples were not acknowledged, they are assumed to be in a component input queue and additional tuples will not be sent. The length of the queue does not take into account size of received events. Thus, the input queue should be shorter if received events are big, but can be longer if events are relatively small. The disadvantage of using Storm reliability API is that it generates twice that many messages as there are events in the system, and our experiments have shown halved event throughput because of the CPU overhead of sending and receiving acknowledgements. We have later solved this problem by using event batching, where only batches are acknowledged, thus minimizing the overhead.

Local throttling can be used to limit input event rates to a constant number and is implemented by Spouts. A Spout knows at runtime how many parallel replicas it runs in and can use this knowledge to request only specified number of events from its input adapter within a time window. If the required input event rate is C, and there are P parallel Spout replicas, each Spout will throttle its input to C/P events per second. The throttling in each task is implemented through use of timers and counting of events that were received within a time window. If quota per time window was reached, the Spout will back off and request new events only after some time elapses.

We also experimented with having a dedicated centralised manager, which would receive batch acknowledgements over standard network sockets and grant permissions to Spouts to emit new batches. The idea was that we would only acknowledge many events at once, thus minimizing acknowledging overhead. We have however encountered problems with components lagging behind and this approach proved to be inferior to the Storm reliability API, since it was creating a bottleneck. Thus, we abandoned this idea.

5.2.6 Load shedding

Since events in Storm are sent on streams using a push-based approach, some care needs to be taken to avoid low memory conditions when system is under heavy load of incoming events. We have already explained that we avoid overflows of component input queues by using event throttling. However, these are not the only cases when system can run out of memory. We have to consider every queue in the system and the possibilities of how it can grow too big.

In particular, event matching queues at individual operators may grow indefinite if matching windows are not specified, or stabilization queues can become too big. The problem does not concern only low memory conditions but also performance of event detection - long queues mean that matching algorithms take longer, which may dramatically decrease event throughput. Therefore we use load shedding to limit degradation of performance.

Load shedding in Step is implemented for the cases of event stabilization and event matching. We try to implement best effort semantics for event detection. If event stabilization queues start overflowing, the oldest stabilizing events will be prematurely released for processing, such that space is made for new events. If event matching queues start overflowing, the oldest events in the queues will be permanently dropped. The disadvantage of this approach is that maximum queue lengths have to be specified at topology start and stay the same change during execution. An alternative and far better approach is specified in [3], where load shedding is applied according to dynamic metrics of topology performance (if performance is poor, events will be automatically dropped).

It should be noted that premature releasing of events from stabilization queues and dropping oldest events from matching queues does not affect correctness of event detection. The safe semantics of Step state that if a complex event is detected, then pattern must have occurred. If load shedding is applied, the backwards direction does not need to hold.

5.3 Step language compiler

In Chapter 4 we described a high-level overview of the compilation process that Step performs. In this section we will detail on some of the compilation phases and explain what they do.

5.3.1 Abstract syntax tree

The first phase of compilation process is construction of abstract syntax tree (AST) from a Step program, by rules specified in the Xtext grammar. The AST is a simple tree representation of a Step program, obtained by process of input parsing (an example can be seen in Figure 5.8). The AST is used to perform *input validation* and *linking*.

Input validation checks whether provided Step program is semantically correct. This is very important, such that we can guarantee that generated topology code will be compilable and will not cause runtime errors when submitted to the cluster. If invalid input is provided, compiler will abort and the offending code will be highlighted in the Step editor with corresponding error message. For validation support we use the Xtext framework and check the following points:

- *Existence of adapters* - whether input and output adapters exist in corresponding directories and whether they are correctly parametrized.
- *Correct naming* - we need to verify uniqueness of identifiers (e.g. aliases of events).
- *Use of predicates with certain operators* - some predicates may only be used in combination with some operators (e.g. previous event field access is allowed only in iteration predicates). Any other use should result in incorrect code.
- *Usage of predicates with respect to union* - we do not allow field accesses for events merged with the union operator, since these are ambiguous. For example, consider the pattern $(A|B) + [A.price > 100]$. The predicate $A.price$ is not available when event B is received at the iteration operator.

- *Correct typing* - we check that arithmetic, comparison and boolean operators in expressions really act on values with the corresponding type.
- *Negation syntax* - the unary syntax of negation pattern must allow for the pattern to be translated into binary exception operator (this cannot be specified in a grammar). In particular only the patterns $!E; E$ or $E; !E; E$ are allowed.

Linking is a process of resolving references between event schemas, external operators and event field accesses. During this process we check whether events in a pattern refer to an existing event schema, and that event field accesses access fields that really exists. This process is highly leveraged by the Xtext framework, which allows us to specify these references directly in the Step grammar. An example of references we need to establish between queries and event schemas is shown in Figure 5.6.

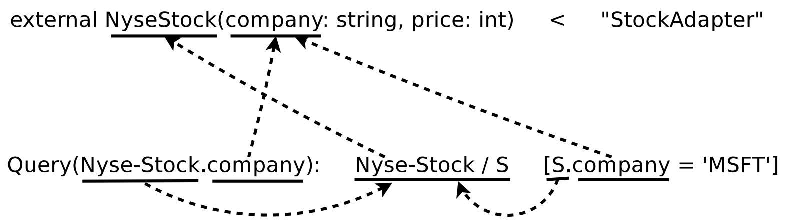

external NyseStock(company: string, price: int) < "StockAdapter"
Query(Nyse-Stock.company): Nyse-Stock / S [S.company = 'MSFT']

Figure 5.6: Example of links that have to be established. After the linking process it is clear where S, *Nyse-Stock* or *company* refer to.

Linking is connected with a concept of scoping - a reference from an element A to an element B can be established only if B is in the same or a sub-scope of A, i.e. it is visible. In Step language we implement only two scopes: *global* and *query* scope. The global scope contains event schemas and query identifiers, meaning that event schemas and query names have to be unique within a whole Step program. On the other hand, the query scope contains all elements that occur in a query pattern. In practice this means that a predicate in one query cannot refer to an external operator from a different query, as they belong to two different query scopes.

5.3.2 Operator tree

Second compilation phase is transformation of AST into an operator tree. The operator tree (illustration in Figure 5.7) does not contain any syntactic elements, but is rather the most abstract representation of user input. We construct one operator tree for each user event query. The tree contains binary and unary operator nodes (i.e. external, union, next, conjunction, projection, exception and iteration operators). Each operator has specified semantics (e.g. operator type) and may have an associated predicate expression (e.g. external, next, conjunction and iteration operators). The expression has a form of a tree and differs from AST in usage of objects instead of syntactic constructs. An operator tree can be seen as a complete abstract specification of event detection semantics that the user intends to use.

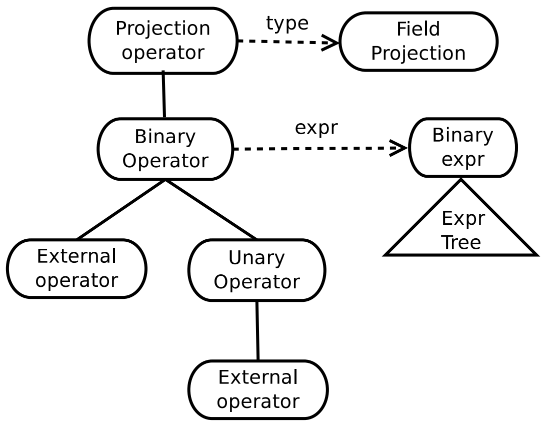


Figure 5.7: Illustration of operator tree.

The transformation of AST into an operator tree firstly translates syntactic constructs into their object representations. For example types in AST are translated to Scala type objects, logical values true and false are translated into boolean objects, and characters +, -, *, / into arithmetic operator classes. The goal is to provide a suitable abstraction for the next phases of compilation.

Secondly, the transformation deals with the cases when syntax of an operator does not correspond to its semantics. In Step, the syntax of event patterns is mostly structured in the same way as the operator tree, thus making it easy to translate abstract syntax tree into an operator tree. However, this is not the case for the exception operator. Recall that the operator's syntax permits patterns $!A; B$ and $A; (!B; C)$, whereas the corresponding binary exception operator semantics are $B \setminus_b A$ and $(A; C) \setminus_d B$. As a result we need to rewrite the AST exception pattern into a binary exception operator pattern. The transformation that we perform is illustrated in Figure 5.8. Other case when the bracketing of the pattern is different (i.e. $(A; !B); C$) is handled similarly.

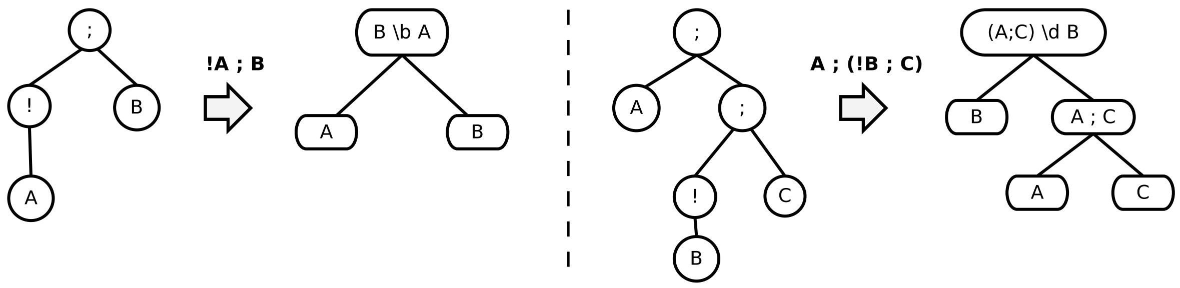


Figure 5.8: Illustration of how syntactic exception pattern $!A; B$ gets translated into exception operator $B \setminus_b A$ and $A; !B; C$ gets translated into $(A; C) \setminus_d B$.

Thirdly, the operator tree is a suitable representation for estimating operator input and output event rates, which will be explained in the next chapter. Also, the tree could be used to perform query rewriting optimizations, for example as explained in Next CEP[13].

5.3.3 Topology graph

Topology graph is a data-structure which represents a real topology. The nodes represent abstract Spouts and Bolts, and arcs represent streams on which these components are connected. The graph contains *all* the information about a real Storm topology and is a very detailed representation. We will not go into much detail here, but only explain the role of the topology graph in the compilation process.

Firstly the topology graph contains information about how many resources a topology will use - for example the number of workers, parallelism of each component or the number of reliability API tasks[14]. Secondly, it contains abstract representations of Bolts and Spouts, which will be later compiled into concrete classes. Each Bolt and Spout knows their parallelism, their input and output streams, and the shape of payloads received and sent on these streams (this is required for expression indexing). Some components might also contain associated predicates, operator types, detection windows, and other details used for code generation (e.g. unique class and package names). Thirdly, the topology graph contains streams that connect Spouts and Bolts. Each stream has an associated name, endpoints and grouping, which can be shuffle or all. Finally, the graph representation also contains a list of payloads, corresponding to the input

[13]However, we have not explored these optimizations, as this was already done by other CEP systems.
[14]Estimation of these will be described in next chapter.

events. Each of them contains fields and types, as declared in event schema, and will be used to generate payload data structures.

Also note that there is only one topology graph, which is built from all operator trees. The common nodes of the trees are deployed as one component (see later optimizations) and each topology node corresponds to an operator - for example external operators correspond to Spouts, other operators correspond to Bolts, but union operator is deployed as merging of streams. To be brief, note that the topology graph looks exactly like a real topology that was explained in the previous chapter.

Expression indexing

When translating operator trees into a topology graph, we perform *expression indexing*. Internal events contain a list of payloads of external events. When internal events are composed together (e.g. by next Bolt) the lists of payloads are concatenated. Some operator may contain a predicate, which needs to access a field of one of these payloads. Accessing a field within a payload is done by using offsets determined at compile time (an example of accessing field B.Fld2 is shown in Figure 5.9). The computation of offsets is done by the *expression indexing* algorithm.

The expression indexing first figures out what payloads will be received at each operator. Examples for input and output payloads for some operators are shown in Figure 5.10. For example if iteration takes as input payloads ps, then it will output payloads $[ps', ps]$, where ps' are the payloads of the first event in iteration sequence, and ps the payloads of the last event. Next and conjunction operators output concatenation of payloads of composed events. Exception outputs the payloads of desired events.

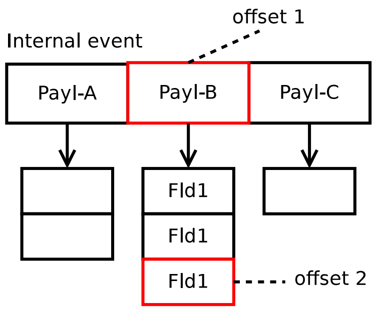


Figure 5.9: An example of how field B.Fld2 can be accessed by using offsets.

Union operator is the most interesting case, since output payload may come from two different input streams. Recall though that these payloads cannot be referenced by predicates and thus we only care about their size[15]. Our rule is to output the bigger payload of the two input ones, by padding payloads if they are smaller. Consider the right-most case in Figure 5.10. If event $A; B$ is received at the union operator, then payloads $[payload_A, payload_B]$ will be output. If event C is received, then we need to pad its payloads from left to the size of 2. Thus, the payload $[null, payload_C]$ will be output. As a result, the output payload size is always fixed, thus enabling for static calculation of offsets.

Thus, for each field access, we can calculate the payload offset and access it in constant time. Generating a code that accesses a field within a payload can then be done by using the field name from the corresponding external event schema.

5.3.4 Code generation

[15]We need to determine at least the size of output payloads, such that expression indexing works for other predicates higher up in the operator tree.

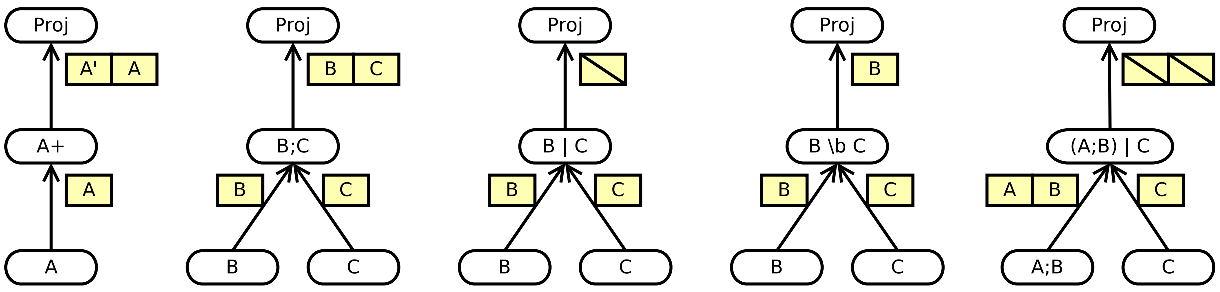


Figure 5.10: Examples of payloads input and output by different operators (yellow boxes are payloads; crossed boxes cannot be referred to in predicates, A' means previous A payload).

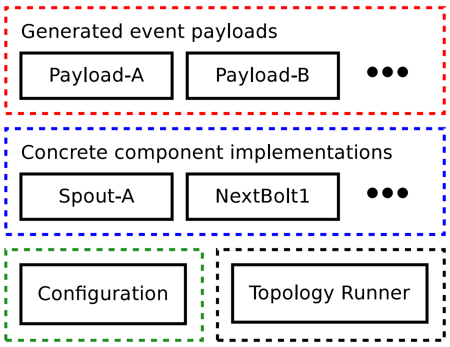


Figure 5.11: The output of compilation process.

After topology graph was constructed, it will be used to generate code, which we call topology fragments. As shown in Figure 5.11, the output of the compilation is a set of Java classes: *configuration*, *topology runner*, *payloads* and *Bolts and Spouts* implementations.

The *configuration* class contains information about which components topology contains, how they are connected on streams, what is their parallelism, how many workers are required, which serializers are used, and what is the event batch size, or the lengths of input, stabilization and matching queues. *Topology runner* contains the main method and uses the configuration to instantiate a topology and submit it to Storm. The *payloads* are generated data structures that will carry values of external events and that will be sent over internal streams. Finally, generated *Bolts and Spouts* are subclasses of abstract components from the runtime framework, and implement their abstract methods. Most important of these need to evaluate predicates, provide matching window durations, component output streams, or initialize input and output adapters.

To see how generated fragments look like, consider the following Step program detecting increasing sequence of Microsoft quotes with some length:

```
topology "Topology"

external Stock(name: string, price: int) < "StockAdapter" [250000.0]

Query(S.price): Stock/S[S.name = 'MSFT']+ [S.price >= prev(S.price) && len > 50]
                > "DiscardingAdapter"
```

The classes that will be generated from this program can be seen in UML diagram in Figure 5.12. Most of the methods in the diagram have obvious meaning. Those to notice are *processNewEvent(extEvent)* in StockSpout, which takes a new external event, converts it into an internal event and sends it to those output streams, for which predicates are satisfied; *select(event, prevEvent)* from IterBolt0, which evaluates iteration predicate; and *project(event)* from ProjectBolt1, which selects the event fields to be output into an output adapter.

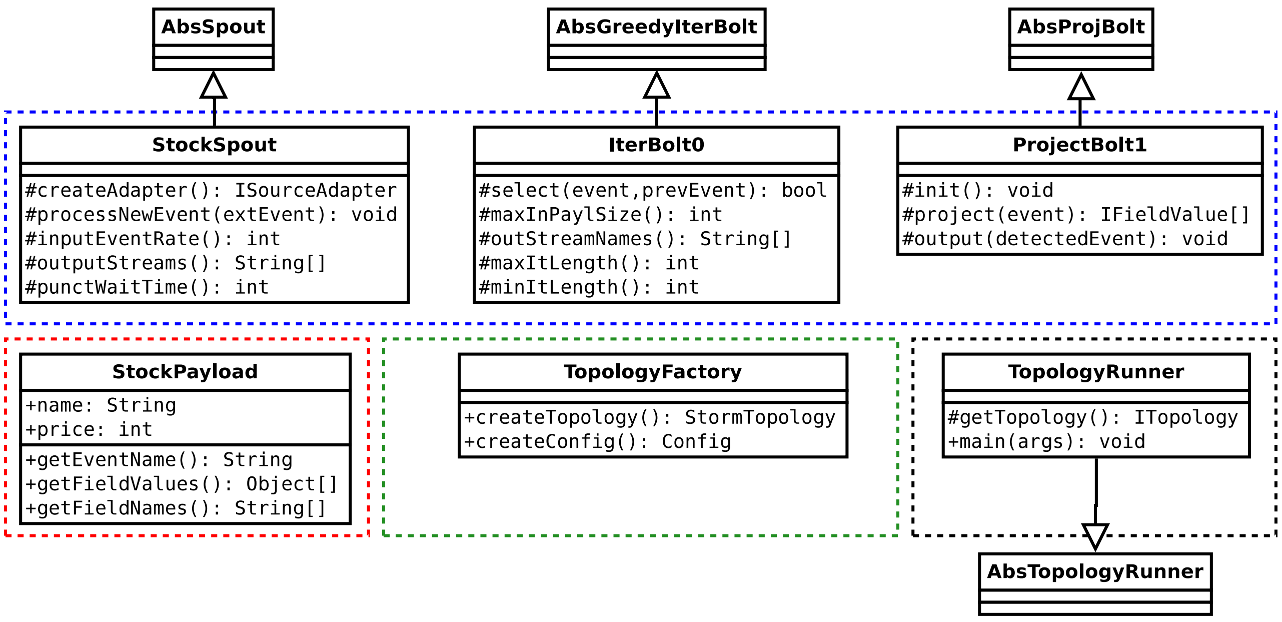


Figure 5.12: The UML diagram of generated topology fragments from the topology graph of the stock query. The classes are color-coded in the same way as in Figure 5.11.

5.4 Step client GUI

In this section we will briefly explain the capabilities of the Step client GUI and how it is built on top of the Eclipse RCP platform. It should be noted that a great part of it (in particular the Step editor) was generated using Xtext, thus sparing us a lot of development time. Some GUI screenshots can be found in Appendix D.

Recall that GUI is implemented as one Eclipse plug-in, which structure can be seen in Figure 5.13. The plug-in contains five components: *cluster overview*, *console*, *monitoring GUI*, *Step editor* and *Step project*.

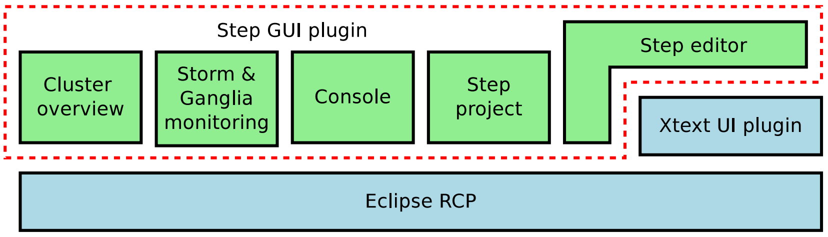


Figure 5.13: The components of Storm GUI build on top of Eclipse RCP platform by using Xtext.

The *cluster overview* part of GUI shows general information about the Storm cluster and topologies that run on it. This information is periodically pulled from the Nimbus service running at the cluster by using the Thrift framework. We display for example the cluster uptime, the number of used workers, the total number of worker slots and the physical node count. For each topology (i.e. Step program) we display the number of tasks it runs, the number of workers it uses, the total count of detected complex events, the number of external events that it consumed, and throughput of consumed input events per second. These statistics are provided by Storm, which counts received events on individual streams. Thus, to count how many events were consumed from an input adapter, Spouts have an extra “dead end” stream (i.e. without subscribers), where new external events are always emitted and counted (note, that this causes only minimal overhead, since events on the stream are not sent).

The *console* part of GUI is used to notify a user of progress or errors that may occur when performing cluster-related tasks, such as topology packaging and submission, or stopping and starting cluster processes.

The *monitoring* part of GUI allows user to monitor resource use across the cluster and view topology information through Storm UI. Resource use is monitored using a distributed monitoring system called Ganglia [40], which shows for each node its CPU, memory, disk and buffer usage, as well as summaries of these. It is a very rich and powerful monitoring tool commonly used in industry. On the other hand, the Storm Web UI can display statistics about topology Spouts, Bolts and streams, in particular their counts of emitted, transferred and received events[16]. The integration of both monitoring systems into Eclipse is simple, as we use the internal Eclipse browser to display both web UIs as web pages.

Another part of GUI is the Xtext-generated *Step editor*, which provides an environment for writing Step programs. Any change to a Step program will automatically cause compilation of the program into a topology, without the user having to explicitly request it. Apart from continuous builds, Step editor offers syntax highlighting for keywords, strings and comments, and auto-completion for keywords. The editor has almost the same capabilities and design as the popular Eclipse Java editor.

To create a Step program, the first step is to create a new Step project. This task is handled by the *Step project* GUI, which will create directory structure for the project, containing sources and build scripts. After the project is formed, new programs may be created in a specific directory and context menus used to submit and kill topologies on the cluster. The Step project GUI also allows for configuration of the cluster - for example specification of nodes' IP addresses and ports.

The implementation of GUI is based on extending Eclipse GUI. Eclipse allows this through the use of so called *extension points*. For example, Step editor is implemented as an editor extension, and new items are added to context menus through menu extensions. The GUI of the newly added elements is implemented by using SWT widgets, which are the standard graphical library for Eclipse.

5.5 Optimizations

5.5.1 Batch processing

We observed that events arriving on streams with high throughput are typically small[17]. However, by profiling runtime topologies we noticed that for these small events Storm does not perform well. More precisely, we observed high CPU load when sending and receiving events, which in some cases accounted for up to 50% of the CPU use. We hypothesise that this could have been due to messaging layer overhead, which had to process each event individually.

It is of course implausible for a system to spend half of its resources on doing I/O. Therefore, we were trying to find a workaround and we found it in terms of event batching (interestingly Storm does not do this). Event batching is simple - the sender component will buffer multiple outgoing events for the same output stream. When the buffer is full, events will be packaged

[16]Storm Web UI is only useful when user understands topology structure, as it displays information only about individual Bolts and Spouts.

[17]Consider for example stock quotes, which arrive on fast streams, but contain only about ten numerical fields and two small string fields.

into an event batch, which will be sent to the corresponding stream. The component receiving an event batch will simply unpack it and process (i.e. first stabilize) all its events one by one, in the order of sending (event batch preserves this order). The data-structure used for batching is shown in Figure 5.14 and is really just a container for variable number of internal events.

The number of events in a batch is configurable, but is fixed per topology. In Chapter 6 we will show what we consider the optimal batch size and also illustrate the incredible performance gain that this simple optimization caused. It may appear that bigger batches are generally better, as they cause better saturation of network and lower CPU resource use due to sending and receiving of events. However, it is also important to note that they lead to higher processing latency.

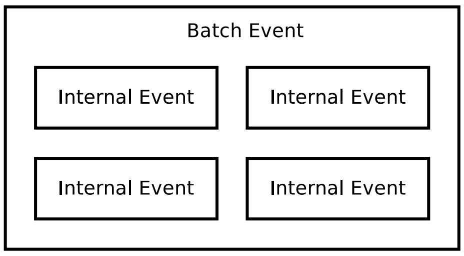


Figure 5.14: The structure of batched events.

Furthermore, some extra care has to be taken to handle punctuations, which are sent separately from event batches. If a punctuation is received on a stream, some events may be released from the stabilization queue. Additionally, with regards to batching, it pays off to release the current batch of outgoing events, even though it might not be full yet. This is purely an optimization with regards to lowering processing latency, as receiving a punctuation means that we may not receive more events to fill the outgoing batch.

5.5.2 Two-phase stabilization

Introduction of batching caused some unexpected performance issues. We found out that a lot of events were being held in the stabilization queues, thus making the stabilization algorithm take much longer. Consider for example an operator, which takes input from S streams, where events are sent in batches of size B. Also assume that events have strictly bigger end timestamps (which will often be the case). After a batch is received at the operator, its contained events will be put onto the stabilization queue. An event will be released from the queue only after it has been acknowledged by events with higher timestamps from all streams. Thus, a batch from each stream has to be received and put onto the queue. This results in a queue of average size $B * S$ and $O(B * S)$ operations are needed to release an event.

Now consider the case without batching. On average there will be only S events in the queue (assuming events arrive from each stream at the same rate). Considering that B is in order of hundreds, the performance of stabilization degrades significantly when using batching. The problem can be solved by an algorithm we call *two-phase stabilization*.

The idea of two-phase stabilization is that we stabilize events not in the order of received batches, but rather in the order in which they would be received without the batching optimization. The operation of the algorithm can be seen in Figure 5.15 and consists of two phases:

- *The first phase* buffers event batches for each input stream. If an event batch was received from all streams, we start removing events. The events are removed in the order of streams, instead of in the order of batches. In other words, the 1st event from the 1st stream is removed, then the 1st event from the 2nd stream, up to the the 1st event from S-th stream. Then we continue by removing the 2nd event from the 1st stream, the 2nd event from the 2nd stream, and so up until the B-th event from the S-th stream. The removed events are sent for stabilization to the second phase. Now suppose that events on one stream get exhausted, and that the last event from the stream had end timestamp t. We continue removing events from other streams in round robin fashion, as long as they have end timestamp less than t. This results in better order or released events, in the case when batches on one stream arrive faster than on the other streams.

- *The second phase* is actual stabilization of events, using the same algorithm as described earlier in this chapter.

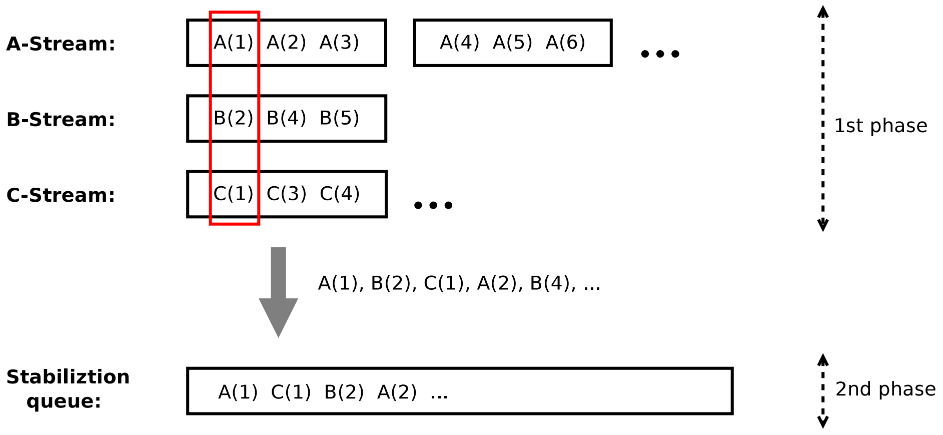


Figure 5.15: Illustration of two-phase stabilization. The first phase buffers event batches for each stream and releases individual events in the order shown by the red box. Second phase is the original stabilization algorithm.

If a batch of size B was received from a stream, it will take constant time to buffer it during the first phase. For any event released during the first phase, the second phase will take $O(N)$ steps, where N is the length of the stabilization queue. Since events are stabilized in the order in which they would be without batching, we achieve the performance of the original algorithm. In fact, the performance will be even better in cases when streams have different event rates. This is because events from fast streams will be sent to stabilization queue only if events from other streams were received, thus not always saturating the stabilization queue. Two-phase stabilization is correct because all events are stabilized by the original correct stabilization algorithm.

Note that with regards to load shedding, batches that are buffered during the first phase will be released if maximum queue sizes per stream are reached. This means that oldest events will be passed to the second stabilization phase. During the first phase we also need to deal with received punctuations. If a punctuation is received, we will remove all events waiting in the first phase of the algorithm, such that progress can be made.

5.5.3 Reuse of common event patterns

Another optimization that is implemented on top of already described system is reuse of common event patterns among different queries. Consider for example the following two queries: $(A\,;\,B)\,|\,C$ and $(A\,;\,B)\,|\,D$. The generated topology for this query is shown in Figure 5.16, and includes the next operator twice, for each specified query. Both of these operators perform the same detection, as they operate on the same input streams. The topology also requires an event to be sent twice to each operator and hence twice that many CPU resources are used when detecting a next pattern.

We can improve the performance in these cases by deploying operators common among multiple queries only once. An example of a topology for the earlier queries utilizing this optimization is shown in Figure 5.17. Note that the next operator now outputs detected events not to one, but

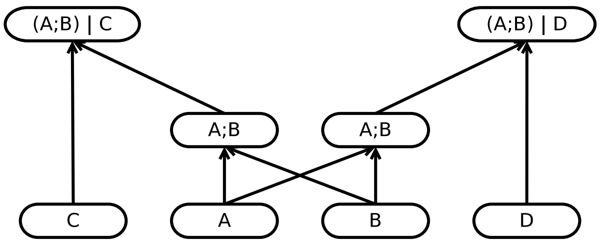


Figure 5.16: The topology of two queries $(A\,;\,B)\,|\,C$ and $(A\,;\,B)\,|\,D$ if common sub-queries are not reused.

to two different streams. The grouping of the new stream will depend on whatever the following operator expects (in the example it will be shuffled) and thus stays the same as without the sub-query reuse optimization.

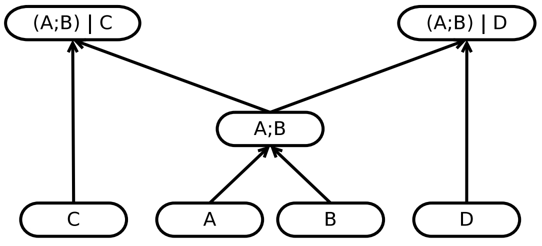


Figure 5.17: The topology of two queries $(A\,;\,B)\,|\,C$ and $(A\,;\,B)\,|\,D$ with common sub-query reuse.

The optimization is performed by compiler during the transformation of the query operator trees into a topology graph. When adding a new operator sub-tree to the topology, we first check whether a Bolt for such sub-tree does not already exist. If it does, we just add an extra output stream to it. Otherwise, we deploy a new Bolt for the operator and recursively continue with deployment of its sub-trees. That is, reuse of common sub-queries happens from top to bottom - we first try to reuse the biggest common pattern and if we do not succeed, we continue with its sub-patterns.

It should be noted that we consider two sub-patterns identical, if they have exactly the same operator trees. For example the patterns $S[S.price > 100]$ and $S[S.price > 100]$ are equal, but are not the same as the pattern $S/T[T.price > 100]$. I.e. for simplicity we do not consider structural equality.

5.6 Acceptance testing framework

The correctness and validity of Step is established in two ways:

- *Unit testing* is used to establish the correctness of the compiler and the runtime framework at a component level. It tests individual complex event detection and compilation algorithms using the JUnit framework.
- *Acceptance testing* is used to establish the correctness of the whole system from its input to its output. That is, we test that the specification of Step programs, their compilation, deployment, and running works correctly as a whole. Since it is not simple to test a distributed system, we will explain our approach, for which we have implemented an acceptance testing framework.

Firstly note that Storm provides us with an ability to simulate a cluster on a single node, a so called *local cluster*. The purpose of this is an ability to try out topologies locally before they are submitted to a real cluster. Local cluster resembles a distributed environment fairly closely. A Zookeeper server is started that holds global Storm state and multiple tasks are spawned for each component according to their parallelism. The events are exchanged between tasks through localhost network interface and arrive on input queues, as in a normal Storm framework. Thus, the local cluster provided by Storm is a good basis, on top of which we built the acceptance testing framework.

The Step acceptance framework takes as input a Step program[18] and a list of events that should be detected by the program. The program will be compiled, run and the events that it detects will be compared against the list of expected events.

The whole process of running a test is illustrated in Figure 5.18. First, Step program is saved into a file and is compiled into topology fragments by the Step compiler. To test variability in topology parameters, the compiler can be parametrized by different event batch sizes or event queue lengths. Generated fragments are then joined with the runtime framework and dynamically compiled and loaded. Afterwards, local Storm cluster is initialized and the compiled topology is submitted to it. Some parameters can be overridden, for example component parallelism can be changed to test correctness of parallelised topology at low input event rates. The local cluster will start running the topology and will be killed after some fixed time. The events detected by the topology are always sent to a mock sink adapter, which stores them. After topology is killed and cluster shut down, the detected events are compared against expected ones, which determines whether the test succeeds or fails.

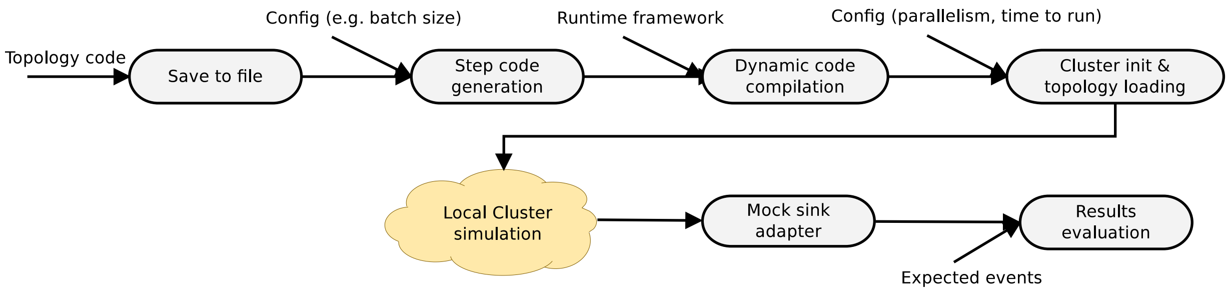


Figure 5.18: The operation of acceptance test framework.

We have implemented 46 acceptance tests (each having multiple event queries) that run on this framework and test individual operators on their own, as well as in combination with complex predicates and other operators. Two examples that illustrate the simplicity of defining acceptance tests are shown in Appendix C.

5.7 Remote profiling

To spot performance bottlenecks, we profiled the Step CEP system at runtime. An interesting aspect of this was remote profiling of topology tasks running on the cluster. For this purpose we used the VisualVM software from Oracle [49], which was connected via JMX connections to jstat deamons and worker processes running at the cluster. We had to configure Storm to start workers with parameters that enable JMX profiling connections. An overview of this set-up can be seen in Figure 5.19.

[18]The Step program has usually parametrized adapters, where parameters specify which events should be emitted

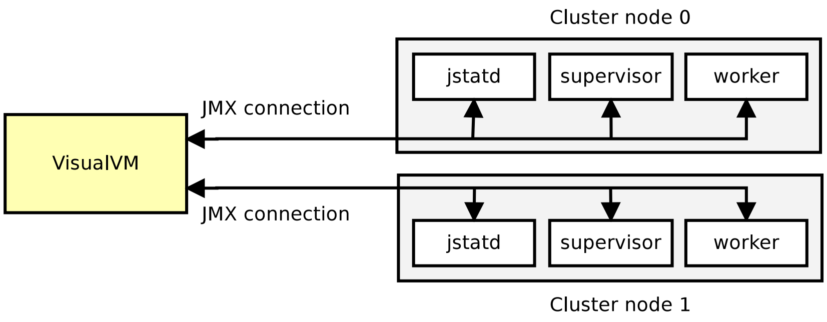
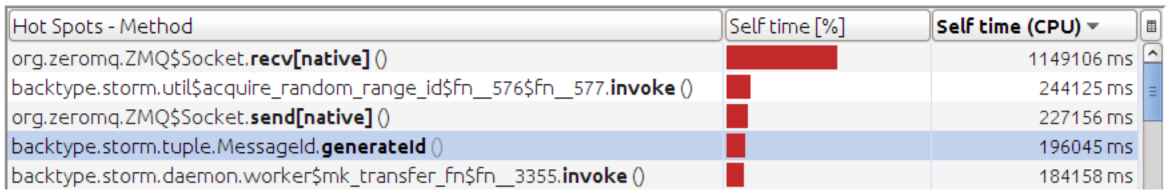


Figure 5.19: Remote profiling of Storm and Step runtime.

For each aspect of complex event detection we could see the CPU time that it used. Using this approach we discovered that too many CPU resources were spent by Storm on receiving and sending events, which led us to implement event batching. Profiling was also used to determine the optimal data structures for the stabilization algorithm. Additionally, after event batching was introduced, it led us to realise that original stabilization algorithm became the bottleneck, which resulted in the implementation of two-phase stabilization. A screenshot of profiling showing that input and output IO consumed too many CPU resources without event batching is shown in Figure 5.20.



Hot Spots - Method	Self time [%]	Self time (CPU)
org.zeromq.ZMQ$Socket.**recv[native]** ()	🟥	1149106 ms
backtype.storm.util$acquire_random_range_id$fn__576$fn__577.**invoke** ()	🟥	244125 ms
org.zeromq.ZMQ$Socket.**send[native]** ()	🟥	227156 ms
backtype.storm.tuple.MessageId.**generateId** ()	🟥	196045 ms
backtype.storm.daemon.worker$mk_transfer_fn$fn__3355.**invoke** ()	🟥	184158 ms

Figure 5.20: Profiling Storm remotely showing bottleneck in the ZeroMQ messaging layer.

5.8 Summary

In this chapter we detailed on implementation aspects of the Step CEP system, as well as our technology choices. We described algorithms provided by the runtime framework, such as event stabilization, throttling and punctuations, as well as algorithms used by the individual operators to match events. We also explained how Step optimizes for efficient serialization and deals with overloads through load shedding. Then we outlined the whole compilation process of a Step program into a runnable Storm topology, and also the functionality of the Step GUI. Finally, some performance optimizations were introduced, as well as a framework for testing the correctness of complex event detection.

Chapter 6

Estimating system parameters

6.1 Goals

Complex event patterns specified in the Step language are compiled into a topology to run on the Storm framework. The topology can be tweaked through changing many parameters, which specify how many resources will be used and how many events per second it will be able to process. Since topologies cannot change at runtime, we need to estimate their values during compilation. We are interested in computing the optimal set of parameters, such that the compiled topologies are capable to detect event patterns at specified input event rates, without being too wasteful. These are the parameters that affect topology performance:

Topology parameter	**Description**
Component parallelism	The number of tasks that each component will run in.
Workers count	The number of workers the topology needs to use.
Acknowledger count	The number of the acknowledger tasks for Storm reliability API.
Batch size	The number of internal events contained in one event batch.
Internal queue sizes	The maximum sizes of component input queues, event matching and stabilization queues.

The most important parameters are *component parallelism* and *workers count*. Increasing parallelism of individual operators increases their event processing throughput. However, setting this parameter too high might waste resources, as unneeded component tasks will be spawned. Furthermore, this may result in lower performance of operators that require input stream replication. Take for example the greedy next operator, which replicates its left stream across all parallel tasks. Unneeded tasks will also receive replicated events, resulting in higher network usage and wasting of CPU resources.

The number of topology workers can be seen as the number of processes that will run topology tasks (i.e. event processing threads). Each worker contains input and output queues for the tasks it runs and handles the messaging IO. We need to determine how many tasks a worker can handle and how many workers can be run on a single CPU core. Apart from workers count and component parallelism, we also need heuristics for the number of acknowledger tasks and the optimal batch size. For simplicity we abandon estimation of internal queue sizes, which we only limit to some maximum values.

6.2 Approach

The basis for determining topology performance parameters are user-specified input event rates. We use these to estimate the input and output event rates for each operator in the topology. This is done by the Step compiler, independently for each event query, by using the operator tree representation of a Step program. In general, the output event rate of an operator is a function of its input rates, as seen in Figure 6.1. Input event rates, the performance of sending and receiving events, and the type of operator output stream are the main factors in determining operator parallelism.

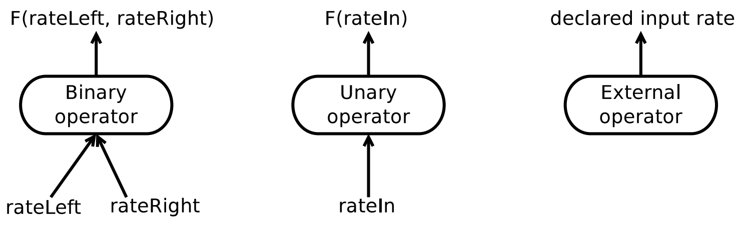


Figure 6.1: Output event rates are calculated as a function of input event rates.

Recall that compilation process merges multiple operator trees into one topology graph. It may be the case that multiple external operators are merged into one Spout, or multiple detection operators are merged into one Bolt due to sub-query reuse. However, this does not affect the operator input event rates. That is, if the operator were estimated to receive x events per second, its corresponding Bolt or Spout would also receive x events per second. To compute the component parallelism, we use the topology graph representation, as it contains information not only about input event rates, but also about predicates and streams.

Each Bolt or Spout has its own model of parallelism, which is based on performance measured during its evaluation. The performance of each operator is evaluated with regards to varying parallelism on a fixed scenario (illustrated in Figure 6.2[1]). Here, enough event sources (Spouts) and event sinks (projection Bolts) are created for operator Bolts, such that the evaluated Bolt always has new events to process and is able to emit detected events. Then we vary the parallelism of the Bolt and plot it against the measured throughput of processed events.

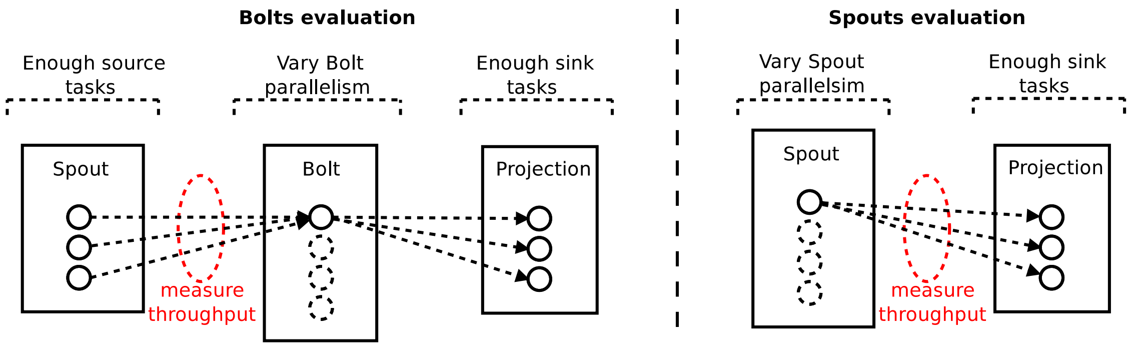


Figure 6.2: Evaluation of individual Bolts and Spouts with respect to parallelism.

The evaluating of Spouts is similar to that of Bolts. For a given Spout parallelism we always create enough sink tasks, such that the Spout has the capacity to emit new events. Then we plot the number of events that a Spout outputs per second with regards to different parallelism. Afterwards, we use the Matlab curve fitting toolbox to process our data, and hence empirically

[1]Exact evaluation scenarios for each operator as well as their performance graphs will be explained in the next chapter.

obtain a model that describes Spouts and Bolts parallelism. In the case of Bolts, the model depends on input event rates and the structure of associated predicates. In the case of Spouts, we also need to include the types and the count of output streams.

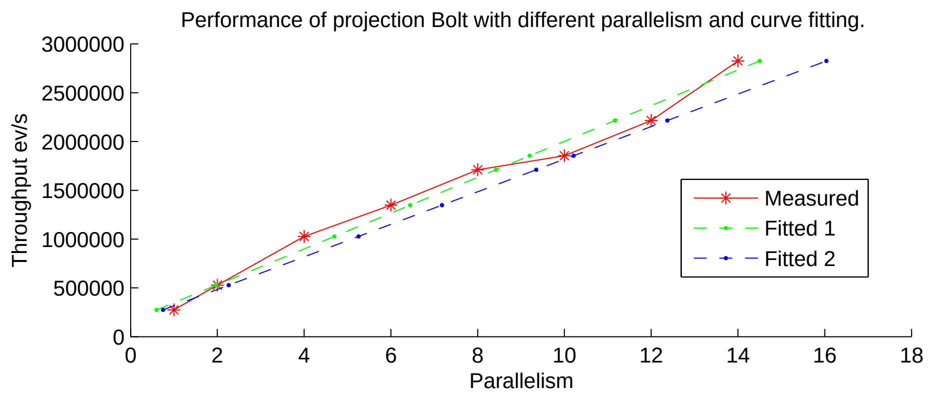


Figure 6.3: Fitting measured data from projection Bolt.

An example of measured projection Bolt performance is shown in Figure 6.3. The evaluation scenario in this case consisted of receiving an event with twenty fields from a Spout and projecting all of its fields to an output adapter that would discard them. Measured data (red line) are then fitted into a model, in this case a linear model depending only on the input arrival rate (green line). We also could have included the number of projection fields in the model, but this would have exponentially increased the number of evaluation experiments. Instead, we try to use conservative scenarios during experiments and any errors or variation introduced by new parameters are approximated with an error margin. To include the error margin into the model, we simply multiply expected input event rate with a constant to obtain higher rate (model with the error margin is displayed by blue line).

We estimate input event rates for each operator, which are then fed into the model to predict component parallelism. This then serves as a base for computing number of required workers and acknowledger tasks, which are estimated using heuristics obtained by observing properties of the Storm platform. It should be noted that our method of modelling performance is hardware-specific. However, the models can be crudely transposed for use on a different hardware. We could for example evaluate one model on a different hardware and determine a performance scaling factor. All other operator models could then be scaled by the same value. We will now describe into more detail how event rates are estimated, what are the performance models of individual operators and which heuristics we use.

6.3 Modelling event rates

For each operator we estimate its input and output event rates. The summary of these can be seen in Table 6.1. The input event rates are straightforward, since they are just a sum of rates of all input streams. The base case is the external operator receiving events from an external stream, which has declared mean event arrival rate. More interesting are operator output event rates, which are functions of the input event rates.

We use the terminology that $rate_{in}$ is the input event rate for unary operators. For any binary operator $P_1\ op_B\ P_2$, we denote $rate_{left}$ to be the rate of events of pattern P_1, and $rate_{right}$ to be the rate of events from pattern P_2. For predicates, dur specifies the maximum duration of a detected composite event and $maxLen/minLen$ the maximum or minimum length of iteration sequence if provided.

Operator type	**Input event rate**	**Output event rate**
External	declared	declared
Projection	$rate_{in}$	$rate_{in}$
Union	$rate_{left} + rate_{right}$	$rate_{left} + rate_{right}$
Conjunction	$rate_{left} + rate_{right}$	$rate_{left} + rate_{right}$
Exception	$rate_{left} + rate_{right}$	$rate_{left}$
Next (greedy)	$rate_{left} + rate_{right}$	$rate_{left}$
Next (any)	$rate_{left} + rate_{right}$	$rate_{left} * rate_{right} * dur$
Iteration (greedy)	$rate_{in}$	$rate_{in}/minLen$
Iteration (any)	$rate_{in}$	$rate_{in} * maxLen$

Table 6.1: Parameters that affect topology performance.

For the *external* operator, the worst case scenario is that on each output stream it will emit all the events that it receives (i.e. its declared input event rate). It should be noted that for simplicity we do not take into account selectivity of logical, arithmetic or boolean predicates that can be associated with operators. This requires more information about the received events, in particular their typical distributions of values and event durations. However, we take into account the duration and length predicates, which may occur in next, conjunction and iteration operators. When estimating event rates, we try to be fairly conservative, so that the CEP detection system can also cope with particularly bad combinations of input events. However, using the worst case scenario is an ineffective estimate for some operators (e.g. conjunction) and we will need to make some simplifying assumptions.

The output event rates of *union* and *projection* are the same as their input rates, since these operators do not drop or create new events. *Conjunction* operator is more interesting, since we need to estimate how many complex events may be output per input event. The operator will output an event only if it overlaps with some other event. Thus, the question is with how many left events can a received right event overlap and vice versa. The worst case scenario is when all events have the same start timestamp and hence all overlap. If the duration predicate is not specified, the output event rate will rise to infinity. Even with specified event duration predicate, the rate[2] would be

$$rate_{left} * dur + rate_{right} * dur$$

This is very pessimistic and unrealistic, as it presumes that within a time window every event would overlap with every other event. Instead, we assume that each received event will overlap at most one other event in the same time window, and hence the output rate will be $rate_{left} + rate_{right}$[3].

The output event rate for the *exception* operator is simply the input event rate of the desired events. This is the worst case that can occur in the absence of undesired events. Similarly, the worst case output event rate of *greedy next* operator is the rate of left events, since a left event can match at most one right event. Thus, only one composite event per left event can be output. The worst case for *any next* is different, since left event can match all right events within a matching window. Within a matching window there will be $rate_{right} * dur$ events, and thus the output rate will be $rate_{left} * rate_{right} * dur$[4].

[2]Justification: each left event will match every right event within a window. There will be $rate_{right} * dur$ right events in a window. Also vice versa, every right event will match every left event within a window.

[3]This assumption is still very conservative. Consider detecting two calls from the same number happening at the same time. An overlap will be detected only as often as a detected fraud occurs (i.e. very rarely).

[4]If maximum duration is not specified, the event rate may be infinitely high. In this case we assume that duration window is only 1 and use the output event rate $rate_{left} * rate_{right}$

Greedy iteration will output one detected sequence and will restart its matching from scratch. If the shortest sequence to be detected is of size $minLen$, then for every $minLen$ events one sequence can be output. Thus, the output event rate is $rate_{in}/minLen$, or $rate_{in}$ if we do not know the minimum iteration length. Here we assume that iteration predicates fail often. If predicates would never fail, we could use the estimation $rate_{in}/maxLen$. Now consider the *any iteration*, which detects any sub-sequence up to length $maxLen$. That means that in the worst case if an event is received, a sub-sequence of size 1, 2, ..., or $maxLen$ can be output. Thus, the estimated worst case output rate is $rate_{in} * maxLen$.

6.4 Parallelism model

Now we will describe our parallelism models for each operator. We will present our theory and the evaluation results on which it is based will be presented in the next chapter.

Consider the structure of a general topology component in Figure 6.4. The time to detect an event can be decomposed into three parts: the time to receive an event, the time to process an event and the time to send a detected event. The time to receive an event and to process it will depend purely on the operator input event rates and its operator semantics (i.e. operator type and its predicate). The time to send an event will depend on the number of subscribers and the type of output stream grouping.

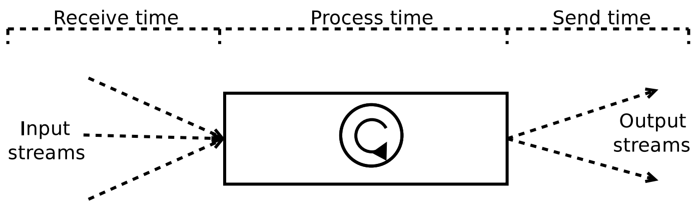


Figure 6.4: Three phases on which Bolt spends most of its time.

We make a simplifying assumption and do not take into account the sub-query reuse optimization[5]. This means that in general operator Bolts will have only one output stream and Spouts will have multiple output streams. As a result, we will model Spouts slightly differently from Bolts.

6.4.1 Model of Spouts

In general, the performance of Spouts will depend on the performance of sending and filtering events. As we can see in Figure 6.5, performance of filtering can be measured in predicate complexity, the number of comparisons that the predicate contains. In this benchmark we used the following predicate, which always evaluates to false (thus avoiding lazy evaluation):

$$x == 0 \mid\mid x == 1 \mid\mid x == 2 \mid\mid ...$$

The figure shows that filtering is a very fast operation. In particular, consider the filtering complexity of 200 (a really big predicate), which results in the filtering throughput of 10^7 events per second. This is roughly ten times more than the maximum task sending throughput. To keep things simple, we exclude the predicate complexity from the Spout model, and include just the performance of sending events.

[5]This optimization would not be applied unless queries share some part.

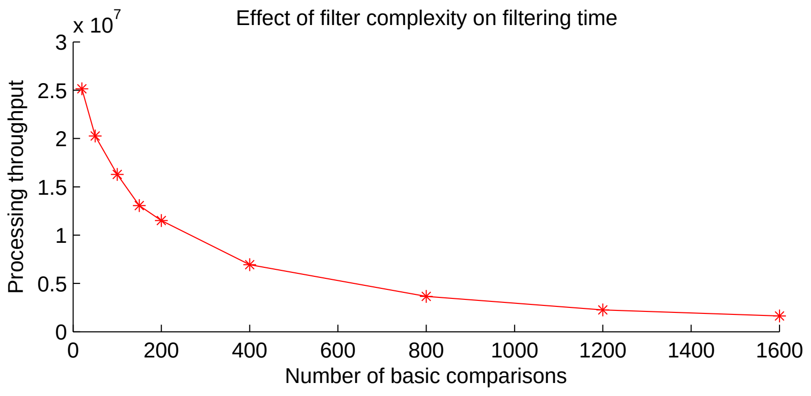


Figure 6.5: Performance of filtering events on predicates.

The performance of sending an event depends on the number of output streams (i.e. the number of subscriber Bolts) and also on their type. Replicating an event to all output streams is obviously slower than sending an event to a shuffled stream. The performance of one Spout task sending events to different types and count of output streams can be seen in Figure 6.6. We notice that the time (here inversely proportional to throughput) to send a single event on a shuffled stream stays constant, whereas for replicated streams it increases linearly with the number of subscribers. However, in the latter case the total number of sent events per second stays roughly the same (the dash blue line)[6]. It is also interesting to see that sending an event to one subscriber on replicated stream is about twice as fast as sending it on a shuffled stream. This is caused by the overhead of randomly shuffling events to decide their destination. These observations suggest that we could use the actual number of events sent as a basis for a performance model. This way we abstract from different stream types.

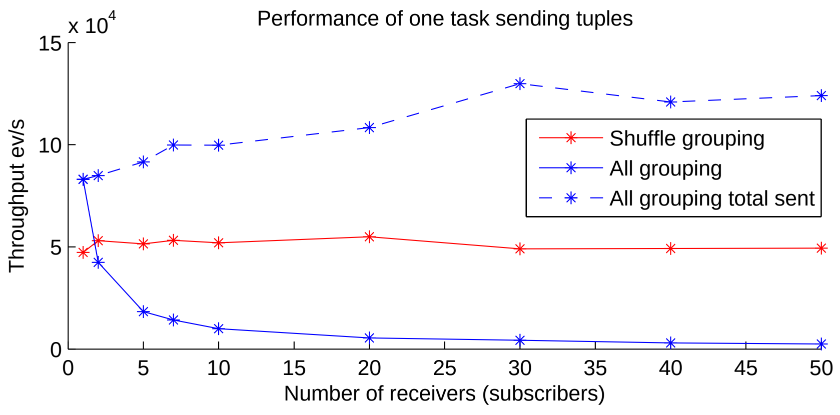


Figure 6.6: Sending throughput for one task with varying number of subscribers and different stream groupings (no event batching).

Let $S_{shuffle}$ be the set of Spout's output streams that are shuffled and S_{all} the set of replicated streams. Also let P_s be the number of subscribers for a stream s (i.e. the parallelism of target component), $rate_{in}$ be the declared Spout input adapter rate and R be a constant ratio of the speed of sending events on a replicated stream, as compared to shuffled stream. We first compute the number of events that must be sent by Spout to different Bolt tasks per second as:

$$rate_{send} = |S_{shuffle}| * rate_{in} + \frac{1}{R} \sum_{s \in S_{all}} P_s * rate_{in}$$

[6]This means that the time to send an event to a single subscriber of a replicated stream is constant

That is, for each shuffled output stream an input event has to be sent only once, but for each replicated stream the event has to be sent as many times as there are subscribers. However, since sending of events for replicated stream is twice as fast as compared to a shuffled stream, we scale the number of events that have to be sent by R. Hence, we obtain the number of events that have to be sent on a single shuffled output stream.

Now we evaluate the sending performance of a simple Spout for only one shuffled output stream, but with changing parallelism. As we will see in the next chapter, the sending throughput increases linearly with increased parallelism, and hence we fit a linear model. Let P be Spout parallelism, K error margin[7] and A, B constants. We obtain the following model for the parallelism of Spouts:

$$P = A * K * rate_{send} + B$$

We compute the parallelism by plugging in the number of events that a Spout has to send (i.e. $rate_{send}$). Since this is a real number, we need to round it up to the nearest integer. The model constants are shown in Appendix A.

6.4.2 Model of Bolts

Modelling of Bolts is simpler than modelling of Spouts. We evaluate each type of Bolt individually with regards to different parallelism. The evaluation will determine what is the usual composite throughput of receiving, processing and sending events for the Bolt. That is, we do not distinguish individual parts of a Bolt, but rather look at its performance as a whole[8]. An alternative approach is to evaluate and model each individual part of a Bolt and try to estimate composite throughput through queuing network analytics. However, this would result in complicated equations, which might not resemble true Bolt performance.

Previously, we assumed that Bolts will typically have only one output stream. Thus, we do not need to include output stream count in our model. However, it would be useful to include the type of output stream, since the performance of a Bolt with replicated output stream may be slower than with a shuffled stream. This would require doubling the amount of evaluation experiments made[9], and hence we omit it for simplicity. We know that the slowest part of a Bolt is processing an event and variations in sending events can be covered by an error margin. Also note that the performance of receiving depends only on input event rates and will be implicitly included in a model. Because of the above reasons, our model depends only on event arrival rates[10]. We will now present the models that we obtained by curve-fitting the evaluated performance of each individual Bolt on a fixed evaluation scenario (concrete constants used are shown in Appendix A). $rate_{in}$ will denote the sum of all input event rates and K the error margin.

For the projection Bolt, we observed that throughput increases linearly with increased parallelism (recall Figure 6.3). Thus, to compute parallelism we use the linear model:

$$P = A * K * rate_{in} + B$$

We use the same model for the exception operator. Since in exception undesired events, which have to be replicated occur only rarely, the performance increases linearly with new replicas.

[7]To cope with imprecise model and filtering overhead, we artificially increase the number of events that have to be sent by 10%.

[8]In the case for Spouts we based our model only on performance of sending events.

[9]Also in the case of replicated streams we would need to take into account target Bolt parallelism.

[10]We also assume that events arrive at the same rates for all input streams.

With slight modification the linear model also appears to be a good fit for the any iteration operator:

$$P = min(A * K * rate_{in} + B, It_{max})$$

The iteration performance rises linearly until It_{max} parallelism is reached and performance gain from replication diminishes. At this point, event replication overhead starts dominating performance gain from parallelising detection. Thus, we bound the parallelism of any iteration by It_{max}. Also, as greedy iteration is not parallelisable, we always set its parallelism to one.

The relationship between event arrival rates and parallelism for next Bolts (greedy or any) and conjunction Bolt nicely fits the exponential equation:

$$P = A * e^{K * rate_{in} * B}$$

This is because the higher parallelism we have, the smaller improvement in throughput we see. In other words, to achieve higher throughput we require exponentially more parallel replicas. This result can be attributed to the fact that with more replicas we need to replicate more events, thus increasing demand for time spent doing IO instead of event processing. However, the exponential growth of parallelism is slow and we will see in the next chapter that each new replica leads to a fair improvement of event processing throughput.

6.4.3 Effects of event timing

So far we have not taken into account effects of operator predicates, in particular the event durations. The described models were obtained by evaluating performance of operators on fixed scenarios with fixed predicates. However, the performance of operators detecting events within some matching windows (i.e. next, conjunction and iteration operators) highly depends on the size of the windows, which is usually specified through maximum event duration or iteration length sequence. To see this, consider the performance of greedy operator for different parallelism and event durations in Figure 6.7[11]. The event throughput is higher if the used duration windows are smaller. This is because smaller windows lead to shorter event matching queues, and hence less operations have to be made during event matching.

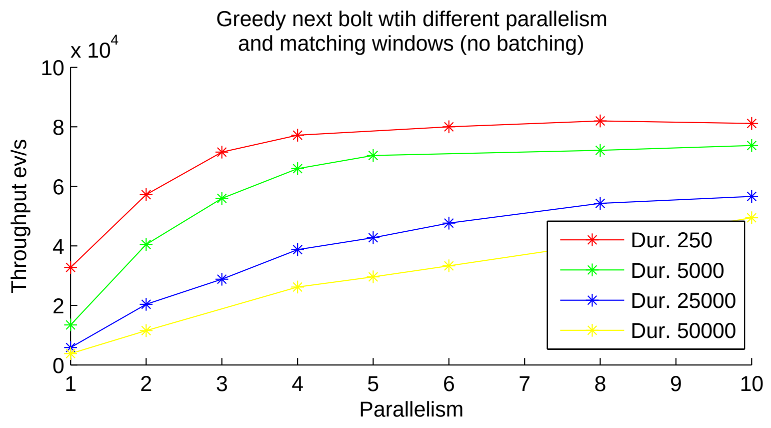


Figure 6.7: Bigger matching windows result in lower throughput.

Our aim is to include duration windows into models of individual operators, to better predict their performance. We demonstrate our approach only on the greedy next operator, but this could be applied to other operators as well. The starting point is to evaluate how the operator's event throughput changes for different duration windows. Thus, we only compare performance

[11]Note that we are not using batching here and hence the performance is low.

of each window size for some fixed parallelism. The goal is to determine how throughput changes with increased matching window size. In the case of greedy next operator we discovered that this follows a rational function of a form $1/x$. Let W be a matching window size and W_a, W_b be constants. Then under fixed parallelism, the throughput for window W can be calculated as:

$$T(W) = \frac{1}{W_a * W + W_b}$$

This just simply says that if we fix parallelism, then increasing window size results in smaller throughput. The exact value is determined by the fitting constants W_a and W_b. Now consider that we have a greedy next Bolt parallelism model for default window W_{def} (e.g. we evaluated the Bolt for window size 5000), which is defined by:

$$P = f(rate_{in})$$

I.e. parallelism is a function of input event rate. However, we would like to take into account the user-specified matching window W_{real}. We can do this by scaling the input event rate (i.e. throughput for default window) by a ratio of W_{def} to W_{real}. Let $rate_{def}$ be the new scaled input event rate resulting from scaling event rate for window W_{real} into event rate for window W_{def}. We compute it as follows:

$$rate_{def} = rate_{in} \frac{T(W_{def})}{T(W_{real})}$$

The new scaled event rate $rate_{def}$ is the number of events per second that the operator would have to process if window W_{def} was used instead of W_{real}. Thus, we can just take the scaled input event rate and plug it into original parallelism model to compute component parallelism:

$$P' = f(rate_{def})$$

We will see how this approach fits real measured performance in the evaluation chapter.

6.5 Modelling of other parameters

6.5.1 Batch size

Since batches significantly improve performance, we are interested in determining an appropriate batch size in terms of the number of internal events, which it carries. To do this, we set up the following experiment: A non-filtering Spout will send event batches over shuffled stream to a projection Bolt, which will discard them. The Bolt will have enough projection tasks, allowing a single Spout task to always emit new events. The number of internal events contained in a batch will be varied, but their payload sizes will be fixed (here 64 bytes). We will measure the number of events that the Bolt can receive per second and the network usage.

The measurements of sending throughput and network utilization for different batch sizes can be seen in Figure 6.8. Note that the experiment was performed over 1 Gbps LAN network. We can see that if batching is not applied (i.e. batch size is 1), the sending throughput is only about 50 000 events per second. The throughput rises in a logarithmic curve, radically up to the event batch of size 100 and then increases at a slow steady rate. The corresponding network usage rises accordingly, but does not saturate the available 1 Gbps capacity.

For small batch sizes (e.g. up to 50) the bottleneck is in high CPU overhead at the Spout. This is caused by generation of message IDs, determination of destination Bolt task, serialization, context switching between native ZeroMQ and Storm, and ZeroMQ overheads of sending a

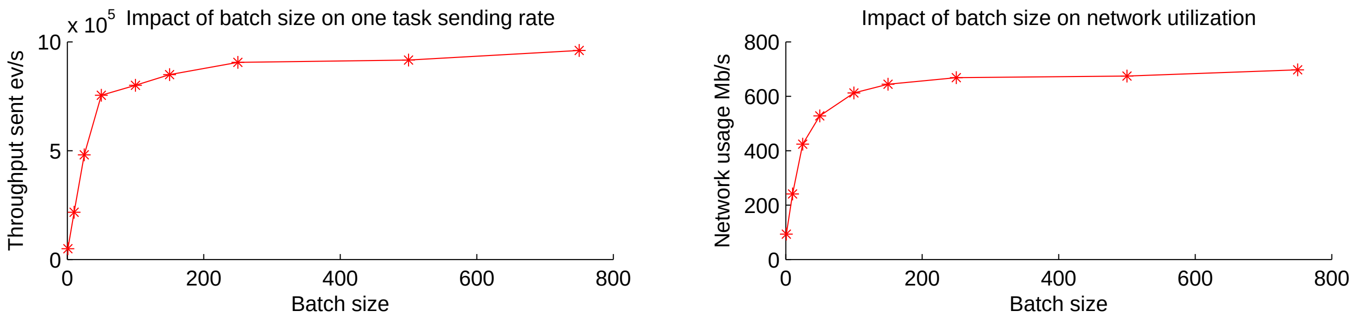


Figure 6.8: Determining optimal batch size.

single message. As we increase the batch size, the CPU usage gets smaller and the bottleneck starts shifting towards network. This is what we want, since we want to free CPU resources for event processing and saturate available network resources. From the figure it can also be seen that the design of Storm allows only to use of up to 80% of available network bandwidth[12].

We chose the batch size which yields a high sending event throughput. This allows CPU resources to be used for event detection instead of IO. Figure 6.8 demonstrates that any size above 300 is good, as it maximizes the throughput. We set our system to use an arbitrary batch size of 500. Going beyond this does not significantly improve performance, but rather results in higher event detection latency.

6.5.2 Workers count

The role of workers is to receive and send events by a dedicated IO thread and to run component tasks. These perform CPU-intensive operations, and we observed that a single task under heavy load will use all the resources of one CPU core. However, usually the task will never be perfectly saturated by incoming events because of the error margin that we include in the models and because of rounding up the required number of tasks. This suggests a joint use of one CPU core by IO thread and the component task, which works well in practice, as we see a CPU use of between 100% (one full core) and 200% (two full cores) depending on the type of component and output streams. For example, a Bolt that outputs events onto one stream will not require to do so much IO as a Spout, which needs to replicate events to many outgoing streams. For Bolts we observe that using one component task per worker uses up to 120% of a single CPU core (in the presence of multiple cores). As a result, we decided that a node with n CPU cores can accommodate up to n workers.

It might be a good idea to run multiple tasks on a single worker, since the tasks could communicate over shared memory, thus lowering IO overhead. If this was the case, a node with n CPU cores could run exactly one worker with n tasks. To verify this, we set up the following experiment: A Spout was sending 64-byte payload events to a projection Bolt, both of them with parallelism 1. The component tasks were run at one worker, on two separate workers on the same machine, and on two separate workers on two different machines. We measured the throughput of sent events per second[13]. Results of repeated runs of these experiments are shown in Table 6.2.

The communication between tasks at the same machine yielded approximately the same through-

[12]By using iperf tool, we measured that a single TCP/IP connection between two nodes can use up to 92% of available bandwidth. Storm is not too far from this.

[13]If communication between two tasks happens through shared memory, the measured throughput should be higher

Positioning of Spout and Bolt	Throughput (ev/s) (batch size 500)
Same worker, same node	368 407
Different workers, same node	832 706
Different workers, different node	825 169

Table 6.2: IO performance measurements resulting from different positioning of tasks.

put as communication between tasks on different machines. This implies that IO between tasks on the same machine is not done through shared memory, which is backed up by the fact that we also observed exchange of events over the localhost network interface. After discussion with Storm developers we realised that Storm does not perform this optimization. Furthermore, we noticed slow performance of exchanging events when tasks run on the same worker. We hypothesise that this is due to a single ZeroMQ thread handling both event receiving and sending[14]. We obtained the same relative results when event batching was not used.

As a result of these measurements, we position one component task per one worker, and each machine will run as many workers as it has CPU cores. Each worker will roughly consume about one CPU core, or a bit more if it is heavy on sending events (e.g. Spouts). Thus, resources will not be wasted. Also note that the worker count is computed by using real numbers for estimated number of component tasks. For example, if model estimated that two components require 1.1 and 1.2 tasks, the total of 2.3 tasks, only three workers will be used for 4 spawned tasks. This may have the disadvantage that some workers might be overloaded, but avoids wasting total of one whole CPU core by unneeded fourth worker.

6.5.3 Acknowledger count

The number of acknowledger tasks for throttling is computed as half of the number of required workers. The heuristic suggested by Storm community is to use the same number of acknowledger tasks as there are workers for the topology. Since batching is always used, we observed that using only half of the tasks is more than enough. However, we do not need to be too precise and can afford to have more acknowledger tasks as needed, since these are extremely lightweight. A precise estimation of their count could be done by modelling acknowledger tasks in the same way that we modelled operators.

6.6 Summary

In this chapter we explained computation of parallelism for each topology component by first estimating operator input and output event rates, and then using a parallelism model obtained by evaluation and curve-fitting of individual Bolts. We also focused on the computation of other parameters, in particular determination of optimal batch size, and the required count of workers and acknowledger tasks. We omitted the performance plots, from which parallelism models originate, as this is described in the next chapter.

[14]We could configure multiple ZeroMQ threads per worker to improve this, but this is inconvenient.

Chapter 7

Evaluation

7.1 Goals

In Chapter 5, we described how correctness of complex event detection is established. This is done by using unit testing at component level and extensive acceptance testing at the system level. Thus, the validity of the Step compiler and the Step runtime framework will not be discussed further. Instead, we are now interested in the performance of complex event detection, and its improvement due to parallelism and optimizations.

Therefore the evaluation goals are the following:

- *Measure performance of individual event detection operators and its improvement due to parallelism*
- *Investigate the applicability and limitations of our models*
- *Determine the role of optimizations in improving performance*
- *Examine performance of real event detection queries*
- *Summarize properties of Storm*

We will first explain the used configuration of the Storm cluster and the method used to measure the performance of topologies. Afterwards, we will evaluate individual operators. The goal is to determine operator event processing throughput with regards to different parallelism, examine how well our models fit the measured data and explain the measurements. Finally, we will evaluate more complex queries applicable to real scenarios of fraud detection or stock monitoring, and describe the most important traits of Storm.

7.2 Our approach

All evaluation was done on the Emulab cluster facility provided by University of Utah and was done by running generated topologies with different parameters on a fixed number of physical nodes. The nodes were 64-bit 2.4GHz Quad Core Xeon CPU machines[1] running Ubuntu 11.10 and connected in a star topology over 1 Gigabit Ethernet, as illustrated in Figure 7.1. We distinguished between two types of nodes: the master node and multiple slave nodes. The master node was used to keep the state of the Storm cluster and run Zookeeper, Ganglia,

[1]2.4 GHz 64-bit Quad Core Xeon E5530 "Nehalem" processor, 5.86 GT/s bus speed, 8 MB L3 cache, 12 GB 1066 MHz DDR2 RAM. Based on the Dell Poweredge R710

Nimbus and Storm UI daemons. Actual complex event detection was done on the slave nodes, which ran Supervisor daemons. The Supervisors were configured to be able to create as many workers as there were CPU cores on a machine (in our case four). The usual set-up consisted of one master node and ten slave nodes.

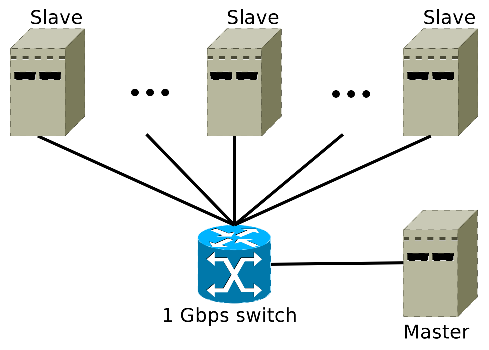


Figure 7.1: Emulab cluster organization during evaluation.

The performance of Storm topologies was measured primarily by event throughput metric, i.e. the number of input events that a topology or individual operator could process during a fixed time period. The topologies were run with the reliable Storm messaging API, such that component input queues were throttled and did not overflow. This meant that apart from topology tasks, there were a number of acknowledger tasks running at the slave nodes, as determined by our models.

For most experiments, an event query was specified in the Step language, compiled into a topology and deployed to the cluster. Then the topology was left to run for 60 seconds, for its processing to settle down and the throughput to become steady[2]. Afterwards, three event throughput measurements were taken, each over a time period of 100 seconds, which were then averaged to yield the final throughput. If the three measurements showed higher standard deviation than 6-10%, the experiment was repeated.

The throughput for each topology was measured by a custom utility called *monitor* (illustration in Figure 7.2). Monitor used the Storm Thrift interface to periodically poll topology statistics from the Storm UI daemon running on the cluster, which was collecting the counts of sent and received events for each topology stream on a second granularity. Depending on a type of experiment, the monitor accordingly summed up statistics for relevant individual streams and determined the throughput of a component. The measurements were collected in spreadsheets and later examined in the Matlab tool. For some topologies the CPU, memory and network resource usage were also measured, which was done by using the Ganglia monitoring system.

Evaluation of a single topology took typically about 7 minutes, which was the reason why we decided to automate some of its aspects. We implemented a program called *batch monitor* which automatically submits, evaluates and kills a single topology with different configurations. The program took its input from an experiment file, which could specify the component parallelism, the length of internal event queues and the number of needed acknowledger tasks. The topology could then be submitted to the cluster with different combinations of these parameters and its results were stored in a spreadsheet file, thus simplifying some tedious work.

[2]The initialization time was used to let input and processing queues of individual components grow to their average length.

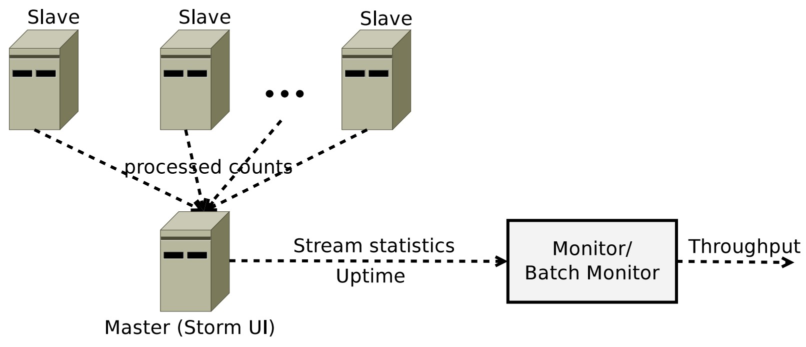


Figure 7.2: Monitoring topology throughput: statistics from different nodes were collected by Storm UI and reported to the monitor.

7.3 Evaluation of individual operators

Figure 6.2 from the previous chapter, explains the evaluation method for individual operators. Each operator was evaluated on a fixed scenario, but with varying parallelism and predicates. The scenarios were chosen such that they were conservative, as we wanted to make sure that the measured performance is easily achievable by typical queries. All operators except the union were compiled into a single parallelisable Bolt, and were tested as a part of a topology.

When evaluating Spouts, the topology needed to have enough projection Bolts, such that they would not block when sending events. Then the sending throughput was measured, by summing up the events that were sent on Spouts' output streams per second. When evaluating operator Bolts, enough parallel Spouts and projection Bolts needed to be created, so that the operator could always process new events and would not block on sending detected events. When measuring Bolt's processing throughput we distinguished between *effective throughput* and *total throughput*. Total throughput measured the total number of events that were processed by all replicas of a Bolt, which could have included replicated events multiple times. Let S_{all} be the set of Bolt's replicated input streams, S_{shuf} the set of shuffled input streams, and E_s be the number of received events from stream s. Then total throughput was calculated as:

$$T_{tot} = [\sum_{s \in S_{all}} E_s + \sum_{s \in S_{shuf}} E_s] \ / \ time$$

On the other hand, the effective throughput indicated how many unique events were processed by a Bolt. This was a more useful measure, since it counted each replicated event only once and corresponded to the number of actual input events processed. We calculated it as follows:

$$T_{eff} = [\frac{\sum_{s \in S_{all}} E_s}{|S_{all}|} + \sum_{s \in S_{shuf}} E_s] \ / \ time$$

Every experiment was performed on input events that were generated by input adapters. The data were often random or followed some fixed pattern. An alternative approach to this would be to set up a central or distributed event queue, such as Kestrel or Kafka, which would contain pre-generated events. Input adapters would then receive events from this queue and possibly replay them many times. This would measure more realistic performance of Spouts, since the time needed for receiving and input event from external source would be included. However, for simplicity we did not use this approach and rather expect input adapter overhead to be included in error margins. Similarly, detected complex events received by output adapters were not logged into an external queue, but discarded.

7.3.1 Spouts

In Figure 6.5 and Figure 6.6 from the previous chapter, we discovered that the predicate filtering time for an event increased linearly with its complexity, but overall was very high. This could be attributed to our approach of using offsets to access event fields in a constant time. We also found that the performance of sending depended much on the type of the output stream and sometimes on the number of subscribers, which was taken into account when modelling Spouts.

To see the effects of parallelism affects on sending and receiving of events, consider Figure 7.3. Here, 64 byte payload events were sent between two Storm components on a shuffled stream, and event batching was not used. When evaluating sending, the target component had parallelism three times higher than the sending, and when evaluating receiving, the sender had three times the parallelism of the receiver[3]. The sending performance increased linearly with parallelism. Doubling the parallelism resulted in 65 - 70% increase in sending throughput. For example, sending event rate at parallelism 4 was measured to be $184,732$, and at parallelism 8 to be $315,694$. Additionally, the event receiving throughput was slightly higher than the sending throughput. This is best visible at parallelism 2, when receiving was about 29% faster than sending. This can be due to higher overhead of sending, which needs to perform event shuffling to determine event destination task[4].

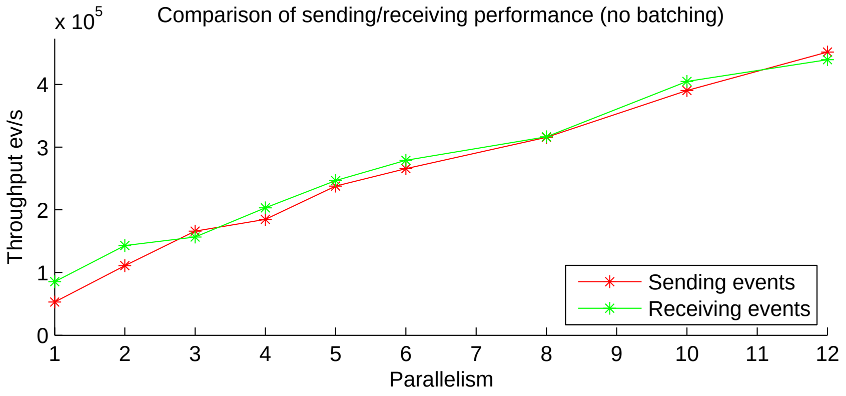


Figure 7.3: Comparison of sending and receiving events.

Consider now Figure 7.4, which demonstrates the improvement of the sending performance with the introduction of batching. For parallelism of 1, the performance increased almost ten times, from $85,330$ to $820,821$ events per second. This is due to the fact that Storm required a lot of CPU resources for sending and receiving events. If we send events individually, each of them needs to be shuffled, serialized and given a new message ID. We also suspect that Storm calls ZeroMQ layer for each event individually, which requires a context switch between JVM and its native code. With events batching, this CPU overhead gets amortized over many events and bottleneck shifts towards network[5]. It is interesting to see that doubling the parallelism resulted in 80% increase in sending throughput, which was more than in the case of sending individual events. This means that batching also leads to a better scalability.

In Figure 7.4 it can be seen that using a linear Spout parallelism model fits the measured data well. Here, the fitted curve was always under the measured data, because of the inclusion of a 10% error margin.

[3]So senders or receivers do not block waiting for each other to consume/provide events.
[4]We consider points 3 and 12 for sending to be outliers.
[5]Recall Figure 6.8, where similar sending event rates lead to 70% network saturation.

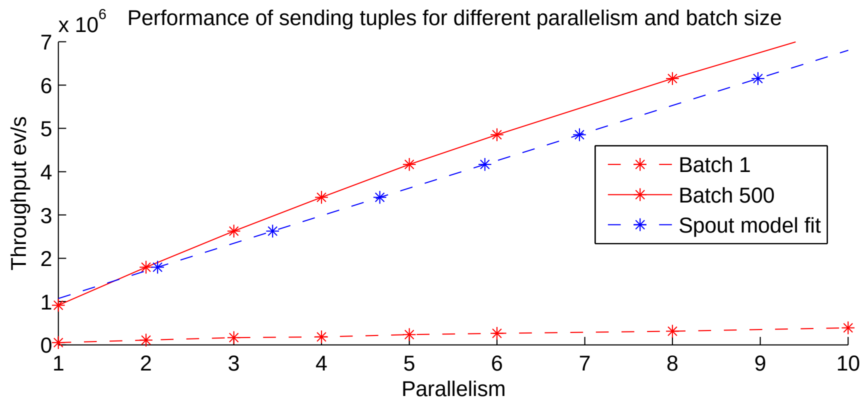


Figure 7.4: Evaluation of Spout sending performance.

7.3.2 Bolts

Projection

During evaluation, a projection Bolt was receiving 64-byte payload events organized into batches of 500 from many Spout tasks. The Bolt performed all projection on eight 8-byte payload fields, and we measured its effective throughput of receiving, processing and discarding events. Since the all projection has the same complexity as the field projection, we did not distinguish between them.

The evaluation results of the projection Bolt can be seen in Figure 7.5. The throughput of projected events increased linearly with parallelism. This was because each projection replica received only a fraction of events, but events were never replicated. Thus, with doubled parallelism, the projection Bolt had almost twice that much capacity to process events. For example a single task was able to project $274,000$ events per second. When we increased the parallelism to 2, the throughput almost doubled to $527,000$, and with parallelism 4 it again doubled to $1,027,000$ events per second.

A linear model for parallelism fits the measured data well. The model without error margins (dashed green line) was not sufficiently conservative, as it sometimes predicted higher performance than the actual. However, the inclusion of 10% error margin (dashed blue line) resulted in a better performance estimation, which was always lower than the measured, thus also leaving some margin for the processing of the output adapters.

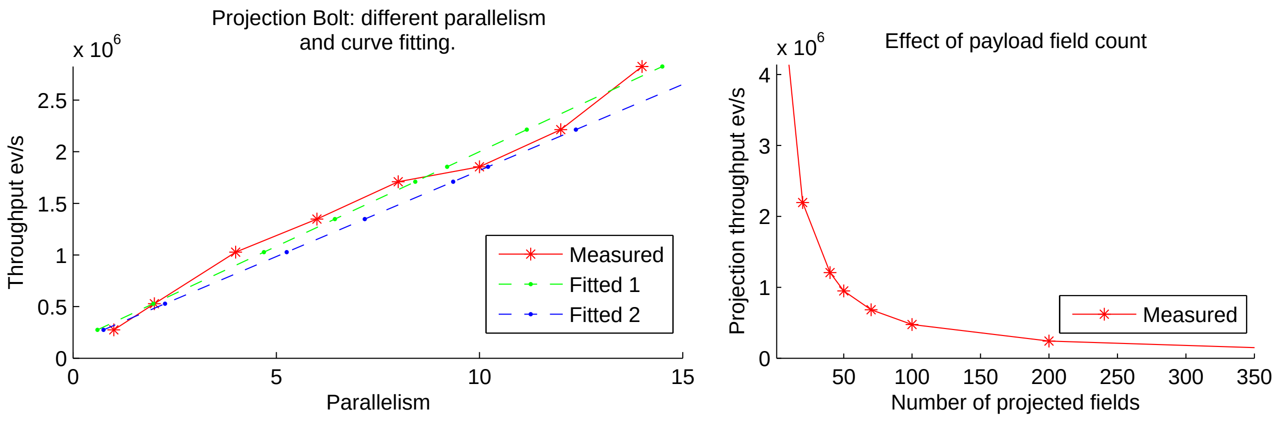


Figure 7.5: Evaluation of the projection Bolt.

Recall that for simplicity we did not include the number of projected fields in the operator model. Figure 7.5 shows that this affects performance of projection. If the total number of fields in a received event was 200, the linear projection algorithm was only capable of processing about 241,000 such events per second. The result could be a halved total processing throughput, which would not be estimated by our model. Thus, it would be useful to also include number of projected fields in the Bolt model.

Next

Both greedy next and any next Bolts were evaluated on the following query, which detected a pair of stock quotes for the Google company, only when the second quote had a higher price:

```
external Stock(name: string, price: int) < "StockSourceAdapter" [500000.0]

NextEval(*):
    Stock/Stock1
      ;[Stock1.name = "GOOG" && Stock2.name = "GOOG" &&
        Stock1.price < Stock2.price && Stock1.price > 0 && dur < X]
    Stock/Stock2 > "DiscardingAdapter"
```

The input stock quotes were generated by input adapters and contained price fluctuating randomly around one value[6]. Every ten consecutive events had the same end timestamp, to simulate multiple stock quotes arriving at the same time. Also to simulate a conservative scenario, when many events get detected, all generated stock quotes were filtered for the Google company. As a result, the number of events detected by the next Bolt was as high as half of the events received.

The evaluation results for the greedy Bolt can be seen in Figure 7.6. We ran the same query with different parallelism and event batch sizes, and measured the effective throughput of received, processed and sent events. Notice that with increased parallelism the throughput did not scale linearly, but rather logarithmically. This was because the next Bolt required one of its streams to be replicated. Increasing parallelism of tasks caused higher IO overhead due to each parallel task receiving all replicated events. The number of total processed events hence increased linearly, but the effective throughput increased only logarithmically. We will explain and validate this further in the next section.

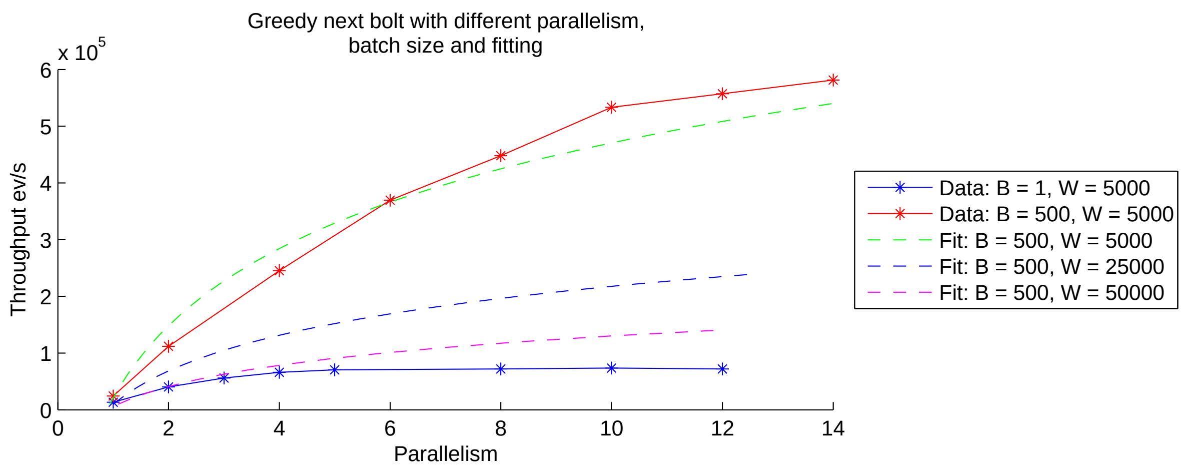


Figure 7.6: Evaluation of the greedy next Bolt (B is batch size and W is event duration).

[6]This would ensure that many events were detected.

Note that the change in batch size from 1 to 500 radically increased the effective throughput. If the parallelism was 2, then not using batching (blue line) resulted in effective throughput of 40, 469, whereas when using batching (red line) the Bolt could process as many as 111, 811 events per second. Also, with batching, the operator scaled better. For example, if we increased parallelism from 2 to 6 in the case without batching, the effective throughput would increase only by 75%. However, with batching, the throughput would increase by 230%. This was because batching significantly reduced IO overhead, and thus overhead of receiving replicated events. This was the main factor that reduced scalability.

It is also interesting to examine how well our greedy next model predicted performance. The model correctly estimated that parallelism increased exponentially with increased throughput. This meant that throughput increased logarithmically with higher parallelism. The fitted model is displayed in Figure 7.6 by dashed lines, each showing estimated performance for different window sizes. The estimated curve for a duration window of 5000 only roughly fit the measured data. The fitted exponential function was too steep. This could be improved with more data, especially for higher parallelism. However, this was not possible due to limited computational resources.

The evaluation of the any next Bolt with different parallelism and matching windows can be seen in Figure 7.7. The evaluation method and query were the same as for the greedy next Bolt. Notice that any next Bolt was significantly slower, as it matched and output many more events for each received. For example, at duration 100 and parallelism 4, any next was capable to detect 323, 605 events per second, whereas the effective throughput of greedy next was 605, 138. Given that the maximum duration of the any next predicate was D, the operator would output on average $D/2$ detected events for each received event[7]. This meant that the Bolt was mostly affected by the performance of sending events, rather than matching them (the windows were relatively small).

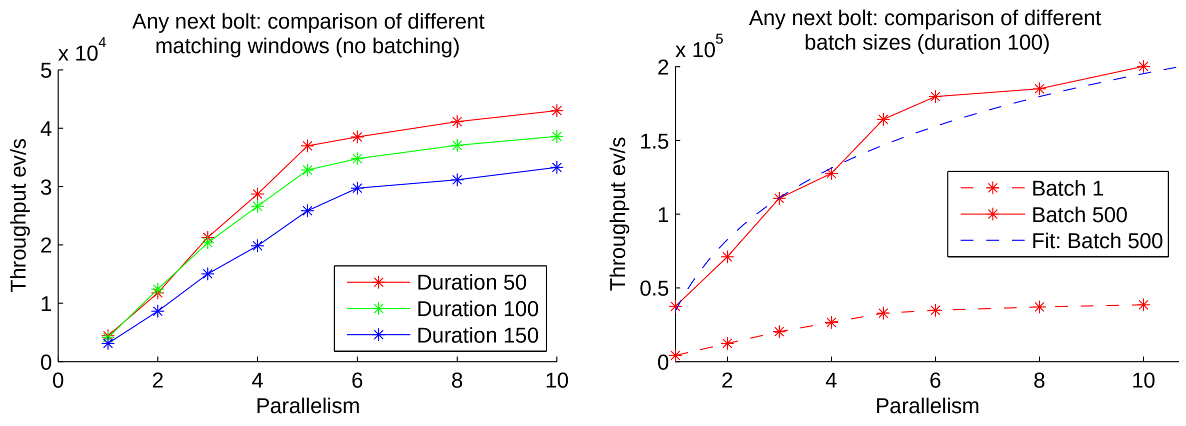


Figure 7.7: Evaluation of the any next Bolt (B is batch size and W is max. event duration).

The scalability of any next Bolt was similar to that of the greedy next Bolt. This was because in both cases replication caused high overhead. Looking at the batched performance, we noticed that effective throughput of any next Bolt also followed adjusted logarithmic curve[8], and our exponential model for parallelism applied. Also notice that the performance of the any next Bolt significantly increased with batching, as this improved the performance of sending events. For parallelism 2 the performance improved by a factor of 6. Also note that bigger event matching windows resulted in lower effective throughput. For example, at parallelism 6 without event

[7] A single quote from left stream would match $D/2$ quotes from the right stream, since all stock quotes were for Google and had random price (i.e. either smaller or bigger/equal).

[8] We can consider measured data for parallelism 5 and 6 to be slight outliers.

batching, increasing the maximum event duration from 50 to 100 resulted in 10% slowdown. For simplicity we did not include these matching windows in the any next model, but this could be done the same way as for the greedy next Bolt.

Note also that the evaluation of any next is fairly conservative. Firstly, we assumed that it detects and sends many events, which is a time consuming operation. Typically, less than a half of matched consecutive events will satisfy next predicate. Secondly, the input event rates are the same for replicated and shuffled streams, which is the worst case scenario with regards to replication[9].

Exception

For evaluation of the exception Bolt we used a scenario where no exception events and only desired events were received. Note that this was a pessimistic scenario, as all received desired events had to be sent further, thus requiring many slow IO operations. As such, this scenario could be also seen as measuring the performance of receiving an event, stabilizing it, and sending it further. We evaluated only the “not before” type of exception Bolt, which we expected to show the same performance as the “not during” type of exception[10]. During evaluation the Bolt was receiving desired events with payload sizes of 64 bytes from a Spout and was sending them to further to a projection Bolt. More precisely, the following query was run:

```
external YesEvent(payl: string) < "FixedStringAdapter(64)" [100000.0]
external NoEvent(payl: string) < "NoSourceAdapter" [0.0]

Query(*): !NoEvent ; YesEvent > "DiscardingAdapter"
```

The measured effective throughput can be seen in Figure 7.8. The use of a linear model can be justified by considering the high variability of the measurements. The estimation error was relatively high and ranged between 4% (parallelism 10) and 28% (parallelism 2 and 4). However, the estimation was pessimistic, and in almost all cases correctly predicted lower performance than the actual. The high variability in seen performance can be the result of a single IO thread having to handle both high receiving and sending event rates. It could be that the thread first sent buffered events and then received new events, causing bursty behaviour and fluctuating throughput.

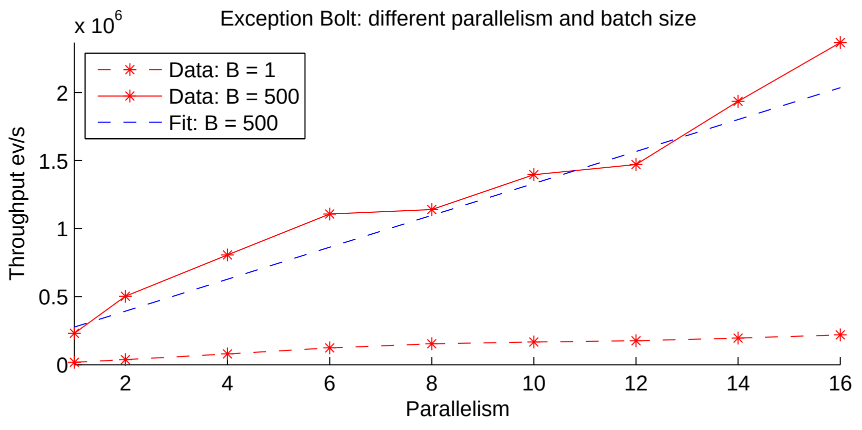


Figure 7.8: Evaluation of the exception operator (B is batch size).

[9]Recall that this is because any next Bolt always replicates the stream with smaller event rate.
[10]From the implementation point of view, the two cases have the same event matching complexity.

From Figure 7.8 we can also see that a maximum achievable throughput of receiving and sending events at parallelism 1 is $230,000$ events per second. This seems to be the limit imposed by the Storm IO implementation. For our hardware configuration this was quite slow and meant that in some cases it would be more useful to group multiple operators into one task. This would remove IO bottleneck, as operators within one task could communicate through shared memory. For example, since the exception operator would mostly forward received events, we could merge it with the following operator in the event pattern. We could also do this for Spouts that emit events into only one Bolt, by merging both operators into one, perhaps by some cost model rules[11].

Also notice that similarly to the other operators, the performance significantly increased when using event batching. For example, at parallelism 2 the experiment with batching performed 13 times faster than without. The exceptionally high increase in performance was due to the fact that the exception Bolt was IO bound and batching improves IO performance. For example, any next Bolt spent more time on matching events, and thus its performance improvement due to batching was not as high as for the exception Bolt.

Conjunction

Conjunction operator was evaluated on the following query:

```
external Event(name: string, x: int, y: int) < "DurationSourceAdapter(0)" [100000.0]

Query(*): Event/A
            ,[A.x + A.y = A.y + A.x && true && true && A.name = "event" && dur < X]
          Event/B > "DiscardingAdapter"
```

The events generated by the input adapter were instantaneous with increasing end timestamps, but always two events had the same timestamp. The conjunction predicate always evaluated to true and also specified the maximum event matching window. The effect of this was that within a window each input event matched exactly one other event and so the input event rates were identical to the output event rate[12]. Since there were always two events with the same timestamp, for duration D the length of the operator queue was $2D$. We varied the operator parallelism and maximum event duration, and measured the effective throughput of receiving, matching and sending detected events.

The evaluation results can be seen in Figure 7.9. Let us first examine the graph on the left side, which demonstrates how performance of the conjunction operator changed with different event duration and parallelism. Bigger matching windows resulted in slower event detection. E.g., for parallelism 1 the event duration of $25,000$ yielded the effective throughput of only 930 events per second, whereas the duration of 250 resulted in 20 times higher performance of $18,860$ events per second. More interesting is that the scalability of the conjunction also depended on the event duration. For example, if using duration 5000 or $25,000$ the effective throughput increased almost linearly with parallelism, whereas on duration 250 the increase was logarithmic. Recall from the next operator that logarithmic increase was caused by IO overhead of receiving replicated events. When using small event duration window this overhead began to dominate the event matching time earlier, as the event matching was fast. On the other hand, using bigger windows meant that IO overhead was just a fraction of event matching time, and resulted in more linear scalability. This result also applies to other operators that require replication of input streams.

[11]Such optimization is done for example in Borealis system and we leave this idea as a future work.

[12]Recall that when modelling event rates, we also assumed that each input event will match exactly one other event within a matching window

Another interesting anomaly is that, with the extremely short event duration of 250, the performance of conjunction started to deteriorate at high parallelism. The reason for this was that at high parallelism the IO overhead of event replication started to dominate the time that the operator spent on matching events. In this case, the exponential model broke down for high parallelism. To fix this, we could bound the conjunction model by some maximum achievable parallelism. We would never scale the operator beyond the maximum parallelism, as we would know that this leads to a smaller effective throughput.

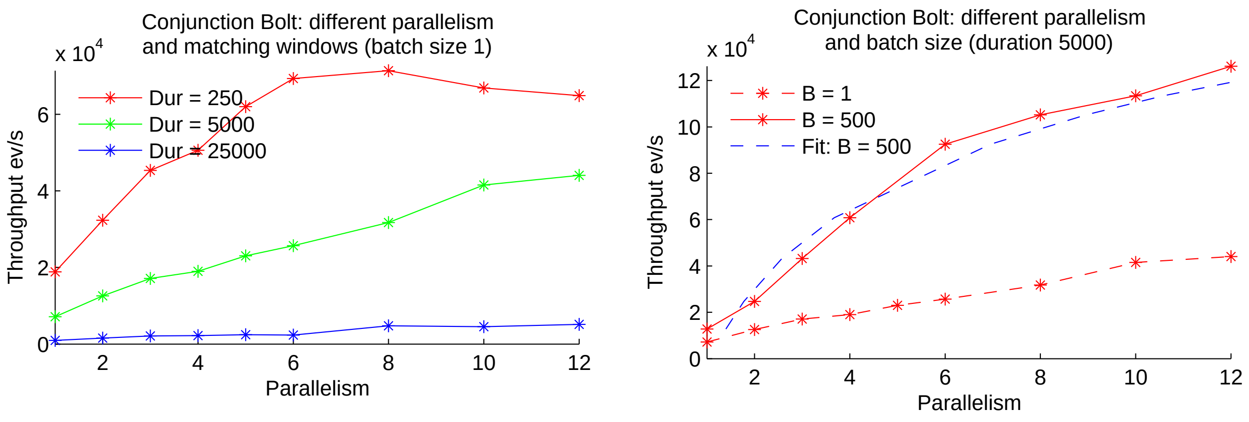


Figure 7.9: Evaluation of the conjunction operator.

Consider now the graph on the right in Figure 7.9, which shows how batch size affects performance for event duration of 5000. Similarly to other operators, we observed a significant performance improvement when using batching. At parallelism 2, batching doubled the effective throughput. Note that the performance improvement was much smaller than that of the exception Bolt. This was because we used large event matching window and the Bolt was heavy on event processing, instead of event IO. Batching optimization improves only event IO. Also note that our exponential parallelism model (dashed blue line) fits the measured data quite well. For a better fit we would need to measure performance for a higher parallelism and need more cluster resources.

Iteration

Both types of iteration operator (greedy and any) were evaluated against the following query:

```
external Stock(name: string, price: int) < "StockSourceAdapter" [100000.0]

Query(*): Stock/S
            +[prev(S.name) = S.name && prev(S.price) != S.price &&
              dur >= 0 && len < X] > "DiscardingAdapter"
```

The query detected sequences of stock quotes of the same company, in which no two consecutive quote prices were the same. Since the performance of the iteration Bolt depended on how many event sequences it stored, the query predicate bounded detected sequences by some maximum length. The input stock quotes were generated for the same company and had prices fluctuating randomly. We varied the maximum length of the iteration sequence and the operator parallelism, and measured the operator's effective throughput.

The results of any iteration evaluation can be seen in Figure 7.10. Examining the plot on the left, increasing maximum sequence length resulted in slower detection. This was because more matching sequences had to be kept at the operator and when a new event arrived, all were extended. However, the effective throughput was not high and the performance did not change significantly with different windows. Consider having a sequence of length L. In the worst case, the any next operator needs to keep all 2^L sub-sequences and with increased L the performance should deteriorate fast. The reason this was not observed, were high operator output event rates. For the given predicate, 10 sequences were detected and output for each input event. This meant that the operator was bound by its sending capabilities and not by the event matching algorithm.

We can justify that any iteration was IO bound also by looking at the plot on the right, which compares performance of different batch sizes. Since input event rates were low, a significant performance improvement due to batching could only be caused by reduction of sending overhead. For example, the effective throughput at parallelism 1 increased by a factor of 20 due to batching, which was even more than for the IO bound exception operator.This was because the input event rates were very low and the output rates were very high. Therefore, since iteration lengths had a small effect on performance, they can be excluded from the operator model. We should rather consider merging any iteration with other operators in the event pattern to minimize sending overhead.

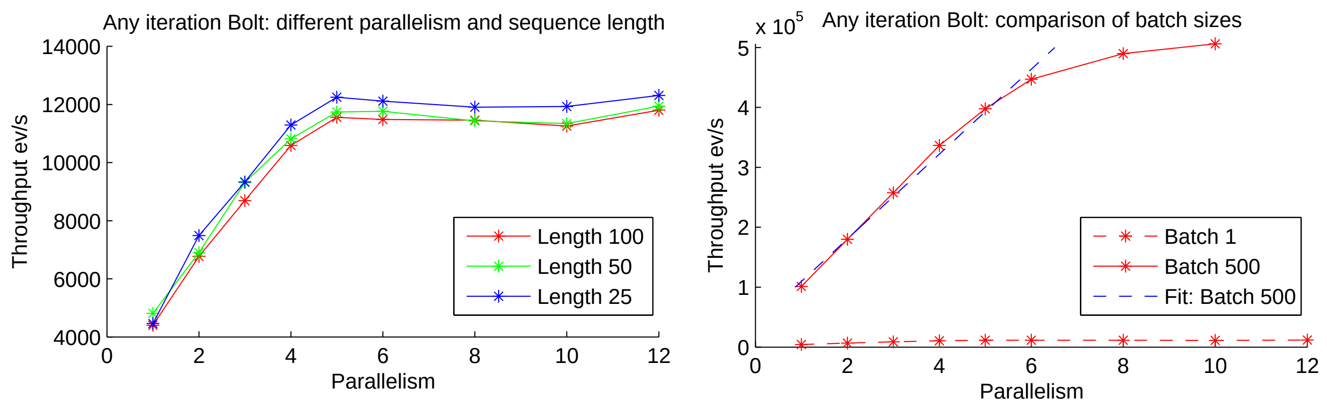


Figure 7.10: Evaluation of any iteration operator.

Recall that we modelled the parallelism of any iteration by a bounded linear model. Figure 7.10 shows, that this resulted in a perfect fit up to the parallelism of 7. Beyond this point the effective throughput did not improve significantly and thus we always limit maximum parallelism to 7 (i.e. the maximum achievable effective throughput is 470, 000 events per second). Alternatively, we could try to fit an exponential parallelism model, which would better account for the observed logarithmic decrease in throughput. However, we could not measure higher parallelism than 10 due to large number of needed projection tasks. As a result, fitting the exponential model on our data is not accurate.

Table 7.1 presents the evaluation results of greedy iteration with different iteration sequence lengths. Even without parallelism, the operator performed much better than any iteration (about five times faster). This was because for N received events the maximum of $N/minLen$ events were detected and sent ($minLen$ is the smallest iteration sequence length). Thus, the operator spent most of its resources on matching events.

Notice that the best performance of greedy iteration was achieved when the maximum iteration length was not too high and not too low. This was because with small iteration length the operator had to output many detected sequences, leading to higher IO overhead. On the other hand, higher iteration sequence length resulted in operator storing more sub-sequences, and hence in higher event matching overhead. The maximum performance was seen with length 50, when there was a balance between event processing and IO.

Max. iteration sequence length	Throughput (ev/s) (batch size 500)
10	439,123
25	424,228
50	592,129
75	526,957
100	524,729

Table 7.1: Performance of greedy iteration Bolt.

7.4 Replication overhead

Consider a next Bolt with parallelism 1, receiving R events from a replicated stream and S events from a shuffled stream. The Bolt will have to match S events with R events, by using $R * S$ comparisons. If we change its parallelism to P, each replica will receive R replicated events and match them against S/P shuffled events[13] by using $R * S/P$ comparisons. Thus, all replicas will do $R * S$ comparisons in total, which is the same as without parallelism.

Even though the number of comparisons that have to be made stays the same, increasing parallelism of operators with replicated streams does not scale linearly. This is because we did not take into account the overhead of receiving events. With parallelism P, each parallel task will receive R replicated and S/P shuffled events, the total received events being $R * P + S$. These events also have to be stabilized, requiring an order of $R * P + S$ stabilization operations. Thus, when we increase parallelism, more time will be spent by each parallel task on receiving and stabilizing replicated events, as compared to matching shuffled events against them. This means that the more CPU resources will be hogged by event receiving, instead of being available for event matching.

We conclude that the reason for which operators with replicated streams scale logarithmically is increased IO overhead of each added parallel task. This can be best seen in Figure 7.11, where we compare the total throughput of any next Bolt with its effective throughput on different parallelism. For the parallelism 1, these measurements are equal, as no events are replicated. With increased parallelism, the total throughput is higher than the effective throughput, because of more events being replicated. We can see that total throughput scales linearly, as each new replica is able to process the same amount of events as any other existing replica. However, the effective throughput scales logarithmically, as with each added replica more time has to be spent on receiving replicated events.

Another way to show that overhead of event replication leads to non-linear operator scalability is to perform the following two experiments: In both experiments we evaluated greedy next Bolt with different parallelism on the same query as in the previous section, but with varying the event rates of replicated and shuffled streams. In the first case, the Bolt would receive X replicated events and X shuffled events[14], and thus would have to perform $X * X$ event matchings. In the second case, the Bolt would receive $X/2$ replicated events and $X * 2$ shuffled events[15], and thus would also have to perform $X * X$ event matchings. We compared the effective Bolt throughput of the two cases.

The results of the experiments can be seen in Figure 7.12. In the second case, the effective throughput was much higher than in the first case, and also appeared to scale linearly. Since the number of comparisons in both experiments stayed the same, the improvement in performance was caused only by the decreased number of replicated events. Thus, replication of events was

[13] S events were shuffled to P streams.

[14] Ratio replicated events to shuffled events is 1:1.

[15] Ratio replicated events to shuffled events is 1:4.

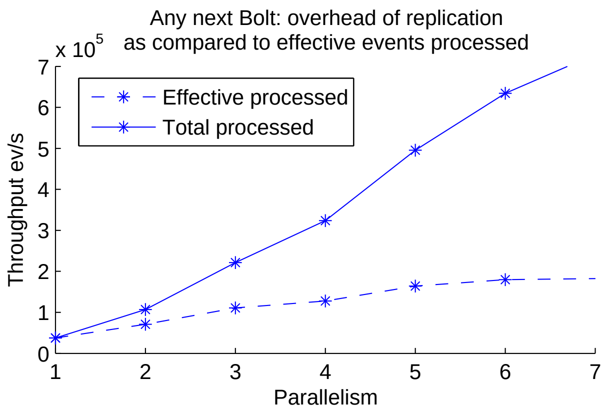


Figure 7.11: Illustration of how many events are processed in total due to replication, as compared to effective processed events.

the main reason why operators did not scale linearly. This experiment also illustrates how processing throughput depended on event rates. We obtained the best scalability if event rates on replicated streams were low.

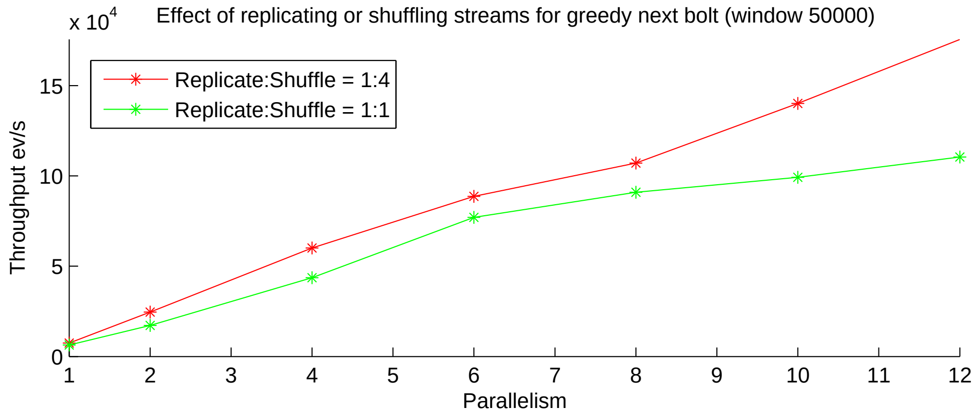


Figure 7.12: Validation of replication overhead hypothesis.

7.5 Stabilization algorithms

Performance of every Bolt is significantly affected by how fast received events can be stabilized. Therefore, we also evaluated the performance of our simple and two-phase stabilization algorithms. Both algorithms were run on the cluster hardware, but not as a part of a topology. We measured the throughput of stabilized events per second. The performance depended on the number of input streams S and the length of the stabilization queue, which was determined by the event timestamps and the input event rates. For example, if events arrived faster from one stream, the queue length would be higher, as opposed to events arriving at the same rates from all streams.

During evaluation, we simulated events to arrive at fast rate from one stream and at the same rates from the other streams. This enabled us to create scenarios where the stabilization queue would grow large, even with small number of streams. Let L be the number of events that arrive on the fast stream, before an event is received from all other slow streams. We can simulate the stabilization queue of size $L + S$ by sending L events from the fast stream and then one event from each slow stream. For example, if $L = 2$ and $S = 3$, we simulate the average queue size of 5 by stabilizing the following pattern of events (A,B,C are different event streams, and instantaneous event timestamps are in brackets):

$$A(1),\ A(1),\ B(2),\ C(3),\ A(4),\ A(4),\ B(5),\ C(5)\ ...$$

The evaluation results for simple stabilization algorithm are shown in Figure 7.13. As the number of input streams S increased, the stabilization throughput decreased at the rate of $1/S$, i.e. the time to stabilize an event increased linearly. This was because with each added stream, every event in the stabilization queue had to be marked by it, which required constant number of operations. Also note that to have 20 input streams is easily achievable in practice. Consider for example a binary operator Bolt, which receives events from two components, each of them having parallelism of 10. With 20 input streams a Bolt was able to stabilize events at the rates from $470,000$ to $820,000$ events per second depending on the parameter L.

If L was large, i.e. events arrived from one stream at much higher rates than from the other streams, the stabilization performance deteriorated. This was because average size of stabilization queue increased, as it had to store many events from the fast stream. If L was small, i.e. input events arrived from all streams at about the same rates, the stabilization throughput went up, as less events were simultaneously present in the stabilization queue.

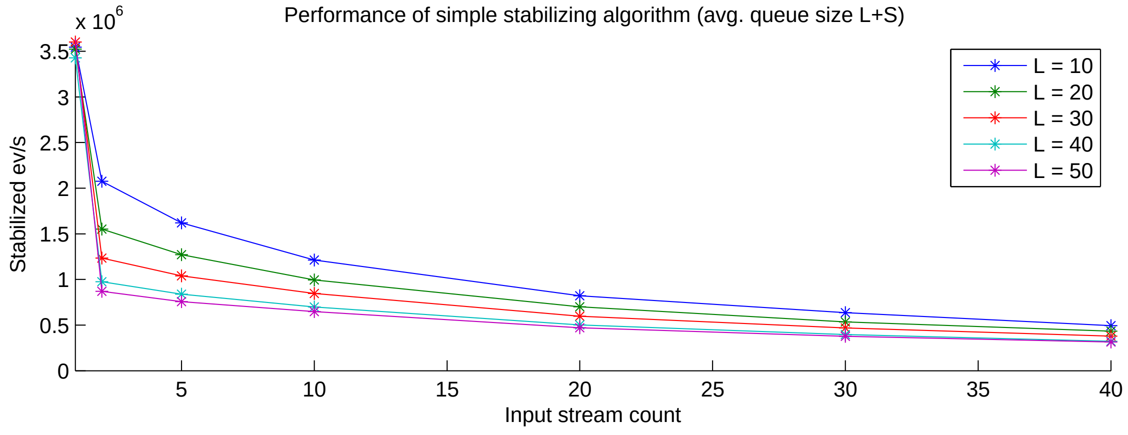


Figure 7.13: Performance of simple stabilization with regards to input stream count and variable stream event rates (high L means that one stream is much faster than others).

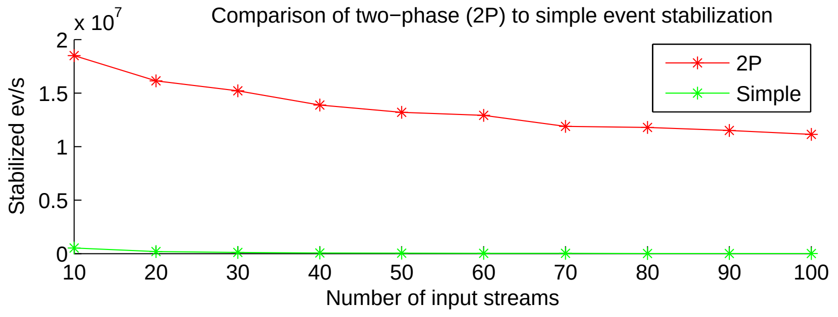


Figure 7.14: Comparison of batch to simple stabilization algorithms.

In Figure 7.14 we compare the simple with the two-phase stabilization algorithm, where events were received in batches on streams with the same input event rates. The simple algorithm enqueued all events in a batch onto the stabilization queue. With many streams this resulted in a very long queue, as events were only removed when batches from all streams were received. The two-phase algorithm solves this problem by enqueueing events in the same order as without batching. In the figure we can see extreme performance improvement. With 10 input streams, the two-phase batching was 35 times faster, and with 20 input streams up to 82 times. With 50 streams, the performance improved by a factor of more than 300. This was because the simple algorithm was very sensitive to multiple events arriving from one stream at the same time. The two-phase algorithm could cope with this by buffering events and stabilizing them in order that keeps the stabilization queue short.

7.6 Evaluation of whole queries

When evaluating bigger queries that consisted of multiple connected operator Bolts, we were interested in observing the performance of individual query components. The first goal was to determine whether the parameters of the compiled topology were sufficient to cope with the declared number of input events. The second goal was to observe how the estimated input event rates for each operator compared to the real measured input event rates. The last goal was to observe whether the modelled topology parameters were optimal, or should be adjusted. In particular, we were concerned about the number of workers and parallelism of each component.

Stock monitoring

The first query that we evaluated consumed generated stock data and detected a spike in a stock price with certain length that occurred for any company. This was done by first detecting an increasing sequence of prices for the same company and also detecting a decreasing sequence, with the use of greedy iteration. Then we used the greedy next to match an increasing sequence with the closest decreasing sequence for the same company, which effectively detected a spike. The evaluation query was following:

```
external Stock(name: string, price: int) < "StockAdapter2" [500000.0]

Spike(S1.name, S1.price):
  (Stock/S1+[prev(S1.price) < S1.price && prev(S1.name) = S1.name && len > 75 && len < 200])
       ;[S1.name = S2.name && dur < 7200]
  (Stock/S2+[prev(S2.price) > S2.price && prev(S2.name) = S2.name && len > 100 && len < 300])
       > "DiscardingAdapter"
```

The input adapter provided events that contained string name and an integer price. The name could belong to one of 30 companies and the price for each company was simulated to increase and decrease. The length of the iteration sequence was governed by a random variable. The events were instantaneous and had strictly higher timestamps.

The evaluation results for this query where half a million events were input per second can be found in Table 7.2. We see that the maximum topology event processing rate was slightly higher - $518,000$ input events per second. For this input event rate and computed parallelism of 1 the Spout was emitting events at a rate of one million events per second. However, increasing its parallelism to two did not improve the performance, so our Spout model was correct. The cause of the performance bottleneck was the greedy next iteration, which could not be parallelised. The event rate of $518,000$ was also correct, since similar values were measured during evaluation of individual operators. The input event rates to the iteration Bolts were also estimated correctly at half a million events per second.

The measured input event rate of the next Bolt was $1,644$, which was smaller than the rate of $11,666$ events estimated by our model. This is because our estimation did not take into account iteration predicate, which filtered some sequences out, resulting in lower input rate.

Because of the low input event rate, there was no need to parallelise the next Bolt. Similarly, the projection Bolt was estimated to process only $5,000$ events per second and thus did not have to be parallelised. The real effective throughput was merely 197 events per second because of the filtering in the previous operators.

We could improve the event rate estimations by measuring them at runtime and dynamically adjusting component parallelism (similarly to the Borealis system). However, Storm does not allow this. Instead, we could try to estimate selectivity of operators, but this requires some knowledge about the input data. For example, if we knew that the stock values were evenly distributed in the range between 0 and 1000, the predicate $price < 500$ could result in estimating only half of the detected events.

	Stock Spout	Iter. Bolt 1	Iter. Bolt 2	Next Bolt	Proj. Bolt
Component parallelism	1	1	1	1	1
Measured eff. throughput	1,037,645 sent	518,788	518,856	1,644	197
Estimated input rate	500,000	500,000	500,000	11666	5000

Table 7.2: Performance measurements and estimation of the stock monitoring query. Throughput and input rates are in events per second.

Recall that to estimate the number of required workers, we sum up the real computed parallelism of components. In this case, even though there were 5 topology tasks, the model suggested to use only 3 workers. Their usage of cluster resources can be seen in Table 7.3. As we can see, each worker used 140% of a CPU, which is more than the desired 100%. The reason for this is not under-estimation of CPU resources required by topology tasks, but rather non-inclusion of IO and acknowledging overhead in our model[16]. We can remedy the under-estimation by observing that IO tasks use between 20% and 40% of the CPU, and increasing the number of required workers by this flat rate.

The total network use by all workers was at the rate of 221 Mbit/s, which meant that each worker handled IO at a low average rate of 73 Mbit/s (i.e. 7.3% of available capacity per node). When measuring memory usage, each worker had a fixed JVM heap size of 784 MB, in total 2304 MB. This meant that the rest of 2181 MB were used by the native ZeroMQ layer (i.e. cache and input event queues), Supervisors and Nimbus processes.

	Worker count	CPU/worker	Network Mbit/s	Memory total (MB)
Measured resource use	3	140%	211	4,485
Ideal parameters	4	105%	n/a	n/a

Table 7.3: Resource use of the stock monitoring query.

Mobile fraud

Another evaluated query was detects one type of mobile fraud (further explained in [31]): an adversary (Eve) with secret number performs many missed calls to different customers. The adversary has a premium account, which means that any calls made to her are above the standard price, earning her money. Victims (Alice) that receive a missed call are often curious and call the adversary back, while being charged. The query for the detection of this fraud can be expressed as:

```
    external Call(fromNo: string, toNo: string, unitPrice: int,
```

[16]Recall that we assumed that there will be some capacity left for IO tasks, as each topology task cannot be fully utilized.

```
              accNo: string, userTyp: string, fromArea: string,
              toArea: string) < "CallAdapter" [180000.0]

   Fraud(Eve.accNo, Eve.fromNo):
      (Call/Eve[dur=0 && Eve.userTyp="secret"]
         ;[Eve.fromNo=Alice.toNo && dur<3600]
      Call/Alice[Alice.unitPrice > 10])
         +[prev(Eve.accNo)=Eve.accNo && len > 10 && len < 50] > "DiscardingAdapter"
```

The topology generated for this query consists of one Spout with two predicates, greedy next Bolt and greedy iteration Bolt. The next Bolt detected a missed call followed by a returned call from a victim. The iteration Bolt detected a sequence of such actions per adversary. The input events were generated and contained increasing end timestamps. Twenty-five percent of all calls were made by the adversary and were instantaneous events. Other 25% of calls were made by victims and had random duration. The rest of the calls were between other customers. The calls were made between 100 customers, each with their own number and billing account. Only one of the customers was the adversary.

The evaluation results for this query can be seen in Table 7.4. We evaluated the query for event arrival rates of 180,000 calls per second. Our utility for throttling delivered input events at the rate of 174,000 per second[17]. The Spout had to send events at double of this rate, which was easily handled by the parallelism of 1. Also, most of the events were filtered, and so only about a half of the estimated events were delivered to the next Bolt. Because of the predicate filtering, the measured effective throughput of the next, iteration and projection Bolts was lower than the estimated, thus proving that our event rates estimation was very conservative.

	Spout input	Next Bolt	Iter. Bolt	Proj. Bolt
Component parallelism	1	6	1	1
Throttled eff. throughput	174,295	169,215	45,373	295
Maximum eff. throughput	227,096	211,793	54,411	397
Estimated input rate	180,000	360,000	180,000	18,000

Table 7.4: Performance measurements and estimation of the fraud detection query. Throughput and input rates are in events per second.

We also measured the maximum performance of this topology with the same parallelism, but without throttling. The maximum throughput was 30% higher, with the topology being able to process about 227,000 events per second. At the maximum event rate, the bottleneck of the topology was the next Bolt, detecting 211,000 events per second. However, the throughput did reach the estimated maximum of 360,000 events per second. This meant that our parallelism model was not applied correctly. Parallelism under-estimation could be caused by imprecise scaling of next input event rates for a shorter event matching window[18], different Bolt predicate and higher input event rate for the replicated stream[19].

The estimation model did a crude job, but it appeared to have worked well, since the over-estimation of input event rates and under-estimation of greedy next Bolt performance cancelled each other out. The computed component parallelism for this query was optimal. If we decreased the parallelism for the any next Bolt from 6 to 5, the maximum achievable throughput would be only between 168,000 to 175,844 events per second, which would be below the required 180,000. Thus, our parallelism model correctly estimated the optimal parallelism of 6. Interestingly, the

[17]Throttling of input event rates is a bit imprecise due to the use of timings
[18]Because of shorter matching window, input event rate was estimated to be smaller.
[19]The number of events that had to be replicated was higher than shuffled.

threshold rate, at which our model recommends the parallelism of 5, was 175,000 input events per second, which was also correct.

The number of resources used by the query can be seen in Table 7.5. If we used the estimated number of 7 workers, the average CPU use of each worker would be 104%, which was almost ideal when the input event rate was throttled. However, when using the maximum capacity of the topology, the average CPU use increased to 128%. If we were to use 40% more workers, as suggested by the stock monitoring query, the maximum throughput would result in nearly ideal 89% average CPU use per worker. Thus, adding a flat 40% IO processing margin to computed worker count seems a reasonable adjustment for our worker model. Comparing throttled and un-throttled topology, we can also see that the 30% increase in throughput caused also almost a 30% increase in network usage. Also, in the case of maximum throughput, the memory consumption rised to the average of 948 MB per worker. This can be explained by noting that, at the maximum throughput, the input event queues of some tasks would grow to their maximum size due to processing bottleneck.

	Worker count	CPU/worker	Network Mbit/s	Memory total (MB)
Throttled resource use	7 (10)	104 (72.8) %	398	5854
Maximum resource use	7 (10)	128 (89.6) %	501	6635

Table 7.5: Resource use of the fraud detection query. The values in brackets show how many workers would be used and what their CPU usage would be with addition of 40% IO margin.

We can conclude that our model does a crude, but satisfactory job when estimating topology parameters. Our parallelism models for individual Bolts are not very accurate, which could be improved by doing more experiments. However, thanks to conservative estimated event rates, which are always higher than the actual, we can make up for the parallelism model inaccuracy.

7.7 Storm characteristics

We have worked with Storm for six months and consider it to be a simple and powerful framework, which nevertheless contains some limitations. In this section we summarize our opinions on this platform, in particular what are its advantages and disadvantages. We consider the following traits as the main advantages of the Storm framework:

- *Component parallelism.* Storm makes it very easy to arbitrarily parallelise topology components through use of powerful streams groupings. Spawning and distributing parallel tasks is all done by the framework.
- *Scalability.* The Storm platform can be used with dozens or hundreds of nodes, since its global state is distributed across a Zookeeper cluster, acknowledger tasks are independent, and only a small part of management tasks is handled by singleton Nimbus and Storm UI daemons.
- *Simplicity.* Writing simple topologies in Storm is straightforward after initial steep learning curve. In short, users simply need to define components and connect them together on streams in a configuration.
- *Reliability.* Storm offers different reliability APIs, thanks to which it can cope with task or node failures. Also it is fail-fast and behaves correctly after tasks or processes fail, which may recover, provided that the Zookeeper cluster survives.
- *Support for testing.* The local cluster provided by Storm can be used to test and debug topologies locally, before they are submitted to a real cluster.

- *Community.* Storm community grows very fast, is helpful and significantly contributes to new features.

The main disadvantages and difficulties that we had with Storm are summarized in the following points:

- *Installation.* Performing manual install is difficult because of the number of dependencies that Storm requires and its reliance on particular versions. We had to do manual install, since Storm provides an installation script only for Amazon EC2.
- *Documentation and APIs.* The APIs constantly change, as new major features are being added. The framework is young, under development and still contains bugs. Even though documentation is well written now, early in the project it was insufficient.
- *Static topologies.* A topology cannot change at runtime, which severely limits Storm's applicability to complex event detection. For example, it is not possible to dynamically adjust parallelism according to load, or adjust event queue lengths. Also, the user cannot determine the positioning of component tasks at particular nodes, as this function was not yet made available.
- *Slow IO.* Our evaluation shows that Storm incurs very high CPU overhead for sending and receiving events. In particular, this is notable on events with small payloads and on shuffled streams.
- *Intra-worker communication.* Intra-worker communication happens over network interfaces, instead of shared memory. Furthermore, we measured intra-worker communication throughput, which was smaller than inter-worker communication rates. This is likely because of Storm using a single input/output queue per worker.
- *Performance estimation.* Storm is internally a very dynamic system and the measured stream event rates vary significantly, which caused us difficulties during evaluation, as we had to run one topology multiple times. It is also difficult to derive the heuristics for the number of acknowledger tasks and workers, as the Storm documentation does not address these issues.

7.8 Summary

In this chapter we presented the measured performance of individual event detection operators, saw how their increased parallelism leads to higher event detection throughput, and justified how and why the replication of events caused non-linear scalability. We also evaluated how well our parallelism models fit the measured data and examined the performance of event stabilization algorithms. Additionally, we examined the applicability of our model for more complicated queries of stock monitoring and mobile fraud detection, as well as the resource use on a Storm cluster. In the last section we summarized what we consider the main advantages and disadvantages of the Storm framework, based on our project experience.

Chapter 8

Conclusions

We designed a scalable system capable of efficient complex event detection. This system is applicable to many different scenarios, in particular to high-throughput stock monitoring and fraud detection. The basis for the system is a concise and expressive event pattern language with a small number of well-defined operators, which semantics can be adjusted by predicates and through the use of different operator types. We believe that we achieved a good balance between expressibility and speed of complex event detection. Also, the operators are parallelisable, and have a well-structured syntax of event patterns, which resembles their runtime organization.

We based complex event detection on data-flow graphs, where event detection operators correspond to individual nodes and event streams to edges. The data-flow graphs, which we also call topologies, are obtained by compilations of queries specified in the Step language, and they run on the Storm framework in a cluster environment. Since Storm topologies cannot change at runtime, we employed a sophisticated compilation process. This optimizes event serialization by generating payload classes for each input event, speeds up predicate evaluation by using expression indexing, and improves performance by reusing already deployed operators. Additionally, we achieved more efficient event detection through the merging of external operators and fast stabilization algorithms, we improved communication overhead by batching events, decreased latency and dealt with stalled streams through the use of punctuations, and sped up event matching by garbage collecting unmatchable events. We validated the correctness of these algorithms by executing system tests on a custom acceptance testing framework.

Furthermore, we achieved high event detection throughput by distributing event queries at an operator level, and parallelising each operator. We suggested two methods of achieving parallelism, out of which one was implemented. This required replication of some streams and shuffling of others. We saw that this approach led to significant performance improvement and linear scalability for some operators, whereas others were proved to scale non-linearly due to the IO overhead of replication. Additionally, we created a model based on the measured performance, which conservatively estimates input event rates for each operator and computes optimal topology parameters, e.g. the component parallelism, the batch size and the required worker count. The evaluation on real queries has shown that even though our method of estimating parameters is crude, it still computes correct or nearly-correct values, due to the inclusion of error margins.

Even though the most difficulties of this project were caused by the problems when using Storm, we believe that this framework is usable and will become popular in the industry when dealing with continuous data streams and their computations. However, we think that we could build a better complex event detection system, if we designed a custom framework instead. This would enable dynamic query optimizations and would have more predictable performance.

Furthermore, Storm is still under development, as major features are being added and some important optimizations are not implemented, which caused us some additional difficulties.

8.1 Future work

Throughout the report we mentioned a few suggestions that would improve the performance or lead to a more usable CEP system. To conclude, we present these suggestions as a possible future work: *aggregation operators*, *operator merging*, *parallelism through windowing* and *predicate selectivity.*

Aggregation and custom operators The most useful addition to the existing Step CEP system would be an addition of operators that can compute data statistics. The operator predicates could then refer to results of these operators, and the system would be more applicable to stock monitoring or data analysis scenarios. Rather than providing a set of common statistical functions (finding maximum, mean or standard deviation in a sequence), the most useful approach would be to create a general framework, where users could define their own operators. An operator could subclass existing operators and would have to declare its event processing function, the shape of its input and output payloads and the input stream count. Such system could become marketable, as it would be applicable to a broad set of scenarios. However, at the same time it would leverage the difficult parts of complex event detection, such as event batching, stabilization, matching, load-shedding, throttling, submission or topology construction.

Operator merging During the evaluation we found that the performance of operators with high output event rates, such as exception or any iteration, depends critically on the throughput of sending events. In order to reduce the IO overhead, it would be advantageous for these operators to be merged with the following operators in an event pattern, thus forming one parallelisable topology component. Step CEP already implements one case of operator merging - the union operator, which is implicitly included in other Bolts. A similar approach could be used for other IO-heavy operators, however not for processing-heavy operators, such as greedy iteration, or operators with multiple output streams.

Predicate selectivity Since Storm does not allow for dynamic adjustment of operator parallelism, we could attempt to improve our estimation of complex event detection performance by incorporating selectivity of operators into our model. By examining large amounts of data we could possibly find an approximate relationship between predicate complexity and the percentage of filtered events. We could also require users to specify input/output event rates for each operator manually, or to define their typical values distributions. From this we could calculate predicate selectivity. Any of these could improve our model and thus make estimation of topology parameters very accurate.

Parallelism through windowing Recall an alternative approach to parallelism, described in Chapter 4. The approach does not replicate streams, but rather sends overlapping windows of events to each parallel task. It would be interesting to implement this and compare its performance and scalability with our system. Both approaches require replication of events. The superiority of one approach over the other would most likely depend on evaluation scenarios. Our Step runtime framework already contains some support for processing events as part of windows, since it was our first considered approach.

Bibliography

[1] A.Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, J. Widom. STREAM: The Stanford Data Stream Management System, 2004.

[2] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, J. Widom. STREAM: The Stanford Stream Data Manager, March 2003.

[3] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, M. Stonebraker. Load Shedding in a Data Stream Manager. Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003.

[4] N. P. Schultz-Moller, M. Migliavacca, P. Pietzuch. Distributed Complex Event Processing with Query Rewriting. DEBS'09, July 6-9, Nashville, TN, USA.

[5] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. White. Cayuga: A General Purpose Event Monitoring System. In CIDR, 2007, pages 412-422.

[6] , A. Demers, J. Gehrke, M. Hong, M. Riedewald, W. White. Towards Expressive Publish/-Subscribe Systems. In Proc. EDBT, 2006, pages 627-644.

[7] L.Brenna, J. Gehrke, D. Johansen, M. Hong. Distributed Event Stream Processing with Non-deterministic Finite Automata. In DEBS'09, July 6-9, Nashville, TN, USA.

[8] Eugene Wu. High-performance complex event processing over streams. In SIGMOD, 2006, pages 407-418.

[9] Eugene Wu, D. Gyllstrom, H. Chae, Y. Diao, P. Stahlberg, G. Anderson. Sase: Complex event processing over streams. In CIDR, 2007.

[10] P. R. Pietzuch, B. Shand, J. Bacon. A Framework for Event Composition in Distributed Systems. In Proceedings of the 2003 International Middleware Conference, 2003, pages 62-82.

[11] G. Muhl, L. Fiege, P. Pietzuch. Distributed Event-Based Systems, pages 2-34. Springer-Verlag New York, Inc. Secaucus, NJ, USA 2006. ISBN: 3540326510.

[12] M. Sloman. Publish-Subscribe Systems. Distributed Systems Course. Imperial College, Department of Computing, London, UK. December, 2010. Lecture.

[13] S. Yusuf, Survey of Publish Subscribe Communication System. URL http://medianet.kent.edu/surveys/IAD04F-pubsubnet-shennaaz/Survey2.html. December, 2004. Accessed on 2/1/2012.

[14] Tim Bass. What is Complex Event Processing? In the Complex Event Processing Blog, URL http://www.thecepblog.com/. April, 2007.

[15] Introduction to Complex Event Processing. In the first chapter of Red Hat JBoss documentation, URL http://docs.redhat.com/docs/en-US/JBoss_Enterprise_BRMS_Platform/5/html/BRMS_Complex_Event_Processing_Guide/ch01.html. Accessed on 2/1/2012.

[16] IBM Research. Exploratory Stream Processing Systems. URL http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html. Accessed on 3/1/2012.

[17] B. Gedik, I. Thomas, P. S. Yu, H. Andrade, M. Doo, K. Wu. Spade: The System S Declarative Stream Processing Engine. In SIGMOD'08, pages 1123-1134.

[18] L. Neumeyer, B. Robbins, A. Nair, A. Kesari. S4: Distributed Stream Computing Platform. Source, p.170-177. Available at: http://www.computer.org/portal/web/csdl/doi/10.1109/ICDMW.2010.172.

[19] S4 distributed stream computing platform. URL http://incubator.apache.org/s4/. Accessed on 12/1/2012.

[20] S. Gatziu, K. R. Dittrich. Events in an Active Object-Oriented Database System. In proc. of the 1st Intl. Workshop on Rules in Database Systems, Edinburg, 1993.

[21] S. Chakravarthy, D. Mishra. Snoop: an expressive event specification language for active databases. In Data & Knowledge Engineering, vol. 14, issue 1, November 1994.

[22] Abadi, Daniel J., D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik. Aurora: a new model and architecture for data stream management. In the VLDB Journal, vol. 12, issue 2, August 2003.

[23] G. Hebrail. Data Stream Management and Mining. Ecole Normale Superieure, September 2007. Lecture. Available at http://videolectures.net/mmdss07_hebrail_dsmm/. Accessed on 6/1/2012.

[24] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik. The design of the borealis stream processing engine. In CIDR, 2005, pages 277-289.

[25] Data, data everywhere. In The Economist. Available at http://www.economist.com/node/15557443. Accessed on 11/1/2012.

[26] NYSE Euronext. U.S. Options Brochure. Available at www.nyse.com/pdfs/6759_ArcaAmex_BRO.pdf. Accessed on 11/1/2012.

[27] StreamBase. Use cases of the StreamBase platform. Available at http://www.streambase.com/industries. Accessed on 23/5/2012.

[28] G. Cugola, A. Margara. TESLA: A Formally Defined Event Specification Language. DEBS'10, July 12-15, Cambridge, UK.

[29] D. Meier, P. A. Tucker, M. Garofalakis. Filtering, punctuation, windows and synopses. Chapter 3 in StreamDataManagement: N. A. Chaudhury et al. (eds). Springer, 2005.

[30] O. Etzion. The genealogy of event processing players - December 2011 edition. Available at http://epthinking.blogspot.co.uk/2011/12/genealogy-of-event-processing-players.html. Accessed on 2/6/2012.

[31] J. Hynninen. Experiences in mobile phone fraud. Seminar on Network Security. Report Tik-110.501, Helsinki University of Technology.

[32] L. Delamaire, H. Abdou, J. Pointon. Credit card fraud and detection techniques: a review. Banks and Bank Systems, Volume 4, Issue 2, 2009.

[33] N. Marz. Storm Wiki. URL `https://github.com/nathanmarz/storm/wiki`. Accessed on 3/1/2012.

[34] Storm community discussion forum. URL `http://groups.google.com/group/storm-user`. Accessed on 3/1/2012.

[35] ZeroMQ, The Intelligent Transport Layer. URL `http://www.zeromq.org/`. Accessed on 3/1/2012.

[36] Apache Zookeeper homepage. URL `http://zookeeper.apache.org/`. Accessed on 3/1/2012.

[37] Apache Thrift homepage. URL `http://thrift.apache.org/`. Accessed on 3/1/2012.

[38] Amazon Elastic Compute Cloud (EC2). URL `http://aws.amazon.com/ec2/`. Accessed on 4/1/2012.

[39] Emulab. URL `http://www.emulab.net/`. Accessed on 4/1/2012.

[40] Ganglia monitoring system. URL `http://ganglia.sourceforge.net`. Accessed on 28/5/2012.

[41] The Scala programming language documentation. URL `http://docs.scala-lang.org`. Accessed on 30/5/2012.

[42] The Xtext homepage. URL `http://www.eclipse.org/Xtext`. Accessed on 30/5/2012.

[43] Apache Ant user manual. URL `http://ant.apache.org/manual/index.html`. Accessed on 30/5/2012.

[44] Apache Maven user manual. URL `http://maven.apache.org/guides/`. Accessed on 30/5/2012.

[45] JUnit documentation. URL `http://junit.sourceforge.net/`. Accessed on 30/5/2012.

[46] Eclipse Rich Client Platform technical resources. URL `http://www.eclipse.org/home/categories/rcp.php`. Accessed on 30/5/2012.

[47] Matlab Documentation. URL `http://www.mathworks.co.uk/help/techdoc`. Accessed on 30/5/2012.

[48] Kryo serialization homepage. URL `http://code.google.com/p/kryo`. Accessed on 31/5/2012.

[49] VisualVM troubleshooting tool homepage. URL `http://visualvm.java.net`. Accessed on 14/6/2012.

Appendix A

Model constants

Table A.1 shows the list of all constants that we use in different component parallelism models. These constants were obtained by fitting evaluated data from Chapter 7 and are used by the Step compiler to estimate parallelism of topology components.

Component model	Constant	Value
Spout	A	1.427e-06
	B	-0.6818
	R	2.058
Projection Bolt	A	5.448e-06
	B	-0.9008
Conjunction Bolt	A	1.08
	B	1.829e-05
Exception Bolt	A	7.748e-06
	B	-1.353
Next (greedy) Bolt	A	1.087
	B	4.325e-06
	W_a	1.381e-09
	W_b	1.685e-05
Next (any) Bolt	A	0.6142
	B	1.36e-05
	K_2	0.2
Iteration (any) Bolt	A	1.4113e-05
	B	-0.54603
	It_{max}	7
Common	K	1.1

Table A.1: Summary of constant values used in different component models.

Appendix B

Xtext grammar for event patterns

```
grammar step.Step hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate step "http://www.Step.step"

Model:
  "topology" name=STRING
  primitiveEvents+=PrimitiveEvent+
  complexEvents+=ComplexEvent+;

Event:
  PrimitiveEvent
  | ComplexEvent;

PrimitiveEvent:
  "external" name=ID "(" fields+=Field ("," fields+=Field)* ")"
  "<" source=STRING
  "[" rate=MYFLOAT "]";

Field:
  name=ID ":" type=("real" | "int" | "string" | "bool");

ComplexEvent:
  name=ID "(" ("*" | (fieldRefs+=FieldReference ("," fieldRefs+=FieldReference)*)) ")"
  ":" pattern=UnionPattern
  ">" sink=STRING;

BooleanExpression returns Expression:
  ComparisonExpression ({BooleanExpression.e1=current} op=("&&" | "||") e2=BooleanExpression)?;

ComparisonExpression returns Expression:
  ArithmeticExpression ({ComparisonExpression.e1=current}
    op=("!=" | "contains" | "=" | "<=" | ">=" | "<" | ">") e2=ComparisonExpression)?;

ArithmeticExpression returns Expression:
  BasicExpression ({ArithmeticExpression.e1=current}
    op=("+" | "-" | "*" | "/") e2=ArithmeticExpression)?;

BasicExpression returns Expression:
  {IntExpression} number=MYINT
  | {FloatExpression} number=MYFLOAT
  | {StringExpression} string=STRING
  | {BoolExpression} bool=("true" | "false")
  | {FieldSelectExpression} fieldRef=FieldReference
  | {PreviousFieldSelectExpression} "prev" "(" fieldRef=FieldReference ")"
  | {DurationExpression} "dur" typ=("<"|">"|"="|"<="|">=") dur=MYINT
  | {IterationLengthExpression} "len" typ=("<"|">") len=MYINT
  | {NegatedExpression} "!" expr=BasicExpression
  | {UnaryMinusExpression} "-" expr=BasicExpression
  | "(" BooleanExpression ")";

UnionPattern returns Pattern:
  NextPattern ({UnionPattern.pattern1=current} "|" pattern2=UnionPattern)?;

NextPattern returns Pattern:
  ConjunctionPattern ({NextPattern.pattern1=current} type=(";"|";?")
    ("[" condition=BooleanExpression "]")? pattern2=NextPattern)?;

ConjunctionPattern returns Pattern:
  IterationPattern ({ConjunctionPattern.pattern1=current} ","
    ("[" condition=BooleanExpression "]")? pattern2=ConjunctionPattern)?;

IterationPattern returns Pattern:
  BasicPattern ({IterationPattern.pattern1=current} type=("+?"|"+")
    ("[" condition=BooleanExpression "]")?)?;

BasicPattern returns Pattern:
  PrimitiveEventRef
  | {EventNegation} "!" pattern=BasicPattern
  | ’(’ UnionPattern ’)’;

PrimitiveEventRef:
  event=[PrimitiveEvent|ID] ("/" aliasName=ID)? ("[" condition=BooleanExpression "]")?;

// this is a reference to event in a pattern, which in turn refers to external event declaration
FieldReference:
  eventRef=[PrimitiveEventRef|ID] "." field=[Field|ID];

/**
 * TERMINALS
 */
terminal MYINT returns ecore::EInt:
  (’-’)? (’0’..’9’)+;

terminal MYFLOAT returns ecore::EBigDecimal:
  (’-’)? (’0’..’9’)+ (’.’ (’0’..’9’)+)?;

terminal ID:
  ’^’? (’a’..’z’ | ’A’..’Z’ | ’_’) (’a’..’z’ | ’A’..’Z’ | ’_’ | ’0’..’9’)*;

terminal STRING:
  ’"’ (’\\’ (’b’ | ’t’ | ’n’ | ’f’ | ’r’ | ’u’ | ’"’ | "’" | ’\\’) | !(’\\’ | ’"’))* ’"’ |
  "’" (’\\’ (’b’ | ’t’ | ’n’ | ’f’ | ’r’ | ’u’ | ’"’ | "’" | ’\\’) | !(’\\’ | "’"))* "’";

terminal ML_COMMENT:
  ’/*’->’*/’;

terminal SL_COMMENT:
  ’//’ !(’\n’ | ’\r’)* (’\r’? ’\n’)?;

terminal WS:
  (’ ’ | ’\t’ | ’\r’ | ’\n’)+;
```

Appendix C

Acceptance tests examples

The following acceptance test is for testing greedy next operators with filtering predicates. We can see that the test is very concise. Firstly it specifies tested Step program, which takes input from a parametrized adapter (the parameters are events that it will emit). Then, we just need to specify the events that should be detected and the test is ready to be run.

```
@Test
def testCond {
    val query =
        """
            topology "GN3Topology"

            external A(a: int, b: bool, c: real) <
                "ParamIntBoolFloatAdapter(
                    new int[]{1, 2, 3},
                    new boolean[]{true, false, true},
                    new double[]{1.1, 2.3, 1.0},
                    new int[] {1, 2, 3}
                )" [100.0]

            Q1(A1.c, A2.c): A/A1;[A1.c < A2.c]A/A2 > "MockSinkAdapter"
            Q2(A1.c, A2.c): A/A1[A1.b];[A2.b]A/A2 > "MockSinkAdapter"
        """;
    val expected = List(
        mkEvent(1, 2, List(mkField("A1", "c", 1.1), mkField("A2","c",2.3))),
        mkEvent(1, 3, List(mkField("A1", "c", 1.1), mkField("A2","c",1.0)))
    );

    runTest(query, "GN3Topology", expected)
}
```

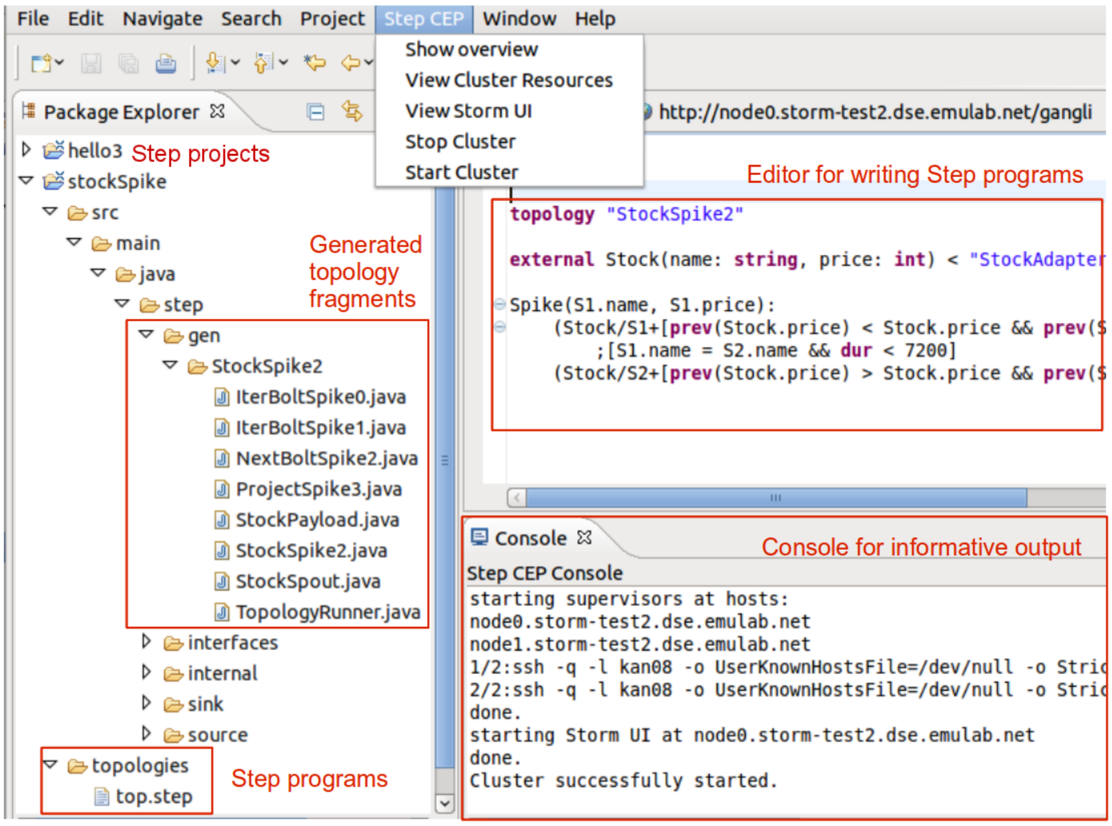
The following test again shows simplicity of acceptance framework, where only topology and expected events have to be specified. Here, the query is fairly complicated, using events from four different streams and joining them using the union, any next and any iteration operators.

```
@Test
def testWithIterationMany {
    val query =
        """
        topology "U5Topology"
            external A(a: int) <
                "ParamIntAdapter(new int[] {11}, new int[] {1}, new int[] {1})" [100.0]

            external B(a: int) <
                "ParamIntAdapter(new int[] {22}, new int[] {2}, new int[] {2})" [100.0]

            external C(a: int) <
                "ParamIntAdapter(new int[] {33}, new int[] {3}, new int[] {3})" [100.0]

            external D(a: int) <
                "ParamIntAdapter(new int[] {44}, new int[] {4}, new int[] {4})" [100.0]

            Q1(*): (A|(B ;? C ;? D))+? > "MockSinkAdapter"
        """;
    val expected = List(
        // seq of length 4
        mkEvent(1, 4, List(mkField("A", "a", 11), mkField("B","a",22),
                           mkField("C","a",33), mkField("D","a",44))),

        // seq of length 1
        mkEvent(1, 1, List(mkField("A", "a", 11))),
        mkEvent(2, 4, List(mkField("B","a", 22), mkField("C","a", 33),
                           mkField("D","a", 44)))
    );

    runTest(query, "U5Topology", expected)
}
```

Appendix D

Step client GUI

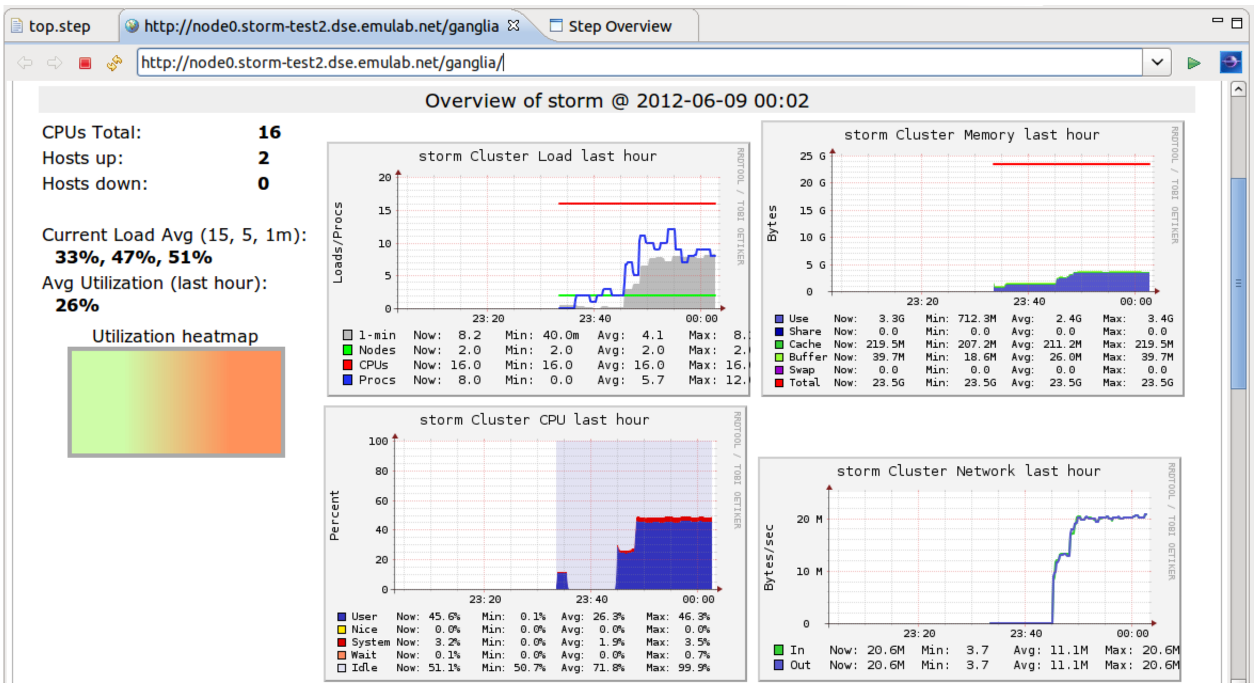
This appendix shows some screenshots of the Step client GUI and illustrates how it is integrated into the Eclipse platform. The screenshots are explained by their captions.



```
topology "StockSpike2"

external Stock(name: string, price: int) < "StockAdapte

Spike(S1.name, S1.price):
    (Stock/S1+[prev(Stock.price) < Stock.price && prev(S
        ;[S1.name = S2.name && dur < 7200]
    (Stock/S2+[prev(Stock.price) > Stock.price && prev(S
```

```
starting supervisors at hosts:
node0.storm-test2.dse.emulab.net
node1.storm-test2.dse.emulab.net
1/2:ssh -q -l kan08 -o UserKnownHostsFile=/dev/null -o Stric
2/2:ssh -q -l kan08 -o UserKnownHostsFile=/dev/null -o Stric
done.
starting Storm UI at node0.storm-test2.dse.emulab.net
done.
Cluster successfully started.
```

Figure D.1: This figure shows the Step editor used for writing topologies that also supports basic syntax highlighting. The project directories generated after the creation of a new Step project are also displayed. The directories contain automatically generated topology fragments as well as the runtime framework and input/output adapters. Context menus enable topology deployment/killing and switching between individual GUI views. The console view shows information about the currently executed task (e.g. starting a cluster).


top.step
http://node0.storm-test2.dse.emulab.net/ganglia
Step Overview
http://node0.storm-test2.dse.emulab.net/ganglia/
storm Cluster Load last hour
Loads/Procs
storm Cluster Memory last hour
Bytes
storm Cluster CPU last hour
Percent
storm Cluster Network last hour
Bytes/sec
23:20
23:40
00:00
RRDTOOL / TOBI OETIKER

Overview of storm @ 2012-06-09 00:02

CPUs Total:	**16**
Hosts up:	**2**
Hosts down:	**0**

Current Load Avg (15, 5, 1m):
33%, 47%, 51%
Avg Utilization (last hour):
26%

Utilization heatmap

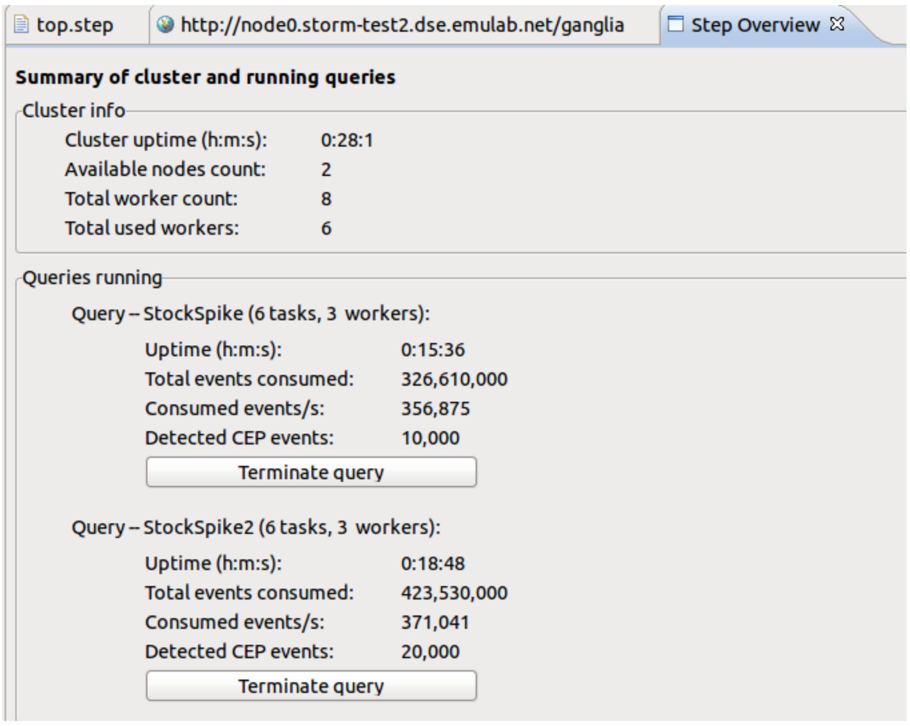
```
▨ 1-min   Now:  8.2   Min: 40.0m   Avg:  4.1   Max:  8.
🟩 Nodes   Now:  2.0   Min:  2.0    Avg:  2.0   Max:  2.
🟥 CPUs    Now: 16.0   Min: 16.0    Avg: 16.0   Max: 16.
🟦 Procs   Now:  8.0   Min:  0.0    Avg:  5.7   Max: 12.
```

```
🟦 Use    Now:   3.3G  Min: 712.3M  Avg:   2.4G  Max:   3.4G
🟦 Share  Now:   0.0   Min:   0.0   Avg:   0.0   Max:   0.0
🟩 Cache  Now: 219.5M  Min: 207.2M  Avg: 211.2M  Max: 219.5M
🟩 Buffer Now:  39.7M  Min:  18.6M  Avg:  26.0M  Max:  39.7M
🟪 Swap   Now:   0.0   Min:   0.0   Avg:   0.0   Max:   0.0
🟥 Total  Now:  23.5G  Min:  23.5G  Avg:  23.5G  Max:  23.5G
```

```
🟦 User   Now: 45.6%  Min:  0.1%  Avg: 26.3%  Max: 46.3%
🟨 Nice   Now:  0.0%  Min:  0.0%  Avg:  0.0%  Max:  0.0%
🟥 System Now:  3.2%  Min:  0.0%  Avg:  1.9%  Max:  3.5%
🟧 Wait   Now:  0.1%  Min:  0.0%  Avg:  0.0%  Max:  0.7%
⬜ Idle   Now: 51.1%  Min: 50.7%  Avg: 71.8%  Max: 99.9%
```

```
🟩 In   Now: 20.6M  Min:  3.7  Avg: 11.1M  Max: 20.6M
🟦 Out  Now: 20.6M  Min:  3.7  Avg: 11.1M  Max: 20.6M
```

Figure D.2: Ganglia cluster resource monitoring system integrated into Eclipse. It displays total CPU, memory, buffers, network and disk usage of a whole cluster, as well as statistics for each individual nodes.



Summary of cluster and running queries

Cluster info

Cluster uptime (h:m:s):	0:28:1
Available nodes count:	2
Total worker count:	8
Total used workers:	6

Queries running

Query – StockSpike (6 tasks, 3 workers):

Uptime (h:m:s):	0:15:36
Total events consumed:	326,610,000
Consumed events/s:	356,875
Detected CEP events:	10,000

Query – StockSpike2 (6 tasks, 3 workers):

Uptime (h:m:s):	0:18:48
Total events consumed:	423,530,000
Consumed events/s:	371,041
Detected CEP events:	20,000

Figure D.3: Step overview shows some basic information about the cluster and summaries about all topologies running (note that this information in not contained in the Storm UI). Also, individual topologies can be terminated here.