Distributed Complex Event Processing with Query Rewriting

Nicholas Poul Schultz-Møller
Department of Computing
Imperial College London
United Kingdom
nps07@doc.ic.ac.uk

Matteo Migliavacca
Department of Computing
Imperial College London
United Kingdom
migliava@doc.ic.ac.uk

Peter Pietzuch
Department of Computing
Imperial College London
United Kingdom
prp@doc.ic.ac.uk

ABSTRACT

The nature of data in enterprises and on the Internet is changing. Data used to be stored in a database first and queried later. Today timely processing of new data, represented as events, is increasingly valuable. In many domains, complex event processing (CEP) systems detect patterns of events for decision making. Examples include processing of environmental sensor data, trades in financial markets and RSS web feeds. Unlike conventional database systems, most current CEP systems pay little attention to query optimisation. They do not rewrite queries to more efficient representations or make decisions about operator distribution, limiting their overall scalability.

This paper describes the NEXT CEP system that was especially designed for query rewriting and distribution. Event patterns are specified in a high-level query language and, before being translated into event automata, are rewritten in a more efficient form. Automata are then distributed across a cluster of machines for detection scalability. We present algorithms for query rewriting and distributed placement. Our experiments on the Emulab test-bed show a significant improvement in system scalability due to rewriting and distribution.

1. INTRODUCTION

Many application domains need timely processing of new data [8]. For example, a financial application may monitor a continuous stream of credit card transactions for fraud patterns. In environmental sciences, sensor networks output infinite sequences of measurements that may be processed to respond quickly to physical events, such as an increase in monitored pollution levels. On the Internet, RSS feeds from weblogs generate vast amounts of real-time information that may be analysed to infer opinions of many people. In general, the requirement of instant processing of new data is at odds with the conventional database model, as implemented by database management systems (DBMSs), in which data is first stored and indexed and only queried afterwards.

In contrast, *complex event processing* (CEP) systems regard new data (or updates to existing data) as a stream of *events* [19]. Users can specify queries as high-level *patterns of events.* The CEP system detects complex events matching these queries and notifies clients of matches in soft real-time. For example, a credit card processing company may monitor fraudulent credit card use by specifying patterns that detect concurrent transactions caused by the same credit card at geographically distant locations.

Detecting complex patterns in high-rate event streams requires substantial CPU resources. To make a CEP system scale in the number of concurrently detectable patterns, it needs additional computational resources by distributing detection across a set of machines or use the existing resources more efficiently through query rewriting. However, the majority of current CEP systems only support execution on a single machine and the semantics of their query languages make automated query rewriting challenging.

In this paper, we describe the design, implementation and evaluation of the NEXT *CEP system* that focusses on distributed event detection with query rewriting. The system uses an expressive automata-based approach for complex event detection. Users specify event patterns in a high-level, SQL-like language with six operators that are translated to low-level event automata. The language facilitates rewriting of expressions into equivalent ones and the automata model eases deployment of detection operators across multiple machines. The system makes optimisation and placement decisions according to cost functions derived from the resource consumption of event automata. Guided by the cost model, our system rewrites queries containing two commonly used operators, “next” and “union”, to reduce their CPU consumption. The system also greedily selects distributed deployment plans to perform event processing on a cluster of machines while reusing already-existing operators.

Our evaluation on the Emulab networks test-bed shows that query rewriting increases the supported number of queries with the “next” operator by up to 100% and with the “union” operator of up to 20%. We also demonstrate how distribution enables the system to scale linearly.

In summary, the main contributions of this paper are:

1. A *high-level event pattern language* based on a new expressive *event automata model* that is suitable for query rewriting and distribution (§3);
2. A *cost model* for queries that allows automated optimisation (§4.2);

3. Algorithms for *query rewriting* into more efficient representations (§4.3) and for *distributed query deployment* with operator reuse (§4.4);
4. The design and implementation of a *prototype CEP system* in Erlang illustrating the above techniques (§5) and an *evaluation* on the Emulab test-bed (§6).

1.1 Detection of Credit Card Fraud

We describe an application scenario — the detection of credit card fraud — that illustrates the scalability challenges in CEP and shows how efficient distribution can provide relief. Credit card fraud has been committed for many years and remains a major problem. Just in the US alone, 9.91 million people were victims of credit card fraud in 2007 with a total loss of US$ 52.6 billion [18]. Fraud typically happens when the credit card owner uses the card in a shop. During the payment, an employee can easily run the card through a card reader without the owner's knowledge. Later the copied credit card information is sold to other criminals who manufacture a new card and commit the actual fraud.

There are several patterns indicating such behaviour in credit card transaction logs, which can be detected by a CEP system. A fast detection of suspicious patterns enables banks to contact customers early in the fraud or even automatically alert users about suspicious credit card transactions by sending text messages to their mobile phones. A typical fraud pattern is that the criminal starts by testing the credit card with a few small purchases followed by larger purchases. In our high-level query language described in §3, this can be expressed as follows:

```
SELECT * FROM ( t S1 ; t S2 ; t L )
 WHERE FILTER(S1.acc = S2.acc), FILTER(S2.acc = L.acc),
       S1.amount < 100 AND S2.amount < 100 AND
       L.amount > 250 AND (S1, L) OCCURS WITHIN 12 hours
```

Properties of data sources. We make the following assumptions about the workload of a CEP system for credit card fraud detection. The system processes transactions from multiple sources. The sources are credit card processing companies responsible for a given credit card brand in a given region, such as Northeastern US.

We take credit card transactions made with Visa^{TM} in the US as an example. According to the Visa website [26], they processed 27.612 billion transactions in 2007. This means an average of 875 credit transactions per second based on a uniform distribution. Assuming that 80% of transactions occur in the 8 hours of the day, this gives an event rate of 2100 transactions per second. To model this, we assume that credit card transactions follow a Poisson distribution with an average of 2100 events per second. Note that because of the high average rate ($\mu = 2100$), the rate does not vary significantly ($SD = \sqrt{\mu} \approx 46$), i.e., the probability of it being in the interval 1950–2250 events per second is 99.9%.

We further assume that the information in each transaction event consists of an identifier and a timestamp, an account number, an amount, currency and country fields and a short description. Given realistic field sizes, this makes the size of transactions range from 29–292 bytes with an average of 150 bytes. The event size can be modelled by a normal distribution with an average of 150 bytes and a spread of 30 bytes. This gives an average data rate of approx. 300 kB per second per source. This amount of data can easily be handled by a fast network connection.

Case for distributed detection. As estimated above, the data rate from a single source is relatively small. However, if a query correlates transaction events from several different sources, then hundreds of thousands of events have to be processed per second. In this case, the CPU resources of a single machine may become a bottleneck and limit system scalability. When the system runs out of CPU resources, it has to buffer events or discard them. Buffering events only provides temporary relief and is only effective under transient overload conditions. A distributed CEP system with query optimisation can distribute detection queries across machines and reduce the cost of individual queries by rewriting them into more efficient forms.

2. RELATED WORK

A range of systems have been used to detect events using different detection approaches. This section provides an overview of previous work, particularly focusing on proposals for query optimisation in event processing.

Complex event processing systems. The goal of CEP systems is the fast detection of event patterns in streams. Specification languages for event patterns are frequently inspired by regular languages and therefore have automata-based semantics [21]. In addition to commercial CEP systems, such as *ruleCore CEP Server*, *Coral8 Engine* and *Esper*, several open research prototypes exist.

Cayuga [14, 13] is a high performance, single server CEP system developed at Cornell University. Event streams are infinite sequences of relational tuples with interval-based timestamps. Its event algebra has six operators: projection, selection, renaming, union, conditional sequence and iteration. Event algebra expressions are detected by non-deterministic finite automata, which can detect unbounded sequences. To achieve high performance, Cayuga uses custom heap management, indexing of operator predicates and reuse of shared automata instances. However, Cayuga does not support automated query rewriting and distributed detection. The distribution of Cayuga automata is complicated by the fact that it merges event algebra expressions into a single automaton, which would have to be partitioned across nodes for distribution.

The *DistCED* system [24] is a Java-based CEP framework that uses extended, non-deterministic finite state automata. Event patterns are specified using six operators in a simple, yet expressive core CE language: concatenation, sequence, iteration, alternation, timing and parallelisation. To avoid the problem of incorrect detection due to late arrival of events that were delayed by the network, a guaranteed detection policy requires detectors to wait until events have become stable. Although the work proposes automata distribution, no cost model or algorithms are given.

More recently, Akdere et al. [4] describe how plan-based optimisation can exploit temporal properties of event sources to reduce network communication costs. Since this work focusses on network communication costs, we believe that it is orthogonal to ours. We do not consider the cost of sending events from event sources to source proxies in our system, as we assume the network not to be a bottleneck.

Data stream processing systems. Systems such as *Aurora* [1], *Esper* [17], *Stream* [5] and *TelegraphCQ* [10] process data streams according to relational continuous queries, e.g., specified in CQL [6]. *Windows* over streams convert infinite sequences to finite relations for relational algebra operators. The Stream system exploits overlap between queries through shared state but, to our knowledge, none of these above centralised systems explore automated query rewriting.

Borealis [2] is a system that distributes queries across a set of Aurora engines. Monitoring components collect performance statistics about deployed queries. These statistics are used by a set of hierarchically-organised *query optimisers* that maximise the quality-of-service of queries by making decisions about load shedding, choice of operator implementations and query reuse. A performance problem is first handled by a local optimiser. If the problem cannot be resolved at this level, it is cascaded to a neighbourhood optimiser and eventually to a system-wide optimiser. In contrast, the focus of our work is on query rewriting before deployment, which is only possible by taking language semantics into account. Query rewriting is difficult to achieve in Borealis because of its "box and arrows" operator semantic and the fact that operators can change at runtime.

Much previous work focussed on load-balancing and load-shedding in distributed stream processing systems. The *Medusa* [11] system uses price-based contracts for dynamic load-balancing in a cluster deployment [7]. A proposal for load-balancing under dynamic conditions was made by Xing et al. [28]. Their approach minimises load variance to reduce tuple processing delays using a greedy online algorithm.

For operator deployment, Ahmad et al. [3] propose to choose greedy deployment plans that reduce bandwidth usage of queries. This favours placements that co-locate operators with their child operators to avoid network communication, which is similar to our approach. It also attempts to place operators at nodes that are close in network distance. An algorithm for wide-area operator placement based on hierarchical clustering of nodes is described by Seshadri et al. [25]. The clustering according to an objective function limits the search space of placements. Pietzuch et al. [23] argue for network-centric operator placement that minimises network usage by ensuring short communication paths between operators. This is orthogonal to our approach that ignores the impact of the network (except for a constant penalty).

Active and distributed DBMSs. In a DBMS, it is natural to treat updates to the database as events and associate them with actions. An *active DBMS* processes events according to *triggers* defined as *event-condition-action* (ECA) rules. Automated optimisation of triggers is difficult because of their expressiveness and imprecise semantics. Another challenge is the lower detection performance due to the overhead of the DBMS.

SAMOS [20] is an active DBMS that uses coloured Petri nets for event detection. This allows the representation of partially-detected events. Event patterns are specified using six operators, whose semantics is defined by the underlying Petri nets. However, the Petri nets for even simple event patterns are complex, making automated rewriting difficult.

In *Snoop* [9], a declarative event language defines point-based logical events from interval-based physical events. Composite event patterns are constructed from five operators. Although Snoop's event language could support query rewriting, it lacks formal semantics and its expressiveness is limited to finite sequences. Snoop's temporal model requires unique timestamps, which is unrealistic under distribution.

There is a large body of work on query optimisation for minimising query response time in distributed DBMSs [29, 16]. Optimisation algorithms, such as *INGRES* and *R**, are based on greedy heuristics and exhaustive enumeration of all possible query execution plans. This is infeasible in a large-scale CEP system with many processing nodes. In addition, CEP systems require cost models that take the long-lived nature of queries into account.

3. CEP MODEL

In this section we introduce the language for describing event patterns along with the necessary event and temporal models that form the basis of our automata-based detection approach. Our high-level language is compiled into a *core language* consisting of six operators. The semantics of the core language is defined by its *event automata*.

3.1 Events and Event Streams

We use an event model that is similar to the ones found in Cayuga [14, 13] and DistCED [24]. An event e is defined as a tuple $\langle s, \mathbf{t} \rangle$ where s is a multiset of fields, as in the relational data model, defined by the schema $\mathbb{S}$, and $\mathbf{t}$ is a sequence of timestamps $\mathbf{t} = [t_s, \ldots, t_e]$ where the first timestamp t_s is the start time of an event and the last timestamp t_e is the end time. Each timestamp is discrete and, for *primitive events*, $\mathbf{t}$ is a pair of timestamps[1]. Compositions of primitive events, *composite events*, contain all timestamps (and fields) of the primitive events that constitute them. The set of all possible events is denoted by $\mathbb{E}$ and all events $e \in \mathbb{E}$ satisfy $t_s \le t_e$. Events with $t_s < t_e$ have a *duration* while events with $t_s = t_e$ are called *instantaneous events*. There are two special events: ϵ — the *empty event* (similar to the empty word in traditional automata) and the *failed-detection event* $\emptyset$. Their use will become apparent in §3.3.

Events *arrive* on *event streams* that are infinite sequences of events with the same schema. A schema and a stream uniquely define an *event type*. A stream has a *source* and zero or more *sinks*. Events are sent from the source in ascending order of t_e. If t_e is the same for two or more events, then the events are sent in ascending order of t_s. If $\mathbf{t}$ is equal for two or more events, then they are sent in arbitrary order. An example of this ordering is (only t_s and t_e are shown): $[1, 2], [3, 5], [4, 5], [4, 6], [4, 6]$. Events arrive without reordering by the underlying communication channel.

3.2 Temporal Model

The temporal model must define what it means for two events to occur *after* another and it must resolve the choice of immediate successor of an event. It should also support the rewriting and optimisation of queries. Therefore it is important for a common operator, the "next" operator ';', which, informally, detects one event occurring after another, to be associative: Given three event patterns $E1$, $E2$ and $E3$, the patterns $E1; (E2; E3)$ and $(E1; E2); E3$ should produce the same composite events. This property allows a pattern $E1; (E2; E3)$ to be optimised if $E1$ occurs rarely, while $E2; E3$ occurs often. By rewriting the pattern to $(E1; E2); E3$, fewer events have to be processed.

In the example in Fig. 1, there are three composite events matching $(E2; E3)$ (namely $[e_b, e_e]$, $[e_c, e_e]$ and $[e_d, e_e]$), while there is only one matching $E1; (E2; E3)$ (namely $[e_a, e_b, e_e]$). If the pattern could be rewritten as $(E1; E2); E3$, then there

[1] A common use for this interval is to accommodate clock inaccuracy of sources.

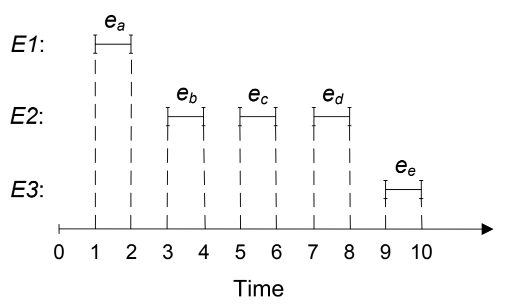


Figure 1: Example of events matching three different patterns $E1$, $E2$ and $E3$.

would be only one composite event matching $E1; E2$ and one primitive event matching $E3$, resulting in fewer processed events. Clearly, only one of the events $[e_b, e_e]$, $[e_c, e_e]$ and $[e_d, e_e]$ can be the immediate successor of e_a to allow the next operator to be associative. The immediate successor of an event should intuitively be the *first* that occurs fully after the event. With this choice of immediate successor, the next operator is associative:

$$E1; (E2; E3) \ni [e_a] \biguplus [e_b, e_e] = [e_a, e_b] \biguplus [e_e] \in (E1; E2); E3,$$

where $\biguplus$ concatenates two composite events.

Therefore we adopt the *complete-history* temporal model formulated in [27] because it is the only temporal model that has an associative next operator and also fulfils other desirable properties that make a system implementable, such as a unique immediate successor. A temporal model is described by $\langle \mathbb{T}, \prec, \text{SUCC}, \bigotimes \rangle$ where $\mathbb{T}$ is the set of all possible timestamps, $\prec$ is a partial ordering on $\mathbb{T}$, SUCC is the successor function $\text{SUCC} : \mathbb{T} \times 2^{\mathbb{T}} \to 2^{\mathbb{T}}$ that takes a timestamp $\mathbf{t}$ and a set of candidate timestamps $\mathcal{F}$ and produces the set of immediate successors $\text{SUCC}(\mathbf{t}, \mathcal{F})$. $\bigotimes$ is the composition operation that takes two timestamps $\mathbf{t_1}, \mathbf{t_2} \in \mathbb{T}$ and produces the timestamp $\mathbf{t_1} \bigotimes \mathbf{t_2}$ for the corresponding composite event.

As described above, primitive events have pairs of timestamps. This extends to the composite case where the sequence of all timestamps of all primitive events constituting a composite event $(t_1, t_2, \ldots, t_n)$ define its timestamp. The partial ordering introduced above is thus defined as

$$\mathbf{t_1} \prec \mathbf{t_2} \equiv (t_{s1}, \ldots, t_{e1}) \prec (t_{s2}, \ldots, t_{e2}) \Rightarrow t_{e1} < t_{s2}.$$

The successor function is defined as

$$\text{SUCC}(\mathbf{t}, \mathcal{F}) = \{\tau \in \mathcal{F} | \mathbf{t} \prec \tau \wedge \neg \exists \rho \in \mathcal{F}.\mathbf{t} \prec \rho \sqsubseteq \tau\},$$

where $\sqsubseteq$ is the slightly modified lexicographical ordering from the end of the sequences of timestamps (effectively a tie-breaker for events with the same end time point):

$$\begin{aligned} \sigma \sqsubseteq \tau \text{ if } & \sigma[\ell(\sigma) - i] < \tau[\ell(\tau) - i] \wedge \\ & \sigma[\ell(\sigma) - k] = \tau[\ell(\tau) - k] \text{ for } k < i \text{ or} \\ & \ell(\tau) < \ell(\sigma) \wedge \sigma[\ell(\sigma) - i] = \tau[\ell(\tau) - i] \text{ for all } i < \ell(\tau) \end{aligned}$$

where $\ell(x)$ is the length of the timestamp sequence x and $x[i]$ is the ith time point. Two examples of sequences σ and τ which fulfill $\sigma \sqsubseteq \tau$ are: $\sigma = [2, 6, 8]$, $\tau = [3, 6, 8]$ and $\sigma = [1, 2, 4, 6, 8]$, $\tau = [4, 6, 8]$.

3.3 Automata-Based Event Detection

Next we describe our automaton model and define the semantics of the core language. An *event automaton* is a tuple $\langle S, T, s_0, s_d, s_t, in, out \rangle$ where

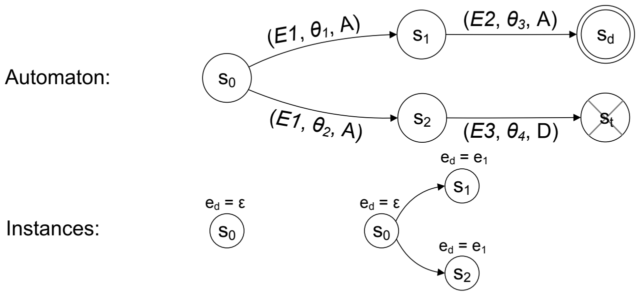


Figure 2: Example of automaton with two instances.

S:	is the set of states of the automaton;
T	$T \subseteq S \times S \times \mathbb{E} \times \Theta \times F$ are the transitions between states. Each transition is labeled with an event type $E \in \mathbb{E}$, a predicate $\theta \in \Theta$ and a *transformation function* F. A predicate $\Theta : \mathbb{E} \times \mathbb{P}(\mathbb{E}) \to \mathbb{B}$ is a function that takes an event and a set of events and returns true or false (described below);
s_0:	$s_0 \in S$ is the start state;
s_d:	$s_d \in S$ is the accepting state or, equivalently, the detection of an event pattern;
s_t:	$s_t \subset S$ is the set of failed-detection states;
in :	is the stream input into the automaton;
out:	is an output stream of detected composite events.

There are three types of states: *ordinary*, *timing states* and *failed-detection states*. The first is analogous to states in traditional finite automata. A timing state starts an internal timer when entered. After a specified relative time period, it emits an instantaneous *timer event* ($t_s = t_e$). It has a unique event type but otherwise can label any transition in the automaton. The failed-detection state is a state that emits a failed-detection event when entered. This allows an operator to "fail" when a given event occurs while the operator is detecting another (composite) event.

The current state of an automaton is called an *instance* and is described by a tree with nodes corresponding to states. Each leaf node represents a *sofar detected composite event* denoted e_d. The root of an instance is the start state s_0 with $e_d = \epsilon$ (no event detected). A path from the root node to a leaf describes a sequence of detected events. The set of states of all leaf nodes is called the *active states set* of the instance and is denoted by S_{active}. We call a node in the tree with two or more children a *choice-point.*

An example of an event automaton with two instances is shown in Fig. 2. The right instance has processed an event $e_1 \in E1$. The set of active states for the left and right instances are $\{S_o\}$ and $\{S_1, S_2\}$, respectively, and the right instance has S_0 as a choice-point.

An automaton operates as follows: There is always a new instance with s_0 as the (only) active state and $e_d = \epsilon$. The timestamps of ϵ are equal to the time when the instance was *spawned.* Given a set of active states, the *input domain* for the next event(s), denoted $E_{\text{d}}(\in \mathbb{E})$, is all the input streams with event types that label a transition from some state $s \in S_{\text{active}}$. Usually there will only be a *single* next event but there may be several simultaneous events and thus a set of events could be input. If the input domain is empty, then the instance is *stuck* and therefore discarded. This also occurs after a pattern was matched.

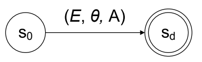
When the next event(s) $E \subset E_{\text{d}}$ occur after the currently detected composite event(s), each event $e \in E$ is processed at each leaf node as follows: For each transition from s, labeled by e, where e and e_d satisfy the predicate of the transition, a new child node is added with $f(e, e_d)$ as the detected event. f is a transformation function that transforms the two events e and e_d into a new composite event. For example, $f_A(e, e_d) = e_d \uplus e$ is the concatenation function (A for *add*) and $f_D(e, e_d) = e_d$ discards events (D for *drop*). However, f could be also be any aggregation function.

If no events in E cause any transitions, then the branch from the leaf node to the nearest choice-point is *discarded*. This corresponds to a "wrong guess" with respect to the pattern being matched. If all branches are discarded, then the instance itself is discarded. If a failed-detection state is reached, the instance is also discarded and a failed-detection event $\emptyset$ is sent on the output stream. If an instance of an automaton at any point reads a failed-detection event, it will also be discarded and emit a $\emptyset$. When an instance reaches the accepting state, detected composite event e_d of s_d is sent once on output stream *out*.

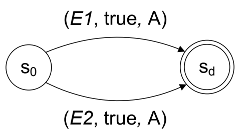
3.4 Core Language for Event Patterns

The core language defines the basic "building blocks" of event pattern queries and our high-level language is compiled into these constructs. Next we present the six operators and their semantics described by event automata.

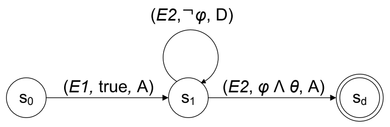
The **filter operator** E_θ detects event type E that satisfies predicate θ. The corresponding automaton is:



The **union operator** $E1|E2$ detects event type $E1$ or $E2$. Note that the simultaneous occurrence of two events matching $E1$ and $E2$, respectively, results in two correctly detected events. The semantics are given by the automaton below:



The **next operator** $E1; E2_{\varphi,\theta}$ detects the next occurrence of $E2$ satisfying θ after $E1$, skipping any intermediate event $E2$ not satisfying φ:

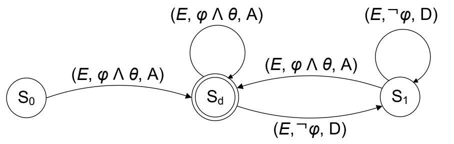


The predicate θ allows the operator to filter some events and then fail on others. Consider the following query from the application scenario in §1.1:

```
SELECT * FROM ( t T1; t T2; t T3 )
 WHERE FILTER(T1.acc = T2.acc), FILTER(T2.acc = T3.acc),
       T1.amount > 1000 AND T2.amount > 1000
        AND T3.amount > 1000
```

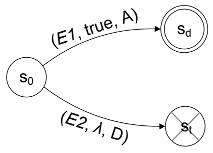
This query matches three consecutive transactions of the same account that are larger than 1000. It discards automaton instances for transactions below this value. Note that this cannot be expressed with Cayuga's conditional sequence operator, which would spawn instances until the pattern is detected, and exhaust memory if the pattern never occurs.

The **iteration operator** $E+_{\varphi,\theta}$ detects one or more consecutive events E that together with e_d satisfy θ. Events not satisfying φ are skipped:



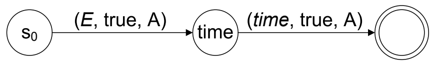
This operator is similar to the Kleene Plus operator in regular expressions. However, its evaluation may require unbounded memory when the θ predicate never evaluates to false. This is due to the temporal model, in which all constituting primitive events of a composite event must be stored. A solution would be to introduce a renaming operator (as described in [14]) to define aggregate functions or to discard old instances after a given number of spawned instances.

The **exception operator** $E1\backslash E2_\lambda$ detects event type $E1$ but fails if event $E2$ satisfying λ occurs. This operator supports the cancellation of an event pattern after a subpattern was detected:



This operator has interesting applications, especially when composed with the next presented operator, the time operator. For example, it can be use to cancel a pattern for fraud detection because the associated account was closed.

The **time operator** $E@_{time}$ detects a relative time point after event E occurred, as specified by time expression *time*. When entering a time state, a *time event* is emitted after the specified time duration:



An interesting design question arises as to why the language needs a time operator when the occurrence of two events within a time period can be checked by the timestamps in the events (once both have occurred). The problem is that if the first stream is high rate and events on the second stream rarely occur, then this approach may exhaust memory, irrespective of how small the time period is. With the exception operator, detection can be "cancelled" after the time period expired and instances are discarded.

3.5 High-Level Event Query Language

Since our high-level language is intended for human use, two design requirements were readability and a syntax similar to SQL. Existing languages interleave source specification and predicates, which makes queries hard to read and reason about [13, 1]. Therefore we wanted to to separate predicates from event patterns, while still making it easy to compile into the core language. Our query language consists of two sub-languages: the *Stream Definition Language* for defining sources and sinks and the *Event Query Language* (EQL) for expressing queries.

The **Stream Definition Language** defines the different sources and sinks in the system, which are connected through network communication channels. Sources and sinks *register* under a unique identifier *id* and submit a schema that describes their event type. They may also provide a wrapper that translates their internal event representation to the one prescribed by their schema.

The **Event Query Language** specifies queries of the form:

`SELECT` *selection* `FROM` *event_pat* `WHERE` *qualifications*

where *selection* is a field selection expression, such as `*` for all fields, *event_pat* is an event pattern and *qualifications* is a list of predicates.

The *event_pat* is built recursively from four basic operators: next (`;`), exception (`\`), union (`|`) and iteration (`+`). The operators map to the ones introduced in §3.4 with the exception of the filter operator, the time operator and predicates, which are specified (implicitly in the case of the filter and time operators) in the *qualifications*. Operands are subpatterns and, in the base case, a primitive event source identified by a single or a pair of *id*entifiers. A pair specifies a source and an *alias*. This allows predicates to refer to a particular event in patterns with several events from the same source, similar to how self-joins are handled in SQL.

Finally *qualifications* are a comma-separated list of two types of predicates: filter predicates, `FILTER(` *pred_expr* `)`, or regular predicates, *pred_expr*. When the high-level language is compiled to core operators, predicates are associated with individual operators and qualifications are evaluated by core operators. The syntax and semantics for predicates is as expected from standard boolean operators, except for time and field constraints.

A *time constraint* is of the form:

`(`*id1* `,` *id2*`) OCCURS WITHIN` *time*

where *id1* and *id2* are identifiers of two streams in the event pattern and *time* is a relative time expression, such as `1 min` or `10 msec`. A time constraint evaluates to true if and only if the two events specified by the identifiers occur within the time period.

A *field constraint* can have one of two forms:

field `CONTAINS` *string* (1)
field_or_val comp_op field_or_val (2)

where *field* is a field name, *string* is a string, *field_or_val* is a field name or an arithmetic expression and *comp_op* is a comparison operator.

A field is either a *simple field*, *id*`.`*f*, or an *aggregate field*, `PREV(`*id*`.`*f*`)` or `ALL(`*id*`.`*f*`)`. A comparison operator evaluates to true for a simple field if and only if the field value in the occurring event of that type makes the comparison true. When the aggregate field `PREV` is used, the operator evaluates to true if and only if the value of the field in the immediately previous event of that type (in the so far detected composite event e_d) makes the comparison true. If there is no previous event in e_d, the comparison is vacuously true. Finally, a comparison with the aggregate field `ALL` evaluates to true if and only if the field values of all previously detected events of that type make the comparison true.

For example, the following query detects composite events that consist of one or more transactions with some currencies followed by a transaction with a currency different from all the previous ones:

```
SELECT * FROM t T1+; t T2
         WHERE ALL(T1.currency) <> T2.currency
```

We can now describe the semantics of the two field constraints given above. (1) The first field constraint evaluates to true if and only if the specified field is of type string and its value has *string* as a substring. For aggregate fields, the constraint evaluates to true if and only if the fields of the previous/all previous events contains *string* as a substring. (2) The second field constraint evaluates to true if and only if the operator is defined for the two fields/values and evaluates to true (with the usual semantics defined for standard data types). An arithmetic expression is composed of the four binary operators (`*`, `/`, `+`, `-`) and the unary minus operator (`-`). If an arithmetic expression is undefined (e.g., division by zero), the field constraint evaluates to false.

4. QUERY OPTIMISATION

Event queries can be executed in different ways, each with given resource consumption and performance characteristics. These depend on the structure of event patterns and on the distribution of operators to processing nodes. The goal of query optimisation is to find an event pattern structure and deployment plan with the *best* possible behaviour. This can be achieved by rewriting event patterns to better equivalent patterns, reusing already deployed operators and selecting efficient deployment plans for new operators. Before describing our query optimisation approach, we state our assumptions and the underlying cost model.

4.1 Optimisation Goals and Assumptions

The three main resources in a distributed event processing system are memory, CPU time and network bandwidth. In many applications, such as the one described in §1.1, CPU resources become a bottleneck due to the computational overhead of pattern detection, even at low event rates. This limits maximum throughput and scalability in terms of the number of concurrent queries supported by the system. Low detection latency is frequently another desirable property.

Our primary optimisation goal for *query rewriting* is to minimise operators' CPU usage to enable the system to support more operators. The secondary objective is to lower detection latency. It turns out that these goals are, in fact, not conflicting — lower processing time of an event also results in lower latency.

With respect to *distributed deployment plans* for operators, we aim to reuse existing operators to minimise CPU usage and latency. Receiving, processing and sending network packets consumes CPU resources and traversing network links increases latency. A secondary goal is to minimise variance of CPU usage. High variance due to processing bursts may result in temporary overload conditions and high scheduling overhead.

We make the following assumptions about the CEP system and its deployment environment: (1) The deployment nodes are dedicated servers in a data centre with only the CEP system running. (2) Network links between nodes are not congested. This is a reasonable assumption in a data centre with high capacity links. (3) The source streams are

Poisson distributed and the average event rate is available or can be measured. This assumes that events occur independently, as it is the case in our application scenario from §1.1. (4) Operators only consume CPU cycles when processing events, yielding the CPU when idle.

4.2 Cost Model

The cost model provides a quantitative measure of a query plan's quality. To simplify the cost model, we make the following assumptions: (1) Our cost model describes asymptotic costs of operators. It only permits reasoning about relative costs, as absolute costs depend on a range of other factors, such as processor speeds and programming language overheads. (2) We ignore the cost of predicate evaluation. Efficient algorithms for predicate indexing (with known resource demands) could be used to extend the cost model [13]. (3) Finally, we ignore the selectivity of operators, i.e., the number of events not filtered by predicates, and assume worst-case output sizes. There exists an extensive body of work in database query optimisation on estimating operator selectivities. We assume CPU consumption to be a function of event rates only.

We use the *worst-case rate* μ_s of complex events in all cost equations. Event rates may be lower due to events filtered by predicates. Table 1 summarises the worst case event rates for different operators as a function of input rates. Note that the event rate of the next operator only depends on the rate of the first subpattern as each event $E1$ results in at most one complex event $E1; E2$. The worst-case event rate of the iteration operator is unbounded because there may be an infinite number of instances outputing events when a new event arrives. Therefore, we leave the cost equations for the iteration operator to future work. It would requires additional statistics, such as predicate selectivities.

A complex event detected by an operator o has a *detection time*, $\mathrm{DT}(o)$. It is the average time between complex events detected by o. For a primitive source s with event type E, $\mathrm{DT}(s) = 1/\mu_s(E)$ because sources are assumed to be Poisson distributed with average rate $\mu_s(E)$. For any operator, the detection time depends on the detection time of subpatterns, as shown in Table 1. The detection times of the filter, exception, iteration and time operators are equal to the detection times of the subpatterns (and $E1$ in the case of the exception operator) because they immediately output a composite event (assuming negligible processing delay). In contrast, the next operator waits for the second event pattern $E2$ to occur after the first event $E1$ has occurred. This results in a detection time $\mathrm{DT}(E1)+\mathrm{DT}(E2)$. The detection time of the union operator is the time between two events from its subpatterns occurring.

Cost of query. The cost function used for query rewriting is given by: $\mathrm{cost}(Q) = \sum_{o \in Q} \mathrm{CPU}(o)$ where Q is a query and $\mathrm{CPU}(o)$ is the CPU cost of operator o measured as CPU time used per unit time. The models for CPU usage of different operators are described below.

Cost of filter operator. Its CPU cost is equal to the number of events processed per time unit. On average this is $\mu_s(E)$ (ignoring the cost of predicate evaluation).

Cost of exception operator. Similarly, the CPU cost is equal to the total number of events processed per time unit. As the exception operator has two streams as input, the cost is $\mu_s(E1) + \mu_s(E2)$.

Cost of union operator. The union operator also has two streams as input. Its cost is therefore $\mu_s(E1) + \mu_s(E2)$.

Cost of time operator. The CPU cost is equal to the average number of processed events $\mu_s(E)$.

Cost of next operator. The cost function of the next operator differs significantly from other operators due to an effect we call *bursting*. It occurs when a next operator $E1; E2_{\varphi,\theta}$ has event streams with $\mu_s(E1) > \mu_s(E2)$. As a result, one or more instances are spawned for each event $\in E1$. When the next event $\in E2$ occurs, multiple complex events are detected and output simultaneously — a burst.

We define the *burst size* $\mathrm{b_{size}}(E)$ as the number of complex events in a burst and the *burst rate* $\mathrm{b_{rate}}(E)$ as the number of bursts occurring per second. The burst rates and sizes of most operators are straight-forward and are shown in Table 1. For the next operator, the burst size is recursively defined by: $\mathrm{b_{size}}(E1; E2) = \mathrm{b_{size}}(E1)\ \mathrm{b_{rate}}(E1)\ \mathrm{DT}(E2)$ because $\mathrm{DT}(E2)$ is the average time between two events $\in E2$ and $\mathrm{b_{rate}}(E1)\ \mathrm{DT}(E2)$ is the number of bursts of $E1$, each of size $\mathrm{b_{size}}(E1)$, that occurred in that time period.

The burst rate of the next operator is $\mathrm{b_{rate}}(E1; E2) = 1/\max[\mathrm{DT}(E1), \mathrm{DT}(E2)]$ because a burst is sent from the next operator when a burst of $E2$ occurs. The average time between two consecutive bursts $\in E2$ is $\mathrm{DT}(E2)$. However, the burst rate cannot exceed the time between two bursts of $E1$ ($\mathrm{DT}(E1)$) since at least one event $\in E1$ must occur before $E2$ due to the semantics of the next operator.

Therefore the cost of the next operator is as follows. First, the operator has to process all events $E1$, which adds a term of size $\mu_s(E1)$. The number of bursts of $E1$ that occur before a burst of $E2$ is $\mathrm{DT}(E2)/\mathrm{DT}(E1)$. Each $E1$ burst is of size $\mathrm{b_{size}}(E1)$. Thus $\frac{\mathrm{DT}(E2)}{\mathrm{DT}(E1)}\ \mathrm{b_{size}}(E1)$ instances have to process the $\mathrm{b_{size}}(E2)$ number of $E2$ events. This processing occurs $\mathrm{b_{rate}}(E2)$ times per second (each time an $E2$ burst occurs). However, if the burst rate of $E2$ is higher than $E1$, there are events $\in E2$ that have no matching $E1$ events. These remaining events are processed (and discarded) by the operator at a cost of $\max(\mathrm{b_{rate}}(E2) - \mathrm{b_{rate}}(E1), 0)\ \mathrm{b_{size}}(E2)$. This gives a total cost of:

$$\begin{aligned}\mathrm{cost}(E1; E2) &= \mu_s(E1) + \\ &\frac{\mathrm{DT}(E2)}{\mathrm{DT}(E1)}\ \mathrm{b_{size}}(E1)\ \mathrm{b_{size}}(E2)\ \mathrm{b_{rate}}(E2) + \\ &\max(\mathrm{b_{rate}}(E2) - \mathrm{b_{rate}}(E1), 0)\ \mathrm{b_{size}}(E2)\end{aligned}$$

Cost of deployment plan. The cost of an operator deployment plan includes both the cost of placing an operator on a different node than its suboperators, which results in network traffic, and also the variance of CPU utilisation of different nodes. The cost function is given by:

$$\begin{aligned}\mathrm{cost}(Q, P) = \sum_{o \in Q} \Bigg[&\sum_{o_2 \in \mathrm{sub}(o)} \Big(\delta(P(o), P(o_2)) \cdot \mu_s(o_2)\Big) + \\ &\mathrm{has}(o, P(o)) \cdot \mathrm{CPU}(o) + \sum_{o_2 \in \mathrm{ops}(o, P(o))} \mathrm{CPU}(o_2) \Bigg]\end{aligned}$$

where P maps operators to nodes, $\mathrm{sub}(o)$ are the suboperators of operator o, $\mathrm{ops}(o, n)$ are the operators excluding o running on node n. $\delta(n_1, n_2)$ is defined to return 1 if and only if $n_1 = n_2$ and 0 otherwise. $\mathrm{has}(o, n)$ returns 0 if node n already has operator o running and 1 otherwise.

As can be seen from the cost function, a deployment of operator o is expensive if its suboperators are placed on an-

Operator (o)	$\mu_s(o)$	Detection time	Burst size ($\mathrm{b}_{\mathrm{size}}(o)$)	Burst rate ($\mathrm{b}_{\mathrm{rate}}(o)$)
E_θ	$\mu_s(E)$	$\mathrm{DT}(E)$	$\mathrm{b}_{\mathrm{size}}(E)$	$\mathrm{b}_{\mathrm{rate}}(E)$
$E1\backslash E2_\lambda$	$\mu_s(E1)$	$\mathrm{DT}(E1)$	$\mathrm{b}_{\mathrm{size}}(E1)$	$\mathrm{b}_{\mathrm{rate}}(E1)$
$E1\|E2$	$\mu_s(E1)+\mu_s(E2)$	$\frac{1}{\mathrm{b}_{\mathrm{rate}}(E1)+\mathrm{b}_{\mathrm{rate}}(E2)}$	$\frac{\mathrm{b}_{\mathrm{size}}(E1)\ \mathrm{b}_{\mathrm{rate}}(E1)+\mathrm{b}_{\mathrm{size}}(E2)\ \mathrm{b}_{\mathrm{rate}}(E2)}{\mathrm{b}_{\mathrm{rate}}(E1)+\mathrm{b}_{\mathrm{rate}}(E2)}$	$\mathrm{b}_{\mathrm{rate}}(E1)+\mathrm{b}_{\mathrm{rate}}(E2)$
$E1;E2_{\varphi,\theta}$	$\mu_s(E1)$	$\mathrm{DT}(E1)+\mathrm{DT}(E2)$	$\mathrm{b}_{\mathrm{size}}(E1)\ \mathrm{b}_{\mathrm{rate}}(E1)\ \mathrm{DT}(E2)$	$\frac{1}{\max(\mathrm{DT}(E1),\mathrm{DT}(E2))}$
$E+_{\varphi,\theta}$	unbounded	$\mathrm{DT}(E)$	unbounded	unbounded
$E@_{time}$	$\mu_s(E)$	$\mathrm{DT}(E)+time$	$\mathrm{b}_{\mathrm{size}}(E)$	$\mathrm{b}_{\mathrm{rate}}(E)$

Table 1: Worst-case event rates of complex events (ignoring predicates), average detection times and burst sizes and rates of operators.

other node than o or if the total CPU utilisation of the selected node is high. However, the cost is lowered if the operator already exists on the selected node. This favours deployment plans that reuse operators, deploy suboperators on the same node as parent operators and do load-balancing.

4.3 Query Rewriting

Next we give the transformation rules and algorithms for finding optimal, equivalent patterns with lower CPU cost for three operators.

Patterns with union operator. The union operator is commutative and associative and following equivalences hold: $E1|E2 \equiv E2|E1$ and $E1|(E2|E3) \equiv (E1|E2)|E3$. Now, consider three of the 12 equivalent patterns of $E1|(E2|E3)$: (1) $E1|(E2|E3)$, (2) $E2|(E1|E3)$ and (3) $E3|(E1|E2)$. Pattern (1) is optimal with respect to CPU cost if $\mu_s(E1) > \max_{i=2,3}[\mu_s(E_i)]$. The reason is that an event $e \in E1$ is then only processed by one operator instead of two, as in patterns (2) and (3).

In general, the problem of finding the most efficient pattern is solvable by enumerating all equivalent patterns and calculating the CPU cost for each. As there are $(n+1)! \cdot \frac{(2n)!}{(n+1)!n!} = \frac{(2n)!}{n!}$ different patterns with n union operators, this is infeasible for larger patterns. (The number of ways n internal nodes can be arranged in a binary tree (i.e., the parentheses) is given by the Catalan number $C_n = \frac{(2n)!}{(n+1)!n!}$. For each tree, there are $(n+1)!$ different ways to place the leaves (i.e., subpatterns).)

Instead, we propose a greedy algorithm for calculating the optimal solution in $O(n \log n)$ steps. An event pattern can be thought of as a tree with union operators as internal nodes and subpatterns (other operators or primitive streams) as leaves. Let E_i be a leaf with an associated event rate $\mu_s(E_i)$ and let $d_T(E_i)$ denote the depth of the leaf in the tree. The cost incurred by E_i is $\mu_s(E_i)\ d_T(E_i)$ since each event has to be processed by all union operators on the path to the root operator. The total cost of the tree is $\sum_i \mu_s(E_i)\ d_T(E_i)$.

This problem is isomorphic to finding the optimal prefix code for compression of data, i.e., a set of bit strings, in which no string is the prefix of another. A prefix code is also represented by a binary tree: Each internal node represents either a one or a zero and each leaf represents the frequency of a character. The bits on the path from the root to a leaf give the character encoding. Now, the cost of a binary tree is $\sum_{c \in C} f(c)\ d_T(c)$ where C is the set of characters and $f(c)$ is the frequency of character c. The problem of finding an optimal prefix code is solvable by the greedy *Huffman* algorithm. An adopted version of this algorithm can be applied to rewrite union expressions. We omit the proof that closely follows the one given in [12].

Fig. 3 shows the pseudo code for the union rewriting algorithm. It first builds a sorted set of subpatterns in the event pattern that are operands of a union operator (line 2). It

```
1 findOptimalUnion(e_pat)
2     Q = extractUnionOperands(e_pat)
3     n = Q.size()
4     for(i = 0; i < n - 1; i++)
5         l = Q.extractMinRate()
6         r = Q.extractMinRate()
7         newRate = l.rate() + r.rate()
8         Q.insert(new UnionOperator(l, r), newRate)
9     return Q.extractMinRate()
```

Figure 3: Pseudo code for finding the optimal equivalent pattern of union operators.

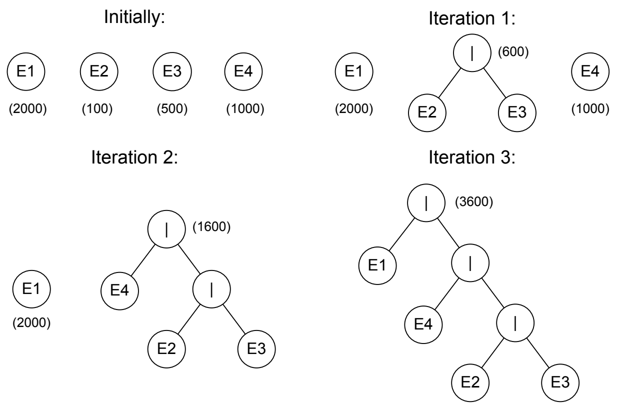


Figure 4: Example of the union rewriting algorithm applied to event pattern $E1|E2|E3|E4$ with the given rates. The optimal pattern is $(E1|(E4|(E2|E3)))$ with a cost of 3600 events/second.

then constructs the resulting tree of union operators (lines 4–8) by iteratively extracting the two event patterns in `Q` with the lowest rate (CPU cost). After $n-1$ iterations, the set consists of a single element, the optimised pattern of union operators (line 9). Fig. 4 gives an example of the operation of this algorithm.

Patterns with next operator. The next operator is associative and thus $E1;(E2;E3) \equiv (E1;E2);E3$ holds. The cost of each of the two event patterns depends on the properties of $E1$, $E2$ and $E3$. As in the case of the union operator, the lowest cost pattern can be found by enumerating all equivalent patterns and computing each cost. There are $\frac{(2n)!}{(n+1)!n!}$ enumerations where n is the number of next operators and, as in the previous case, this is infeasible.

We solve the problem of finding the optimal pattern of next operators using a dynamic programming approach. Fig. 5 shows the algorithm that iteratively finds optimal subexpressions of increasing length `l` until an optimal (sub)expression with length `n` is found (for-loop in lines 4–13). The optimal subexpressions are kept in list `list`, which initially holds all subexpressions of length 1. As optimal subexpressions are found, they are appended to the list. The for-loop in lines 5–13 finds the `n-l` optimal subexpressions of length `l`. For each subexpression of length `l` starting at position `i` and

```
1 findOptimalNext(e_pat)
2     list = extractNextOperands(e_pat)
3     n = list.size()
4     for(l = 2; l =< n; l++)
5         for(i = 0; i < n-l+1; i++)
6             best = new InfinityCost()
7             j = i+l
8             for(k = i; k < j; k++)
9                 ele1 = list.get(indexOf(i, k, n))
10                ele2 = list.get(indexOf(k+1, j, n))
11                if(cost(ele1, ele2) < best.cost())
12                    best = New NextOperator(ele1, ele2)
13            list.add(best)
14    return list.get(indexOf(0, n-1, n))
```

Figure 5: Pseudo code for finding the optimal equivalent pattern of next operators.

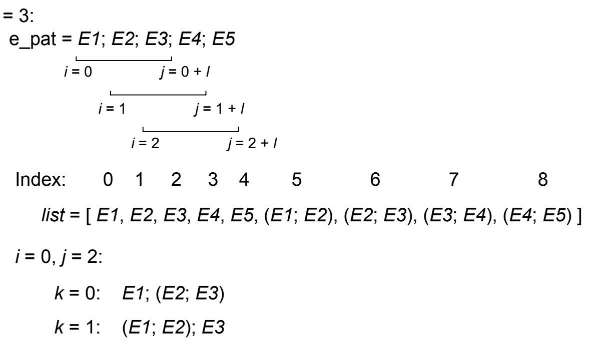


Figure 6: Sample execution of the next rewriting algorithm for pattern $E1; E2; E3; E4; E5$.

ending at position `j=i+l-1`, the optimal expression is found by finding the two subexpressions that concatenated start at `i` and end at `j`, and have the lowest cost (for-loop in lines 8–12). The function `indexOf` returns the index of the optimal (previously found) subexpression of length `l` (starting at `a` and ending at `b` in the pattern) in `list`.

The time complexity of the algorithm is $\Theta(n^3)$ due to the three for-loops. Its space complexity is $\Theta(n^2)$ because the `list` contains $\sum_{i=1}^{n} i$ elements at termination.

A snapshot of a sample execution of the algorithm is shown in Fig. 6. The algorithm is at line 8 and is about to compute the optimal pattern starting at index 0 with length 3. As shown in the figure, there are two different patterns to consider: $E1; (E2; E3)$ and $(E1; E2); E3$. The one with the lower cost is appended to the list in line 13. After that, the same computations are performed again for the next pattern of length 3, $E2; E3; E4$ ($i = 1$) and so on.

Patterns with exception operator. The exception pattern $E1; (E2\backslash E3)$ is equivalent to $(E1; E2)\backslash E3$ because the terminating pattern $E3$ only influences the composite event detection after the next operator has detected $E1$, which does not depend on $E2$. This allows the next rewriting algorithm to be used.

4.4 Operator Distribution

We developed a simple greedy algorithm for choosing operator deployment plans. The algorithm reuses already deployed operators and deploys the remaining operators in a bottom-up fashion. Existing operators are stored in a hash map to achieve fast look-up. First, a submitted query is traversed top-down to find the largest equivalent deployed operator in the hash map, if any, starting with the entire query. If found, the (sub)expression is replaced with a marker containing the operator identifier and location to allow operators to be connected once deployed. Next, the remaining operators are deployed bottom-up. The location of each operator is selected by recursively placing the left and right subexpressions of the operator and then the operator itself. An operator is placed by calculating the *cost* of placing the operator on each node and selecting the lowest cost node, as described in §4.2. This approach has a time complexity of $O(Q\ N)$ where Q is the number of operators in the submitted query and N is the number of nodes.

Information about operators' deployments are stored incrementally, as efficient locations are found, to support operator reuse even within a single query. To avoid requesting information about hosted operators from nodes themselves, we store information about deployment plans at a central location. Memory usage is proportional to the number of operators times the average size of the *abstract operator tree* (AOT) that describes each operator. To reduce memory usage, the hash value of AOTs could be used as key and AOTs could be stored on disk and retrieved on-demand.

The advantage of our distribution approach is that it selects good deployment plans while having low time complexity. However, as the algorithm is greedy, it may not find the optimal solution. In particular, as suboperators are placed independently of each other, they may not be placed on the same node, even if this would result in a better solution after placing the parent operator. This could be addressed by post-processing deployment plans using a meta-heuristic, such as simulated annealing. We leave this to future work.

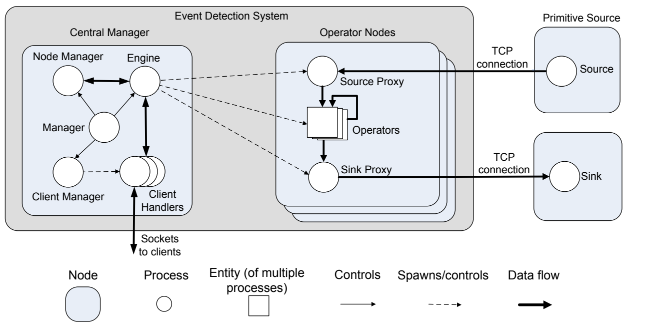


Figure 7: Overview of the Next CEP system design.

5. THE NEXT CEP SYSTEM

In this section, we describe the design and implementation of the NEXT CEP system. As shown in Fig. 7, it consists of a *Central Manager*, *Operator Nodes* and *Source/Sink Proxies*.

The *Central Manager* receives, processes and optimises queries and instantiates them on available physical Operator Nodes. These tasks are divided between different processes. The Node Manager monitors which Operator Nodes are available for operator distribution. The Client Manager accepts connections from clients and spawns a client handler that receives queries, sends them to the Engine and returns responses from the Engine to clients. The Engine parses queries, registers sources and sinks and optimises queries received from clients and spawns necessary Operators and Sink and Source Proxies on the available nodes according to the query deployment plan. The Engine also keeps track of running queries and statistics used for query optimisation.

We chose a centralised management design for its simplicity. It also facilitates query optimisation: Since the Central

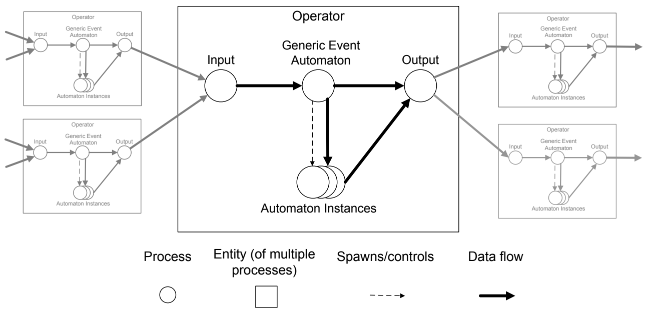


Figure 8: Design of operators and their interaction with other operators.

Manager has knowledge of all operators and their deployment, it is possible to reuse existing operators when new queries are submitted.

The *Operator Nodes* execute the operators and proxies. The actual sources and sinks are not part of the system and are connected through TCP connections to ensure ordering of packets according to our event model from §3.1. The operators on an Operator Node can send events to each other and also to operators on other nodes in the network.

Operators. Each operator has four types of processes, as illustrated in Fig. 8. Incoming events are received by an *Input* process and are then processed by a *Generic Event Automaton*. This automaton is instantiated with a specification of the particular operator type to execute (e.g., the next operator). We chose this approach for its flexibility and extensibility but it suffers from reduced operator performance. The Generic Event Automaton spawns *Automaton Instances* according to the operator's semantics. An *Output* process sends detected complex events to other operators or sink proxies and also allows operators to be reused.

A problem inherent to distribution is the lack of global time [22]. We assume that sources are synchronised using the Network Time Protocol (NTP), which results typically in 1–50 ms accuracy. On the other hand correct detection does not depend on clock synchronisation of Next nodes since we implement a *guaranteed detection policy*, as found in the *DistCED* [24] system. Under this policy, events have to become *stable* before being consumed by operators. An event is stable if no other (delayed) event in the system should be processed instead. Since channels do not reorder events, stability of events can be achieved by having the Input delay events from other operators or sources until the *next event* has been received.

A benefit of the Next CEP design is that operators are share-nothing processes communicating through events that can thus be distributed transparently. Therefore more CPU resources can be utilised resulting in higher throughput at the cost of increased latency. This is relevant in our credit card fraud detection scenario from §1.1. Most queries include patterns involving only transactions of the same card. As a card is only used infrequently, e.g., once a day on average, and the data rate is 2100 transactions per second, there will be a large number of automaton instances waiting for the next transaction. By distributing these instances to multiple machines, our system can then handle the workload.

Prototype implementation. To simplify development relating to concurrency and distribution, we decided to implement the system in *Erlang*. Erlang is a functional language designed for distributed and fault-tolerant computing based on the actor model. In this model, processes (actors) only interact by asynchronous message passing within a single host or between hosts. Erlang processes are lightweight language-level threads that are mapped to operating system threads. Therefore they have small memory footprints and lower context-switching overheads.

The drawback of Erlang is reduced execution performance. Although the Erlang virtual machine is capable of executing native, pre-compiled code, it remains substantially slower than systems programming languages, such as C or C++. However, for the purpose of this work, we were interested in *relative* performance improvements due to query optimisation techniques, as opposed to absolute performance numbers. Although our absolute performance remains several orders of magnitude lower than that of state-of-the-art CEP implementations, we believe that it nevertheless shows the validity of our proposed optimisation techniques. The absolute performance of our prototype implementation could be improved by creating operator-specific (as opposed to generic) event automata in C and calling these from Erlang.

To evaluate our query optimisation techniques, we wanted a prototype implementation with predictable operator performance. This allowed us to validate the cost functions used in the query optimisation. As a result, our implementation does not use the "simple solution" everywhere. For example, we implemented an efficient algorithm using general balanced trees to compute event stability in the Input process. Our implementation also maintains an index of automaton instances for fast event dispatching.

6. EVALUATION

Our evaluation had two main goals: assessing the potential efficiency gains of query optimisation and evaluating the system behaviour when deployed on multiple machines. Consequently, we separate the two aspects by first measuring query optimisation improvements on a single machine and then analysing system behaviour for the distributed case.

Methodology. Our tests were performed on the *Emulab* test-bed [15]. In the single node case, we deployed the Next CEP system on a Xeon 64-bit 3GHz machine that hosted the operators, while the manager, the sources and the sinks were hosted on other machines. For the distributed case, we allocated 40 850Mhz Pentium III machines for the operators and again separate machines to measure the system and inject events. Nodes were connected to a 100-Mbps LAN.

We set up 8 sources publishing Poisson-distributed events. The mean of the distribution was chosen exponentially increasing for each source, ranging from one event per second, for the slowest one, to 128 events per second, for the fastest one. The combined average of all sources was 255 events per seconds. To stress CPU usage, messages were kept small, only including timestamps and theirs ids.

We choose three query workloads, next, union and mix, to investigate the rewriting of the two main operators, union and next, and also to represent realistic queries as featured in the credit card fraud detection scenario. We composed queries of the next and union workloads by uniformly generating trees of a given size and randomly allocating sources to leaves. Queries of the mix workload were built by recursively nesting union or next subtrees of two operators. While the system without query rewriting used the query as issued, the

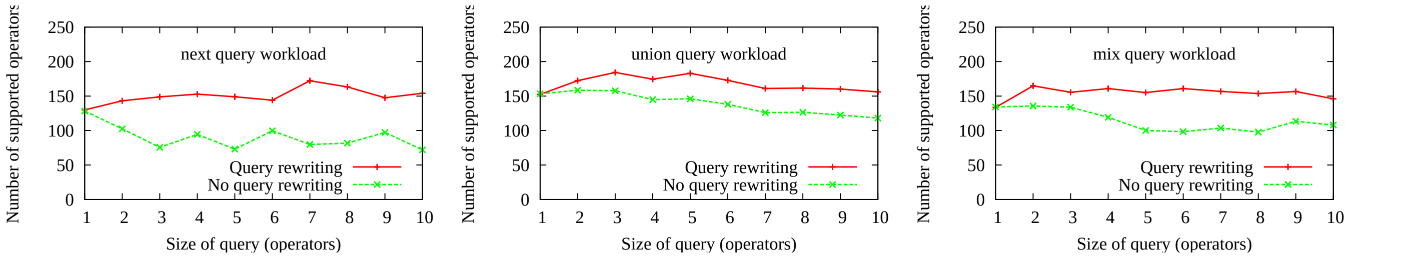


Figure 9: Scalability improvement due to query rewriting for next, union and mix query workloads as a function of query size (single node case).

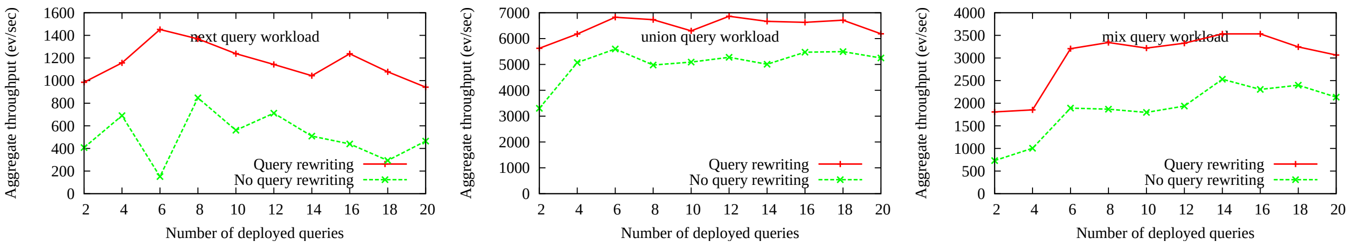


Figure 10: Throughput improvement due to query rewriting for next, union and mix query workloads as a function of query number (single node case).

rewriting query compiler of the NEXT CEP system rewrote it to a more optimal form before deploying it.

The additional resources due to query rewriting can be exploited to increase the number of operators supported concurrently by the system or to support sources with higher event rates. To test the first case, we measured the maximum number of operators that could be run on a single machine without overloading it and repeated this with query rewriting turned off. To determine overload, we first timestamped events at the input queue of operators, where events are stabilised, and again when events left the queue to enter the operator. This enabled us to distinguish, for each event, the portion of the latency measured at the sink caused by event stabilisation and the part caused by queueing and processing delays. We declared the system overloaded when the 80th percentile of processing delay grew above one second in a 10-second observation window.

(1) Single node case. We conducted several experiments changing the query size and computed averages over multiple runs. The 5-run average results on a single node are given in Fig. 9. For each query workload, the graphs shows the maximum number of non-overloaded operators that can be supported with and without query rewriting, as a function of query size. The effect of query rewriting is significant for next queries and increases with query size, as the union operator is less costly than the next operator the performance gains for **union** queries are less substantial, also improving for larger queries. The result for the mix workload is between the next and union.

Fig. 10 shows the dual scalability aspect by fixing the number of deployed queries and testing the maximum rate of complex event detection. In this test, we registered a variable number of 5-operator queries and increased the source rates up to the maximum throughput achievable by the system. The graph for the next workload shows that query rewriting increases the achievable throughput substantially, doubling it in many cases. The throughput improvement for union is still significant, about 20% on average, with mix still showing an intermediate behaviour between the two.

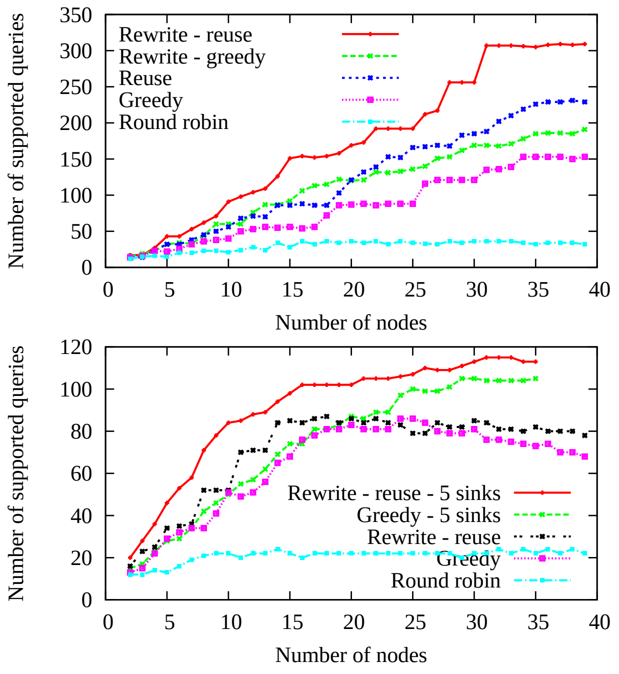


Figure 11: Scalability improvement for next (top) and union (bottom) query workloads as a function of node number (distributed case).

(2) Distributed case. To test scalability in the multi-node case, we measured the number of supported 5-operator next and union queries as we added more operator machines. The distribution algorithm plays a crucial role: for the next workload (top chart in Fig. 11), the number of queries supported by Round robin placement grows slowly up to 15 nodes and is flat thereafter. Using our Greedy placement approach, scalability is improved considerably and is further increased after enabling the planner's Reuse feature. The Rewrite graphs in the figure show that query rewriting improves upon Greedy and Reuse: with 40 nodes, Rewrite improves Greedy by 24% and Reuse by 34%. When query rewriting is applied before reuse, query rewriting increases reuse opportunities by finding equivalent patterns and rewriting them in a more efficient way.

For the union workload (bottom chart in Fig. 11), the system saturates at around 80 queries. This can be explained by considering that each next query outputs 32 events/sec on average (i.e., the rate of the first source), while the 5-operator union query produces six times that, 191 events/sec on average. With 80 queries, the union workload outputs about 15,000 events/sec vs. 2500 events/sec for next. We determined that this caused the sink proxy machine to became overloaded. When we removed this bottleneck by allocating 5 machines as sink proxies, the distance between Rewrite and Greedy improved to be up to 50% at about 10 nodes (as shown). We believe that this demonstrates how our distributed NEXT CEP system facilitates the addition of new resources to remove bottlenecks.

7. CONCLUSIONS

In this paper, we describe the NEXT CEP system, a distributed event processing system with automated query rewriting and distributed deployment implemented in Erlang. The system uses a new high-level event query language for expressing event patterns that supports rewriting and separates patterns from predicates. Patterns are detected through a simple, yet expressive automaton-based approach. The CPU usage of different operators is estimated using a cost model that takes the burstiness of operators into account. We show how patterns with three operators (union, next and exception) can be rewritten to have lower costs. We also describe a greedy algorithm for selecting deployment plans that reuse existing operators.

For future work, we plan to extend the cost model to include operator selectivities and processing costs of predicates. In addition, we want to examine the interplay of language level optimisation techniques with dynamic optimisation at query run-time. We believe that our cost model will also be applicable to this case and enable the CEP system to reconsider query optimisation decisions as resource availability and properties of event sources change.

8. REFERENCES

[1] D. Abadi, D. Camey, U. Cetintemel, M. Chemiack, et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 2003.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, Asilomar, CA, January 2005.

[3] Y. Ahmad and U. Çetintemel. Network-Aware Query Processing for Stream-based App. In *VLDB*, 2004.

[4] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based Complex Event Detection across Distributed Event Sources. In *VLDB*, 2008.

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, et al. STREAM: The Stanford Data Stream Management System. http://infolab.stanford.edu/stream, 2004.

[6] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2), 2006.

[7] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *Proc. of NSDI'04*, San Francisco, CA, Mar. 2004.

[8] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, et al. Monitoring Streams - A New Class of Data Management Applications. In *VLDB*, 2002.

[9] S. Chakravarthy and D. Mishra. Snoop: an Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[10] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, et al. Scalable Distributed Stream Processing. In *CIDR*, Asilomar, CA, January 2003.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press and McGraw-HilDyanl, 2nd edition, 2003.

[13] A. Demers, J. Gehrke, M. Hong, B. Panda, et al. Towards Expressive Publish/Subscribe Systems. In *EDBT*, 2006.

[14] A. Demers, J. Gehrke, M. Hong, B. Panda, et al. Cayuga: A Genaral Purpose Event Monitoring System. In *CIDR*, pages 412–422, 2007.

[15] Emulab's homepage. `http://www.emulab.net/`.

[16] R. S. Epstein and M. Stonebraker. Analysis of Distributed Data Base Processing Strategies. In *VLDB*, pages 92–101, 1980.

[17] Esper's homepage. `http://esper.codehaus.org`.

[18] Federal Bureau of Investigation. Financial Crimes Report to the Public Fiscal Year 2007. Technical report, Federal Bureau of Investigation, 2007.

[19] L. Fiege, G. Muhl, and P. R. Pietzuch. *Distributed Event-based Systems.* Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 2006.

[20] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *RIDE-ADS*, pages 2–9, 1994.

[21] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[22] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[23] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, et al. Network-Aware Operator Placement for Stream-Processing Sys. In *ICDE*, 2006.

[24] P. R. Pietzuch, B. Shand, and J. Bacon. A Framework for Event Composition in Distributed Systems. In *Middleware*, Rio de Janeiro, Brazil, jun 2003.

[25] S. Seshadri, V. Kumar, and B. F. Cooper. Optimizing Multiple Queries in Distributed Data Stream Systems. In *NetDB*, page 25, 2006.

[26] Visa Inc. Operational Performance Data. `http://corporate.visa.com/md/st/`, 2008.

[27] Walker White and Mirek Riedewald and Johannes Gehrke and Alan Demers. What is "Next" in Event Processing? In *SIGMOD-SIGACT-SIGART*, 2007.

[28] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *ICDE*, pages 791–802, 2005.

[29] M. T. Özsu and P. Valduriez. *Principles of Distributed Databases.* Prentice-Hall Inc., 2nd edition, 1999.