Efficient Pattern Matching over Event Streams *

Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA, USA
{jagrati, yanlei, dpg, immerman}@cs.umass.edu

ABSTRACT

Pattern matching over event streams is increasingly being employed in many areas including financial services, RFID-based inventory management, click stream analysis, and electronic health systems. While regular expression matching is well studied, pattern matching over streams presents two new challenges: Languages for pattern matching over streams are significantly richer than languages for regular expression matching. Furthermore, efficient evaluation of these pattern queries over streams requires new algorithms and optimizations: the conventional wisdom for stream query processing (i.e., using selection-join-aggregation) is inadequate.

In this paper, we present a formal evaluation model that offers precise semantics for this new class of queries and a query evaluation framework permitting optimizations in a principled way. We further analyze the runtime complexity of query evaluation using this model and develop a suite of techniques that improve runtime efficiency by exploiting sharing in storage and processing. Our experimental results provide insights into the various factors on runtime performance and demonstrate the significant performance gains of our sharing techniques.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

Algorithms, Design, Performance, Theory

Keywords

Event streams, pattern matching, query optimization

1. INTRODUCTION

Pattern matching over event streams is a new processing paradigm where continuously arriving events are matched against complex patterns and the events used to match each pattern are transformed into new events for output. Recently, such pattern matching over streams has aroused significant interest in industry [28, 30, 29, 9] due to its wide applicability in areas such as financial services [10], RFID-based inventory management [31], click stream analysis [26], and electronic health systems [16]. In financial services, for instance, a brokerage customer may be interested in a sequence of stock trading events that represent a new market trend. In RFID-based tracking and monitoring, applications may want to track valid paths of shipments and detect anomalies such as food contamination in supply chains.

While regular expression matching is a well studied computer science problem [17], pattern matching over streams presents two new challenges:

Richer Languages. Languages for pattern matching over event streams [10, 15] are significantly richer than languages for regular expression matching. These event pattern languages contain constructs for expressing sequencing, Kleene closure, negation, and complex predicates, as well as strategies for selecting relevant events from an input stream mixing relevant and irrelevant events. Of particular importance is Kleene closure that can be used to extract from the input stream a finite yet unbounded number of events with a particular property. As shown in [15], the interaction of Kleene closure and different strategies to select events from the input stream can result in queries significantly more complex than regular expressions.

Efficiency over Streams. Efficient evaluation of such pattern queries over event streams requires new algorithms and optimizations. The conventional wisdom for stream query processing has been to use selection-join-aggregation queries [3, 7, 8, 24]. While such queries can specify simple patterns, they are inherently unable to express Kleene closure because the number of inputs that may be involved is *a priori* unknown (which we shall prove formally in this paper). Recent studies [10, 26, 34] have started to address efficient evaluation of pattern queries over streams. The proposed techniques, however, are tailored to various restricted sets of pattern queries and pattern matching results, such as patterns without Kleene closure [34], patterns only on contiguous events [26], and pattern matching without output of complete matches [10].

The goal of this work is to provide a fundamental evaluation and optimization framework for the new class of pattern queries over event streams. Our query evaluation framework departs from well-studied relational stream processing due to its inherent limitation as noted above. More specifically,

*This work has been supported in part by NSF grants CCF 0541018 and CCF 0514621 and a gift from Cisco.
*Authors of this paper are listed alphabetically.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

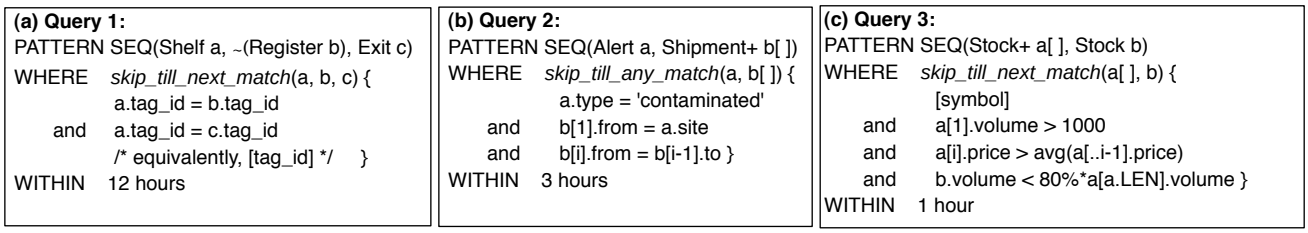


Figure 1: Examples of event pattern queries.

the design of our query evaluation framework is based on three principles: First, the evaluation framework should be sufficient for the full set of pattern queries. Second, given such full support, it should be computationally efficient. Third, it should allow optimization in a principled way. Following these principles, we develop a data stream system for pattern query evaluation. Our contributions include:

- **Formal Evaluation Model**. We propose a formal query evaluation model, NFA^b, that combines a finite automaton with a match buffer. This model offers precise semantics for the complete set of event pattern queries, permits principled optimizations, and produces query evaluation plans that can be executed over event streams. The NFA^b model also allows us to analyze its expressibility in relation to relational stream processing, yielding formal results on both sufficiency and efficiency for pattern evaluation.
- **Runtime Complexity Analysis**. Given the new abstraction that NFA^b-based query plans present, we identify the key issues in runtime evaluation, in particular, the different types of non-determinism in automaton execution. We further analyze worst-case complexity of such query evaluation, resulting in important intuitions for runtime optimization.
- **Runtime Algorithms and Optimizations**. We develop new data structures and algorithms to evaluate NFA^b-based query plans over streams. To improve efficiency, our optimizations exploit aggressive sharing in storage of all possible pattern matches as well as in automaton execution to produce these matches.

We have implemented all of the above techniques in a Java-based prototype system and evaluated NFA^b based query plans using a range of query workloads. Results of our performance evaluation offer insights into the various factors on runtime performance and demonstrate significant performance gains of our sharing techniques.

The remainder of the paper is organized as follows. We provide background on event pattern languages in Section 2. We describe the three technical contributions mentioned above in Section 3, Section 4, and Section 5, respectively. Results of a detailed performance analysis are presented in Section 6. We cover related work in Section 7 and conclude the paper with remarks on future work in Section 8.

2. BACKGROUND

In this section, we provide background on event pattern languages, which offers a technical context for the discussion in the subsequent sections.

Recently there have been a number of pattern language proposals including SQL-TS [26], Cayuga [10, 11], SASE+ [34, 15], and CEDR [5].[1] Despite their syntactic variations, these languages share many features for pattern matching over event streams. Below we survey the key features of pattern matching using the SASE+ language since it is shown to be richer than most other languages [15]. This language uses a simple event model: An event stream is an infinite sequence of events, and each event represents an occurrence of interest at a point in time. An event contains the name of its event type (defined in a schema) and a set of attribute values. Each event also has a special attribute capturing its occurrence time. Events are assumed to arrive in order of the occurrence time.[2]

A pattern query addresses a sequence of events that occur in order (not necessarily in contiguous positions) in the input stream and are correlated based on the values of their attributes. Figure 1 shows three such queries.

Query 1 detects shoplifting activity in RFID-based retail management [34]: it reports items that were picked at a shelf and then taken out of the store without being checked out. The PATTERN clause specifies a ***sequence*** pattern with three components: the occurrence of a shelf reading, followed by the non-occurrence of a register reading, followed by the occurrence of an exit reading. Non-occurrence of an event, denoted by '$\sim$', is also referred to as ***negation***.

Each component declares a variable to refer to the corresponding event. The WHERE clause uses these variables to specify ***predicates*** on individual events as well as across multiple events (enclosed in the '{' '}' pair). The predicates in Query 1 require all events to refer to the same tag_id. Such equality comparison across all events is referred to as an equivalence test (a shorthand for which is '[tag_id]'). Finally, the query uses a WITHIN clause to specify a 12-hour ***time window*** over the entire pattern.

Query 2 detects contamination in a food supply chain: it captures an alert for a contaminated site and reports a unique series of infected shipments in each pattern match. Here the sequence pattern uses a ***Kleene plus*** operator to compute each series of shipments (where '+' means one or more). An array variable b[ ] is declared for the Kleene plus component, with b[1] referring to the shipment from the origin of contamination, and b[i] referring to each subsequent shipment infected via collocation with the previous one. The predicates in WHERE clearly specify these constraints on the shipments; in particular, the predicate that compares $b[i]$ with $b[i-1]$ ($i > 1$) specifies the collocation condition between each shipment and its preceding one.

[1] There have also been several commercial efforts and standardization initiatives [9, 28, 29]. The development of these languages is still underway. Thus, they are not further discussed in this paper.

[2] The query evaluation approach that we propose is suited to an extension for out-of-order events, as we discuss more in §8.

Query 3 captures a complex stock market trend: in the past hour, the volume of a stock started high, but after a period when the price increased or remained relatively stable, the volume plummeted. This pattern has two components, a Kleene plus on stock events, whose results are in $a[\,]$, and a separate single stock event, stored in b. The predicate on $a[1]$ addresses the initial volume. The predicate on $a[i]$ ($i > 1$) requires the price of the current event to exceed the average of the previously selected events (those previously selected events are denoted by $a[..i-1]$). This way, the predicate captures a trend of gradual (not necessarily monotonic) price increase. The last predicate compares b to $a[a.\text{LEN}]$, where $a.\text{LEN}$ refers to the last selected event in $a[\,]$, to capture the final drop in volume.

Besides the structure and predicates, pattern queries are further defined using the ***event selection strategy*** that addresses how to select the relevant events from an input stream mixing relevant and irrelevant events. The strategy used in a query is declared as a function in the WHERE clause which encloses all the predicates in its body, as shown in Figure 1. The diverse needs of stream applications require different strategies to be used:

Strict contiguity. In the most stringent event selection strategy, two selected events must be contiguous in the input stream. This requirement is typical in regular expression matching against strings, DNA sequences, etc.

Partition contiguity. A relaxation of the above is that two selected events do not need to be contiguous; however, if the events are conceptually partitioned based on a condition, the next relevant event must be contiguous to the previous one in the same partition. The equivalence tests, e.g., [symbol] in Query 3, are commonly used to form partitions. Partition contiguity, however, may not be flexible enough to support Query 3 if it aims to detect the general trend of price increase despite some local fluctuating values.

Skip till next match. A further relaxation is to completely remove the contiguity requirements: all irrelevant events will be skipped until the next relevant event is read. Using this strategy, Query 1 can conveniently ignore all the readings of an item that arise between the first shelf reading and an exit or register reading. Similarly, Query 3 can skip values that do not satisfy the defined trend. This strategy is important in many real-world scenarios where some events in the input are "semantic noise" to a particular pattern and should be ignored to enable the pattern matching to continue.

Skip till any match. Finally, skip till any match relaxes the previous one by further allowing non-deterministic actions on *relevant* events. Query 2 illustrates this use. Suppose that the last shipment selected by the Kleene plus reaches the location X. When a relevant shipment, e.g., from X to Y, is read from the input stream, skip till any match has two actions: (1) it selects the event in one instance of execution to extend the current series, and (2) it ignores the event in another instance to preserve the current state of Kleene closure, i.e. location X, so that a later shipment, e.g., from X to Z, can be recognized as a relevant event and enable a different series to be instantiated. This strategy essentially computes transitive closure over relevant events (e.g., all infected shipments in three hours) as they arrive.

Finally, each match of a pattern query (e.g., the content of a[ ] and b variables for Query 3) is output as a composite event containing all the events in the match. Two ***output formats*** are available [15, 28]: The default format returns all matches of a pattern. In contrast, the non-overlap format outputs only one match among those that belong to the same partition (for strict contiguity, treat the input stream as a single partition) and overlap in time; that is, one match in a partition is output only if it starts after the previous match completes. Language support is also available to compute summaries for composite events and compose queries by feeding events output from one query as input to another [15, 28]. These additional features are not a focus of this paper and can be readily plugged in the query evaluation framework proposed below.

3. FORMAL SEMANTIC MODEL

After describing event pattern queries, we study their evaluation and optimization in the rest of the paper. In this section, we present a formal evaluation model that offers precise semantics for this new class of pattern queries (§3.1). We also offer compilation algorithms that translate pattern queries to representations in this model, thereby producing query evaluation plans for runtime use (§3.2). This model further allows us to analyze its expressibility in relation to relational stream processing, yielding formal results on both sufficiency and efficiency for pattern evaluation (§3.3) .

3.1 An Evaluation Model: NFA^b Automaton

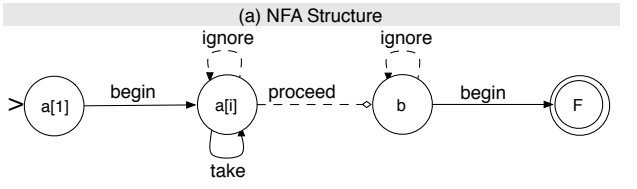
Our query evaluation model employs a new type of automaton that comprises a nondeterministic finite automaton (NFA) and a match buffer, thus called NFA^b, to represent each pattern query. Formally, an NFA^b automaton, $A = (Q, E, \theta, q_1, F)$, consists of a set of states, Q, a set of directed edges, E, a set of formulas, θ, labelling those edges, a start state, q_1, and a final state, F. The NFA^b for Query 3 is illustrated in Figure 2.[3]

States. In Figure 2(a), the start state, $a[1]$, is where the matching process begins. It awaits input to start the Kleene plus and to select an event into the $a[1]$ unit of the match buffer. At the next state $a[i]$, it attempts to select another event into the $a[i]$ ($i > 1$) unit of the buffer. The subsequent state b denotes that the matching process has fulfilled the Kleene plus (for a particular match) and is ready to process the next pattern component. The final state, F, represents the completion of the process, resulting in the creation of a pattern match.

In summary, the set of states Q is arranged as a linear sequence consisting of any number of occurrences of singleton states, s, for non-Kleene plus components, or pairs of states, $p[1], p[i]$, for Kleene plus components, plus a rightmost final state, F. A singleton state is similar to a $p[1]$ state but without a subsequent $p[i]$ state.

Edges. Each state is associated with a number of edges, representing the actions that can be taken at the state. As Figure 2(a) shows, each state that is a singleton state or the first state, $p[1]$, of a pair has a forward *begin edge*. Each second state, $p[i]$, of a pair has a forward *proceed edge*, and a looping *take edge*. Every state (except the start and final states) has a looping *ignore edge*. The start state has no edges to it as we are only interested in matches that start with selected events.

[3] Our NFA^b automata are related to the left-deep automata in [10]. The main differences are that NFA^b employ an additional buffer to compute and store complete matches and can support the compilation of a wider range of queries (more see §7).



(b) Basic Formulas on Edges

θa[1]_begin = a[1].volume>1000	**θa[i]_take** = a[i].symbol=a[1].symbol ∧ a[i].price>avg(a[..i-1].price)	**θb_begin** = b.symbol=a[1].symbol ∧ b.volume<80%*a[a.LEN].volume ∧ b.time<a[1].time+1 hour
	θa[i]_proceed = True	
	θa[i]_ignore = ¬(a[i].symbol=a[1].symbol ∧ a[i].price>avg(a[..i-1].price)	**θb_ignore** = ¬(b.symbol=a[1].symbol ∧ b.volume<80%*a[a.LEN].volume)

(c) Example Formulas after Optimization

θ*a[i]_take = θa[i]_take ∧ a[i].time<a[1].time+1 hour
θ*a[i]_ignore = θa[i]_ignore ∧ a[i].time<a[1].time+1 hour
θ*a[i]_proceed = θb_begin ∨ (¬θ*a[i]_take ∧ ¬θ*a[i]_ignore)

Figure 2: The NFA^b Automaton for Query 3.

Each edge at a state, q, is precisely described by a triplet: (1) a ***formula*** that specifies the condition on taking it, denoted by θ_{q_edge}, (2) an operation on the input stream (i.e., consume an event or not), and (3) an operation on the match buffer (i.e., write to the buffer or not). Formulas of edges are compiled from pattern queries, which we explain in detail shortly. As shown in Figure 2(a), we use solid lines to denote begin and take edges that consume an event from the input and write it to the buffer, and dashed lines for ignore edges that consume an event but do not write it to the buffer. The proceed edge is a special ϵ-edge: it does not consume any input event but only evaluates its formula and tries proceeding. We distinguish the proceed edge from ignore edges in the style of arrow, denoting its ϵ behavior.

Non-determinism. NFA^b automata may exhibit nondeterminism when at some state the formulas of two edges are not mutually exclusive. For example, if $\theta_{p[i]\_\text{take}}$ and $\theta_{p[i]\_\text{ignore}}$ are not mutually exclusive, then we are in a nondeterministic skip-till-any-match situation. It is important to note that such non-determinism stems from the query; the NFA^b model is merely a truthful translation of it.

NFA^b runs. A run of an NFA^b automaton is uniquely defined by (1) the sequence of events that it has selected into the match buffer, e.g., e_3, e_4 and e_6, (2) the naming of the corresponding units in the buffer, e.g., $a[1]$, $a[2]$, and b for Query 3, and (3) the current NFA^b state. We can inductively define a run based on each begin, take, ignore, or proceed move that it takes. Moreover, an **accepting run** is a run that has reached the final state. The semantics of a pattern query is precisely defined from all its accepting runs. These concepts are quite intuitive and the details are omitted in the interest of space.

Pattern queries with negation and query composition are modeled by first creating NFA^b automata for subqueries without them and then composing these automata. In particular, the semantics of negation is that of a nested query, as proposed in [34]. For instance, Query 1 from Figure 1 first recognizes a shelf reading and an exit reading that refer to the same tag; then for each pair of such readings it ensures that there does not exist a register reading of the same tag in between. To support negation using NFA^b, we first compute matches of the NFA^b automaton that includes only the positive pattern components, then search for matches of the NFA^b automaton for each negative component. Any match of the latter eliminates the former from the answer set.

3.2 Query Compilation using NFA^b

We next present the compilation rules for automatically translating simple pattern queries (without negation or composition) into the NFA^b model. Composite automata for negation or composed queries can be constructed afterwards by strictly following their semantics. The resulting representations will be used as query plans for runtime evaluation over event streams.

Basic Algorithm. We first develop a basic compilation algorithm that given a simple pattern query, constructs an NFA^b automaton that is faithful to the original query. In the following, we explain the algorithm using Query 3 as a running example.

Step 1. *NFA^b structure*: As shown in Figure 2, the PATTERN clause of a query uniquely determines the structure of its NFA^b automaton, including all the states and the edges of each state.

The algorithm then translates the WHERE and WITHIN clauses of a query into the formulas on the NFA^b edges.

Step 2. *Predicates*: The algorithm starts with the WHERE clause and uses the predicates to set formulas of begin, take, and proceed edges, as shown in Figure 2(b).[4] It first rewrites all the predicates into conjunctive normal form (CNF), including expanding the equivalence test [symbol] to a canonical form, e.g., a[i].symbol = a[1].symbol. It then sorts the conjuncts based on the notion of their *last identifiers*. In this work, we call each occurrence of a variable in the WHERE clause an identifier, e.g., $a[1]$, $a[i]$, $a[a.\text{LEN}]$, and b for Query 3. The last identifier of a conjunct is the one that is instantiated the latest in the NFA^b automaton. Consider the conjunct "b.volume $< 80\% * a[a.\text{LEN}]$.volume". Between the identifiers b and $a[a.\text{LEN}]$, b is instantiated at a later state.

After sorting, the algorithm places each conjunct on an edge of its last identifier's instantiation state. At the state $a[i]$ where both take and proceed edges exist, the conjunct is placed on the take edge if the last identifier is $a[i]$, and on the proceed edge otherwise (e.g., the identifier is $a[a.\text{LEN}]$). For Query 3, the proceed edge is set to True due to the lack of a predicate whose last identifier is $a[a.\text{LEN}]$.

Step 3. *Event selection strategy*: The formulas on the ignore edges depend on the event selection strategy in use. Despite a spectrum of strategies that pattern queries may use, our algorithm determines the formula of an ignore edge at a state q, θ_{q_ignore}, in a simple, systematic way:

Strict contiguity:	False
Partition contiguity:	¬ (partition condition)
Skip till next match:	¬ (take or begin condition)
Skip till any match:	True

As shown above, when strict contiguity is applied, θ_{q_ignore} is set to False, disallowing any event to be ignored. If partition contiguity is used, θ_{q_ignore} is set to the negation of the partition definition, thus allowing the events irrelevant to a partition to be ignored. For skip till next match, θ_{q_ignore} is set to the negation of the take or begin condition depending

[4] For simplicity of presentation, we omit event type checks in this example. Such checks can be easily added to the edge formulas.

on the state. Revisit Query 3. As shown in Figure 2(b), $\theta_{a[i]\_\text{ignore}}$ is set to $\neg\theta_{a[i]\_\text{take}}$ at the state $a[i]$, causing all events that do not satisfy the take condition to be ignored. Finally, for skip till any match, θ_{q_ignore} is simply set to True, allowing any (including relevant) event to be ignored.

Step 4. *Time window*: Finally, on the begin or proceed edge to the final state, the algorithm conjoins the WITHIN condition for the entire pattern. This condition is simply a predicate that compares the time difference between the first and last selected events against the specified time window.

Optimizations. In our system, the principle for compile-time optimization is to push stopping and filtering conditions as early as possible so that time and space are not wasted on non-viable automaton runs. We highlight several optimizations below:

Step 5. *Pushing the time window early*: The WITHIN condition, currently placed on the final edge to F, can be copied onto all take, ignore, and begin edges at earlier states. This allows old runs to be pruned as soon as they fail to satisfy the window constraint. Despite the increased number of predicates in all edge formulas, the benefit of pruning non-viable runs early outweighs the slight overhead of predicate evaluation. Figure 2(c) shows $\theta_{a[i]\_\text{take}}$ and $\theta_{a[i]\_\text{ignore}}$ after this optimization for Query 3.

Step 6. *Constraining proceed edges*: We next optimize a proceed edge if its current condition is True and the subsequent state is not the final state, which is the case with Query 3. At the state $a[i]$, this proceed edge causes nondeterminism with the take (or ignore) edge, resulting in a new run created for every event. To avoid non-viable runs, we restrict the proceed move by "peeking" at the current event and deciding if it can satisfy the begin condition of the next state b. We disallow a proceed move in the negative case. An exception is that when the take and ignore edges at $a[i]$ both evaluate to False, we would allow an opportunistic move to the state b and let it decide what can be done next. The resulting $\theta_{a[i]\_\text{proceed}}$ is also shown in Figure 2(c).

It is important to note that while our compilation techniques are explained above using pattern queries written in the SASE+ language [15], all the basic steps (Steps 1-4) and optimizations (Steps 5-6) are equally applicable to other pattern languages [5, 11, 26].

3.3 Expressibility of NFA^b

In this section, we provide an intuitive description of the expressibility of the NFA^b model, while omitting the formal proofs in the interest of space (detailed proofs are available in [1]). We briefly describe the set, $\mathcal{D}(\text{NFA}^b)$, that consists of the stream decision problems recognizable by NFA^b automata.

PROPOSITION 3.1. *$\mathcal{D}(NFA^b)$ includes problems that are complete for nondeterministic space $\log n$ ($NSPACE[\log n]$) and is contained in the set of problems recognizable by read-once-left-to-right $NSPACE[\log n]$ machines [32].*

The idea behind the proof of the first part of Proposition 3.1 is that a single Kleene plus in a skip-till-any-match query suffices to express directed graph reachability which is complete for NSPACE[$\log n$]. Query 2 is an example of this. Conversely, an NFA^b reads its stream once from left to right, recording a bounded number of fields, including aggregates, each of which requires $O(\log n)$ bits.

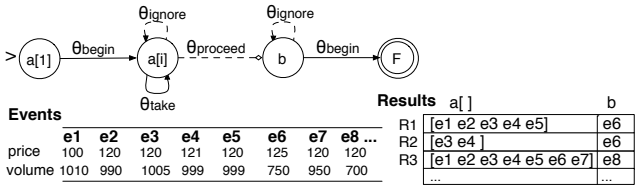


Figure 3: Example pattern matches for Query 3.

We can also prove that any boolean selection-join-aggregation query (a subset of SQL that relational stream systems mostly focus on) is in $\mathcal{D}(\text{NFA}^b)$. Furthermore as is well known, no first-order query even with aggregation can express graph reachability [21]. Thus, Query 2 is not expressible using just selection-join-aggregation. Formally, we have

PROPOSITION 3.2. *The set of boolean selection-join-aggregation queries as well as the set of queries in regular languages are strictly contained in $\mathcal{D}(NFA^b)$.*

Finally, full SQL with recursion [4] expresses all polynomial-time computable queries over streams [18], so this is a strict superset of $\mathcal{D}(\text{NFA}^b)$. However, this language includes many prohibitively expensive queries that are absolutely unnecessary for pattern matching over event streams.

4. RUNTIME COMPLEXITY

Having presented the query evaluation model and compilation techniques, we next turn to the design of a runtime engine that executes NFA^b-based query plans over event streams. The new abstraction that these query plans present and the inherent complexity of their evaluation raise significant runtime challenges. In this section, we describe these challenges in §4.1 and present analytical results of the runtime complexity in §4.2. Our runtime techniques for efficient query evaluation are presented in the next section.

4.1 Key Issues in Runtime Evaluation

The runtime complexity of evaluating pattern queries is reflected by a potentially large number of simultaneous runs, some of which may be of long duration.

Simultaneous runs. For a concrete example, consider Query 3 from Figure 2, and its execution over an event stream for a particular stock, shown in Figure 3. Two patterns matches R_1 and R_2 are produced after e_6 arrives, and several more including R_3 are created after e_8. These three matches, R_1, R_2, and R_3, overlap in the contained events, which result from three simultaneous runs over the same sequence of events.

There are two sources of simultaneous runs. One is that an event sequence initiates multiple runs from the start state and a newer run can start before an older run completes. For example, e_1 and e_3 in Figure 3 both satisfy $\theta_{a[1]\_\text{begin}}$ and thus initiate two overlapping runs corresponding to R_1 and R_2. A more significant source is the inherent nondeterminism in NFA^b, which arises when the formulas of two edges from the same state are not mutually exclusive, as described in §3.1. There are four types of nondeterminism in the NFA^b model:

Take-Proceed. Consider the run initiated by e_1 in Figure 3. When e_6 is read at the state $a[i]$, this event satisfies both $\theta_{a[i]\_\text{take}}$ and $\theta_{a[i]\_\text{proceed}}$, causing the run to split by

taking two different moves and later create two distinct yet overlapping matches R_1 and R_3. Such take-proceed nondeterminism inherently results from the query predicates; it can occur even if strict or partition contiguity is used.

Ignore-Proceed. When the event selection strategy is relaxed to skip till next match, the ignore condition $\theta_{a[i]\_\text{ignore}}$ is also relaxed, as described in §3.2. In this scenario, the ignore-proceed nondeterminism can appear if $\theta_{a[i]\_\text{ignore}}$ and $\theta_{a[i]\_\text{proceed}}$ are not exclusive, as in the case of Query 3.

Take-Ignore. When skip till any match is used, $\theta_{a[i]\_\text{ignore}}$ is set to True. Then the take-ignore nondeterminism can arise at the $a[i]$ state.

Begin-Ignore. Similarly, when skip till any match is used, the begin-ignore nondeterminism can occur at any singleton state or the first state of a pair for the Kleenu plus.

Duration of a run. The duration of a run is largely determined by the event selection strategy in use. When contiguity requirements are used, the average duration of runs is shorter since a run fails immediately when it reads the first event that violates the contiguity requirements. In the absence of contiguity requirements, however, a run can stay longer at each state by ignoring irrelevant events while waiting for the next relevant event. In particular, for those runs that do not produce matches, they can keep looping at a state by ignoring incoming events until the time window specified in the query expires.

4.2 Complexity Analysis

For a formal analysis of the runtime complexity, we introduce the notion of *partition window* that contains all the events in a particular partition that a run needs to consider. Let T be the time window specified in the query and C be the maximum number of events that can have the same timestamp. Also assume that the fraction of events that belong to a particular partition is p (as a special case, strict contiguity treats the input stream as a single partition, so p = 100%). Then the size of the partition window, W, can be estimated using TCp.

The following two propositions calculate a priori worst-case upper bounds on the number of runs that a pattern query can have. The proofs are omitted in this paper. The interested reader is referred to [1] for details of the proofs.

PROPOSITION 4.1. *Given a run ρ that arrives at the state $p[i]$ of a pair in an NFA^b automaton, let $r_{p[i]}(W)$ be the number of runs that can branch from ρ at the state $p[i]$ while reading W events. The upper bound of $r_{p[i]}(W)$ depends on the type(s) of nondeterminism present:*
(i) Take-proceed nondeterminism, which can occur with any event selection strategy, allows a run to branch in a number of ways that is at most linear in W.
(ii) Ignore-proceed nondeterminism, which is allowed by skip-till-next-match or skip-till-any-match, also allows a run to branch in a number of ways that is at most linear in W.
(iii) Take-ignore nondeterminism, allowed by skip-till-any-match, allows a run to branch in a number of ways that is exponential in W.

PROPOSITION 4.2. *Given a run ρ that arrives at a singleton state, s, or the first state of a pair, $p[1]$, in an NFA^b automaton, the number of ways that it can branch while reading W events, $r_{s/p[1]}(W)$, is at most linear in W when skip-till-any-match is used, otherwise it is one.*

Given an NFA^b automaton with states q_1, q_2, ..., $q_m = F$, the number of runs that can start from a given event e, $\tilde{r}_e$, grows with the number of the runs that can branch at each automaton state except the final state. That is, $\tilde{r}_e = r_{q_1}(W_1)\ r_{q_2}(W_2) \dots\ r_{q_{m-1}}(W_{m-1})$, where W_1, W_2, ..., W_{m-1} are the numbers of events read at the states q_1, q_1, ..., q_{m-1} respectively, and $\sum_{i=1}^{m-1} r_{q_i}(W_i) = W$. Obviously, $\tilde{r}_e \leq |\max_{i=1}^{m-1} r_{q_i}(W_i)|^{m-1}$. Then all the runs that can start from a sequence of events e_1, ..., e_W is at most $W|\max_{i=1}^{m-1} r_{q_i}(W_i)|^{m-1}$. Following Propositions 4.2 and 4.2, we have the following upper bounds on the total number of runs for a query:

COROLLARY 4.3. *In the absence of skip till any match, the number of runs that a query can have is at most polynomial in the partition window W, where the exponent is bounded by the number of states in the automaton. In the presence of skip till any match, the number of runs can be at most exponential in W.*

These worst case bounds indicate that a naive approach that implements runs separately may not be feasible. In particular, each run incurs a memory cost for storing a partial or complete match in the buffer. Its processing cost consists of evaluating formulas and making transitions for each input event. It is evident that when the number of runs is large, the naive approach that handles runs separately will incur excessively high overhead in both storage and processing.

Importance of sharing. The key to efficient processing is to exploit sharing in both storage and processing across multiple, long-standing runs. Our data structures and algorithms that support sharing, including a shared match buffer for all runs and merging runs in processing, are described in detail in the next section. In the following, we note two important benefits of such sharing across runs.

Sharing between viable and non-viable runs. Viable runs reach the final state and produce matches, whereas non-viable runs proceed for some time but eventually fail. Effective sharing between viable runs and non-viable runs allow storage and processing costs to be reduced from the total number of runs to the number of actual matches for a query. When most runs of a query are non-viable, the benefit of such sharing can be tremendous.

Sharing among viable runs. Sharing can further occur between runs that produce matches. If these runs process and store the same events, sharing can be applied in certain scenarios to reduce storage and processing costs to even less than what the viable runs require collectively. This is especially important when most runs are viable, rendering the number of matches close to the total number of runs.

Coping with output cost. The cost to output query matches is linear in the number of matches. If a query produces a large number of matches, the output cost is high even if we can detect these matches more efficiently using sharing. To cope with this issue, we support two output modes for applications to choose based on their uses of the matches and requirements of runtime efficiency. The verbose mode enumerates all matches and returns them separately. Hence, applications have to pay for the inherent cost of doing so. The compressed mode returns a set of matches (e.g., those ending with the same event) in a compact data structure, in particular, the data structure that we use to implement a shared match buffer for all runs. Once provided with a decompression algorithm, i.e., an algorithm to

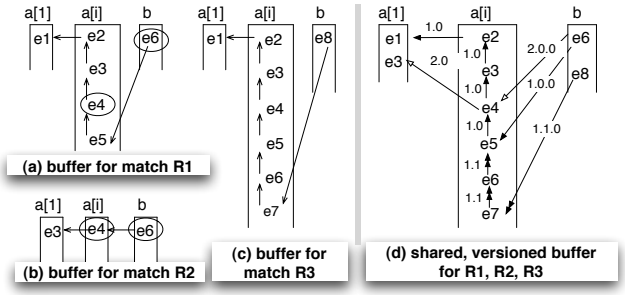


Figure 4: Creating a shared versioned buffer for Q3.

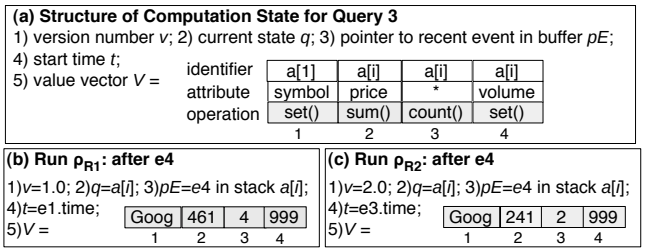


Figure 5: Computation state of runs for Q3.

retrieve matches from the compact data structure, applications such as a visualization tool have the flexibility to decide which matches to retrieve and when to retrieve them.

5. RUNTIME TECHNIQUES

Based on the insights gained from the previous analysis, we design runtime techniques that are suited to the new abstraction of NFA^b-based query plans. In particular, the principle that we apply to runtime optimization is to share both storage and processing across multiple runs in the NFA^b-based query evaluation.

5.1 A Shared Versioned Match Buffer

The first technique constructs a buffer with compact encoding of partial and complete matches for ***all*** runs. We first describe a buffer implementation for an individual run, and then present a technique to merge such buffers into a shared one for all the runs.

The individual buffers are depicted in Figure 4(a)-(c) for the three matches from Figure 3. Each buffer contains a series of stacks, one for each state except the final state. Each stack contains pointers to events (or events for brevity) that triggered begin or take moves from this state and thus were selected into the buffer. Further, each event has a predecessor pointer to the previously selected event in either the same stack or the previous stack. When an event is added to the buffer, its pointer is set. For any event that triggers a transition to the final state, a traversal in the buffer from that event along the predecessor pointers retrieves the complete match.

We next combine individual buffers into a single ***shared*** one to avoid the overhead of numerous stacks and replicated events in them. This process is based on merging the corresponding stacks of individual buffers, in particular, merging the same events in those stacks while preserving their predecessor pointers. Care should be taken in this process, however. If we blindly merge the events, a traversal in the shared buffer along all existing pointers can produce erroneous results. Suppose that we combine the buffers for R_1 and R_2 by merging e_4 in the $a[i]$ stack and e_6 in the b stack. A traversal from e_6 can produce a match consisting of e_1, e_2, e_3, e_4, and e_6, which is a wrong result. This issue arises when the merging process fails to distinguish pointers from different buffers.

To solve the problem, we devise a technique that creates a shared ***versioned*** buffer. It assigns a version number to each run and uses it to label all pointers created in this run. An issue is that runs do not have pre-assigned version numbers, as the non-determinism at any state can spawn new runs. In this technique, the version number is encoded as a dewey number that dynamically grows in the form of $id_1(.id_j)*$ $(1 \leq j \leq t)$, where t refers to the current state q_t. Intuitively, it means that this run comes from the id_1^{th} initiation from the start state, and the id_j^{th} instance of splitting at the state q_j from the run that arrived at the state, which we call an *ancestor run*. This technique also guarantees that the version number v of a run is *compatible* with v' of its ancestor run, in one of the forms: (i) v contains v' as a prefix, or (ii) v and v' only differ in the last digit id_t and id_t of v is greater than that of v'.

A shared versioned buffer that combines the three matches is shown in Figure 4(d). All pointers from an individual buffer now are labeled with compatible version numbers. The erroneous result mentioned above no longer occurs, because the pointer from e_6 to e_4 with the version number 2.0.0 is not compatible with the pointer from e_4 to e_3 (in the $a[i]$ stack) with the version number 1.0.

As can be seen, the versioned buffer offers compact encoding of all matches. In particular, the events and the pointers with compatible version numbers constitute a *versioned view that corresponds exactly to one match*. To return a match produced by a run, the retrieval algorithm takes the dewey number of the run and performs a traversal from the most recent event in the last stack along the compatible pointers. This process is as efficient as the retrieval of a match from an individual buffer.

5.2 NFA^b Execution with Multiple Runs

Each run of NFA^b proceeds in two phases. In the ***pattern matching*** phase, it makes transitions towards the final state and extends the buffer as events are selected. In the ***match construction*** phase, it retrieves a match produced by this run from the buffer, as described in the previous section. Our discussion in this section focuses on algorithms for efficient pattern matching.

5.2.1 Basic Algorithm

We first seek a solution to evaluate individual runs as efficiently as possible. Our solution is built on the notion of ***computation state*** of a run, which includes a minimum set of values necessary for future evaluation of edge formulas. Take Query 3. At the state $a[i]$, the evaluation of the take edge requires the value $avg(a[..i-1].price)$. The buffer can be used to compute such values from the contained events, but it may not always be efficient. We trade off a little space for performance by creating a small data structure to maintain the computation state separately from the buffer.

Figure 5(a) shows the structure of the computation state

for Query 3. It has five fields: 1) the version number of a run, 2) the current automaton state that the run is in, 3) a pointer to the most recent event selected into the buffer in this run, 4) the start time of the run, and 5) a vector V containing the values necessary for future edge evaluation. In particular, the vector V is defined by a set of columns, each capturing a value to be used as an instantiated variable in some formula evaluation.

Revisit the formulas in Figure 2. We extract the variables to be instantiated from the right operands of all formulas, and arrange them in V by the instantiation state, then the attribute, and finally the operation. For example, the 1^{st} column in the V vector in Figure 5(a) means that when we select an event for $a[1]$, store its symbol for later evaluation of the equivalence test. The 2^{nd} and 3^{rd} columns jointly compute the running aggregate $avg(a[..i-1].price)$: for each event selected for $a[i]$, the 2^{nd} column retrieves its price and updates the running sum, while the 3^{rd} column maintains the running count. The 4^{th} column stores the volume of the last selected $a[i]$ to evaluate the formula involving b.

For each run, a dynamic data structure is used to capture its current computation state. Figure 5(b) and 5(c) depict the computation state of two runs ρ_{R1} and ρ_{R2} of the NFA^b for Query 3. Their states shown correspond to R_1 and R_2 after reading the event e_4 in Figure 3.

When a new event arrives, each run performs a number of tasks. It first examines the edges from the current state by evaluating their formulas using the V vector and the start time of the run. The state can have multiple edges (e.g., take, ignore, and proceed edges at the state $a[i]$), and any subset of them can be evaluated to True. If none of the edge formulas is satisfied, the run fails and terminates right away; common cases of such termination are failures to meet the query-specified time window or contiguity requirements. If more than one edge formula is satisfied, the run splits by cloning one or two child runs. Then each resulting run (either the old run or a newly cloned run) takes its corresponding move, selects the current event into the buffer if it took a take or begin move, and updates its computation state accordingly.

Finally, we improve the basic algorithm when the non-overlap output format described in §2 is used. Recall that this format outputs only one match among those that belong to the same partition and overlap in time. Since we do not know a priori which run among the active ones for a particular partition will produce a match first, we evaluate all the runs in parallel as before. When a match is actually produced for a partition, we simply prune all other runs for the same partition from the system.

5.2.2 Merging Equivalent Runs

To improve the basic algorithm that evaluates runs separately, we propose to identify runs that overlap in processing and merge them to avoid repeated work. The idea again stems from an observation of the computation state. If two runs, despite their distinct history, have the same computation state at present, they will select the same set of events until completion. In this case, we consider these two runs ***equivalent***. Figure 6 shows an example, where Query 3 is modified by replacing the running aggregate $avg()$ with $max()$. The structure of its computation state is modified accordingly as shown in Part (b). The column in bold is the new column for the running aggregate $max()$ on $a[i]$. Parts

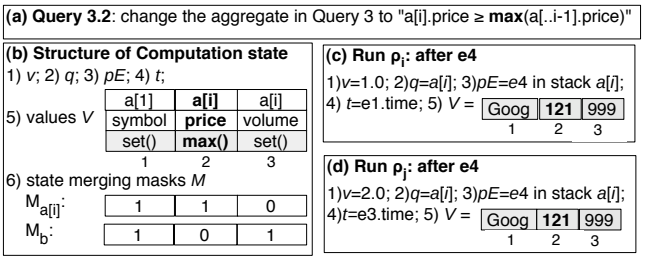


Figure 6: An example for merging runs.

(c) and (d) show two runs after reading the event e_4 from the stream in Figure 3: they are both at the state $a[i]$ and have identical values in V. Their processing of all future events will be the same and thus can be merged.

The merging algorithm is sketched as follows. The first task is to detect when two runs become equivalent, which can occur at any state q_t after the start state. The requirement of identical V vectors is too stringent, since some values in V were used at the previous states and are no longer needed. In other words, only the values for the evaluation at q_t and its subsequent states need to be the same. To do so, we introduce an extra static field M, shown in Figure 6(b), that contains a set of bit masks over V. There is one mask for each state q_t, and the mask has the bit on for each value in V that is relevant to the evaluation at this state. At runtime, at the state q_t we can obtain all values relevant to future evaluation, denoted by $V_{[t...]}$, by applying the mask $(M_{q_t} \vee M_{q_{t+1}} \vee \ldots)$ to V. Two runs can be merged at q_t if their $V_{[t...]}$ vectors are identical.

Another task is the creation of a combined run, whose computation state will be extended with all the version numbers and start times of the merged runs. The version numbers of the merged runs are cached so that later in the match construction phase, we can identify the compatible predecessor pointers for these runs in the shared buffer and retrieve their matches correctly. We also need to keep the start times of the merged runs to deal with expiration of runs. Recall that a run expires when it fails to meet the query-specified time window. Since the merged runs may have different start times, they can expire at different times in execution. To allow the combined run to proceed as far as possible, we set the start time of the combined run as that of the youngest merged one, i.e., the one with the highest start time. This ensures that when the combined run expires, all its contained runs expire as well. Finally, when the combine run reaches the final state, match construction is invoked only for the contained runs that have not expired.

5.2.3 Backtrack Algorithm

For purposes of comparison, we developed a third algorithm called the backtrack algorithm for evaluating pattern queries. This algorithm was inspired by a standard implementation for pattern matching over strings and its adaptation in [26] as a basic execution model for event pattern matching. The basic idea is that we process a single run per partition at a time, which we call the ***singleton*** run for the partition. The singleton run continues until either it produces a match or fails, while the evaluation of any runs created during its processing, e.g., as a result of nondeterminism, is postponed. If the singleton run fails, then we backtrack and process another run whose evaluation was

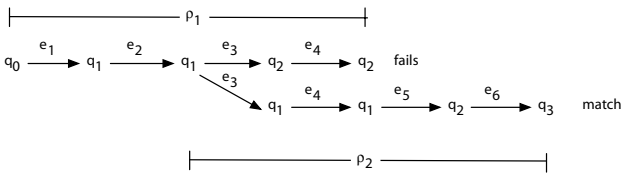


Figure 7: An example for the Backtrack algorithm.

previously postponed for the partition. If the singleton run produces a match, we may backtrack depending on the output format: we backtrack if all results are required; we do not if only non-overlapping results are needed.[5]

We adapted the implementation of our basic algorithm described in §5.2.1 to implement the backtrack algorithm. We highlight the changes through the example given in Figure 7. In this example, ρ_i represents run i, q_j state j, and e_k an event that occurs at time k. We describe how the backtrack algorithm evaluates the event stream $e_1, e_2, e_3, e_4, e_5, e_6$ for a generic query with a single Kleene plus component:

- e_1 creates a new run, ρ_1, at the start state, q_0. ρ_1 becomes the singleton run.
- e_3 results in a nondeterministic move at q_1. We create run ρ_2 and add it together with the id of its current state (q_1) and the id of the current event (e_3) to a stack holding all postponed runs. ρ_1 remains as the singleton run because it is proceeding to the next NFA^b state.
- Process ρ_1 until it fails with event e_4 at state q_2.
- Backtrack by popping the most recently created run, ρ_2 in this example, from the stack. Resume processing ρ_2 (the new singleton run) at state id q_1 by reading events in the buffer starting from e_3.
- ρ_2 produces a match with e_6.

If we view the creation of runs as a tree that expands during event processing, the backtrack algorithm processes runs in a depth first search manner. That is, we process the singleton run until it either fails or produces a result and then we backtrack to the most recent run that was created during the processing of the singleton run. Our basic algorithm, on the other hand, expands the "run tree" in a breadth first search manner; it creates and evaluates all runs at once.

5.3 Memory Management

There are a number of data structures that grow in proportion to the size of the input event stream. Since the input event stream is infinite, consistent performance over time can only be achieved by actively maintaining these data structures. To this end, we prune data structures incrementally and reuse expired data structures whenever possible.

There are two key data structures that we actively prune using the time window during runtime. One is the shared match buffer. After each event is processed, we use the timestamp of this event and the time window to determine the largest timestamp that falls outside the window, called the pruning timestamp. We use the pruning timestamp as a key to perform a binary search in each stack of the match buffer. The binary search determines the position of the most recent event that falls outside the window. We prune the events (more precisely, container objects for those events) at and before this position from the stack. Similarly, we prune events from a global event queue in the system using the pruning timestamp.

To further optimize memory usage, we reuse frequently instantiated data structures. As objects are purged from the match buffer, we add them to a pool. When a new stack object is requested, we first try to use any available objects in the pool and only create a new object instance when the pool is empty. Recycling stack objects as such limits the number of object instantiations and quiesces garbage collection activity. Similarly, we maintain a pool for NFA^b run objects, i.e., the dynamic data structures that maintain the computation state of runs. Whenever an NFA^b run completes or fails, we add it to a pool to facilitate reuse.

6. PERFORMANCE EVALUATION

We have implemented all the query evaluation techniques described in the previous sections in a Java-based prototype system containing about 25,000 lines of source code. In this section, we present results of a detailed performance study using our prototype system. These results offer insights into the effects of various factors on performance and demonstrate the significant benefits of sharing.

To test our system, we implemented an event generator that dynamically creates time series data. We simulated stock ticker streams in the following experiments. In each stream, all events have the same type, STOCK, that contains three attributes, symbol, price, volume, with respective value ranges [1-2], [1-1000], [1-1000]. The price of those events has the probability p for increasing, $\frac{1-p}{2}$ for decreasing, and $\frac{1-p}{2}$ for staying the same. The values of p used in our experiments are shown in Table 1. The symbol and volume follow the uniform distribution.[6] We only considered two symbols; adding more symbols does not change the cost of processing each event (on which our measure was based) because an event can belong to only one symbol.

Table 1: Workload Parameters

Parameter	Values used
$Prob_{price\_increase}$	0.7, 0.55
ES, event selection strategy	(s_2) partition-contiguity (s_3) skip-till-next-match
$P_{a[i]}$, iterator predicate used in Kleene closure	(p_1) True; (p_2) $a[i].price > a[i-1].price$; ($p_3$) $a[i].price > aggr(a[..i\text{-}1].price)$ $aggr = max \mid min \mid avg$
W, partition window size	500-2000 events
Result output format	all results (default), non-overlapping results

Queries were generated from a template " PATTERN(STOCK+ $a[\,]$, STOCK b) WHERE ES $\{[symbol] \wedge a[1].price$ %500==0 $\wedge$ $P_{a[i]}$ $\wedge$ $b.volume$ <150$\}$ WITHIN W", whose parameters are explained in Table 1. For event selection strategy, we considered partition contiguity (s_2) and skip till next match (s_3) because they are natural choices for the domain of stock tickers. The iterator predicate used in Kleene closure, $P_{a[i]}$, was varied among three forms as listed in Table 1. Note that

[5]Regular expression matching in network intrusion detection systems (NIDS) [19, 35] is also relevant to event pattern matching. However, we did not choose to compare to NIDS because regular expressions can express only a subset of event queries, as stated in §3.3, and most NIDS use deterministic finite automata (DFA) that would explode to an exponential size when handling non-determinism [35], which abound in event queries.

[6]The distributions for price and volume are based on our observations of *daily* stock tickers from Google Finance, which we use to characterize *transactional* stock tickers in our simulation.

take-proceed non-determinism naturally exists in all queries: for some event e, we can both take it at the state $a[i]$ based on the predicate on price, and proceed to the next state based on the predicate on volume. The partition window size W (defined in §4.2) was used to bound the number of events in each partition that are needed in query processing.

The performance metric is throughput, i.e., the number of events processed per second. In all experiments, throughput was computed using a long stream that for each symbol, contains events of size 200 times the partition window size W. All measurements were obtained on a workstation with a Pentium 4 2.8 Ghz CPU and 1.0 GB memory running Java Hotspot VM 1.5 on Linux 2.6.9. The JVM allocation pool was set to 750MB.

6.1 Effects of Various Factors

To understand various factors on performance, we first ran experiments using the shared match buffer (§5.1) and the basic algorithm that handles runs separately (§5.2.1). In these experiments, the probability of price increase in the stock event stream is 0.7.

Expt 1: varying iterator predicate and event selection strategy ($ES \in (s_2, s_3)$, $P_{a[i]} \in (p_1, p_2, p_3)$, W=500). In this experiment, we study the behavior of Kleene closure given a particular combination of the iterator predicate (p_1, p_2, or p_3) and event selection strategy (s_2 or s_3). For stock tickers with an overall trend of price increase, p_3 using the aggregate function *max* performs similarly to p_2, and p_3 using *avg* is similar to p_3 using *min*. Hence, the discussion of p_3 below focuses on its use of *min*.

Figure 8(a) shows the throughput measurements. The X-axis shows the query types sorted first by the type of predicate and then by the event selection strategy. The Y-axis is on a logarithmic scale. These queries exhibit different behaviors, which we explain using the profiling results shown in the first two rows of Table 2.

Table 2: Profiling Results for Expt1

	p_1s_2	p_1s_3	p_2s_2	p_2s_3	p_3s_2	p_3s_3
match length	250	250	4.5	140	250	250
num. runs/time step	2	2	0.01	2	2	2
matching cost(%)	61	61	100	67	67	66
construction cost(%)	39	39	0	33	33	34

For the predicate p_1 which is set to True, s_2 and s_3 perform the same because they both select every event in a partition, producing matches of average length 250. Simultaneous runs exist due to multiple instances of initiation from the start state and take-proceed non-determinism, yielding an average of 2 runs per time step (we call the cycle of processing each event a time step).

For p_2 that requires the price to strictly increase, s_2 and s_3 differ by an order of magnitude in throughput. Since p_2 is selective, s_2 tends to produce very short matches, e.g., of average length 4.5, and a small number of runs, e.g., 0.01 run per time step. In contrast, the ability to skip irrelevant events makes s_3 produce longer matches, e.g., of average length 140. Furthermore, s_3 still produces 2 runs per time step: due to the *ignore-proceed* nondeterminism that s_3 allows (but s_2 does not), a more selective predicate only changes some runs from the case of *take-proceed* nondeterminism to the case of *ignore-proceed*.

Finally, p_3 requires the price of the next event to be greater than the minimum of the previously selected events. This predicate has poor selectivity and leads to many long matches as p_1. As a result, the throughput was close to that of p_1 and the difference between s_2 and s_3 is very small.

In summary, selectivity of iterator predicates has a great effect on the number of active runs and length of query matches, hence the overall throughput. When predicates are selective, relaxing s_2 to s_3 can incur a significant additional processing cost.

We also obtained a cost breakdown of each query into the pattern matching and pattern construction components, as shown in the last two rows of Table 2. As can be seen, pattern matching is the dominant cost in these workloads, covering 60% to 100% of the total cost. Reducing the matching cost is our goal of further optimization.

Expt 2: varying partition window size ($ES \in (s_2, s_3)$, $P_{a[i]} \in (p_1, p_2, p_3)$). The previous discussion was based on a fixed partition window size W. We next study the effect of W by varying it from 500 to 2000. The results are shown in Figure 8(b). We omitted the result for p_1s_2 in the rest of the experiments as it is the same as p_1s_3.

The effect of W is small when a selective predicate is used and the event selection strategy is s_2, e.g., p_2s_2. However, the effect of W is tremendous if the predicates are not selective, e.g., p_1 and p_3, and the event selection strategy is relaxed to s_3. In particular, the throughput of p_1s_3 and p_3s_3 decreases quadratically. Our profiling results confirm that in these cases, both the number of runs and the length of each match increase linearly, yielding the quadratic effect.

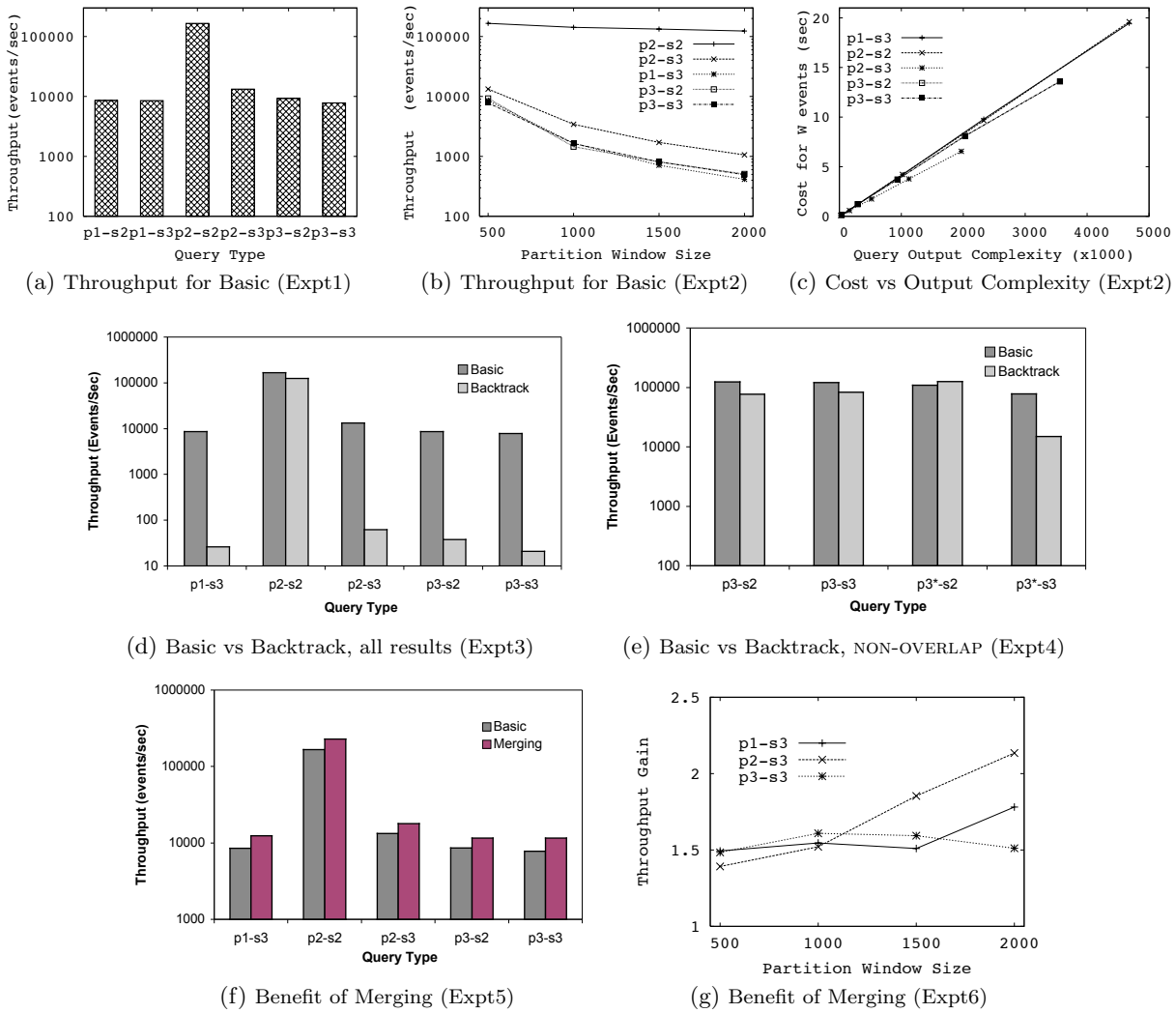
We further explore the efficiency of our algorithm by taking into account the effect of W on the *query output complexity*, defined as $\sum_{\text{each match}}$(length of the match). It serves as an indicator of the amount of computation needed for a query. Any efficient algorithm should have a cost linear in it. Figure 8(c) plots the processing cost against the output complexity for each query, computed as W was varied. It shows that our algorithm indeed scales linearly. The constants of different curves vary naturally with queries. The effect of further optimization will be to reduce the constants.

6.2 Basic versus Backtrack Algorithms

Recall from §5.2 that our basic algorithm evaluates all runs simultaneously when receiving each event. In contrast, the backtrack algorithm, popular in pattern matching over strings and adapted in [26] for event pattern matching, evaluates one run at a time and backtracks to evaluate other runs when necessary. We next compare these two algorithms.

Expt3: all results. In this experiment we compare the two algorithms using the previous queries and report the results in Figure 8(d). These results show that the throughput of our basic algorithm is 200 to 300 times higher than the backtrack algorithm across all queries except for p_2s_2, where the basic algorithm achieves a factor of 1.3 over the backtrack algorithm.

The performance of the backtrack algorithm is largely attributed to repeated backtracking to execute all the runs and produce all the results. The throughput results can be explained using the average number of times that an event is reprocessed. The backtrack algorithm reprocesses many events, e.g., an average 0.6 time for each event for queries using s_3, resulting in their poor performance. In contrast, our basic algorithm never reprocesses any event. The only case of backtrack where this number is low is p_2s_2 with short duration of runs, yielding comparable performance. As can



(a) Throughput for Basic (Expt1) (b) Throughput for Basic (Expt2) (c) Cost vs Output Complexity (Expt2)

(d) Basic vs Backtrack, all results (Expt3) (e) Basic vs Backtrack, NON-OVERLAP (Expt4)

(f) Benefit of Merging (Expt5) (g) Benefit of Merging (Expt6)

Figure 8: Experimental Results.

be seen, if all results are to be generated, our basic algorithm is a much better choice.

Expt4: non-overlapping results. We next compare these two algorithms when only non-overlapping results are required. The difference from the case of all results is that we may not need to examine all simultaneous runs to produce such results. However, it is unknown a prior which run among the active ones will produce a result first. In this experiment, we instrumented both algorithms to return shortest non-overlapping results.

We first reran all previous queries with NON-OVERLAP. These queries exhibit similar trends for the two algorithms, as illustrated using p_3s_2 and p_3s_3 in the 1st and 2nd groups of bars in Figure 8(e). We observed that this set of queries are ideal for the backtrack algorithm since little backtracking is needed. This is because the last predicate *b.volume* <150 is not selective given uniform distribution of volume in events. Hence, once a run starts, it is likely to produce a match during a short period of time, eliminating the need to backtrack. Our basic algorithm runs fast for a similar reason: even though it executes multiple runs at the same time, the quick generation of a result allows it to prune other runs, hence reducing its overhead of executing multiple runs.

We next modified the query workload so that a random run is less likely to produce a result—requiring each algorithm to search through runs to generate the result. To do so, we extended the two queries using predicate p_3 with another pattern component: PATTERN(STOCK+ a[ ], STOCK b, STOCK c), where $c.price > a[1].price \wedge c.price < a[a.\text{LEN}].price$. We denote the modified queries using p_3^* and show its results using the 3rd and 4th groups of bars in Figure 8(e). In particular, the basic algorithm yields an order of magnitude higher throughput than the backtrack algorithm for $p_3^*s_3$. In this workload, c's predicates are selective so most runs fail at this state, causing the backtrack algorithm to repeatedly invoke backtracking and try other runs.

6.3 NFA^b with Merging of Runs

We showed in the above experiments that our basic algorithm provides comparable or better performance than the backtrack algorithm in most workloads tested. In the following experiments, we omit the backtrack algorithm and extend the basic algorithm with merging of equivalent runs.

Expt 5: merging runs of Expt 1. In this experiment, we applied the merging algorithm to the queries in Expt 1.

Figure 8(f) shows the results; it also includes the numbers of the basic algorithm from Figure 8(a) for comparison. As can be seen, the merging algorithm yields significant performance gains over the basic algorithm for all the queries tested, e.g., achieving a factor of 1.5 for p_1s_3, 1.4 for p_2s_3, and 1.5 for p_3s_3. It ran all of the expensive queries, which either use less selective predicates such as p_1 and p_3 or the event selection strategy s_3, at over 10,000 events per second.

Expt 6: merging runs of Expt 2. We further applied merging to all the queries in Expt 2 when the partition window size W was varied. We focused on the three queries using s_3, namely, p_1s_3, p_2s_3, and p_3s_3, because they are more expensive than (if not the same as) their counterparts using s_2. Recall that the partition window size affects query output complexity; a larger size leads to more query matches and longer matches, hence a higher sequence construction cost. Since sequence construction is common to both the basic and merging algorithms, to better understand their differences we turned off sequence construction in this experiment and measured only the cost of sequence matching.

Figure 8(g) shows the benefits of merging in *throughput gain*, defined as new_throughput/old_throughput. As can be seen, merging offers remarkable overall throughput gains, ranging from a factor of 1.4 to 2.1. Results of the three queries can be further explained using two factors shown in Table 3: the sharing opportunity, captured by the percentage of runs that were successfully merged in columns 2 to 4 of the table, and the overhead of maintaining the match buffer, captured by the total cost of buffer updates in columns 5 to 7.

These three queries represent interesting combinations of these two factors. The predicate p_1 ($= true$) allows the most sharing opportunity: all of the overlapping runs for the same partition can be merged. However, p_1 leads to long query matches, hence a high overhead of maintaining the match buffer as events are selected. The predicate for price strictly increasing, p_2, still allows significant sharing: as soon as two runs have the same price from the last selected events, they can be merged. In addition, p_2 is selective so the buffer update cost is relatively low. As a result, p_2 achieves higher throughput gains for large values of W. Finally, the predicate p_3 requires two runs to agree on the minimum price in order to be merged, offering a somewhat less sharing opportunity. Since it is not selective, it also has a high buffer update cost as p_1. Combining both factors, p_3 achieves only limited throughput gains for large values of W.

Table 3: Profiling Results for Expt6

	Runs merged (%)			Buffer update cost (sec)		
W	p_1s_3	p_2s_3	p_3s_3	p_1s_3	p_2s_3	p_3s_3
500	47.9	47.8	47.8	11.9	5.4	12.0
1000	50.9	48.3	45.8	175.8	62.3	205.4
1500	66.6	55.7	44.9	363.6	132.6	454.2
2000	75.3	60.4	50.6	670.3	221.3	840.4

Other experiments. To explore the effects of different data characteristics, we also varied the probability p for stock price increase in the event stream. We briefly summarize the results below. When a smaller value of p is used, e.g., $p = 0.55$, the queries using predicate p_1 have the same performance because they simply select every event in each partition. In comparison, the queries using predicates p_2 and p_3 produce fewer matches and hence all achieve higher throughput numbers. The advantage of our basic algorithm over backtrack still holds, but with a smaller margin. On the other hand, the difference in throughput between the merging and basic algorithms is even higher. This is because fewer matches mean a smaller cost of sequence construction, and the benefit of merging is magnified in the presence of a low sequence construction cost common to both algorithms.

7. RELATED WORK

Much related work has been covered in previous sections. We discuss broader areas of related work below.

Event languages for active databases [14, 13, 6, 23, 22, 36] offer temporal operators including sequencing and Kleene closure, but do not support complex predicates to compare events. As we showed in this paper, such predicates are crucial in pattern definition. Those languages also lack efficient implementation over high-volume streams.

Traditional pub/sub systems [2, 12] offer predicate-based filtering of individual events. Our system significantly extends them with the ability to match complex patterns across multiple events. Cayuga [10, 11] supports patterns with Kleene closure and event selection strategies including partition contiguity and skip till next match, but does not allow output of complete matches. In comparison, our system supports more event selection strategies and output of complete matches both in the evaluation model and in runtime optimization. The Cayuga implementation focuses on multi-query optimization, which is directly applicable when our system is extended to handle multiple queries.

Sequence databases [27, 26] offer SQL extensions for sequence data processing. SEQUIN [27] uses joins to specify sequence operations and thus cannot express Kleene closure. SQL-TS [26] adds new constructs SQL to handle Kleene closure, but restricts pattern matching to only contiguous tuples in each relevant partition. Its optimization based on predicate containment can be integrated into our system for workloads that exhibit such containment relationships.

Many recent event systems [16, 25, 31, 34] offer relatively simple event languages and stream-based processing. These systems lack important constructs for pattern matching such as Kleene closure and choices of event selection strategies. In particular, our work significantly extends prior work [34] with Kleene closure and event selection strategies, two features that fundamentally complicate event pattern matching, a formal rich model NFA^b for evaluation and theoretical analysis, and a suite of sharing techniques. The evaluation framework of [34] is further extended to handle out-of-order events [20]. SASE+ [15] provides a rich language for pattern matching but lacks implementation details.

There have also been theoretical studies on the underlying model of complex event processing. CEDR [5] proposes a new temporal model that captures the duration of events and analyzes the consistency of event processing for out-of-order streams. A related study [33] designs a temporal model for such events that has a bounded representation of timestamps and offers associativity of the sequencing operator. These results can be applied to guide the extension of our system to handle events with duration.

8. CONCLUSIONS

In this paper, we studied the evaluation and optimization of pattern queries over event streams. We rigorously defined a query evaluation model, the NFA^b automata, analyzed

its expressibility, and provided compilation techniques for building query plans based on this model. We also analyzed the runtime complexity and developed sharing techniques for efficient runtime evaluation. Our system could process tens of thousands of events per second for fairly expensive queries and offers much higher throughput for cheaper queries. Our sharing techniques also produced remarkable performance benefits, ranging from 40% to 110%, over a spectrum of query workloads.

We plan to continue our research in a few directions. We will extend our system to handle out-of-order events by augmenting the NFA^b model with techniques including invalidation and re-computation: The NFA^b proceeds as before and, when receiving an out-of-order event, invalidates some of the existing runs affected by the event and recomputes from valid intermediate steps. In addition, our current implementation of queries with negation and composition is strictly based on the query semantics. We will explore new opportunities for optimization that are particularly suitable for these queries. Finally, we will study robust pattern matching over uncertain events that are produced from a variety of sensor networks.

9. REPEATABILITY ASSESSMENT RESULT

Figures 8(a), 8(b), 8(d), 8(e) have been verified by the SIGMOD repeatability committee.

10. REFERENCES

[1] J. Agrawal, Y. Diao, et al. Efficient pattern matching over event streams. Technical Report 07-63, University of Massachusetts Amherst, 2007.

[2] M. K. Aguilera, R. E. Strom, et al. Matching events in a content-based subscription system. In *PODC*, 53–61, 1999.

[3] A. Arasu, S. Babu, et al. CQL: A language for continuous queries over streams and relations. In *DBPL*, 1–19, 2003.

[4] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD*, 16–52. 1986.

[5] R. S. Barga, J. Goldstein, et al. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 363–374, 2007.

[6] S. Chakravarthy, V. Krishnaprasad, et al. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, 606–617, 1994.

[7] S. Chandrasekaran, O. Cooper, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[8] M. Cherniack, H. Balakrishnan, et al. Scalable distributed stream processing. In *CIDR*, 2003.

[9] Coral8. `http://www.coral8.com/`.

[10] A. J. Demers, J. Gehrke, et al. Towards expressive publish/subscribe systems. In *EDBT*, 627–644, 2006.

[11] A. J. Demers, J. Gehrke, et al. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.

[12] F. Fabret, H.-A. Jacobsen, et al. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD*, 115–126, 2001.

[13] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Rules in Database Systems*, 23–39, 1993.

[14] N. H. Gehani, H. V. Jagadish, et al. Composite event specification in active databases: Model & implementation. In *VLDB*, 327–338, 1992.

[15] D. Gyllstrom, J. Agrawal, et al. On supporting kleene closure over event streams. In *ICDE*, 2008. Poster.

[16] L. Harada and Y. Hotta. Order checking in a CPOE using event analyzer. In *CIKM*, 549–555, 2005.

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2006.

[18] N. Immerman. *Descriptive Complexity*. Graduate Texts in Computer Science. Springer, New York, 1999.

[19] S. Kumar, B. Chandrasekaran, et al. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS*, 155–164, 2007.

[20] M. Li, M. Liu, et al. Event stream processing with out-of-order data arrival. In *Int'l Conf. on Distributed Computing Systems Workshops*, page 67, 2007.

[21] L. Libkin and L. Wong. Unary quantifiers, transitive closure, and relations of large degree. In *Proc. of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STAC)*, 183–193, 1998.

[22] D. F. Lieuwen, N. H. Gehani, et al. The Ode active database: Trigger semantics and implementation. In S. Y. W. Su, editor, *ICDE*, 412–420, 1996.

[23] R. Meo, G. Psaila, et al. Composite events in chimera. In *EDBT*, 56–76, 1996.

[24] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[25] S. Rizvi, S. R. Jeffery, et al. Events on the edge. In *SIGMOD*, 885–887, 2005.

[26] R. Sadri, C. Zaniolo, et al. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.

[27] P. Seshadri, M. Livny, et al. The design and implementation of a sequence database system. In *VLDB*, 99–110, 1996.

[28] Pattern matching in sequences of rows. SQL change proposal. `http://asktom.oracle.com/tkyte/row-pattern-recogniton-11-public.pdf`. 2007.

[29] StreamBase. `http://www.streambase.com/`.

[30] Truviso. `http://www.truviso.com/`.

[31] F. Wang and P. Liu. Temporal management of RFID data. In *VLDB*, 1128–1139, 2005.

[32] I. Wegener. *Branching programs and binary decision diagrams: theory and applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[33] W. M. White, M. Riedewald, et al. What is "next" in event processing? In *PODS*, 263–272, 2007.

[34] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, 407–418, 2006.

[35] F. Yu, Z. Chen, et al. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS*, 93–102, 2006.

[36] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *ICDE*, 392–399, 1999.