Approximating the Temporal Order of Events in Big Data Streams

Baltus, Nora

Award date:
2021

Department of Mathematics and Computer Science
Database Systems Group

Approximating the Temporal Order of Events in Big Data Streams

Nora Baltus

Assessment Committee:

dr. Odysseas Papapetrou (Supervisor)

dr. Renata Medeiros de Carvalho

dr. Nikolay Yakovets

July 20, 2021

Abstract

In the era of big data, it is becoming increasingly important to analyze massive data streams of events for patterns, using pattern matching queries. Applications such as network intrusion detection systems utilize these queries to signal an attack. Pattern matching queries rely on the underlying temporal order of events to detect when events occur in sequence. Solutions that provide an exact answer to these queries have to store the full data stream to support arbitrary queries over historical data. However, storing the full stream while maintaining low query times can be challenging given the data's large volume and velocity.

This thesis looks into the possibilities of approximating an answer to pattern matching queries without storing the complete dataset to provide faster query times than are currently possible using exact methods and provide support for arbitrary ad-hoc queries that are not initially known. To this end we aim to approximate the underlying temporal order of events, which we then use to answer such queries. We propose a solution that splits the stream into small windows of events, that each hash to a different Bloom filter. The resulting Bloom filters are maintained in a tree-based structure, where Bloom filters in nodes higher up in the tree result from merging the Bloom filters of the node's children. The temporal order of events can then be inferred from the positions of the Bloom filters the events are hashed to. In this report, we describe the construction of this data structure and show how it can be used to answer different types of pattern matching queries over data streams. Experiments show that our approximate solution can indeed answer pattern matching queries faster than an already efficient solution that provides an exact answer.

Contents

Chapter 1

Introduction

1.1 Motivation

In the era of big data, increasingly many applications continuously generate massive streams of data. As such, there is a growing need to efficiently analyze the data streams for patterns. One example of such an application is that of a computer network, where intrusion detection systems constantly monitor streams of network traffic for patterns that could signal an attack, known as attack signatures. These patterns rely on the underlying concept of temporality between events to detect when events occur in sequence. Intrusion detection is only one example application from a broader field that analyzes streams for patterns. Two main categories exist within this field [1]. The first is data stream mining, which assumes the patterns are not known a priori. Data stream mining aims to detect interesting patterns based on a set of requirements, such as occurrence frequency or similarity between events [2]. The second category is Complex Event Processing (CEP) which, contrary to data mining, assumes the patterns of interest are known and provided as a query to which the system returns all matches [3].

Given the large volume and velocity of the data, it is challenging to analyze large data streams using traditional methods, such as databases, as streams cannot be stored in their entirety efficiently, while maintaining fast response times. To counter this problem, approximate stream analysis solutions exist. These solutions aim to summarize the stream using special data structures, called synopses or sketches, that approximate the answer to questions that would otherwise require access to all events in the stream [4]. A common characteristic of approximate stream analysis techniques is that they provide a guarantee on the possible error size, where the trade-off is generally between storage space and the accu-

racy of the approximation. Many approximate stream analysis solutions solve problems common to data stream mining, such as determining the number of frequent or distinct items in a stream [4]. However, there is a lack of approximate data structures developed specifically for Complex Event Processing.

The lack of approximate solutions for Complex Event Processing likely stems from the fact that CEP generally requires the extraction of sequences that match a pattern such that they can be processed further elsewhere [5]. Since approximate data structures cannot store all events, an exact reconstruction of sequences of events that match a pattern would be impossible. However, systems that return all matches may be slow as they require access to the entire data stream. Thus, approximate solutions could provide results to pattern matching queries much faster. In other cases, the system's primary goal is to take immediate action when a match occurs, as in the case of intrusion detection, which could issue a warning when a match is detected. The persistence of the whole stream may not be feasible in these cases, even though these systems could also benefit from finding pattern matches in historical data. For example, in the case of intrusion detection, it could be interesting to detect whether a newly emerged attack signature, that was not known at the start of the stream, has occurred in the past and at which points in time. To this end, we propose a data structure that approximates the temporal order of events in a stream, and we show how this data structure can approximate answers to queries that are common to Complex Event Processing.

1.2 Problem Statement

Pattern matching relies on the underlying temporal order of events, which generally requires access to the complete set of events in the stream, along with an indicator of their ordering, such as a timestamp or the event's position within the stream. The primary goal of this thesis is to answer such pattern matching queries without having to store the entire dataset, thereby providing an approximate answer to order-based queries. Such a solution could benefit applications that cannot store the complete dataset yet would like to detect arbitrary pattern matches over previous data. Furthermore, it could also provide faster response times than exact solutions, as a small summary of the data is queried rather than the complete dataset.

There are several challenges to solving this problem. The first challenge is to develop an appropriate data structure from which we can infer the order of events in the underlying dataset, in small space. The second challenge is to bound

the probability that the solution returns an incorrect answer, such that it cannot exceed a value that the user finds acceptable. Finally, the proposed solution must be fast to construct, update, and query to ensure that it is usable in a streaming setting.

1.3 Contributions

The contributions of this thesis are summarized as follows.

- We construct a sketch from which we can approximate the temporal order of different events, through which we can answer pattern matching queries over data streams.
- We provide a set of query types supported by the data structure and demonstrate how to evaluate these types of queries in the most efficient manner.
- We provide a theoretical analysis of the data structure, which includes an approach for determining the appropriate configuration based on a desired false-positive rate and memory constraints.
- We compare the speed of our approximate solution to that of a solution that provides exact answers to pattern matching queries.
- We analyze the performance of our solution further using synthetic datasets.

1.4 Outline

The rest of the report is organized as follows. Chapter 2 defines concepts used in the report and highlights any assumptions that may differ from the standard definitions. Chapter 3 provides an overview of work that has been done in the area of answering pattern matching queries over data streams and attempts that have been made to define sketches with a temporal component. Chapter 4 describes the solution and the query types it supports. It further depicts how different queries are evaluated using the data structure, and provides a theoretical analysis of the solution's false-positive rate and its running time complexity. Chapter 5 shows how the solution compares to an exact Complex Event Processing solution in terms of speed and shows the results of further experiments using synthetic datasets. Finally, Chapter 6 summarizes the results of this thesis and discusses the solution's strengths and shortcomings, as well as possibilities for future work.

Chapter 2

Preliminaries

This chapter defines concepts used in the rest of the report and describes how our underlying assumptions may vary from the standard definitions. We start by describing big data streams along with characteristics of synopses and sketches that are used to summarize them. In particular, we focus on Bloom filters, which form the basis of the solution proposed in this report. Finally, we look at concepts from Complex Event Processing, as this forms the context within which the solution must work, and provide an overview of finite automata which many CEP solutions use to detect patterns.

2.1 Big Data Streams

Big data streams result from the continuous generation of data that is common in fields such as network monitoring, financial services, and web applications. For this application, we assume that a data stream is a sequence of events, $< e_1, e_2, \ldots e_N >$, where the events are homogeneous and represent only a single data type, such as IP addresses, sensor readings, or actions within a business process. Furthermore, we assume that the moment at which the system processes an event defines its temporal position. Therefore, our definition is a simplification, as the order in which events arrive may not necessarily correspond to the order in which they were generated.

When querying big data streams, the queries can be either continuous or ad-hoc [6, 7]. Continuous queries are often predefined queries that require continuous monitoring of the stream to respond to queries in real-time, as in intrusion detection systems. On the other hand, ad-hoc queries are generally not predefined and often require access to historical data, making these queries more complex to

answer [6]. The solution described in this report focuses on answering arbitrary ad-hoc queries that arrive while the stream is being processed. The solution is therefore not aimed at signaling the occurrence of predefined patterns in real-time.

Big data streams are often difficult to analyze using databases. Even when the entire stream can be stored, it often requires persistence to secondary storage, making it slow to query, as disk accesses are generally the biggest bottleneck in terms of speed [8]. In order to avoid having to persist all data, stream analysis solutions use synopses or sketches, which are special data structures that maintain a summary of the stream seen thus far [4]. These structures occupy less space than the original data and are often specific to a particular type of query. An example of a sketch is the Bloom filter [10], which is used to answer set-membership queries and is described in more detail in Section 2.2 below. However, as only a summary of the data is stored, these solutions often provide only approximate answers with a guarantee on the possible error instead of exact solutions that guarantee the answer is correct. Given that this summary has to be constructed and queried in real-time as the stream is processed, sketches should be fast to update and query. These two properties are therefore the most important points, apart from the accuracy of the returned results, that the solution proposed in this report should be evaluated on.

An interesting property in data analysis is that data often becomes less important as it ages. Therefore, many stream analysis solutions work with time-decay models that take into account only the most recent events, or assign weights to lower the importance of events that are considered outdated [7]. A popular time-decay method is the sliding window model [11], which only takes into account the current window of events. Such windows can be either (i) **time-based**, indicating that the elements in the window have all occurred within a given time interval, such as within the last hour, or (ii) **count-based**, indicating that only the W most recent events are included in the window. Figure 2.1 illustrates the concept of a sliding window, using a count-based window $W = 4$.

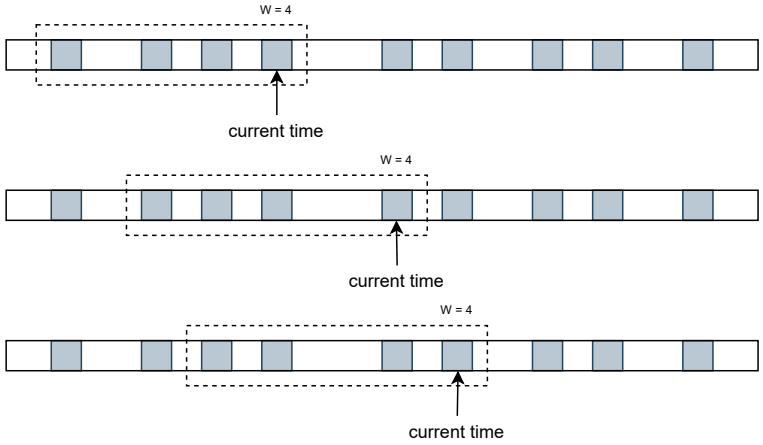


Figure 2.1: An example of a count-based sliding window with $W = 4$. In this figure, the colored squares indicate the events in the stream. The last item in the window is the latest event that has arrived.

2.2 Bloom Filters

A Bloom filter [10] is an approximate data structure that answers set-membership queries which determine whether an event has occurred in a set. A Bloom filter consists of an array of m bits along with k different and pairwise-independent hashing functions, that each map an element to a position in the array of bits. The bits in the bit array are initially all set to *0* and are set to *1* whenever an event hashes to that bit. Figure 2.2 illustrates this.

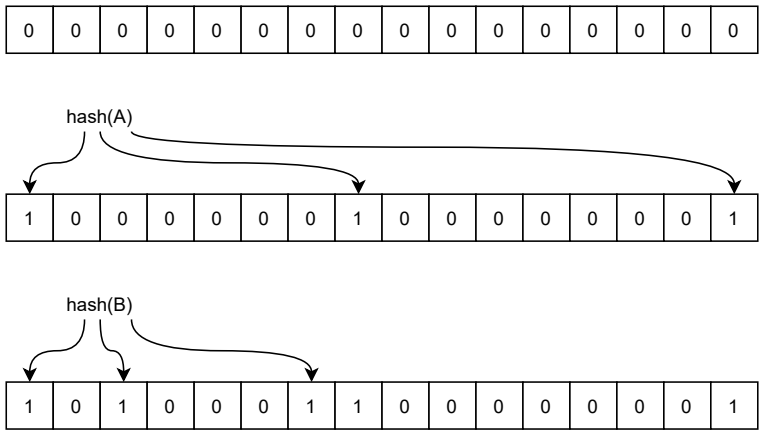


Figure 2.2: The insertion of two events in a Bloom filter, assuming $k = 3$ hashing functions.

Bloom filters are known for their small space requirements and fast update and query times. A Bloom filter can guarantee that an event has not occurred, but

it can only approximate whether an event has occurred. However, this approximation can have a high accuracy.

To check whether an event has occurred in the Bloom filter, the event is hashed using the same k hashing functions, and each resulting bit is checked to see whether it has been hashed to before (it is set to *1*) or not (it is set to *0*). If at any point a *0* is found, then the Bloom filter can guarantee that the event has not occurred. Hence, there are no false negatives. However, the occurrence of k bits set to *1* might indicate a false positive if other elements hashed to the same bits. It is for this reason that Bloom filters cannot support deletions, as the bits are not exclusive to any events. However, the probability that the Bloom filter returns a false positive can be controlled by modifying the number of bits (m) and hashing functions (k).

Note that every hashing function has a $\frac{1}{m}$ probability of setting a bit to *1*. Therefore, the probability that a bit has not been set to *1* by a hashing function is $(1 - \frac{1}{m})$. Since there are k hashing functions that hash n events, the probability that any bit is not set to *1* is $(1 - \frac{1}{m})^{kn}$. By taking the complement, we then get the probability that a bit is set *1*: $1 - (1 - \frac{1}{m})^{kn}$. The probability that k hashing functions all return *1* is given by the formula below, which defines a Bloom filter's false positive rate.

$$fp = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k \tag{2.1}$$

In order to obtain a desired false-positive rate fp for an expected number of events n, the appropriate number of bits m and hashing functions k can be calculated using the following formulas [12].

$$m \geq -\frac{n \ln fp}{(\ln 2)^2} \tag{2.2}$$

$$k = \ln 2 \cdot \frac{m}{n} \tag{2.3}$$

An interesting property of Bloom filters is that we can take their union and intersection, assuming they share the same k hashing functions and both have bitarrays of the same size m. Their union can be constructed by performing a bitwise OR on their bit arrays, thereby providing the union of their sets. Likewise, their intersection can be constructed using a bitwise AND, which results in the intersection of their sets [13]. In this work, we make extensive use of the ability to merge Bloom filters by taking their union.

As Bloom filters require k hashing functions that map an event independently and at random to guarantee the desired false-positive rate, we make use of MurmurHash3 [14], which is known for its speed. We generate k different values by providing k different seeds to the hashing function, and mapping each of the k results to one of the m bits.

2.3 Complex Event Processing

Complex Event Processing (CEP) is the processing of streams of events to detect patterns specified by the user. These patterns are primarily temporal in nature but can encompass additional predicates on attributes of events. The output of a CEP program is generally another stream that consists of the sequences of events that satisfied the pattern. Most queries are specified using a language that resembles SQL with support for an additional PATTERN-clause that allows the patterns to be specified using operators that are common to regular expressions [5]. Giatrakos et al. [3] identify several pattern matching operators used by CEP solutions. Depending on the solution, these operators may or may not be defined hierarchically. The following subset of operators is of interest to us.

- **Sequence** (A, B): When an event B temporally follows an event A.
- **Disjunction** $(A \vee B)$: When an event A or an event B occurs, independent of their temporal order.
- **Conjunction** $(A \wedge B)$: When both an event A and an event B occur, independent of their temporal order.
- **Negation** $(\neg A)$: When an event A does not occur.
- **Iteration** $(A*)$: When an event A can occur 0 or more times, in sequence.
- **Windowing**: When the events should occur within a specific time window, $[t_{start}, t_{end}]$, where t_{start} is the starting timestamp of the interval and t_{end} is the ending timestamp of the interval.

A common characteristic of Complex Event Processing is that events are not always expected to occur strictly in sequence (known as strict-contiguity) but are also allowed to be followed by any number of irrelevant events. Selection policies describe the different ways irrelevant events are allowed to be bypassed [3].

- **Strict-contiguity**: The matching events must follow each other strictly in sequence.
- **Skip-till-any**: Any number of irrelevant events may occur in-between the matching events, including any number of matching events themselves. For a sequence *SEQ(A,B)*, we can imagine that it contains a wildcard character between A and B that allows any event to be matched, including the events A and B.
- **Skip-till-next**: Any number of irrelevant events may occur in-between the matching events until the next matching event occurs. For a sequence *SEQ(A,B)*, any events may follow A until the first event B occurs.

Finally, reuse policies describe whether events can be part of multiple matches [3]. Events can either be reused, consumed, or reused a maximum number of times. In this work, we assume that all events can be reused.

As a demonstration of the different selection policies, we now determine the occurrences of a sequence *SEQ(A,B)* given the following example of a data stream. Here, A_i, B_i, etc. represent the events in the stream, and the index i is used only to clarify which events matched the query.

$$A_1, B_1, C_1, B_2, A_2, A_3, C_2, B_3, A_4, D_1, C_3, B_4, A_5, B_5, C_4, B_6$$

- ***SEQ(A,B)* under strict-contiguity:** $(A_1, B_1), (A_5, B_5)$
- ***SEQ(A,B)* under skip-till-any:** (A_1, B_1), (A_1, B_2), (A_1, B_3), (A_1, B_4), (A_1, B_5), (A_1, B_6), (A_2, B_3), (A_2, B_4), $(A_2, B_5) \ldots (A_5, B_6)$
- ***SEQ(A,B)* under skip-till-next:** (A_1, B_1), (A_2, B_3), (A_3, B_3), (A_4, B_4), (A_5, B_5)

2.4 Finite Automata

Many works in the field of Complex Event Processing rely on finite automata for the detection of patterns [3]. Finite automata are a key concept in the field of Computer Science, and are often used for pattern matching applications, ranging from the parsing of programs to searching for a string within a text [15]. A finite automaton can be defined as a tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states

- Σ is a finite set of input symbols
- δ is a transition function, $\delta : Q \times \Sigma \rightarrow Q$
- q_0 is the starting state
- and F is a set of final, accepting states.

Figure 2.3 provides an example of an automaton that can be used to detect the pattern *SEQ(A, A, B, C)* under the strict-contiguity selection policy. Here, the circles depict the states and the arrows depict the transitions that can be taken if a specific input symbol is read. The final states are depicted using double circles and the starting state can be recognized by an incoming transition that has no source. To match the sequence of events, we start at q_0 and take a transition that matches the current input symbol. When a final state is reached, it implies that a match has been found.

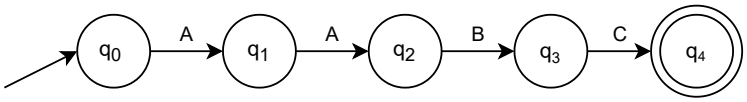


Figure 2.3: An example of a DFA that matches a sequence *SEQ(A,A,B,C)* under the strict-contiguity selection policy.

The previous figure is an example of a Deterministic Finite Automata (DFA), where each input symbol can be associated with at most one transition from any state. If from a state there exist multiple transitions for a given input symbol, then the automaton is said to be a Non-deterministic Finite Automaton (NFA). Therefore, in an NFA, multiple states can be active at a given time, whereas in a DFA at most one state can be active.

Even though finite automata are undoubtedly the most popular method for detecting patterns, they are known to be inefficient in the field of network intrusion detection, which is closely related to Complex Event Processing. As described earlier, intrusion detection systems analyze streams of network data for patterns that could signal an attack, and often use the same regular expression operators as Complex Event Processing to specify a pattern [16]. An example of such a system is the open-source Snort [17]. The inefficiency of finite automata stems from the fact that NFAs are slow, but have small space requirements, while DFAs are fast, but can suffer from a state-explosion problem. The state-explosion problem typically results from pattern matching queries that include wildcards, which allow any number of arbitrary events to be matched. This can lead to an exponential

growth of states in DFAs [16, 18]. Yu et al. [16] provide an overview of the space and processing complexity per character for each type of automaton. For a pattern query of length n, an NFA requires $O(n)$ storage space, while the processing complexity per character is $O(n^2)$, as all n states can be active at the same time in the worst case. For a DFA, the storage space requirement is $O(\Sigma^n)$ in the worst case, while the processing complexity per character is constant.

These problems that are inherent to automata provide an additional motivation to find a solution for answering pattern matching queries in data streams based on different techniques. In the next chapter we look at some of the works that utilize automata to answer Complex Event Processing queries, as well as alternative methods for detecting patterns in streams.

Chapter 3

Related Work

This chapter first provides an overview of related work that has been done in the area of answering pattern matching queries in data streams. It mainly consists of solutions that have been proposed in the context of Complex Event Processing. The chapter then describes work that has been done with regards to adding a temporal component to sketches, and finally it looks at sketches that have been proposed to answer different types of pattern matching problems.

3.1 Complex Event Processing

3.1.1 Automata-based Solutions

One of the most well-known solutions proposed for Complex Event Processing is SASE [5]. SASE was introduced to monitor high-volume streams, such as RFID data, in real-time. However, it can also support queries over historical data in combination with a relational database that stores the events [19]. Given a stream of events, SASE looks for matches to user-specified patterns and transforms matching sequences of events into a format more suitable for external monitoring applications. SASE assumes that the temporal order of events coincides with their order in the stream, which is similar to our assumption regarding the input stream of events. However, contrary to our assumption that events consist only of the event's name, SASE accepts events that come with a set of attributes as well. To query events, SASE introduces a user-friendly language that resembles SQL, which supports the detection of sequences of events that assume a temporal order, under all three selection policies. Events in a sequence can represent either the occurrence of an event or its non-occurrence, using a negation operator. It further supports predicates on attributes associated with an event and a

time window within which the pattern of events must have occurred. At the core of SASE are Non-deterministic Finite Automata (NFA), where successive events within a query translate to successive states within the NFA. These states contain self-loops with wildcards, so any number of event matches can occur before proceeding to a successive state. Furthermore, the transitions can contain predicates on attributes of events. Since SASE returns the sequences of events that matched a pattern, the authors use a stack alongside the automaton, which allows for the reconstruction of the original matching sequence. Negation is not handled directly within the NFA, but is instead done by finding the timestamps of the events that precede and follow the negated event, and checking whether the negated event occurred within that time interval. SASE+ [20] extends SASE by providing support for the Kleene-closure operator, which allows for a finite, but unbounded number of events to be matched. Given that SASE+ continuously monitors an infinite stream for patterns, it relies on termination criteria that indicate when the Kleene-closure operator should terminate. For skip-till-next queries, the Kleene-closure terminates when the next event in the query occurs. Other termination criteria include a maximum time window constraint and a constraint on a value of the last matching event. Since our solution is only aimed at detecting patterns in the stream processed up until the arrival of a query, there is no need to define explicit termination criteria.

Another popular CEP solution that makes use of NFAs is FlinkCEP [22], which is a Complex Event Processing library for Apache Flink [23], an open-source stream processor. FlinkCEP is similar to SASE [5] in that it generates an NFA for each query, with predicates on the transitions of the automaton to test for additional conditions. Unlike SASE, it allows the user to define the patterns using Java or Scala. It further provides support for all three selection policies and allows for patterns to be combined, thereby providing more flexibility than other CEP solutions.

As described in Chapter 2, finite automata are known to have performance issues when detecting patterns in network intrusion detection systems [16]. Zhang et al. [21] confirm these issues in the context of CEP. The performance issues are said to be caused by expensive selection policies such as skip-till-any and expensive operators such as the Kleene-closure. As stated by [16], Zhang et al. [21] also note that the runtime complexity of these solutions is given by the number of active states that the automaton can be in at any given time. When a state has multiple outgoing transitions that can be taken, then each transition initiates a new clone of the automaton, which in turn must also be cloned for any non-

deterministic transitions that occur in any of its states [21]. Solutions that are not based on finite automata could provide more flexibility in terms of answering these expensive queries.

3.1.2 Tree-based Solutions

An example of a tree-based solution for Complex Event Processing is ZStream [24]. The authors describe the following three limitations of NFA-based solutions that they wish to address using a tree-based solution.

1. NFA-based solutions have a fixed order of evaluation, determined by the sequence of states, which limits the number of evaluation options.

2. It is difficult to model negation using NFAs that have predicates on their transitions.

3. It is difficult to handle concurrent events, such as the conjunction of events, using NFAs.

ZStream transforms a query into a tree, where the internal nodes represent the operators and predicates. The leaves of the tree have buffers that store the events and the internal nodes have buffers that store the intermediate results of a (sub)query. These buffers are used to process the events in batches. At every step in the tree, the buffers are sorted according to increasing end-time, such that the system does not have to perform further time comparisons. The goal of the tree-based structure is to treat the evaluation of queries as that of a join-operator in a database context. To this end the authors transform the query into different plans such that the order in which the operators are evaluated changes, but the final result stays the same. ZStream then assigns a CPU cost to each operator and evaluates the query under the most cost-efficient plan. Using a flexible evaluation model, the authors address the first limitation of NFA-based solutions. The second limitation which states that NFAs cannot model negation easily is addressed by ZStream by pushing the negation operator down in the tree to filter unnecessary results early in the evaluation. Finally, the last limitation regarding the difficulty of evaluating conjunctions using NFAs can simply be handled using their structure by selecting all matching events from the input buffers without taking order of the events into account.

3.1.3 Other Solutions

One of the most recent works in the area of CEP is that by Mavroudopoulos et al. [25], who propose a system to detect and return arbitrary sequences in event log files. These files are assumed to consist of timestamped events grouped according to traces, which can represent a specific business process instance, for example. Contrary to the solutions presented above, this solution does not work in a streaming setting. Instead, it aims to detect arbitrary sequences in an offline-manner, but with support for the concatenation of new logs. Mavroudopoulos et al. support sequence detection under the following two selection policies: strict-contiguity and skip-till-next matching. Their solution does not provide support for pattern matching operators that are common to CEP, so queries they answer are considered to be a form of sequence detection rather than full-blown pattern matching. The solution proposed by Mavroudopoulos et al. consists of two components: a pre-processing component and a query processor. The pre-processing component is responsible for the construction of special indexes which are stored in a key-value database. The query processor then utilizes these indexes to provide a fast response to different types of queries. The most important index for sequence detection is the inverted index, which for each pair of events, assigns a list of triples that contain the trace ID and the timestamps of the events within that trace $\{(trace_{id}, ts_a, ts_b)\}$. The selection policy determines how the inverted index is constructed. For strict contiguity, all pairs of consecutive events are added as an index, whereas the skip-till-next policy requires a more complex construction of event pairs. For the construction of skip-till-next indexes, the system determines all distinct events, and for each distinct event e_i and any events e_j that follow it in the trace, it adds an index for the pair (e_i, e_j) until the next event e_i is reached in a trace. Once the indexes are constructed, the system can answer a sequence detection query *SEQ(A,B)* by finding all traces that contain the event A as the starting point (A, e_i), and then finding all pairs that contain e_i as the starting point (e_i, e_j), until finally a pair (e_j, B) is reached. The timestamps belonging to the events help ensure that at each step in the detection process, the same event instances are used that were matched in a previous step. The authors compared their work to SASE [5] and found it to be faster when querying larger datasets for sequences with a large number of events.

Given that this solution most closely resembles the problem of answering arbitrary ad-hoc queries, whereas other solutions focus more on answering continuous queries, it is this work which we will compare our proposed solution to in

Chapter 5.

3.2 Temporal Sketches

We now look at work that has been done to add temporal component to sketches.

The work by Peng et al. [26] focuses on developing a probabilistic data structure based on Bloom filters that supports temporal membership queries. In their paper, they define a temporal membership query $q(x, [s, e])$ as a test of whether an event x has occurred at least once in the time-range $[s, e]$. They describe two data structures that they refer to as persistent Bloom filters (PBF-1 and PBF-2) that can be used to answer such queries. Both persistent Bloom filters work on the same underlying principle, which is to build a hierarchy of Bloom filters for different time ranges. PBF-1 does this by building a binary tree of Bloom filters, where Bloom filters in levels higher up the tree are a union of the Bloom filters in its children. To limit the number of Bloom filters, PBF-1 assumes a minimum time-granularity of g at the leaf level, and makes use of an additional Bloom filter for queries that require a smaller time-granularity. This additional Bloom filter, which the authors refer to as SBF, stores each event along with its timestamp in the form of a tuple (a, t), where a is the event and t is time at which the event occurred. SBF answers a query $q(x, [s, e])$ by splitting the query into multiple membership-queries, one for each possible pair (a, t), for each t in the range $[s, e]$. For queries with a higher time-granularity, PBF-1 answers the query by performing a single membership-query for each Bloom filter that occurred in the cover of the interval $[s, e]$. In Chapter 4, we propose a structure similar to that of PBF-1 to answer pattern matching queries over data streams. In PBF-2 the authors minimize the number of Bloom filters by using only a single Bloom filter for each level in the tree, where the time granularity doubles with every level. The tree consists of $L = \lceil \log_2 T \rceil + 1$ levels, where T is the current upper bound on the time range that is stored in the structure. Each level l within the tree is assigned a Bloom filter that stores events with a time granularity of at most 2^{L-1-l}. When an event is inserted, it is inserted into all Bloom filters as a pair (e, t) where e is the name of the event and the value of t depends on the Bloom filter's level within the tree and is equal to $\lceil t_e / 2^{L-1-l} \rceil$, where t_e is the actual time of event e. To answer a temporal membership query (e, t), each Bloom filter is then queried with the time value corresponding to the current Bloom filter's level within the tree. While PBF-2 can be used to infer the termporal order of events in a data stream, it is not suitable for answering pattern matching queries as this would require us to

explicitly query for multiple different timestamp and event combinations.

Several other works make use of a hierarchical structure to establish an order between the events that are hashed to it. These hierarchies are often used to answer range queries which test whether an event occurred within a given range. Range queries naturally assume an underlying order between the events, although this order may not always be temporal in nature. One such work is the Adaptive Range Filter (ARF) by Alexiou et al. [27], which was proposed to detect whether a record in a database with a key in a certain range can be considered cold data, implying that it does not get accessed frequently. To this end, the authors propose ARF, which can be used to used to check for matches within a range of values. Adaptive Range Filters are binary trees, where the leaves indicate whether any of the keys in its range contain cold data by setting a single bit to 0 or 1. Internal nodes contain the range intervals of its children and provide easy navigation to the leaves. While ARFs can be used to test if any of the values in a range have a specific property, we are interested in determining which values belong to a given range, as this allows us to determine the order of events.

3.3 Pattern Matching Bloom Filters

Bloom filters have previously been used for pattern matching in other areas. These solutions either solve completely different problems using similar techniques or use techniques that cannot directly be applied to pattern matching in the context of Complex Event Processing. An example of work that solves a different problem, but with a structure similar to ours, is the Hierarchical Bloom Filter Tree by Lillis et al. [28], which was developed to match a collection of seized files to collections of files that are known to be incriminating, as is common in the field of forensics. The solution is an extension of the *MRSH-v2* algorithm [29] which generates similarity digests for a file in the form of a collection of Bloom filters. A similarity digest is a compression of a file that can be used to check for similarities without having to compare the full contents of the file. Such a digest is calculated by splitting a file into chunks and hashing each chunk to 5 bits in a Bloom filter until it is considered full, before proceeding to a new Bloom filter. The resulting collections of Bloom filters of different files are then checked for similarity [28]. The authors aim to speed this up by introducing a tree of Bloom filters, where each chunk of 5 bits is hashed as a leaf in the tree (until the leaf is full) and different leaves are merged to create the internal nodes. To find out if a file is similar to another file in a collection, its hashed chunks are calculated and are queried against the

nodes in the tree. If several consecutive hashed chunks all return 1, then a match is found. The proposed structure can determine similarity between files using fewer comparisons than the original *MRSH-v2*. While this solution is not directly relevant to Complex Event Processing, it shows how a similar structure can be used to answer different types of pattern matching problems.

Another work that uses Bloom filters for pattern matching is that by Moraru and Andersen [30], which is aimed at matching a large number of patterns against a large corpus. This is technically similar to the problem we aim to solve if we consider the data stream to be the corpus and the large number of patterns to be the different types of patterns we wish to match. To this end, the authors develop Feed-Forward Bloom Filters (FFBF), which are built from the patterns that need to be matched and are used to reduce the size of the corpus. It does this by checking whether fragments of the corpus generate a match, and discards any fragments that do not match. It further uses all matching fragments to perform a similar test as described above, but with the patterns as the corpus and the matching fragments as the FFBF. Finally the remaining patterns and fragments can be matched using an exact pattern matching algorithm. While this solution is interesting for determining exact patterns, it cannot provide support for pattern matching operators that are common to Complex Event Processing. It is also not suitable in a streaming setting.

Chapter 4

Solution

This chapter describes the solution, the types of queries it supports, and different query evaluation techniques that minimize the query time. Finally, it provides an analysis of the running time and describes how the system can be configured to guarantee a user-specified false-positive rate.

4.1 Description

The goal of the solution is to impose an order between events without storing the events themselves or indicators of their temporal positions within the stream. We first describe a simplified version of the solution, which will be extended further on in this section.

To impose a temporal order between events, we split the stream into C small count-based windows of events and assign a separate Bloom filter to each window. This results in an a sequence of Bloom filters, where the events that hash to a Bloom filter bf_i have occurred prior to the events hashed to a Bloom filter that follows it, bf_{i+1}, where $i \in [0, B]$ and B represents the total number of different Bloom filters. Figure 4.1 illustrates this.

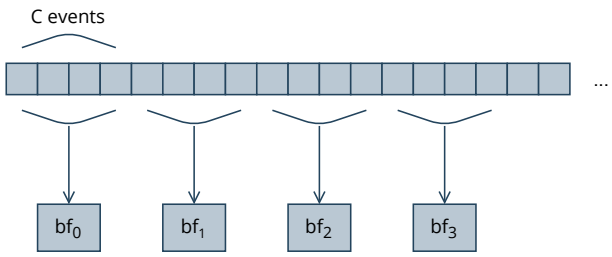


Figure 4.1: The data stream split into windows of C events that all hash to a different Bloom filter.

To check whether an event A occurred before another event B using this structure, we check all Bloom filters in the sequence until we find the first Bloom filter that contains A. We then check all Bloom filters to the right of the one that contains A for B. If a Bloom filter returns *True* then we can conclude that A occurred before B. If none contain B, then either B did not occur at all, it occurred before A, or it is hashed to the same Bloom filter as A.

Since a Bloom filter itself does not define an order between the events hashed to it, the events that hash to the same Bloom filter are assumed to have occurred at the same time. If in the example above B only hashed to the same Bloom filter as A, then we would not be able to infer their temporal order. Therefore, the size of the windows that split the stream defines the granularity with which we can distinguish the order of different events. This granularity in part defines the accuracy with which we can answer a query; the larger the number of events hashed to a single Bloom filter, the fewer orderings we can distinguish. It is for this reason that a smaller number of Bloom filters would not be desirable. However, as the resulting structure is a sequence of B Bloom filters, it could take $O(B)$ time to query. While we cannot save on the number of events hashed to a single Bloom filter, we can adapt the structure to make it more efficient to query. As described in Chapter 2, an interesting property of Bloom filters is that we can take their union, by taking a bitwise-OR of their bit arrays, assuming they share the same k hashing functions and the Bloom filters all use the same number of bits m. To this end, we construct a tree-like structure of Bloom filters, where the Bloom filter in a node higher up the tree results from merging the Bloom filters in the node's children. For ease of discussion, we assume that each internal node, except the root, has exactly two children. However, the solution can support any number of children, which we refer to as the tree's fanout.

The tree is constructed as follows. Algorithms 1 and 2 show the pseudo-code for this process. First, we pick a value C that defines the count-based window of events that should hash to the same Bloom filter. We then process the stream and hash the first C events to the first Bloom filter bf_0 (Algorithm 1, Line 5). Before proceeding to the following C events, we add a node for this Bloom filter as a leaf within the tree (Algorithm 1, Lines 7-14). The node has two main properties; namely, the Bloom filter and the interval of elements hashed to it, to support queries within interval ranges. For a Bloom filter bf_i, its interval is of the form $[i \cdot C + 1, (i + 1) \cdot C]$, where $i \in [0, L]$ and L represents the total number of leaves within the tree, which corresponds to the total number of original, unmerged, Bloom filters. In the case of the first Bloom filter, this interval is $[1, C]$, $[C + 1, 2C]$

for the second, $[2C + 1, 3C]$ for the third, etc (Algorithm 1, Line 10). It is also useful to keep intervals of the timestamps of the events hashed to a Bloom filter for queries within time intervals. This can be done by taking the timestamp of the first event hashed to it as the starting point of the interval, and the timestamp of the last event hashed to it as the interval's ending point. However, for ease of discussion we omit this and assume only count-based intervals. After the first Bloom filter is stored as a leaf within the tree, a new Bloom filter is created to handle the next C events within the stream (Algorithm 1, Line 12). Whenever new leaves are inserted, the algorithm checks whether nodes can be merged to form a new parent node (Algorithm 2, Lines 3, 7). A group of nodes is a candidate for merging if it meets the following two requirements.

- **Level constraint:** The nodes in the group are all on the same level δ within the tree. The leaves are at level 0 and the root is at level h, where h is the height of the tree.
- **Density constraint:** The Bloom filter that results from merging the Bloom filters of the nodes within the group must have a density D with $D \leq \frac{m}{2}$, where D represents the number of bits that have been set to 1, and m is the size of a single Bloom filter.

The density constraint is based on the observation that the false positive rate of a Bloom filter is minimized if its density does not exceed 50% [13]. For every two nodes that can be merged, we create a new node (Algorithm 2, Lines 4-5) and define its Bloom filter as the union of the Bloom filters of its children (Algorithm 2, Lines 6, 8). We then make the nodes that have been merged children of the newly created merged node (Algorithm 2, Lines 9-13). Note that nodes that can no longer be merged lead to several different Bloom filters at the top of the tree. These top-level Bloom filters are crucial for providing fast answers to user queries. Figure 4.2 shows the construction of the tree and Figure 4.3 shows how fully-merged nodes can accumulate at the top of the tree. Note that the root serves no purpose other than to provide easy access to its children.

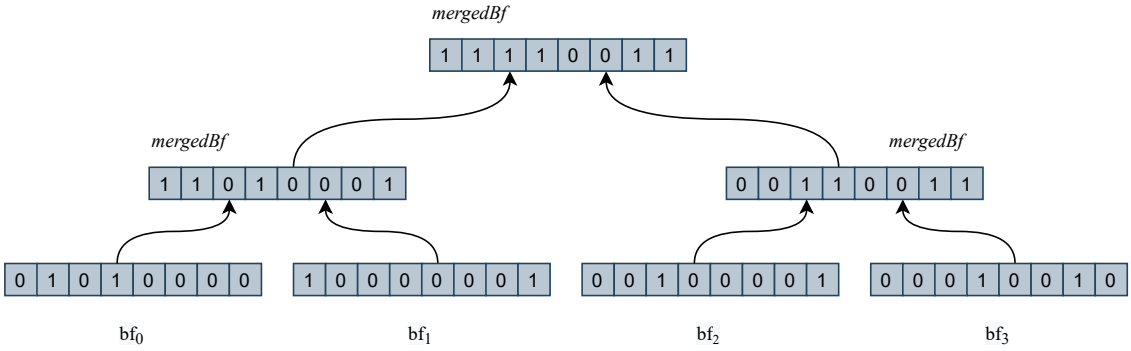


Figure 4.2: The bitwise-OR merging of Bloom filters to form a tree.

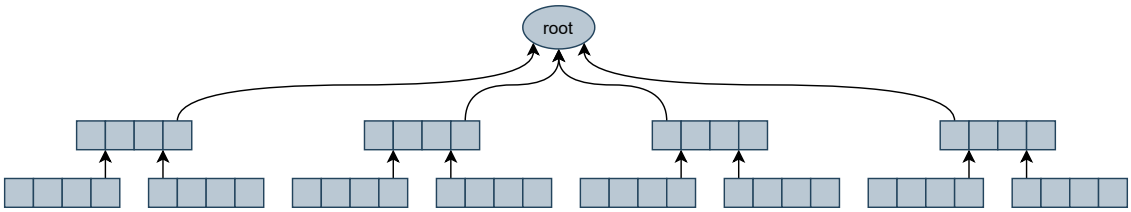


Figure 4.3: Fully-merged nodes lead to several different Bloom filters at the top of the tree.

Algorithm 1: process()

Input: stream of events, count-based window size C

```
1  count = 1
2  intervalStart = 1
3  initialize new node root
4  initialize new Bloom filter, bf
5  for each event in the stream do
6      add event to bf
7      if count % C == 0 then
8          initialize new node newLeaf
9          newLeaf.bf = bf
10         newLeaf.interval = [intervalStart, count]
11         root.addLeaf()
12         create new bloom filter bf
13         intervalStart = count + 1
14     end
15     count++
16 end
```

Algorithm 2: addLeaf()

Input: node to be added as leaf

```
1  add node as a child of root
2  for each pair (child1, child2) of neighboring children of root do
3      if child1 and child2 are on the same level then
4          initialize new node mergedNode
5          initialize new Bloom filter bf
6          bf = child1.bf ∪ child2.bf
7          if bf.density <= m/2 then
8              mergedNode.bf = bf
9              remove child1 as child of root
10             remove child2 as child of root
11             add child1 as child of mergedNode
12             add child2 as child of mergedNode
13             add mergedNode as child of root
14         end
15     end
16 end
```

4.2 Query Types

The solution presented above aims to provide various levels of support for the operators and selection policies described in Chapter 2. Due to the order of events being indistinguishable within the leaves, we cannot support the selection policies in their current forms. However, if we relax the definitions such that we (i) ignore the detection of pattern matches within a leaf, and (ii) assume that strict-contiguity applies only between a leaf and its direct neighbor, instead of an event and the event that occurs directly after it, then we can provide support for relaxed versions of the selection policies. Note that since many interesting CEP queries already require that we skip over any irrelevant events, these relaxed definitions return results that are still valuable; they may only fail to detect pattern matches that occur within a single leaf.

The relaxed selection policies are defined as follows.

- **Strict-contiguity:** The matching events must occur in contiguous leaves.
- **Skip-till-any:** Any number of irrelevant nodes may occur in-between the nodes of matching events.
- **Skip-till-next:** Any number of irrelevant nodes may occur in-between the nodes of matching events until the first node containing a matching event is found.

The solution can further support all operators from Chapter 2, except for iteration. However, a relaxed version of iteration can be achieved by querying for the same event in sequence. Note that only the sequence operator supports nested operators, but it is not allowed to receive other SEQ-operators as arguments. The operators can be provided as a query using the following syntax. All operators may receive multiple arguments, except for *Windowing*.

- **Sequence:** *SEQ(*$e_1, \ldots, e_n$*)*
- **Disjunction:** *OR(*$e_1, \ldots, e_n$*)*
- **Conjunction:** *AND(*$e_1, \ldots, e_n$*)*
- **Negation:** *NOT(*$e_1, \ldots, e_n$*)*
- **Windowing:** $[t_{start}, t_{end}]$

More formally, assume e is a single event, then the following grammar defines the language with which we can query for patterns. This language was inspired by the operators used in SASE [5].

$$op \ \rightarrow \ e \mid OR(e_1, \ldots, e_n) \mid AND(e_1, \ldots, e_n) \mid NOT(e_1, \ldots, e_n) \mid SEQ(op_1, \ldots, op_n)$$

Given these relaxed selection policy definitions, the semantics of the operators need further clarification. For example, conjunction *AND(A,B)* assumes that both A and B occur, independent of their order, implying that we need to find a match for either *SEQ(A,B)* or *SEQ(B,A)*. There are two cases in which we return a match for the *AND*-operator:

1. When all events occur in the Bloom filter of a single node.
2. When a match is found in subsequent, but not necessarily strictly-contiguous nodes for all permutations of the events. For example, for *AND(A,B,C)* we check for the occurrence of *SEQ(A,B,C)* $\vee$ *SEQ(A,C,B)* $\vee$ *SEQ(B,A,C)* $\vee$ *SEQ(B,C,A)* $\vee$ *SEQ(C,A,B)* $\vee$ *SEQ(C,B,A)*.

In case (2), the nodes should only be strictly-contiguous if the selection policy requires it. Similarly, the *OR*-operator is supposed to return a match when the Bloom filter of a node contains any of the events.

The *NOT*-operator has a special meaning since it is often used to ensure that an event does not occur until another event occurs. This has similarities with the *Until*-operator from Temporal Logic [31]. Since there are no common semantics for the *NOT*-operator, we assume that it can only occur between two other operators. When *NOT(A)* occurs between two different operators, then the match starts from the node immediately following the last match until the first match of the operator that follows it. This is similar to how SASE [5] handles negation, as described in Chapter 3. When multiple events are passed to *NOT*, we check that none of the nodes on a path between the matches of other operators contain any of the events passed to *NOT*.

We further assume that only *SEQ* of events that do not contain other operators works in combination with the **skip-till-any** selection policy. All other operators assume either **strict-contiguity** or **skip-till-next**.

Finally, the solution supports two different types of queries based on the type of results they return: (i) **Simple queries**, which determine whether a pattern occurred and the result is either *True* or *False*, and (ii) **Complex queries**, which find

all intervals in which a pattern occurred, and the result is a list of the matching intervals.

4.3 Query Evaluation

To evaluate pattern matching queries, we walk the tree in search for matches. The main benefit of using a tree-based data structure to answer these queries is that most queries that would otherwise require an exhaustive scan of the data structure can already be answered by looking at the top-most levels of the tree. For example, when pattern queries contain events that have not been seen in the stream, then assuming that no false positive occurs, the algorithm can immediately return a response after looking only at the top-most nodes in the tree. Note that these top-most nodes may also contain Bloom filters that have not been merged fully yet, as indicated by Figure 4.4, where the pink nodes on a lower level in the tree can still be merged. In the rest of this report, we refer to all pink nodes in Figure 4.4 as top-level nodes t. Furthermore, simple queries that only question the occurrence of a single pattern match can often also be answered by looking only at the nodes in the upper levels of the tree. However, complex queries that require all matching intervals as a response may be more expensive to process. This section looks at query evaluation plans designed to minimize the number of walks in the tree.

When a query arrives, the events passed as arguments to the operators are extracted and tested for occurrence against the top-most nodes in the tree. The only exception to this rule is the argument of the negation operator, since an upper-level node can return *False* for a sub-query *NOT(A)*, yet when walking down the tree, we may still find nodes where this predicate holds. Figure 4.5 illustrates this. We therefore always walk down the tree until we reach nodes that can guarantee that the event did not occur within an interval.

If any arguments in the query did not occur, then we immediately return *False* or an empty list of intervals, depending on the query type.

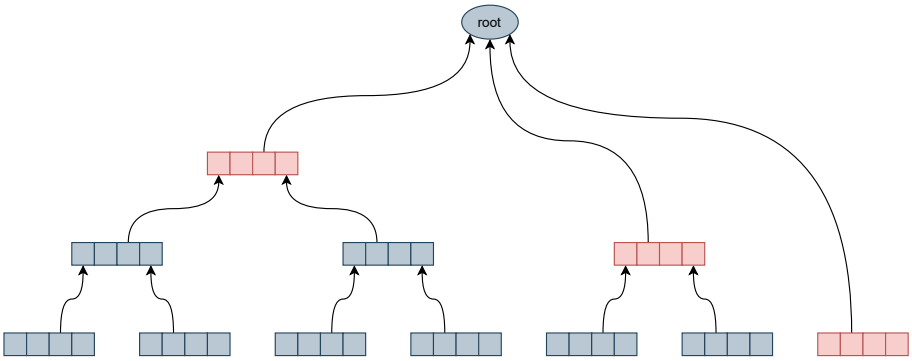


Figure 4.4: The pink nodes indicate the top-level nodes t.

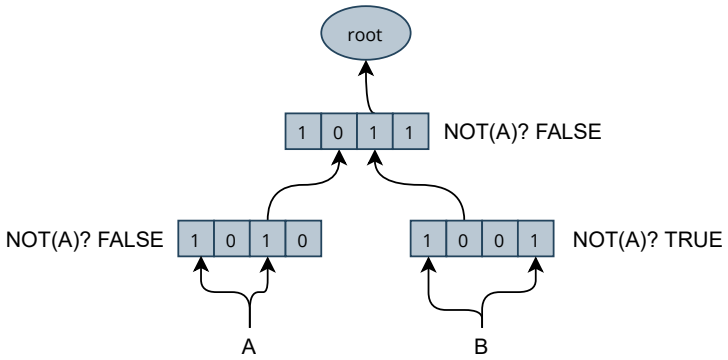


Figure 4.5: A query *NOT(A)* that cannot be answered by looking at a top-level node.

4.3.1 Strict-contiguity

Queries that assume a strict-contiguity of events in sequence are processed as follows. For a query *SEQ(A,B)*, we start by finding the top-most nodes that contain events A and B. We then walk down to the leaves in search for A only in the sub-trees of top-level nodes that also contain B. At every leaf that contains A, we then check its right neighbor for B. Note that in some cases, the leaf that contains A might be the right-most leaf of a sub-tree. In that case, the top-level node of the sub-tree that contains the leaf might not contain B. In these cases, we check whether B is hashed to the right neighbor of the top-level node of A, and include the sub-tree of the top-level node that only contains A in our search as well. Figure 4.6 illustrates this. When the query checks for multiple events in sequence *SEQ(A,B,C)*, then we immediately check for C in the right neighbor of the leaf that contains B. If the query is simple, the algorithm returns *True* after the first match is found or *False* if no match exists. If the query is complex, the

corresponding interval is saved and the search continues starting from the leaf that contains the last event.

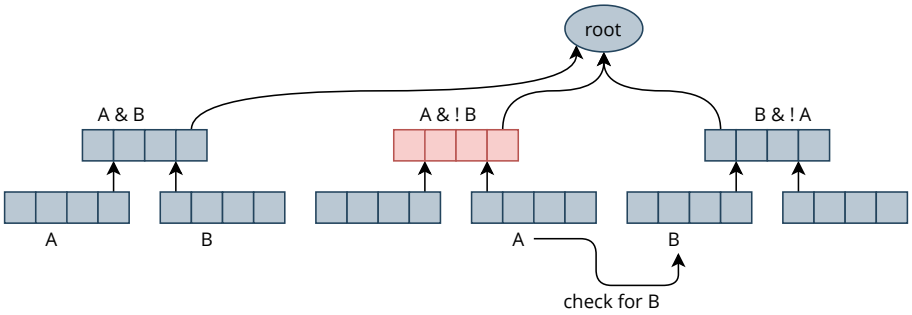


Figure 4.6: Strict-contiguity query evaluation of $SEQ(A, B)$ where A and B do not hash to the same top-level node.

4.3.2 Skip-till-next

Skip-till-next queries are processed as follows. We hereby make the distinction between events that hash to the same top-level node and those that do not.

When two events do not hash to the same top-level node and the first top-level node that contains B follows that of A, then we either return *True* or save the interval and continue the rest of the search starting from the node that contains B. If no such ordering can be found, and the events do not hash to the same top-level node, then we return *False* or an empty list of intervals.

If the two events do hash to the same top-level node, then we walk down the tree until we reach a level at which A occurs in a Bloom filter that does not contain B. We then have to ensure that at some point in the tree, B was found in a node that occurred to the right of the one that contains A. To avoid having to walk up the tree, we keep a stack that finds the first node that contains B to the right of the one that contains both events, on every level in the tree. We then query this stack when we reach a level where a Bloom filter contains A, but no longer contains B. If the stack is empty, then we continue the search starting from the top-level node to the right of the top-level node of the current sub-tree. If the stack is not empty, we take the interval of the last node that was added to the stack as the ending point of the interval for $SEQ(A, B)$ and continue the search starting from this interval.

If the query contains more than two events, such as $SEQ(A, B, C)$, then for each matching interval we take the starting point of A and the ending point of C and add the resulting interval to the list of intervals to be returned.

Note that this evaluation technique stops as soon as we reach a level where two events stop hashing to the same Bloom filter, so the accuracy of the returned interval is given by the granularity of the level. For better accuracy, we may choose to walk down to the leaves to find the exact intervals within which the matching events were found. The desired level, and therefore the accuracy, can be adjusted on a per-query basis.

4.3.3 Skip-till-any

Skip-till-any queries are answered in two different ways, depending on the query type. If the goal is to return *True* or *False*, then for a query *SEQ(A,B,C)*, we do the following.

1. Find the first top-level node that contains A, t_A.
2. Starting from t_{A+1}, find the first top-level node that contains B, t_B.
3. Starting from t_{B+1}, find the first top-level node that contains C, t_C.

If at any point we cannot find the first top-level node of an event in the list of nodes that occur after the previous event's node, then it may indicate that the pattern can still be matched if the pair of events hashed to the same top-level node. We then use backtracking, starting from the first event where the search failed, and check if the top-level node of the previous event also contains the current event. If this is not the case, then we return *False*. Otherwise, we walk down the sub-tree of the top-level node that contains both events. At each level of the tree, we then repeat the steps above until either a match is found or no such match exists. If no match exists, we cannot immediately return *False* unless we have reached the starting event. We then repeat the backtracking algorithm for the previous two events in the query, and stop only when the pattern match fails because an element does not occur in a top-level node or when we reach the starting event without being able to find a match.

If the goal is to return all intervals, the process is more complex, since for every A we have to find all possible intervals for B that occur after A, and for every such B we have to find all intervals C that occur after every B. To answer such queries, we make a list of all the leaves that contain the events, and for each event, we add a tuple where the first element is the event and the second element is the leaf that contains it. We then construct the list of matching intervals from this list using an exhaustive search of the list. Note that **skip-till-any** queries that return intervals

are not only the most expensive to process, but also the least useful, as larger matching intervals will automatically encompass the intervals of smaller matches. As such, **skip-till-any** should ideally only be used to answer simple queries that return *True* or *False*.

4.3.4 Operators

Finally, while most operators such as *SEQ* and *OR* are straightforward to answer, some require additional steps, such as Windowing, Negation, and Conjunction. Their evaluation methods are described below.

Windowing

To answer queries within an interval (window), the algorithm first calculates the nodes that form a cover of the window and then queries only these nodes and their children, if necessary. Whenever complex queries are answered, and the starting point of the next interval search is an internal node, then we also calculate the set of nodes that forms a cover from that node's starting point to the end of the query window.

Negation

To answer queries that contain a negation operator *NOT*, we first skip over any negation operators and find matches only for the operators that occurred before and after the *NOT*-operator. Then for each pair of matches, we determine the interval of nodes between any two matching nodes and ensure that the events passed to *NOT* do not occur within these nodes.

Conjunction

Finally, as described in Section 4.2, Conjunction is answered by checking the disjunction of all permutations of events passed to the *AND*-operator. As such, we first calculate the different permutations and create a new query for each permutation. We then return the set of all intervals that matched the resulting queries.

4.4 Theoretical Analysis

4.4.1 False Positive Analysis

The solution described in this chapter makes extensive use of Bloom filters, which are known to cause false positives that can be configured by adjusting the number of hashing functions and bits that the Bloom filter uses. This thesis aims to provide an approximate data structure that can detect patterns in a data stream with a desired false-positive rate fpq that the user provides as a parameter. Further required parameters are the desired window size C, the maximum available space M, and the expected size of the stream N, as well as the expected length of a typical query s, although this last value is optional and is set to 2 by default. Based on these input values, which are highlighted in Table 4.1, the algorithm must determine an appropriate configuration for the Bloom filters to guarantee the desired false-positive rate fpq, while remaining within the space bounds. Table 4.1 provides an overview of the variables that are used in the following calculations.

N	The expected size of the stream.
M	The total available memory.
C	The count-based window size that describes how many events hash to a single Bloom filter leaf.
s	The desired query length.
fpq	The desired false-positive rate for a query, specified by the user.
fp	The false-positive rate of a single Bloom filter.
m	The number of bits of a single Bloom filter.
k	The number of hashing functions.
T	The total number of nodes in the tree.
q	The total number of Bloom filters probed as part of a query.
t	The number of top-level nodes.
d	The total number of nodes in a sub-tree.
L	The total number of leaves in the tree.
l	The number of leaves in a single, fully-merged sub-tree.
n	The number of distinct events in a sub-tree.
$\hat{n}$	The number of non-distinct events in a sub-tree.
p	The number of pairs in a query.

Table 4.1: Variable definitions. The highlighted values are input values that are used to configure the system.

The false-positive rate of the solution depends on the length of the query, which corresponds to the number of arguments. For this analysis, we assume a complex query that returns all intervals for four events in sequence $SEQ(A, B, C, D)$ with no additional operators, under a skip-till-next selection policy. We first calculate the false-positive rate of a query that consists of a single event and then

extend the resulting equation to determine the appropriate configuration for the system given the full query. Note that we first provide a derivation of the formulas needed to compute an appropriate configuration. Only the final equations (4.16 - 4.18) take the input values of the user into account to derive the final configuration. Furthermore, all equations below provide bounds on the values, since they assume a perfectly balanced and fully-merged tree. In practice, the tree will never be fully-merged or balanced.

As described in Chapter 2, the probability that a single Bloom filter returns a false positive is given by the following formula:

$$fp = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k} \tag{4.1}$$

The false-positive rate of a query can be calculated as follows. Assume we have to probe q different Bloom filters as part of a query, then we want to know the probability that at least one of the q Bloom filters returns a false positive. To calculate this, we first determine the probability that none of these Bloom filters return a false positive:

$$\prod_{i=1}^{q}(1 - fp) = (1 - fp)^{q} \tag{4.2}$$

Then the probability that at least one results in a false positive is:

$$fpq = 1 - (1 - fp)^{q} \tag{4.3}$$

Note that since no false negatives occur, these do not have to be taken into account. Replacing the formula for fp in (4.3) gives the following equation.

$$fpq = 1 - \left(1 - \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k}\right)^{q} \tag{4.4}$$

From (4.3), we can calculate the false-positive rate fp of the individual Bloom filters as follows.

$$fp = 1 - \sqrt[q]{1 - fpq} \tag{4.5}$$

Figure 4.7 shows how the false-positive rate of a query fpq depends on the number of Bloom filters that are probed in the process q, and their individual false-positive rates fp. It is clear that the number of Bloom filters that are accessed as

part of the query are a significant contributing factor to the false-positive rate of the query. However, note that the false-positive rates of the Bloom filters are the highest in the fully-merged top-level nodes, as these contain the most events. In general, a query is not answered by looking only at the top-level nodes in the tree. Therefore, any false-positives that occurred in a Bloom filter higher up in the tree may be corrected as we walk down the tree. This means that the probabilities that the top-level Bloom filters do not return a false positive provide an upper bound on the probabilities that any of their children do not return a false positive. As such, we can set $q = t$, where t is the number of top-level nodes, since a complex query has to look at all top-level nodes, and in many cases their children as well, to find all matching intervals.

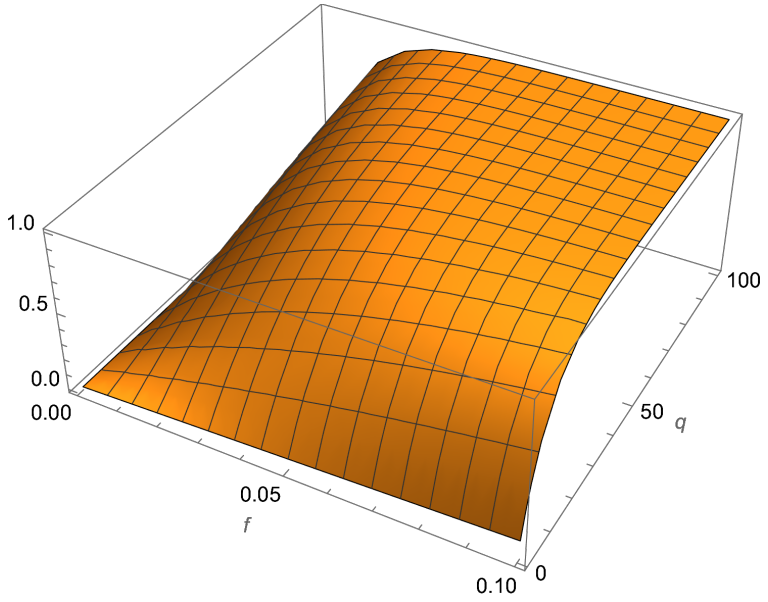


Figure 4.7: The output represents the false-positive rate of a query fpq in terms of fp (displayed as f here) and different values for q.

Figure 4.8 shows how the value of q impacts the false-positive rates of single Bloom filters, assuming $fpq = 0.0001$. From this figure, we can see that supporting a greater number of top-level nodes (in this case q) requires smaller false-positive rates for the q Bloom filters. Since top-level Bloom filters can already answer several order-based queries, we would still like to have as many of these as we can fit in memory. However, the memory complexity increases as a consequence.

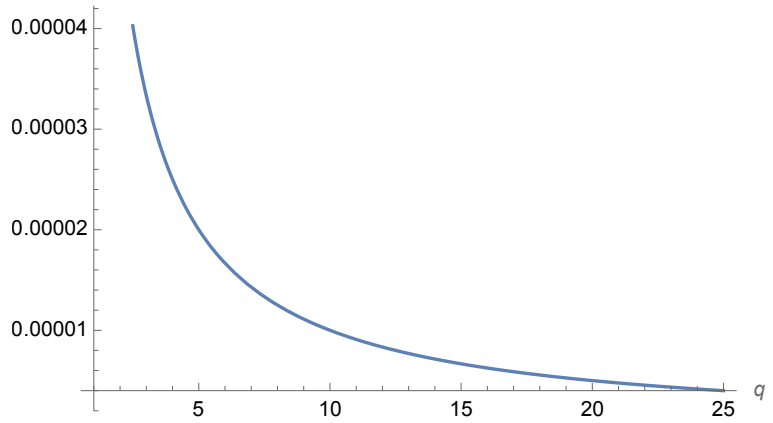


Figure 4.8: The output represents the false-positive rate fp required for single Bloom filters given $fpq = 0.0001$ and different values for q.

Since M is the maximum available memory assigned to the data structure, the size of the bit arrays of all the Bloom filters in the tree must be less than M. This constraint is given by the following equation, where T represents the total number of nodes in the tree.

$$T \cdot m \leq M \tag{4.6}$$

The total number of nodes in the tree T can be calculated by multiplying the total number of sub-trees, given by the top-level nodes t which form their roots, and the total number of nodes in each sub-tree d.

$$T = t \cdot d \tag{4.7}$$

The total number of sub-trees t can be determined by dividing the total number of leaves L by the number of leaves in each sub-tree l.

$$t = \frac{L}{l} \tag{4.8}$$

Note that the total number of leaves in the tree is equal to the expected size of the stream divided by the count-based window size C that splits the stream into different leaves. Since these are both input parameters, L can be determined entirely from the input.

$$L = \frac{N}{C} \tag{4.9}$$

The number of leaves l per sub-tree t depends on the height of the tree h. Assuming a binary fanout and a fully-merged sub-tree, l can be determined as follows.

$$l = 2^{h-1} \tag{4.10}$$

Using the height of the tree h, we can also determine the total number of nodes in a single sub-tree using the following equation.

$$d = \sum_{i=0}^{h-1} 2^i = 2^h - 1 \tag{4.11}$$

Equation (4.7) can now be rewritten as:

$$T = \frac{(2^h - 1)L}{2^{h-1}} \tag{4.12}$$

From Chapter 2, we know that m can be calculated as follows.

$$m \geq -\frac{n \ln fp}{(\ln 2)^2} \tag{4.13}$$

Replacing the formula for fp (4.5) into the formula above gives us the following equation. Note that we replaced q with $t = \frac{L}{2^{h-1}}$ from (4.8) and (4.10).

$$m \geq -\frac{n \ln (1 - \frac{L}{2^{h-1}} \sqrt{1 - fpq})}{(\ln 2)^2} \tag{4.14}$$

The memory constraint can then be rewritten as follows.

$$-\frac{(2^h - 1)L}{2^{h-1}} \cdot \frac{n \ln (1 - \frac{L}{2^{h-1}} \sqrt{1 - fpq})}{(\ln 2)^2} \leq M \tag{4.15}$$

Given this constraint in (4.15), we now want to maximize $q = \frac{L}{2^{h-1}}$ to provide as many order-of-arrival distinctions at the top of the tree as possible.

Note that the equation above depends on the distinct number of elements that will be hashed to a Bloom filter. Since n is maximal for top-level nodes, we now determine the expected number of distinct items in a top-level node based on the number of leaves in its sub-tree. Since a top-level node has $l = 2^{h-1}$ leaves and each leaf consists of C items, then we have $\hat{n} = 2^{h-1}C$ non-distinct items in the leaves of a sub-tree. To determine the expected number of distinct items after having seen $\hat{n} = 2^{h-1}C$ items in the stream, we note that in many streams the different events are not distributed uniformly. One of the most common distributions for data streams is the power law distribution, and more specifically the Zipf distribution [32]. Given that many datasets that follow Zipf's Law are also found to follow Heap's Law [33], which is used to determine the vocabulary given a collection of tokens [34], we can use Heap's Law to estimate the number of distinct events n in a sub-tree, after having seen $\hat{n}$ events. Heap's Law is given by

the following formula, where $30 \leq K \leq 100$ and $\beta \approx 0.5$ [34]. For the rest of the equations we assume $K = 60$ and $\beta = 0.5$.

$$n = K(\hat{n})^{\beta} = K(2^{h-1}C)^{\beta} \tag{4.16}$$

We then get the following constraint, in which only the height of the tree h is a variable.

$$-\frac{(2^h - 1)L}{2^{h-1}} \cdot \frac{(60(2^{h-1}C)^{0.5}) \ln\left(1 - \frac{L}{2^{h-1}} \sqrt{1 - fpq}\right)}{(\ln 2)^2} \leq M \tag{4.17}$$

We then solve $q = \frac{L}{2^{h-1}}$ for h under the constraint above, which gives us the total number of top-level nodes. Once we have obtained h we can calculate m using (4.14) and k using the equation below from Chapter 2.

$$k = \ln 2 \cdot \frac{m}{n} \tag{4.18}$$

For both m and k we use the distinct number of events n obtained by (4.16).

Figure 4.9 shows how the total storage M (in bits) grows under different heights of the tree, for $fpq = 0.0001$ and $N = 1,000,000$ events.

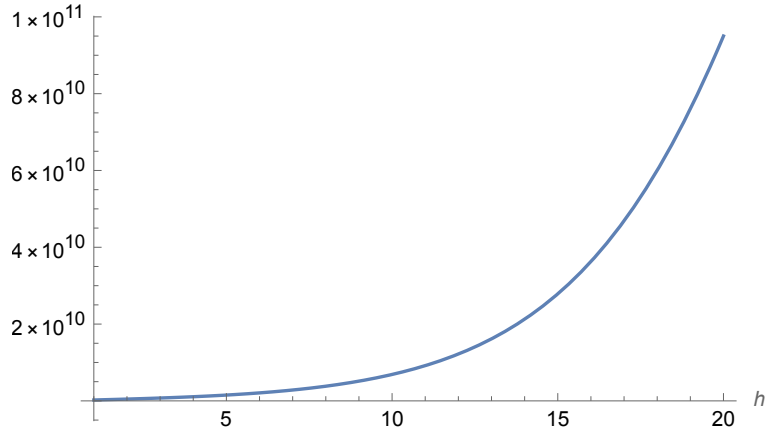


Figure 4.9: The output represents the total memory (in bits) required to store $N = 1,000,000$ events with $fpq = 0.0001$ given different heights h.

Since the false-positive rate fpq of a query depends on the length of the query, which is determined by the number of arguments, the number of top-level nodes that are probed q grows with the length of the query. For a query $SEQ(A, B, C, D)$ of length four, we probe the Bloom filters twice for every pair. Since there are three pairs in this query: $SEQ(A, B)$, $SEQ(B, C)$, $SEQ(C, D)$, the q Bloom filters have to be probed six times. We therefore solve for h using $q = \frac{6L}{2^{h-1}}$ instead of

$q = \frac{L}{2^{h-1}}$ and calculate the appropriate values for k and m, using the technique described above.

4.4.2 Running Time Analysis

The running time complexity of the resulting data structure can be split into three main parts: insertion, merging, and querying. The insertion of an item from the stream takes $O(k)$ time as it needs to be hashed using k hashing functions. Since this is done for N events, insertion takes $O(kN)$ time in total. Whenever C items are processed, a node is created for the Bloom filter and any eligible nodes are merged to form a new node in a level higher up the tree. For any leaf with Bloom filter bf_i with $i \in [0, L]$, $\frac{N-C(L-i)}{C}$ leaves have been inserted into the tree before it. Therefore, the worst-case cost of merging is $\sum_{i=0}^{L} \frac{N-C(L-i)}{C} \cdot O(m)$, as the merging of two bit arrays takes $O(m)$ time and in the worst case all nodes may need to be merged. Note that this ignores any fully-merged nodes. Finally, the cost of querying is determined by the type of query, its length, and the level at which we can answer it. For each pair of arguments, every Bloom filter that is probed in the process is queried twice: $O(2q)$. However q can vary greatly depending on the query. In the absolute worst-case, when answering a complex query that requires us to return all intervals, we may have to walk down to the leaves for every possible leaf and perform two Bloom filter probes of $O(k)$ per level on the path down to the leaves. We then have to repeat this for each possible pair. Let p be the number of pairs in a query, then the absolute worst-case running time is $O(pL \cdot \log_2 d)$, if every pair of leaves contains a match. For this reason, the solution should only be used to return all intervals for patterns that can be considered rare.

Chapter 5

Experiments

This chapter compares the solution proposed in this report to the exact Complex Event Processing method proposed by Mavroudopoulos et al. [25]. It further contains experiments using synthetic datasets that analyze the time needed to construct the data structure as well as answer different types of queries on datasets of varying lengths. Finally, it provides a discussion of the results.

5.1 Comparisons to Exact Solution

5.1.1 Experimental Setup

The goal of the following experiments is to demonstrate how the approximate solution proposed in this report compares to exact pattern matching solutions in terms of the time it takes to respond to queries. To this end, we compare our solution to the solution proposed by Mavroudopoulos et al. [25], which aims to detect sequences of events in event log files. The primary reason why we compare to this work is because it also focuses on providing fast answers to arbitrary queries over the full dataset, whereas other solutions may be optimized for continuous solutions instead. It further is one of the most recent contributions in the field of Complex Event Processing, and has shown to be significantly faster at answering longer skip-till-next sequence queries than SASE [5], which is considered state-of-the-art. Mavroudopoulos et al. [25] support sequential pattern matching under the strict-contiguity and skip-till-next selection policies. Since the results of our relaxed definition of strict-contiguity will differ greatly from the results of those that return matches according to the original definition, we will only compare queries that are evaluated under the skip-till-next selection policy. Furthermore, as [25]. do not support any operators outside of the sequence-operator, we shall

also limit our queries to those that only test for sequence matches without special operators such as *AND*, *OR*, and *NOT*.

All experiments described below were conducted on a machine running Ubuntu 20.04 (LTS) x64 with 2 2.7GHz vCPUs and 16GB of RAM.

5.1.2 Datasets

For the datasets, we use the two largest real-world datasets that [25] use in their experiments, which are from the Business Process Intelligence (BPI) challenges by 4TU [35]. The first and largest dataset is that of the BPI 2017 challenge [36], which consists of 1,202,267 events relating to a loan application process of a Dutch financial institute. The second dataset is that of the BPI 2020 Challenge [37], which consists of 36,796 events relating to travel expense claims from a university. Both event logs are in eXtensible Event Stream (XES) format, which is an XML-based standard for event log files that is commonly used in the field of process mining [38]. Each XES-file can be regarded as single log that consists of traces, which in turn consist of events. Traces group the events according to certain characteristics. For example, in the case of the BPI 2020 dataset, each trace represents a single travel expense claim and the events comprise of steps taken to process a claim. Below is an excerpt of an event from a trace in the BPI 2020 dataset [37].

```
<event>
  <string key="id" value="st_step 149911_0"/>
  <string key="org:resource" value="STAFF MEMBER"/>
  <string key="concept:name" value="Request For Payment
    SUBMITTED by EMPLOYEE"/>
  <date key="time:timestamp" value="2017-01-12T14:52:53.000+01:00"/>
  <string key="org:role" value="EMPLOYEE"/>
</event>
```

From each event, the properties of interest are the name property **concept:name** and the timestamp property **time:timestamp**. The timestamp determines the temporal order of events, and the name is the event's identifier in queries.

5.1.3 Comparison

The solution proposed by Mavroudopoulos et al. [25] starts with a pre-processing step to extract special indexes of the form $\{trace_{id}, time_A\ time_B\}$ for each pair of

consecutive events (A, B) in a trace. Here, *trace*$_{id}$ represents the trace in which both events took place and the times indicate the timestamps at which the events occurred. These indexes are stored in the key-value database Cassandra [39]. The query executor then uses the resulting indexes to construct a response to sequence detection queries. It is important to note that Mavroudopoulos et al. only detect sequences in a single trace, with no overlaps between traces. This means that if an event A occurred in *trace*$_0$ and an event B occurred in *trace*$_1$, then they would not be able to detect the sequence *SEQ(A,B)* by analyzing the events in the consecutive traces *trace*$_0$ and *trace*$_1$. This poses a problem, as our solution is aimed at detecting pattern matches over larger ranges of events, and we fail to detect pattern matches between events that all hash to the same Bloom filter. If the number of events in a trace is small, such as in the case of BPI 2020, which has a mean of 5.3 events per trace [25], then we cannot detect any of the pattern matches [25] can detect. Likewise, [25] cannot detect any of the patterns our solution can detect. However, this problem only occurs if we read the original file as an input stream, and assume the events of all traces are ordered in terms of increasing timestamps. This generally is not the case for event logs of business processes, as an event belonging to a trace is often temporally followed by an event that occurs in a different trace. For example, in a business process of a web-shop, where each trace represents a separate order, a simplified and fictional event log could look as follows.

```
<trace>
    <event>
        <string key="concept:name" value="Order placed for item #1" />
        <date key="time:timestamp value="2021-01-01" />
    </event>
    <event>
        <string key="concept:name" value="Item #1 sent" />
        <date key="time:timestamp value="2021-01-05" />
    </event>
</trace>
<trace>
    <event>
        <string key="concept:name" value="Order placed for item #10" />
        <date key="time:timestamp value="2021-01-02" />
    </event>
    <event>
        <string key="concept:name" value="Item #10 sent" />
```

```
        <date key="time:timestamp value="2021-01-10" />
    </event>
</trace>
```

Here, the timestamp of an order for item #1 (2021-01-01) is not temporally followed by item #1 being sent (2021-01-05), but is instead followed by an order for item #10 (2021-01-02). We therefore pre-process the event log file and order the events according to increasing timestamps.

As described earlier, [25] detect sequences per trace instead of over the complete dataset, whereas our solution will find matches over combined traces. Modifying the event log file to include all events in a single trace was not found to be a reliable option for comparison as the solution by Mavroudopoulos et al. [25] was not optimized for this, and was further found to return only a single matching sequence per trace. To provide a fair comparison between the two solutions, we therefore use two different approaches when querying the event log using our solution.

- **Approach 1:** This approach returns all matching sequences over the full dataset, thereby ignoring the traces events belong to. This approach will return more results than that of [25].
- **Approach 2:** This approach conforms our solution to the use case of detecting sequences of events in traces. To this end, we pre-process the event log file and append a unique trace identifier to the name of each event as follows: *$trace_{id}$-event-name*. When querying for a sequence of events *SEQ(A, B)*, we then perform a query *SEQ(A_i, B_i)* for each $i \in [0, \tau]$, where τ is the total number of traces in the event log file. The time taken to execute the query is then the sum of the times taken to run the queries for all traces.

The experiments are conducted as follows. For each dataset, we generate 100 queries of different lengths, ranging from 2 to 8 events, and repeat each experiment 10 times. The results of running the experiments on the BPI 2020 [37] dataset are shown in Table 5.1, and the results of identical experiments on the BPI 2017 [36] dataset are shown in Table 5.2

Query Length	Mavroudopoulos et al.	Proposed Solution (Approach 1)	Proposed Solution (Approach 2)
2	0.0143s	0.0040s	0.0362s
3	0.0481s	0.0032s	0.0309s
4	0.0749s	0.0031s	0.0320s
5	0.1896s	0.0029s	0.0307s
6	0.3024s	0.0029s	0.0295s
7	0.4468s	0.0025s	0.0308s
8	0.6800s	0.0021s	0.0318s

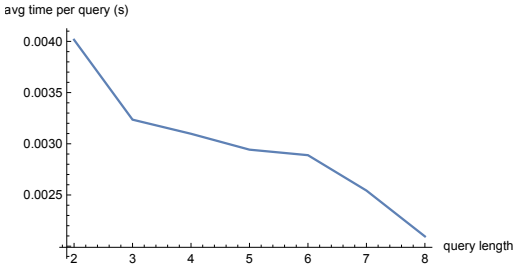
Table 5.1: BPI 2020 - Average query time (per query) of the solution proposed by [25]. compared to the solution proposed in this report, under different query lengths.

Query Length	Mavroudopoulos et al.	Proposed Solution (Approach 1)	Proposed Solution (Approach 2)
2	0.07654s	0.0705s	4.6239s
3	0.27445s	0.0668s	4.3471s
4	0.63936s	0.0665s	4.1347s
5	1.24048s	0.0644s	4.5579s
6	2.12963s	0.0644s	4.5179s
7	3.46484s	0.0619s	4.6149s
8	5.37686s	0.0585s	4.2729s

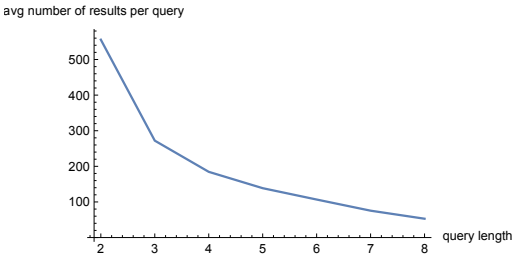
Table 5.2: BPI 2017 - Average query time (per query) of the solution proposed by [25]. compared to the solution proposed in this report under different query lengths.

The results above show that the exact solution by [25] takes longer to process a query as the query length increases, whereas the query time of our solution remains fairly constant, and even shows a slight downwards trend as the length of the query increases. This is not immediately the behavior that was expected, as the running time complexity generally increases as the complexity of the query increases. We therefore perform additional experiments using our solution to determine the cause as well as verify the validity of the results. Figure 5.1 shows two plots using data obtained from repeating the experiment on the BPI 2020 dataset using our standard approach to querying, *Approach 1*. Figure (a) contains the average time taken per query, and (b) shows the average number of events that were returned. There appears to be a small correlation between the number of events that are returned and the time it takes for our solution to process a query. The same experiments were repeated using the BPI 2017 dataset. The results of this experiment are presented in Figure 5.2 and show similar results. These results could be explained by the fact our solution can quickly discard queries that will not result in a match by looking only at nodes in the upper levels of tree. However, the correlation may also simply be caused by conditions within

the running environment of the program. Further experiments are therefore done using queries with an even larger query length. The results of this experiment, which were performed using the BPI 2020 dataset, are shown in Figure 5.3. In this experiment the query length range has been increased from 2-8 to 2-20. This figure shows a clearer correlation between the running time and size of the result set.

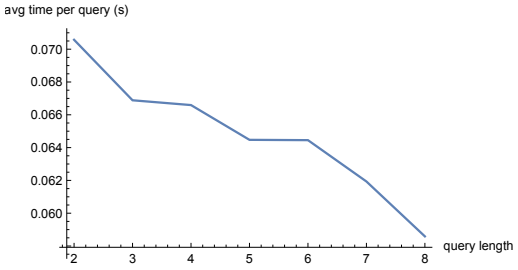


(a) The average time per query given the different query lengths

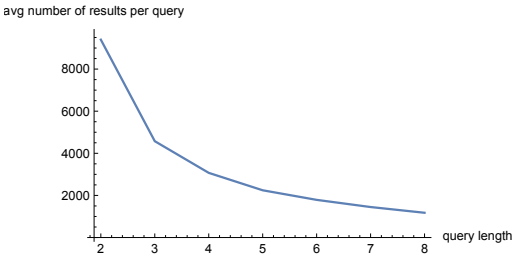


(b) The average number of results returned per query given the different query lengths 2

Figure 5.1: A decrease in query times as the number of returned results decreases, using the BPI 2020 dataset.

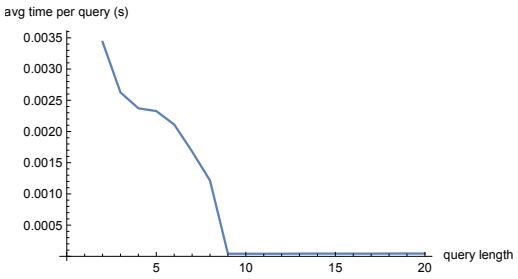


(a) The average time per query given the different query lengths

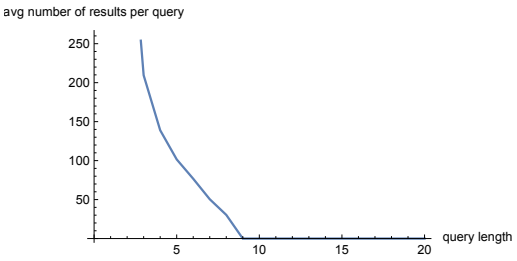


(b) The average number of results returned per query given the different query lengths 2

Figure 5.2: A decrease in query times as the number of returned results decreases, using the BPI 2017 dataset.



(a) The average time per query given the different query lengths



(b) The average number of results returned per query given the different query lengths

Figure 5.3: A decrease in query times as the number of returned results decreases, using the BPI 2020 dataset, but with a larger range of query lengths the the figures above.

5.2 Performance Analysis Using Synthetic Data

The datasets of the BPI challenges [35] used above are limited in that the number of distinct events is incredibly small (19 for BPI 2020 and 26 for BPI 2017), as well as the size of the datasets (36,796 for BPI 2020 and 1M for BPI 2017). It is therefore interesting to analyze how our proposed solution works on larger datasets with a greater number of distinct events. It is further also interesting to analyze how the different operators described in Chapter 4 perform under different conditions. To this end, we generate synthetic datasets by concatenating various public domain books from Project Gutenberg [40]. As described in Chapter 4, data streams are often said to have a Zipfian distribution, which is often also inherent to written text. These synthetic datasets are therefore more likely to mimic real-world streaming data. Each word in the resulting file is treated as an event in the data stream. Table 5.3 shows the properties of the datasets used within the following experiments.

Dataset Name	**Total Size**	**Distinct Items**
D1	511,178	30,439
D2	2,489,295	38,793
D3	5,489,709	52,451
D4	1,705,977	87,130

Table 5.3: Properties of the synthetic datasets.

5.2.1 Construction Time

We first analyze how long it takes to construct the data structure under datasets of different sizes. This experiment was repeated 100 times per dataset and the resulting average values are shown in Table 5.4. The configuration of the data structure is based on the following input values: $M = 2$GB, $N =$ total size of the dataset, $C = 100$, $fpq = 0.0001$, and $s = 3$. These settings will be used for all of the experiments in this section. Note that the data structure is continuously updated as events arrive. These construction times are therefore only an indication of the possible construction and update overhead.

Dataset Name	AVG Construction Time
D1	4.58s
D2	11.09s
D3	51.04s
D4	14.78s

Table 5.4: Average construction times of the data structure.

5.2.2 Query Types

We now look at how different types of queries perform under the different datasets. For these experiments we run 100 random queries of length 3 and repeat each experiment 10 times. All queries are configured to return all matching intervals and are evaluated under both the strict-contiguity as well as the skip-till-next selection policies. The different query types are shown in Tables 5.5 and 5.6 below. Here, the events A, B, and C are replaced with randomly generated events from the dataset at runtime. The values presented in the tables are the average query times per query. Table 5.5 shows the query times under the strict-contiguity selection policy and Table 5.6 shows the results under a skip-till-next selection policy.

Query Type	AVG Time / D1	AVG Time / D2	AVG Time / D3	AVG Time / D4
SEQ(A, B, C)	0.001s	0.005s	0.015s	0.003s
SEQ(A, OR(B, C))	0.013s	0.045s	0.083s	0.004s
SEQ(A, AND(B, C))	0.003s	0.008s	0.025s	0.004s
SEQ(A, NOT(B), C)	0.002s	0.004s	0.010s	0.002s

Table 5.5: Query evaluation under strict-contiguity.

Query Type	AVG Time / D1	AVG Time / D2	AVG Time / D3	AVG Time / D4
SEQ(A, B, C)	0.015s	0.266s	1.650s	0.029s
SEQ(A, OR(B, C))	0.037s	0.944s	6.421s	0.043s
SEQ(A, AND(B, C))	0.023s	0.562s	3.510s	0.029s
SEQ(A, NOT(B), C)	0.012s	0.212s	1.051s	0.018s

Table 5.6: Query evaluation under skip-till-next.

5.3 Discussion

We now provide a discussion of the results from Sections 5.1 and 5.2. The goal of the experiments in Section 5.1 was to determine how the solution compares to a method that provides exact answers to sequence detection queries. The results are promising, as even the most inefficient approach taken to answering queries (Approach 2) can still compete with the solution of [25]. Our original approach, which constructs all possible matches by ignoring traces, provides quite a significant improvement over the exact solution. Where the exact solution gets slower as the length of the query grows, our solution becomes slightly faster, as it can quickly detect when a pattern cannot be matched. It is further interesting to see that the complexity of our solution is not determined by the complexity of the query, but by the number of results that match it. However, while the results on real-life event logs with a very small number of distinct events look promising, datasets with a much larger vocabulary make the solution slower.

The experiments in Section 5.2 were aimed analyzing the behavior of the data structure and queries under a dataset that more closely resembles actual streaming data. While the results are a little slower than for the BPI datasets, they are still reasonable. A closer look at the returned results (not presented here) showed that the more expensive queries, such as *SEQ(A, OR(B,C))* under the dataset *D2* had quite a high number of matching results, whereas faster queries such as *SEQ(A,B,C)* under strict contiguity returned some of the smallest result sets. It is further interesting to see that the operators used in a query pose no real difference to the query time, except for the OR-operator which can be explained by the fact that it matches more results than other operators.

Chapter 6

Conclusion

The goal of this thesis was to approximate answers to pattern matching queries over big data streams, as is common in the field of Complex Event Processing [3]. A solution that provides an exact answer to such queries must generally either store or process the full data stream in search for matches. An approximate solution, that consists only of a small summary of the underlying data, could provide answers significantly faster. Since pattern matching queries assume an underlying temporal order between events to detect when events occur in sequence, our goal was to build an approximate data structure from which the underlying temporal order of events can be inferred. To this end, we proposed to split the stream into small windows of events, that each hash to a different Bloom filter. These Bloom filters are then stored in a tree-based structure to provide fast answers to pattern matching queries. An event B that is hashed to the right of a node that contains an event A is assumed to have occurred at a later timestamp than A. Pattern matching queries can then be answered by comparing the temporal order of the events in the query. The resulting data structure is flexible, as it can support several operators that are common to pattern matching. It has further shown to be faster at detecting patterns than [25], which is one of the latest works in the area of Complex Event Processing. We now discuss the limitations of the solution and provide areas for future work.

Limitations

Even though the solution presented in this report is faster than the exact solution of [25], several shortcomings of our solution limit its usefulness for Complex Event Processing. First, as described in Chapter 2, Complex Event Processing solutions such as SASE [5] extract the matching sequences of events in the stream,

such that they can be analyzed further elsewhere. However, it is not possible to reconstruct the original sequence of events using our approximate solution. A second limitation of our solution is that it only finds matching events whose names correspond to the event names provided in the query. Solutions such as SASE allow the user to limit the matching events further by filtering out events that do not match predicates on the attributes of events. Since our solution cannot associate attributes with events, it cannot provide the same flexibility as exact pattern matching solutions. However, if events are known to have specific attributes, then we can concatenate these attributes' values to the event's name. The second approach described in Section 5.1, where we concatenate the trace ID to the event's name, is an example of this. While this technique allows us to find events that match a specific attribute, it no longer allows us to query events without providing a specific value for the attribute. Furthermore, the attributes of events are not only filter out data, but they can also provide valuable information regarding the cause of an anomaly. For example, if a pattern matching query detects a sequence of a events that could signal fraud, then it would be meaningful to view attribute values that could indicate who was involved. Thus, the solution described in this report may not be suitable for all Complex Event Processing scenarios.

Even though our solution may not be suitable as a standalone Complex Event Processor, it can be meaningful in combination with other persistence methods that are too expensive to query in search for patterns. For example, if streaming data is backed up to secondary storage, then finding the matching intervals of a pattern using our solution can make it much more efficient to the retrieve persisted data.

Another limitation of the solution is the storage space it requires. While the solution uses less storage than would have been the case if the full data stream were stored, the number of Bloom filters it requires is still linear in the number of events in the stream. This space requirement may be too much for many applications, especially if the solution is limited to running on a single machine.

Future Work

As described above, the space complexity of the current solution may grow unbounded with the size of the stream. It would therefore be interesting to incorporate time-decay models, such as the sliding-window model described in Chapter 2. Another option for future work is related to the smart configuration of the Bloom filters, as it now relies on the user predicting what the full size of the stream will

be. It would be interesting to make use of learning algorithms that analyze the stream as it arrives and bases its configuration on statistics and other characteristics of the stream seen thus far. Finally, as Complex Event Processing currently relies on exact solutions, it would be interesting to find other sketches that can efficiently detect patterns over streams, preferably using even less storage space than the sketch provided in this report.

Bibliography

[1] Lange, Moritz, Arne Koschel, and Irina Astrova. "Dealing with Data Streams: Complex Event Processing vs. Data Stream Mining." International Conference on Computational Science and Its Applications. Springer, Cham, 2020.

[2] Aggarwal, Charu C. Data mining: the textbook. Springer, 2015.

[3] Giatrakos, Nikos, et al. "Complex event recognition in the big data era: a survey." The VLDB Journal 29.1 (2020): 313-352.

[4] Cormode, Graham, et al. "Synopses for massive data: Samples, histograms, wavelets, sketches." Foundations and Trends in Databases 4.1–3 (2012): 1-294.

[5] Gyllstrom, Daniel, et al. "SASE: Complex event processing over streams." arXiv preprint cs/0612128 (2006).

[6] Babcock, Brian, et al. "Models and issues in data stream systems." Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. 2002.

[7] Garofalakis, Minos, Johannes Gehrke, and Rajeev Rastogi, eds. Data stream management: processing high-speed data streams. Springer, 2016.

[8] Vitter, Jeffrey Scott. "External memory algorithms and data structures: Dealing with massive data." ACM Computing surveys (CsUR) 33.2 (2001): 209-271.

[9] Zhang, Hao, et al. "In-memory big data management and processing: A survey." IEEE Transactions on Knowledge and Data Engineering 27.7 (2015): 1920-1948.

[10] Bloom, Burton H. "Space/time trade-offs in hash coding with allowable errors." Communications of the ACM 13.7 (1970): 422-426.

[11] Datar, Mayur, et al. "Maintaining stream statistics over sliding windows." SIAM journal on computing 31.6 (2002): 1794-1813.

[12] Broder, Andrei, and Michael Mitzenmacher. "Network applications of bloom filters: A survey." Internet mathematics 1.4 (2004): 485-509.

[13] Papapetrou, Odysseas, Wolf Siberski, and Wolfgang Nejdl. "Cardinality estimation and dynamic length adaptation for bloom filters." Distributed and Parallel Databases 28.2 (2010): 119-156.

[14] Aappleby. github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp. (Last accessed: 5th of July, 2021).

[15] Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. "Introduction to automata theory, languages, and computation." Acm Sigact News 32.1 (2001): 60-65.

[16] Yu, Fang, et al. "Fast and memory-efficient regular expression matching for deep packet inspection." Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems. 2006.

[17] "Snort - Network Intrusion Detection & Prevention System." https://www.snort.org. [Last accessed July, 2021].

[18] Wang, Xiang, et al. "Hyperscan: a fast multi-pattern regex matcher for modern cpus." 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019.

[19] Gyllstrom, Daniel, et al. "SASE: Complex event processing over streams." arXiv preprint cs/0612128 (2006).

[20] Diao, Yanlei, Neil Immerman, and Daniel Gyllstrom. "Sase+: An agile language for kleene closure over event streams." UMass Technical Report (2007).

[21] Zhang, Haopeng, Yanlei Diao, and Neil Immerman. "On complexity and optimization of expensive queries in complex event processing." Proceedings of the 2014 ACM SIGMOD international conference on Management of data. 2014.

[22] FlinkCEP Complex event processing for Flink. https://ci.apache.org/projects/flink/flink-docs-release-1.13 /docs/libs/cep/. [Last accessed July, 2021].

[23] Apache Flink: Stateful Computations over Data Streams. https://flink.apache.org/. [Last accessed July, 2021].

[24] Mei, Yuan, and Samuel Madden. "Zstream: a cost-based query processor for adaptively detecting composite events." Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. 2009.

[25] Mavroudopoulos, Ioannis, et al. "Sequence detection in event log files." EDBT. 2021.

[26] Peng, Yanqing, et al. "Persistent bloom filter: Membership testing for the entire history." Proceedings of the 2018 International Conference on Management of Data. 2018.

[27] Alexiou, Karolina, Donald Kossmann, and Per-Åke Larson. "Adaptive range filters for cold data: Avoiding trips to siberia." Proceedings of the VLDB Endowment 6.14 (2013): 1714-1725.

[28] Lillis, David, Frank Breitinger, and Mark Scanlon. "Hierarchical Bloom filter trees for approximate matching." arXiv preprint arXiv:1712.04544 (2017).

[29] Breitinger, Frank, and Harald Baier. "Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2." International conference on digital forensics and cyber crime. Springer, Berlin, Heidelberg, 2012.

[30] Moraru, Iulian, and David G. Andersen. "Exact pattern matching with feed-forward bloom filters." Journal of Experimental Algorithmics (JEA) 17 (2012): 3-1.

[31] Emerson, E. Allen, and Joseph Y. Halpern. "Decision procedures and expressiveness in the temporal logic of branching time." Journal of computer and system sciences 30.1 (1985): 1-24.

[32] Cormode, Graham, and Shan Muthukrishnan. "Summarizing and mining skewed data streams." Proceedings of the 2005 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 2005.

[33] Lü, Linyuan, Zi-Ke Zhang, and Tao Zhou. "Zipf's law leads to Heaps' law: Analyzing their relation in finite-size systems." PloS one 5.12 (2010): e14139.

[34] Schütze, Hinrich, Christopher D. Manning, and Prabhakar Raghavan. Introduction to information retrieval. Vol. 39. Cambridge: Cambridge University Press, 2008.

[35] BPI Challenges. https://www.tf-pm.org/competitions-awards/bpi-challenge [Last accessed July, 2021].

[36] BPI Challenge 2017. https://data.4tu.nl/articles/dataset/BPI_Challenge_2017/12696884/1 [Last accessed July, 2021].

[37] BPI Challenge 2020. https://data.4tu.nl/collections/BPI_Challenge_2020/5065541 [Last accessed July, 2021].

[38] IEEE 1849-2016 XES Standard. https://xes-standard.org [Last accessed July, 2021].

[39] Apache Cassandra. https://cassandra.apache.org [Last accessed July, 2021].

[40] Project Gutenberg. https://www.gutenberg.org [Last accessed July, 2021].