Sequence detection in event log files

Ioannis Mavroudopoulos
Aristotle University of Thessaloniki, Greece
mavroudo@csd.auth.gr

Theodoros Toliopoulos
Aristotle University of Thessaloniki, Greece
tatoliop@csd.auth.gr

Christos Bellas
Aristotle University of Thessaloniki, Greece
chribell@csd.auth.gr

Andreas Kosmatopoulos
Aristotle University of Thessaloniki, Greece
akosmato@csd.auth.gr

Anastasios Gounaris
Aristotle University of Thessaloniki, Greece
gounaria@csd.auth.gr

ABSTRACT

Sequential pattern analysis has become a mature topic, with a lot of techniques for a variety of sequential pattern mining-related problems. Moreover, tailored solutions for specific domains, such as business process mining, have been developed. However, there is a gap in the literature for advanced techniques for efficient detection of arbitrary sequences in large collections of activity logs. In this work, we make a threefold contribution: (i) we propose a system architecture for incrementally maintaining appropriate indices that enable fast sequence detection; (ii) we investigate several alternatives for index building; and (iii) we compare our solution against existing state-of-the-art proposals and we highlight the benefits of our proposal.

1 INTRODUCTION

Event log entries refer to timestamped event metadata and can grow very large; e.g., even a decade ago, the amount of log entries of a single day was at the order of terabytes for certain organizations, as evidenced in [3]. Due to their timestamp, the log entries can be regarded as event sequences that follow either a total or a partial ordering. The vast amount of modern data analytics research on such sequences is divided into two broad categories.

The first category comprises sequential pattern mining [11], where a large set of sequences is mined to extract subsequences that meet a variety of criteria. Such criteria range from frequent occurrence, e.g., [23, 33] to importance and high-utility [12]. In addition, there are proposals that examine the same problem of finding interesting subsequences in a huge single sequence, e.g., [25]. However, these techniques fail to detect arbitrary patterns, regardless of whether they are frequent or interesting; e.g., they are tailored to a setting where a support threshold is provided and only subsequences meeting this threshold are returned, whereas we target a different problem, that is to return all subsequence occurrences given a pattern.

The second category of existing techniques deals with detecting event sequences on the fly and comprises complex event processing (CEP). CEP is a mature field [14, 34] and supports several flavors of runtime pattern detection. We aim to solve a similar problem to CEP but tailored to a non-streaming case, where pattern queries are submitted over potentially very large log databases. Since logs continuously arrive, we account for periodic index building and we support pattern matching where the elements in the pattern are not strictly in a sequence in the logs, e.g., in the log sequence ABBACC, we are interested in detecting the occurrence of the pattern ABC despite the fact that in the original sequence other elements appear in between the elements in the searched pattern. Given that we relax the constraint of strict contiguity, techniques based on suffix trees and arrays are not applicable. Contrary to CEP, we aim to detect all pattern occurrences efficiently and not only those happening now.

In summary, our contribution is threefold: (i) we propose a system architecture for incrementally maintaining appropriate indices that enable fast sequence detection and exploration of pattern continuation choices; (ii) we investigate several alternatives for index building; and (iii) we compare our solution against existing suffix array-based proposals, focusing on logs from business processes, showing that not only we can achieve high performance during indexing but we also support a broader range of queries. Compared to other state-of-the-art solutions, like Elasticsearch, we perform more efficient preprocessing, while we provide faster query responses to small queries remaining competitive in large queries in the datasets examined; additionally, we build on top of more scalable technologies, such as Spark and Cassandra, and we inherently support pattern continuation more efficiently. Finally, we provide the source code of our implementation.

The structure of the remainder of this paper is as follows. We present the notation and some background next. In Section 3, we introduce the architecture along with details regarding preprocessing and the queries we support. We discuss the index building alternatives in Section 4. The experimental evaluation is in Section 5. In the last sections, we discuss the related work, open issues and present the conclusions.

2 PRELIMINARIES

In this section, we first present the main notation and then we briefly provide some additional background with regards to the techniques against which we compare our solution.

2.1 Definitions and notation

We aim to detect sequential patterns of user activity in a log, where a log contains timestamped events. The events are of a specific type; for instance, in a log recording the user activity on the web, an event type may correspond to a specific type of click and in business processes, an event corresponds to the execution of a specific task. The events are logically grouped in sets, termed as cases or sessions or traces[1], which may correspond to a specific session or the same process instance or, in the generic case, grouped by other user-defined criteria. More formally:

[1] In this work, we use the terms trace, case and session interchangeably.

	Our Method	**Exact rooted subtree matching**
Supported policy	SC, STNM	SC, Tree Matching
Database usage	Yes	No
Preprocess rationale	Indexing of all possible pairs	Indexing of all the subtrees
Query processing rationale	Combination/merging of results of pairs in the query sequence	Binary search in the subtrees space

Table 1: Differences between the technique in [19] and our method

Symbol	Short description
L	the log containing events
A	the set of activities (or tasks), i.e., the event types
E	the set of all events
C	the set of cases, where each case corresponds to a single logical unit if execution, i.e., a session, a trace of a specific business process instance execution, and so on
ev	an event ($ev \in E$), which is an instance of an event type
ts	the timestamp of an event (also denoted as $ev.ts$)
l	the size of A, $\|A\|$
n	the maximum size of a case
m	the size of C, $\|C\|$

Table 2: Frequently used symbols and interpretation

Definition 2.1. (Event Log) Let A be a finite set of activities (tasks). A log L is defined as $L = (E, C, \gamma, \delta, ts, \leq)$ where E is the finite set of events, C is the finite set of Cases, $\gamma : E \rightarrow C$ is a surjective function assigning events to Cases, $\delta : E \rightarrow A$ is a surjective function assigning events to activities, ts records the timestamp denoting the recording of task execution and $\leq$ is a strict total ordering over events belonging to a specific case, normally based on execution timestamps.

The notion of timestamp requires some further explanation. In sessions like web user activity and similar ones, events are usually instantaneous. However, this is not the case in task executions in business processes. In the latter case, the timestamp refers to either the beginning of the activity or its completion, but in any case, logging needs to be consistent. The duration of activities can only be estimated implicitly and not accurately from the difference between the timestamps of an event and its successor, because there may be delays between the completion of an activity and the beginning of the execution of the next activity downstream. However, systematic analysis involving task duration can be conducted only if the exact task duration is captured, which requires extensions to the definition above. Such extensions is out of the scope of this work and are orthogonal to our contributions.

Table 2 summarizes the main notation; $|A|$ is denoted as l, the maximum size of a case is denoted as n, and the size of the set of cases $|C|$ is denoted as m.

Next, to provide the context of the queries we aim to support, we define the two main types of event sequence detection that we employ in this work:

Strict contiguity (SC) , where all matching events must appear strictly in a sequence one after the other without any other non-matching events in-between. This definition is widely employed in both exact subsequence matching and CEP systems and stream processing engines, such as Flink [6].[2] For example, SC applies when we aim to detect pattern occurrences, where a search for a product on an e-shop website is immediately followed by adding this product to the cart without any other action in between.

Skip-till-next-match (STNM) , where strict contiguity is relaxed so that irrelevant events are skipped until we detect the next matching event of the sequential pattern [34]. STNM is required when, for example, we aim to detect pattern occurrences where after three searches for specific products there is no any purchase eventually in the same session.

Example: let us assume that we look for a pattern `AAB`, where `A,B` are activities. Let us also assume that a log contains the following sequence of events `<AAABAACB>`, where the timestamps are implicit by the order of each event. SC detects a pattern occurrence starting at the 2nd position, whereas STNM detects two occurrences; the first one contains the events at the 1st, 2nd and 4th position, while the second one contains the events at the 5th, 6th and 8th position. Note that other types of event sequence detection allow for additional and overlapping results, e.g., to detect a pattern in the 1st, 3rd and 8th position, as discussed at the end of this work [34].

2.2 Exact rooted subtree matching in sublinear time

Strict contiguity (SC) is directly relevant to subsequence matching and tree-based techniques have been used for a long time for finding sub sequences in large datasets. Suffix trees and suffix arrays are commonly used to this end. The method presented in [19] can find subtrees in sublinear time and it has been used to detect possible continuations of a given event sequence in business processes in [27].

In a nutshell, the technique in [19] solves the problem of finding the occurrences of a subtree with m modes in a tree T with n nodes in $O(m + logn)$, after pre-processing the tree. First, the string W corresponding to T is created; this is achieved through traversing the tree in a preorder manner and adding a 0 every time we recur to a previous level. This yields a W of length equal to $2n$. W is then used to create a suffix array A, in which the starting positions of the $2n$ suffices are specified. After discarding those starting with 0, we end up with n suffices. The main property of A is that suffices are sorted by the nodes' order. The subtree to be searched in T is first mapped to a preorder search string, and then a binary search in A is performed.

In Table 1 we present the high-level differences between this method and our proposal. We rely on simple indexing employing a database backend, while, during query processing, the main

[2] https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html

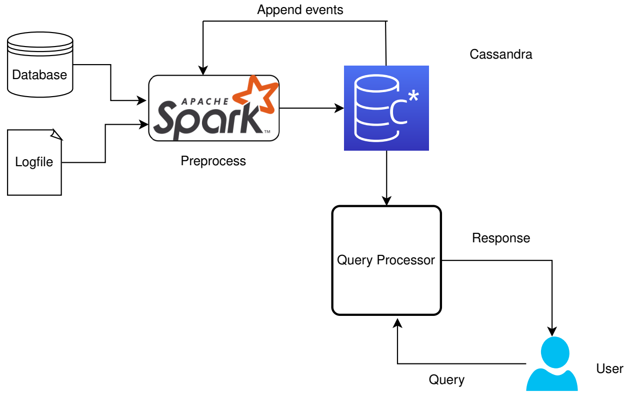

Figure 1: Architecture overview

operation is merging and post-processing of sorted lists, as explained in the next sections. More importantly, we support both STNM and SC pattern types.

3 SYSTEM ARCHITECTURE

There exist several CEP proposals along with fully-fledged prototypes and complete systems that allow users to query for Strict contiguity (SC) or Skip-till-next-match (STNM) patterns, but these are operating in a dynamic environment over a data stream. Therefore, we need to develop a system that can receive adhoc pattern queries over a large collection of logs and process them in an efficient manner. These queries will be defined later in this section and are broadly divided into three main categories (statistics, pattern detection and pattern expansion). We focus on offline pattern detection, but we account for the fact that the logs are constantly growing bigger and bigger. This entails that any practical approach needs to be incremental, i.e., to support the consideration of new logs periodically.

The overview of our proposed architecture is shown in Figure 1. There exists a database infrastructure containing old logs and, periodically, new logs are appended. There are two main components in the architecture. The pre-processing component constructs and/or updates an inverted index that is leveraged during query processing. This index is stored in a key-value database to attain scalability. In our implementation, we have chosen Cassandra[3], because of its proven capability to deal with big data and offer scalability and availability without compromising performance. However, any key-value store can be used in replacement.

The second component is the query processor, which is responsible for receiving user queries, retrieving the relevant index entries and constructing the response.

These two components are described in more detail in the remainder of this section, while indexing is discussed in the next section.

3.1 The pre-processing component

The log database has a typical relational form, where each record corresponds to a specific event. More specifically, each row in the log database contains the trace identifier, the event type, the timestamp and any other application-specific metadata that play no role in our generic solution. The second input of the pre-processing component contains the more recent log entries that have not been indexed yet. For example, if the index is updated on a daily basis, the log file is expected to contain from a few thousand of events up to several millions.

trace: <(A,1), (A,2), (B,3), (A,4), (B,5), (A,6)>		
Pair	**Strict Contiguity**	**Skip till next match**
(A,A)	(1,2)	(1,2),(4,6)
(B,A)	(3,4),(4,5)	(3,4),(5,6)
(B,B)	-	(3,5)
(A,B)	(2,3),(4,5)	(1,3),(4,5)

Table 3: Pairs created per different policy.

Pattern indexing and querying is applied per trace. In other words, for each distinct trace, a large sequence of all its events is constructed sorted by the event timestamps. To this end, the recent logfile is combined with the log database. In addition, and since the trace may span many indexing periods, new log entries need to combined with already indexed events in the same trace in a principled manner to avoid duplicates. If new logged events belong to a trace already started, we extract stored information from the indexing database (the exact procedure will be described in detail shortly).

Based on these trace sequences, we build an inverted indexing of all event pairs. That is, we extract all event pairs from each trace, and for each pair we keep its trace along with the corresponding pair of timestamps. This information is adequate to answer pattern queries efficiently, where these queries may not only refer to pattern detection, but frequency counts and prediction of next events, as discussed in Section 3.2.1. The index contains entries of the following format: (A,B):{(trace12, 2,5),(trace12, 7,11), (trace15,1,6),. . . }. In this example, the pair of event types (A,B) has appeared twice in trace12 at timestamps (2,5) and (7,11), respectively, and once in trace15.

The pre-processing component is implemented as a Spark Scala program to attain scalability. Next, we delve into more details regarding pre-processing subparts.

3.1.1 Creation of event pairs. There are more than one ways to create pair of events in a trace, which depends on the different policy applied. We have already given two policies, namely SC and STNM, which impact on how pairs are created.

Let us assume that a specific trace contains the following sequence of pairs of event types and their timestamps: trace: <(A,1), (A,2), (B,3), (A,4), (B,5), (A,6)>. Table 3 shows the pairs created per different policy. This example shows a simplified representation of the inverted indexing. SC detects only the pairs of events that are consecutive. There is no pair pair (B,B) because there is an event (A) between the two Bs in the trace. As expected, the SC policy creates less pairs per trace and is also easier to implement.

STNM skips events until it finds a matching event, but there are no overlapping pairs, the timestamps of which are intertwined. For example, regarding pair (A,B), we consider only the (1,3) pair of timestamps and not (2,3). The complexity of pair creation in STNM is higher and there are several alternatives that are presented in Section 4.

A final note is that our approach can work even in the absence of timestamps. In that case, the position of an event in the sequence can play the role of the timestamp.

[3] https://cassandra.apache.org/

Algorithm 1 Update index

```
1:  Input : new_events
2:  traces ← transform new_events to traces as in the Seq table
3:  temp ← LastChecked table joined with traces
4:  new_pairs ← [ ]
5:  for all trace in traces do
6:      extract events
7:      for all ($ev_a, ev_b$) do
8:          $lt$ ← temp.get($ev_a, ev_b$).$last\_completion$ for the same trace
9:          if $ev_a.ts > lt$ then
10:             new_pairs += create_pairs($ev_a, ev_b$)
11:         end if
12:     end for
13: end for
14: append new_pairs to the Index table
```

3.1.2 Tables in the indexing database. The pre-processing phase creates and updates a set of tables, which can all be stored in a key-value store, such as Cassandra. The first one contains the trace sequence, so that there is no need to be reconstructed from scratch every time is needed, e.g., to append new events. The second one is the index presented earlier. The other tables are auxiliary ones, which are required during index creation and query answering.

- **Seq** with key: $trace_{id}$ and value: $\{(ev_a, ts_a), (ev_b, ts_b), ...\}$. This table contains all traces that are indexed. It is used to create and update the main index; new events belonging to the same trace are appended to the value list.
- **Index** with a complex key: (ev_a, ev_b) and value containing a list of triples: $\{(trace_{id}, ts_a, ts_b), ...\}$. This is the inverted index, which is the main structure used in query answering.
- **Count** with key a single event type: ev_a and value a list of triples: $\{(ev_b, sum\_duration, total\_completions)$, $(ev_c, sum\_duration, total\_completions), ...\}$. For each event ev_a, we keep a list which contains the total duration of completions for a pair (ev_a, ev_x) and the total number of completions. This is used to find the most frequent pairs where an event appears first and also we can leverage the duration information in case further statistics are required.
- **Reverse Count**, which has exactly the same form of key and value with **Count**, but the statistics refer to pairs that have the event in the key as their second component
- **LastChecked** with complex key a pair (ev_a, ev_b) and value a list of pairs: $\{(trace_{id}, last\_completion), ...\}$. The length of the list is the number of traces in which the pair (ev_a, ev_b) appears. The *last_completion* field keeps the last timestamp of ev_b in a pair detection. This table is used to prevent creating and indexing pairs more than once.

3.1.3 Index update. In dynamic environments, new logs arrive continuously, but the index is not necessarily updated upon the arrival of each new log record. New log events are batched and the update procedure is called periodically, e.g., once every few hours. To avoid the generation of duplicates, the LastChecked table introduced above plays a crucial role. The index update rationale is illustrated in Algorithm 1.

In line 3 of the algorithm, we extract the LastChecked table and keep only its part that refers to the traces that their id appears in new events. In line 10, the *create_pairs* procedure is not specifically described here but can be any of the algorithms presented in Section 4 depending also on the policy employed.

A subtle point is that the index may grow very large. To mitigate this, a separate index table can be used for different periods, e.g., for different months. In addition, the traces corresponding to completed sessions can be safely pruned from the Seq table, along with the corresponding value entries in LastChecked.

Algorithm 2 Pattern detection

```
procedure GETCOMPLETIONS($< ev_1, ev_2, \ldots, ev_p >$)
    previous ← Index.get($ev_1, ev_2$)
    for $i$ =2 to $p - 1$ do
        $idx\_completions$ ← Index.get($ev_i, ev_{i+1}$)
        for all $c$ in $idx\_completions$ grouped by trace do
            new ← [ ]
            for all $pr$ in previous for the same trace do
                if $pr.last\_event.ts == c.first.ts$ then
                    append $c$ to $pr$ and add to new
                end if
            end for
            previous ← new
        end for
    end for
    return previous
end procedure
```

3.2 The query processor component

The architecture described can support a range of pattern queries that are presented in Section 3.2.1. In Section 3.2.2, we give different solutions for predicting subsequent events in a pattern while trading off accuracy for response time.

3.2.1 Different type of queries. The query input is a pattern (i.e., a sequence) of events $< ev_1, ev_2, ev_3, \ldots, ev_p >$ for all supported query types. The query types in ascending order of complexity are as follows:

- **Statistics.** This type of query returns statistics regarding each pair of consecutive events in the pattern. The statistics are those supported by the Count table, namely number of completions and average duration. Also, from the LastChecked table, the timestamp of the last completion can be retrieved. The pairwise statistics can provide useful insights about the behavior of the complete pattern with simple post-processing and without requiring access to any other information. For example, the minimum number of completions of a pair provides an upper bound of the completions of the whole pattern in the query. Also, the sum of the average durations gives an estimate of the average duration of the whole pattern. Finally, the number of completions could be more accurately bounded if all pairs in the pattern are considered instead of the consecutive ones only; clearly, there is a tradeoff between result accuracy and query running time in this case.
- **Pattern Detection.** This query aims to return all traces that contain the given pattern. Query processing starts by searching for all the traces that contain event pair (ev_1, ev_2). At the next step, the technique keeps only the traces where the same instance of ev_2 is followed by e_3 to the pattern; to this end, it finds all the traces that contain (ev_2, ev_3) and keeps those for which ev_2 has the same timestamp in both cases. Up to now, we have found the traces that contain

Algorithm 3 Accurate exploration of events

```
1: Input: pattern $e_1, ev_2, ..., ev_p$
2: candidate_events ⟵ from Count Table get all events that has $ev_p$ as first event
3: propositions⟵ [ ]
4: for all $ev$ in candidate events do
5:     tempPattern ⟵ append $ev$ to pattern
6:     candidate_pairs ⟵ getCompletions(TempPattern)
7:     proposition ⟵ apply time constraints to candidate pairs (optional)
8:     append proposition to propositions
9: end for
10: return propositions sorted according to Equation (1)
```

(ev_1, ev_2, ev_3). The execution continues in the same way up to ev_p, as shown in Algorithm 2. It is trivial to extend the results with further information, such as the starting and ending timestamp.

- **Pattern Continuation.** Another aspect for pattern querying is exploring which events are most likely to extend the pattern in the query. This has several applications, such as predicting an upcoming event given partial pattern in an incomplete trace, or computing the probability of an event to appear in a pattern, based on prior knowledge. In this query, the response contains the most likely events that can be appended to the pattern, based on a scoring function. Equation 1 gives a score for a proposed event. Total completions refer to the frequency of this event to follow the last event in the query pattern, while average duration favors events that appear closer to the pattern in the original traces.

$$Score = \frac{total\_completions}{average\_duration} \qquad (1)$$

3.2.2 Pattern Continuation Alternatives. Exploring events for pattern continuation can be computationally intensive, depending on the log size. Some times, we want accurate responses, while in other cases it is adequate to receive coarser insights so that we can trade accuracy for response time. We present three alternative ways of exploring events, namely one accurate, one fast heuristic and one hybrid that is in between the previous two.

- **Accurate.** In Algorithm 3, we present the outline of this method. In line 2 we use the Count table to find all event pairs that begin with the last event of the pattern and collect the second events of the pairs in the candidate_events list. The procedure getCompletions is already provided in Algorithm 2. We also allow for constraints in the average time between the last event in the pattern and the appended event; these constraints are checked in line 7. The strong point of this approach is that all pattern continuations are accurately checked one-by-one; the drawback is that the response time increases rapidly with the size of log files and the number of different events.

- **Fast.** In Algorithm 4 we perform a heuristic search. We start by finding the upper bound of the total times the given pattern has been completed (lines 3-8). Then, for every possible event ev, we approximate the upper bound if this event is added at the end of the pattern, by keeping the minimum between the max_completions and the total completions of ev (line 11). The strong point of this approach is that it is fast, since it extracts precomputed statistics from the indexing database but the results rely only on approximations.

Algorithm 4 Fast exploration of events

```
1: Input: Pattern $e_1, ev_2, ..., ev_p$
2: max_completions ⟵ $\infty$
3: for all $(ev_i, ev_{i+1})$ in pattern do
4:     count ⟵ Count Table get $ev_i, ev_{i+1}$
5:     if count.total_completions < max_completions then
6:         max_completions ⟵ count.total_completions
7:     end if
8: end for
9: propositions ⟵ [ ]
10: for all ev in Count.get($ev_p$) do
11:     completions ⟵ min(max_completions,ev.total_completions)
12:     append (ev.event,completions,ev.average_duration) to propositions
13: end for
14: return propositions sorted according to Equation (1)
```

Algorithm 5 Hybrid exploration of events

```
1: Input: Pattern $e_1, ev_2, ..., ev_n$
2: Input: topK
3: fast_propositions ⟵ run Algorithm 4 for the input pattern
4: propositions ⟵ run Algorithm 3 for topK of fast_propositions
5: return propositions sorted according to Equation (1)
```

- **Hybrid.** Lastly, in Algorithm 5 we perform a trade off between accuracy and response time. This flavor receives topK as an input parameter. First, the fast alternative runs to provide an initial ranking of possible pattern continuations. Then, for the topK intermediate results, the accurate method runs. In this flavor, the trade-off is configurable. Setting topK to l, the technique degenerates to the accurate, while setting topK to 0 is equal to the fast only alternative.

4 ALTERNATIVES FOR INDEXING EVENT PAIRS

The indexing of event pairs largely depends on the pattern detection policy. For the Strict contiguity (SC) policy, the process is straightforward. For Skip-till-next-match (STNM), there are three flavors. Each trace is processed separately in parallel using Spark. Below, we show the processing per trace; therefore the overall complexity for the complete log needs to be increased by a factor of O(m). The techniques presented in this section refer to the implementation of the *create_pairs* procedure in Alg. 1.

4.1 Strict Contiguity

This method is straightforward: we parse each trace and we add the consecutive trace events in the index. The complexity is $O(n)$, where n is the size of a trace sequence in the log file.

4.2 Skip-till-next-match

The three different ways to calculate the event pairs using the skip-till-next-match (STNM) strategy have different perspectives.

Algorithm 6 Parsing method (per trace)

```
    checkedList ← [ ]
    for i in (0, trace.size-1) do
        inter_events ← [ ]
        if $ev_i$.type not in checkedList then
5:          for j in (i,trace.size) do
                if $ev_i$.type==$ev_j$.type then
                    update inv_index with ($ev_i$, $ev_j$)
                    for all inter_e in inter_events do
                        update inv_index with ($ev_i$, $inter\_e$)
10:                 end for
                    reset inter_events
                else if $ev_j$.type not in inter_events then
                    update inv_index with $ev_i$, $ev_j$
                    append $ev_j$ to inter_events
15:             end if
            end for
            append $ev_i$ to checkedList
        end if
    end for
```

The *Parsing* method computes pairs while parsing through the sequence. The *Indexing* method, first detects the positions of each distinct event and then calculates the pairs. Finally, the *State* method updates and saves a state for the sequence for each new event.

Each method can be used in different scenarios. As we will show in the experimental section, the *Indexing* method dominates in the settings investigated. But this is not necessarily always the case. For example, if we operate in a fully dynamic environment, where new events are appended continuously as a data stream, its easier to keep a state of the sequence than calculating all the pairs from the start. However, in our core scenario where new logs are processed periodically, all three ways apply. In addition, if a domain has a lot of distinct events, i.e., l is very high and much higher than the cardinalities examined, the *Indexing* method becomes inefficient and thus is better to use the *Parsing* one.

Parsing method. The main structure is inv_index, which is a trace-specific part of the Index table. For each trace, the entries of this table are augmented in parallel and since there is no ordering in the values, this is not a problem.

The main rationale is shown in Algorithm 6, which contains two main loops, in line 2 and in line 5, respectively. The idea is to create all the event pars (ev_i, ev_j), in which ev_i is before ev_j. The checkedList prevents the algorithm from dealing with events types that has already been tested. While looping through the trace sequence for an event ev_1, the algorithm appends all new events to inter_events until it finds an event, ev_2 that has the same type as ev_1. When this happens it will create all the pairs of ev_1 with the events in the inter_events list (line 8-10) and will empty it (line 11). After that point, the algorithm proceeds with creating pairs where the timestamp of the first event is now equal to ev_2's timestamp. While updating the index, some extra checks are performed to prevent entering the same pairs twice.

Complexity Analysis. Even though there are two loops iterating the events, the if statement in line 4 can be true only up to l times (where l is the number of distinct elements) and so the complexity is $O(nl^2)$, with n being the length of the trace sequence. The space required is $O(n+l^2)$, for the inv_index and the checkedList.

Algorithm 7 Indexing method (per trace)

```
1: indexer ← map(event_id):[timestmap1,timestamp2,...]
2: for all $ev_a$ in indexer do
3:     for all $ev_a$ in indexer do
4:         CreatePairs(indexer[$ev_a$.tsList],indexer[$ev_b$.tsList]))
5:     end for
6: end for
   procedure CreatePairs(times_a,times_b)
       i,j,prev ← 0,0,-1
       pairs ← [ ]
       while i < times_a.size and j < times_b.size do
5:         if times_a[i] < times_b[j] then
               if times_a[i] > prev then
                   append (times_a[i],times_b[j]) to pairs
                   prev ←times_b[j], i←1, j←1
               else
10:                i←1
               end if
           else
               j←1
           end if
15:    end while
       return pairs
   end procedure
```

Algorithm 8 State method (per trace)

```
1: index ← HashMap(($ev_i$, $ev_j$):[$ts_1$, $ts_2$, $ts_3$...])
2: for all $ev$ in the trace do
3:     Add_New(index,$ev$)
4: end for
5: return index
6: procedure Add_New(index, new_event)
7:     for all combinations where new_event is the 1st event in index do
8:         update state
9:     end for
10:    for all combinations where new_event is the 2nd event in index do
11:        update state
12:    end for
13: end procedure
```

Indexing method. The key idea is to read the whole sequence of events while keeping the timestamps in which every event type occurred (line 1). Then, for every possible combination of events we run the procedure, in which we create the pairs. The procedure is similar to a merging of two lists, while checking for time constrains. More specifically, in line 5, the order of the events is checked and then in line 6, we ensure that there are no overlapping event pairs.

Complexity Analysis. In line 1, we loop once the entire sequence to find the indexes of each distinct event ($O(n)$). Then, the next loops in lines 2-3 retrieve all the possible event pairs ($O(l^2)$) and finally the procedure in line 4, will pass through their indices ($O(n)$). This gives a total complexity of $O(n+l^2n)$, which is simplified to $O(nl^2)$. The total space required is $O(n+l^2)$, for the partial and the pairs. I.e., the complexity is similar to the *Parsing* method.

State method. The algorithm is based on a Hash Map, which contains a list of timestamps for each pair of events. We first initialize the structure by adding all the possible event pairs that can be created (line 1) through parsing the sequence and detecting all distinct event types that appear. Then we loop again through the event sequence. While looping through the sequence we add new event (ev_i) in the structure, by first updating all the the pairs that have ev_i as the first event and then as the second (procedure lines 7-12). During these updates, we ensure that no pair is overlapping. The update operation is as follows. For each (ev_i, ev_j) entry in the HashMap, if the list of the timestamps has even size, we append $ev_i.ts$; otherwise we do nothing. Similarly, for each (ev_j, ev_i) entry in the HashMap, if the list of the timestamps has odd size, we append $ev_i.ts$; otherwise we do nothing. At the end, we trim all timestamp lists of odd size (not shown in the algorithm).

Complexity Analysis. The space complexity is $O(l^2)$ due to the HashMap. In line 2, the loop is passing through all the events in the sequence and for every event executes the procedure Add_new. This procedure has two loops passing through the set of distinct events (l), which gives us a total complexity of $O(nl)$ multiplied by the complexity to access the HashMap, which is $O(1)$ in the average case. Despite this lower complexity, in the evaluation section, we will provide evidence that the overheads due to the HashMap access are relatively high.

Implementation information. We have used Spark and Scala for developing the pre-processing component, which encapsulates the event pair indexing, and Java Spring for the query processor. The source code is publicly available on GitHub.[4],[5]

5 EVALUATION

We used both real-world and synthetic datasets to evaluate the performance of the proposed methods. We start by presenting the datasets, followed by the evaluation of the different flavors of indexing event pairs. Then we compare the preprocess time with the proposal in [19] and Elasticsearch v7.9.1 and finally we show the response time for queries that executed in both methods. In query processing, we also compare against SASE [34].[6] All tests were conducted on a machine with 16GB of RAM and 3.2GHz CPU with 12 cores. Cassandra is deployed on a separate machine with 64GB of RAM and 2GHz CPU. Each experiment is repeated 5 times and the average time is presented.

5.1 Datasets

The real-world datasets are taken from the Business Process Intelligence (BPI) Challenges, and more specifically from the years 2013, 2017 and 2020. BPI13[7] is an event log of Volvo IT incident and problem management. It includes 7,554 traces, which contain 65,533 events in total. The mean, min and max number of events per trace for this dataset are 8.6, 1 and 123, respectively. BPI17[8] is an event log, which corresponds to a loan application of an Dutch financial institute. It includes 31,509 traces, which contain over 1M (1,202,267) events in total. The mean, min and max number of events per trace for this dataset are 38.15, 10 and 180, respectively.

Log file	Number of traces	Activities
max_100	100	150
max_500	500	159
med_5000	5,000	95
max_5000	5,000	160
max_1000	1,000	160
max_10000	10,000	160
min_10000	10,000	15
bpi_2013	7,554	4
bpi_2020	6,886	19
bpi_2017	31,509	26

Table 4: Number of traces and distinct activities for every process-like event log.

From BPI20[9], we use an event log of requesting for payment for a business trip. This is the smaller real-world dataset. It includes 6,886 traces, which contain 36,796 events. The mean, min and max number of events per trace for this dataset are 5.3, 1 and 20, respectively.

We also created synthetic datasets. First with the help of the PLG2[10] tool, we created 3 different processes, with different number of distinct activities (15,95,160). Then by modifying the number of traces per logfile, we created logs that contain from 500 to 400,000 events. The log files are in the XES[11] format. In Figure 2, the distributions of events per trace and unique activities per trace are shown. The purpose of these figures is to provide evidence that our test datasets cover a broad range of real-world trace profiles, thus the experimental results are trustworthy. In general, logs with the terms “med” and “max” in their name have more events per trace and much more unique activities than those with the term “min”. Summary metadata are also in the Table 4. The process-oriented logs are not big, but are used in order to compare our approach against the one in [19], which has been employed in pattern continuation in business processes. This method cannot handle much bigger datasets. To test the scalability of our solution, we employ some additional random datasets that will be introduced separately.

5.2 Evaluating the different ways of indexing pairs

In this section, we evaluate the different flavors that index the event pairs according to the skip-till-next-match (STNM) policy. We aim to find the pros and cons for each flavor in Section 4 and also define the different real life scenarios to use them complementing the discussions already made above. We start the evaluation using the datasets in Table 4. The results are shown in Table 5. The main observation is that all three flavors perform similarly while indexing process-like datasets. When the relative differences are larger (e.g., larger than 30% for bpi_2020), the absolute times are small, so the impact of different decisions is not that important.

These datasets are not big. To better test the potential of the three alternatives, we created log files in which the events were not based on a process. We range the number of traces from 100 to 5000, the number of max events per trace from 50 to 4000

[4] https://github.com/mavroudo/SequenceDetectionPreprocess
[5] https://github.com/mavroudo/SequenceDetectionQueryExecutor
[6] The SASE code repository used in the experiments is https://github.com/haopeng/sase
[7] doi:10.4121/500573e6-accc-4b0c-9576-aa5468b10cee
[8] https://data.4tu.nl/articles/BPI_Challenge_2017/12696884
[9] https://doi.org/10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51
[10] https://plg.processmining.it/
[11] https://xes-standard.org/

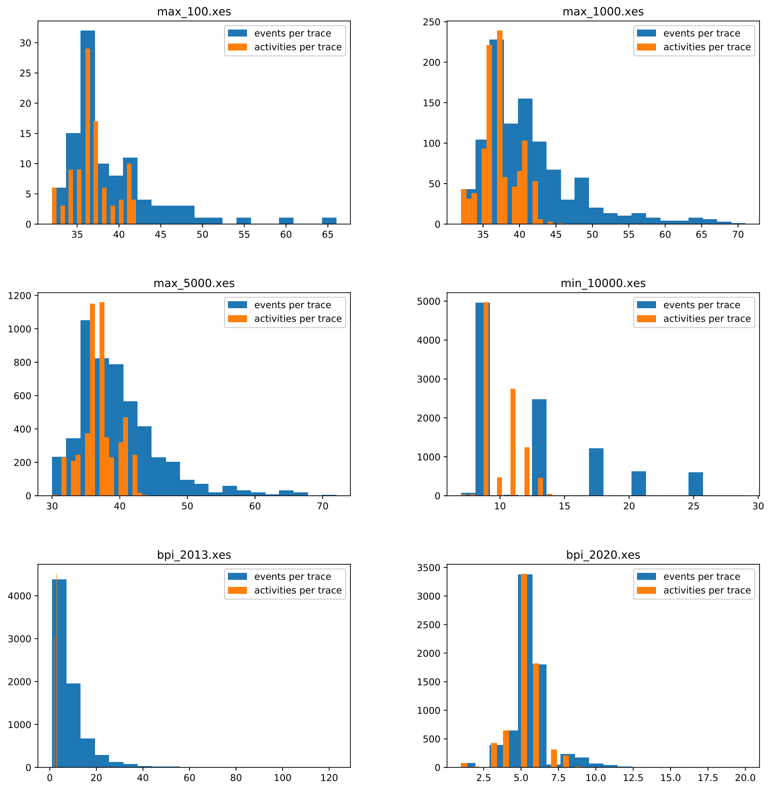

Figure 2: Distributions of the number of events and activities (i.e., unique event types) per trace for every process like log file.

Log file	Indexing	Parsing	State
max_100	4.874	4.49	4.572
max_500	8.454	7.109	7.294
max_1000	10.656	10.407	10.447
med_5000	23.105	22.601	22.417
max_5000	38.152	34.854	38.444
max_10000	79.863	77.964	80.796
min_10000	15.604	13.979	13.625
bpi_2020	6.803	10.384	8.822
bpi_2013	9.528	8.044	8.197
bpi_2017	170.9	171.666	179.352

Table 5: Execution times of different methods (in seconds).

and the number of activities from 4 to 2000. We refer to them as random datasets, due to the lack of correlation between the appearance of two events in a trace, which is not the typical case in practice, and renders the indexing problem more challenging.

The results are presented in Figure 3. In the first plot, we set the number of traces equal to 1000 and the number of different activities equal to 500, while changing the number of max events per trace from 100 to 4000. I.e., we handle up to 4M events. In the second plot, we keep the maximum number of events per trace and distinct activities to 1000 and 100, respectively while increasing the number of traces from 100 to 5000. I.e., we handle up to 5M events. Lastly, we maintain both the number of traces and maximum number of events to 500 and increase the distinct activities from 4 to 2000.

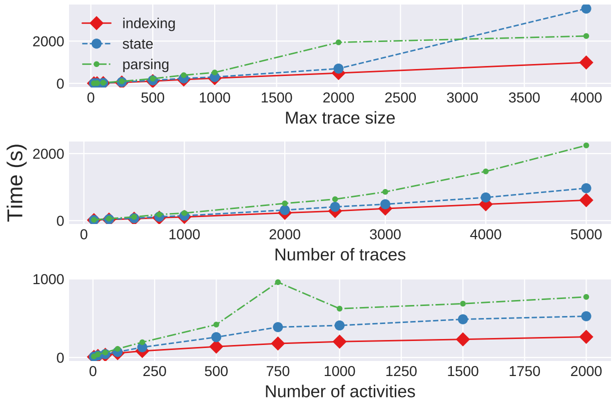

Figure 3: Comparison of execution times of the three different approaches of indexing the event pairs according to the STNM policy for large random logs.

From the Figure 3, we can observe that the Indexing alternative outperforms the other two, in some cases by more than an order of magnitude. The simplicity of this method makes it superior to State, even though the time complexity indicates that the latter is better. The State method performs better than Parsing; especially in the third plot we can see the non-linear correlation between the execution time and the number of distinct activities.

In summary, our results indicate that indexing is the most efficient flavor to use when dealing with log files considered periodically (so that new log entries are a few millions): it has

Log file	[19]	Strict (1 thread)	Strict	Indexing (1 thread)	Indexing	Elasticsearch
max_100	1.054	3.764	3.701	5.398	4.874	0.67
max_500	2.68	5.593	4.649	12.568	8.454	4.68
max_1000	4.458	7.084	5.69	22.544	10.656	10.167
med_5000	6.913	20.361	9.175	113.04	23.105	31.80
max_5000	16.163	25.419	12.452	210.713	38.152	31.41
min_10000	26.64	31.379	8.782	116.318	15.604	38.15
max_10000	37.569	63.975	21.006	734.844	79.863	121.167
bpi_2020	95.269	11.461	8.597	17.908	6.803	14.49
bpi_2013	504.089	12.817	7.918	14.925	9.528	9.973
bpi_2017	very high	451.666	66.284	crash	170.9	364.293

Table 6: Comparison of execution times between [19] and our proposal (time in seconds).

minimum space complexity and it has the best executing time. On the contrary, State is preferable when operating in a dynamic environment, when for example new logs will be appended at the end of every few minutes and some traces will be active for weeks. State allows to save the current state of the log and, when new events are appended, it can calculate the event pairs without checking the previous ones. Even though the space complexity is higher than the other methods, it is expected to dominate in a real dynamic scenario.

5.3 Pre-process comparison

Based on the previous results, we continue the comparison using only the Indexing alternative for the STNM policy. We compare the time for building the index for both SC and STNM against [19], which supports only SC, and against Elasticsearch. The results are presented in Table 6; to provide the full picture we run Spark in two modes, namely using all the available machine cores and using a single Spark executor. The latter allows for direct comparison against non-parallel solutions.

Considering how [19] works, logs that are based on processes are easier to handle. We can split the test datasets of table 4 into three categories, namely small synthetic datasets (100-1000 traces), large synthetic datasets (5000 & 10000 traces) and real datasets (from the BPI challenge). In the first category, Strict performs almost the same as [19], while Indexing has significantly higher execution time, due to the more complex process it runs. In the second category, Strict scales better and achieves better times than [19]. Finally, in real datasets, our method achieves two order of magnitude lower times compared to [19]. When using the bpi_2017 dataset, [19] could not even finish indexing in 5 hours. This is probably based on the large amount of events (≈1.2M) combined with the high number of distinct events per trace. This lead to a very large suffix array, which probably could not fit in main memory and ended up doing an extensive amount of I/Os. For the same dataset, both Indexing and Strict managed to create inverted indexing in less than 3 minutes when using all machine cores.

Compared to Elasticsearch, we can observe that our best performing technique is on average faster for the last two categories (large synthetic and real datasets). In the larger real dataset, building an index to support STNM queries according to our proposal is more than 2.1X faster than Elasticsearch.

Parallelization-by-design is a big advantage of our method; we do not simply employ Spark but we can treat each trace in parallel. Further, parallelization applies to both the event-pair creation and the storage (Cassandra is a distributed database). As shown in Table 6, indexing can run even 10 times faster when using all 12 cores available. This is not the case for the [19] and other solutions, like Elasticsearch. However, there exist some structures that build suffix trees in parallel [2, 13, 20]. But still, the most computational intense process is to find all the subtrees and store them. The number of subtrees is increased with the number of leaves, which depends on the different traces that can be found in a logfile.

Log file	[19]	Our method (2)	Our method (10)
max_100	0.0023	0.007	0.022
max_500	0.0026	0.020	0.029
max_1000	0.0022	0.010	0.050
med_5000	0.0022	0.013	0.280
max_5000	0.0026	0.007	0.230
min_10000	0.0022	0.060	2.200
max_10000	0.0026	0.012	0.400
bpi_2020	0.0059	0.006	0.290
bpi_2013	0.0185	0.034	4.000

Table 7: Comparison response times in seconds

5.4 Query response time

We start by comparing the response time for a single query, between our method and the one in [19]. Since [19] supports the strict contiguity (SC) policy solely, we use this policy to create the inverted index and then execute a pattern detection query, as described in Section 3.2.1. Then, we compare the STNM solutions against Elasticsearch and SASE, which does not perform any preprocessing. We do not employ Elasticsearch in the SC experiments, because it is more suitable for STNM queries; more specifically, supporting SC can be achieved with additional expensive post-processing. Finally, we compare the pattern continuation methods and show the effectiveness of the trade-off between accuracy and response time.

5.4.1 Comparison against [19] for SC. The results of the comparison are shown in Table 7. In the first column, we can see the response time of [19] for the different log files. In the next 2 columns, we have different response times for detection queries, for pattern length equal to 2 and 10, respectively. As discussed in Section 2.2, all subtrees are precalculated and stored, which means that the detection query time is $O(logn + k)$ where n here is the number of different subtrees and k is the number of subtrees that will return. As such, for [19], the response time does not depend on the querying pattern length. On the other hand,

Log file	Elasticsearch	SASE	Our method
pattern length = 2			
max_100	0.006	0.003	0.003
max_500	0.009	0.014	0.006
max_1000	0.009	0.038	0.004
med_5000	0.048	0.958	0.006
max_5000	0.015	1.400	0.005
min_10000	0.145	1.565	0.031
max_10000	0.048	7.024	0.011
bpi_2013	0.071	0.205	0.008
bpi_2020	0.068	0.366	0.040
bpi_2017	0.609	70.491	0.102
pattern length = 5			
max_100	0.011	0.002	0.008
max_ 500	0.018	0.014	0.012
max_ 1000	0.017	0.038	0.013
med_ 5000	0.126	0.999	0.048
max_ 5000	0.037	1.226	0.036
min_ 10000	0.647	1.688	0.525
max_ 10000	0.170	6.413	0.061
bpi_ 2013	0.155	0.233	0.063
bpi_ 2020	0.246	0.534	0.562
bpi_ 2017	4.652	370.142	1.495
pattern length = 10			
max_100	0.020	0.002	0.048
max_500	0.031	0.014	0.039
max_1000	0.032	0.038	0.060
med_5000	0.239	1.010	0.279
max_ 5000	0.075	1.245	0.218
min_10000	1.340	1.712	3.707
max_10000	0.289	6.491	0.373
bpi_2013	0.259	0.229	0.374
bpi_2020	0.440	0.531	4.262
bpi_ 2017	9.661	440.066	11.188

Table 8: Response times for STNM queries in seconds

our proposal incrementally calculates the completion for every event in the pattern as described in Section 3.2.1; as such, the response time depends on the pattern length. In Figure 4, we show how response time increases with respect to the querying pattern length.

The experiments in Table 7 were executed 5 times and we presented the mean response time. However, we have noticed a fluctuation in response times, which is affected by the events in the pattern. Each event has a different frequency in the log files; e.g., starting and ending events are more frequent than some events that correspond to an error. When events in the querying pattern have low frequency, the response time will be shorter because there are fewer traces that need to be tested.

For small patterns, with length equal to 2-5, we get similar response times between the two methods, while [19] is always faster. As pattern length increases, our method's response time increases as well, but we also return as a by-product detection for all the sub-patterns.

In summary, the table shows the penalty we pay against a state-of-the-art technique during subsequence matching; however, the benefits of our approach are more significant: we allow efficient indexing in large datasets and we support, with similar times, pattern queries using the STNM policy.

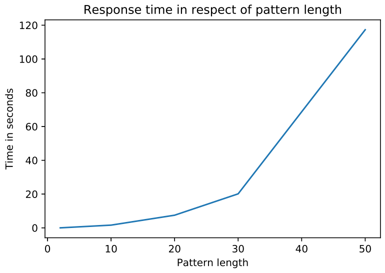

Figure 4: Response time with respect to the querying pattern length

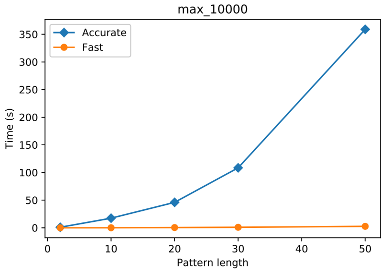

Figure 5: Response time for different pattern continuation methods for different query pattern lengths

5.4.2 Comparison against Elasticsearch and SASE for STNM. In Table 8, we present comparison against SASE and Elasticsearch query response times, when, in each experiment, we search for 100 random patterns. There are two main observations. Firstly, running techniques that perform all the processing on the fly without any preprocessing, such as SASE, yields acceptable performance in small datasets but significantly degrades in larger datasets, such as bpi_2017 and max_10000. In the former dataset, techniques that perform preprocessing are faster by 2 orders of magnitude. Secondly, there is no clear winner between Elasticsearch and our solution. But, in general, we are faster for small queries of pattern size equal to 2 and in all but one dataset for pattern size equal to 5, while Elasticsearch runs faster for pattern length equal to 10. However, for the longest running long queries, our solution is only 15.8% slower. Therefore, we can claim that our solution is competitive for large query patterns. Moreover, we can relax our query method to achieve faster times, as explained in the next part of the evaluation, while we support pattern continuation more efficiently due to the incremental approach of pattern processing that we adopt; i.e., we do not have to repeat the complete query from scratch.

5.4.3 Comparison of pattern continuation alternatives. In Figure 5, we show the response times between Accurate and Fast method for the dataset max_10000. We can see that the Accurate method follows the same pattern as the graph in Figure 4, which is what we expected as it performs pattern detection for every possible subsequent event in the pattern. On the other hand, there is no significant increase of response time for the Fast heuristic with regards of the pattern length.

We are trying to fill this performance gap with the Hybrid alternative. In Figure 6, we use again the max_10000 dataset and a pattern with 4 events and we show the response time with respect to the *topK* parameter given to Hybrid. The response time for both Accurate and Fast is constant, because they do not use this parameter. As expected, the time increases linearly as k increases.

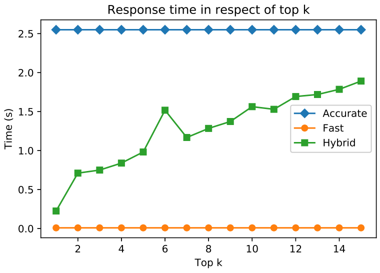

Figure 6: Response time of pattern continuation methods for different *topK* values

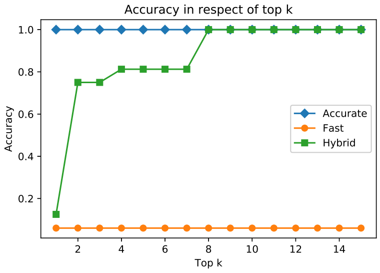

Figure 7: Accuracy of pattern continuation methods for different *topK* values

Fast's execution time is the lower bound and Accurate's is the upper one.

For the same setup, we perform an accuracy test presented in Figure 7. We use as ground truth the events returned from the Accurate method and compute the accuracy as the fraction of the events in the top k propositions from Hybrid that exist in the propositions reported by Accurate, where k is the number of propositions returned from Accurate. The accuracy is increasing as the number of k increases until it reaches 100% for k=8. For the same value, response time is half of the Accurate, as shown in the previous graph. Also in this example we could achieve a 80% accuracy with k=2 and 1/3 of the response time that Accurate would have taken.

6 RELATED WORK

Our work relates to several areas that are briefly described here in turn.

Complex event processing. There are number of surveys presenting scalable solutions for the detection of complex events in data stream. A variety of general purpose CEP languages have been developed. Initially, SASE [30] was proposed for executing complex event queries over event streams supporting SC only. The SQL-TS [24] is an extension to the traditional SQL that supports search for complex and recurring patterns (with the use of Kleene closure), along with optimizations, to deal with time series. In [9], the SASE language was extended to support Kleene closures, which allow irrelevant events in between thus covering the skip-till-next-match (STNM) and skip-till-any-match strategies. An extensive evaluation of the different languages was presented in [34] along with the main bottlenecks for CEP. In addition, the K*SQL [21] language, is a strict super-set of SASE+, as it can work with nested words (XML data). Besides the languages, most of these techniques use automata in order to detect specific patterns in a stream, like [18]. Our technique differs from them as we do not aim to detect patterns on the fly, but instead, to construct the infrastructure that allows for fast pattern queries in potentially large databases. To this end, the work in [22] also uses pair of events to create signatures, but for scalability purposes, this work considers only the top-k most frequent pairs, which yields an approximate solution, whereas we focus on exact answers. In addition, [22] focuses on the proposal of specific index types, whereas we follow a more practical approach, where are indices are stored as Cassandra tables to attain scalability.

Pattern mining. For non-streaming data, a series of methods have been developed in order to mine patterns. The majority of these proposals are looking for frequent patterns; e.g., Sahli et al in [26] proposed a scalable method for detecting frequent patterns in long sequences. As another example, in several other fields such as biology, several methods have developed, which are typically based in statistics (e.g., [1, 15]) and suffix trees (e.g., [10]). Parallel flavors have also been proposed, e.g., in [7]. Other forms of mined patterns include outlying patterns [5] or general patterns with high utility as shown in [32]. It is not trivial to build on top of these techniques to detect arbitrary patterns, because these techniques typically prune non-interesting patterns very aggressively.

Business processes. There are applications in business process management that employ pattern mining techniques to find outlier patterns and clear log files from infrequent behavior, e.g., [16, 28], in order to facilitate better process discovery. Another application is to predict if a trace will fail to execute properly; for example, in [4, 17], different approaches to predicting the next tasks of an active trace are presented. None of these techniques addresses the problem of efficiently detecting arbitrary sequences of elements in a large process database as we do, but the technique in [27] encapsulates our main competitor, namely [19]. Finally, in [8], a query language over business processes was presented to support sub-process detection. The proposed framework can be leveraged to support SC and STNM queries over log entries rather than subprocesses, but this entails using a technique like SASE, without any pre-processing. Our evaluation shows that such techniques are inferior to our solution.

Other data management proposals. The closest work to ours from the field of data management is set containment join queries, e.g., [31]. However, this type of joins does not consider time ordering. An interesting direction for future work is to extend these proposals to work for ordered sequences rather than sets; to date, this remains an open issue.

7 DISCUSSION

We have proposed a methodology to detect pattern according to the Strict contiguity (SC) and Skip-till-next-match (STNM) policy in large log databases, assuming that new events arrive and processed in big batches. However, there are several issues that need to be addressed with a view to yielding a more complete solution. First, sequential patterns, in their more relaxed form, allow for overlappings, which is commonly referred to as the skip-till-any-match policy. Supporting such patterns places additional burden to both the indexing process and query execution. Second, in many cases, assuming a total ordering is restrictive and also, the way some events may be logged, even in the same trace, cannot be regarded as following a total order. For example, in predictive maintenance in Industry 4.0, it is common to group events in large sets ignoring their relative order, e.g., [29]. Extending our approach to operate under partial ordering is an interesting extension. Additionally, judiciously choosing the optimal update period is an open issue and gives rise to a multi-objective problem where low indexing time and result timeliness are contradicting objectives. Finally, the pattern continuation techniques can account for other operation modes, where an event is not appended only at the end, but also at arbitrary places in the query pattern. Our proposal can be easily extended to cover these cases, but we omit details here.

8 CONCLUSION

Despite the big advances in complex event processing and sequential pattern mining, efficient detection of arbitrary subsequences in log databases is an overlooked issue. Our proposal fills this gap and proposes indexing techniques along with query evaluation algorithms that allow the user to detect any patterns according to either the strict contiguity and the skip-till-next-match policy. Compared to subsequence matching techniques that support only strict contiguity, we show that our indexing can scale and also, query processing times are competitive when both approaches are applicable. Compared to Elasticsearch, a state-of-the-art solution, we build the indices faster and we run small queries faster, while we are competitive in large queries. Further, our solution can support exploration of pattern extension alternatives with different trade-offs between running time and accuracy and builds on top of scalable technologies, like Spark and Cassandra.

Acknowledgment. The research work was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the “First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant” (Project Number:1052).

REFERENCES

[1] Alberto Apostolico, Matteo Comin, and Laxmi Parida. 2011. VARUN: Discovering Extensible Motifs under Saturation Constraints. *IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM* 7 (01 2011), 752–26. https://doi.org/10.1109/TCBB.2008.123

[2] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and Uzi Vishkin. 1988. Parallel construction of a suffix tree with applications. *Algorithmica* 3 (11 1988), 347–365. https://doi.org/10.1007/BF01762122

[3] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD Conference*. 975–986.

[4] Michael Borkowski, Walid Fdhila, Matteo Nardelli, Stefanie Rinderle-Ma, and Stefan Schulte. 2019. Event-based failure prediction in distributed business processes. *Information Systems* 81 (2019), 220 – 235. https://doi.org/10.1016/j.is.2017.12.005

[5] Lei Cao, Yizhou Yan, Samuel Madden, Elke A. Rundensteiner, and Mathan Gopalsamy. 2019. Efficient Discovery of Sequence Outlier Patterns. *Proc. VLDB Endow.* 12, 8 (April 2019), 920-932. https://doi.org/10.14778/3324301.3324308

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[7] Alexandra Carvalho, Arlindo Oliveira, Ana Teresa Freitas, and Marie-France Sagot. 2004. A Parallel Algorithm for the Extraction of Structured Motifs. *Proceedings of the ACM Symposium on Applied Computing* 1, 147–153. https://doi.org/10.1145/967900.967932

[8] Daniel Deutch and Tova Milo. 2012. A structural/temporal query language for Business Processes. *J. Comput. System Sci.* 78, 2 (2012), 583 – 609. https://doi.org/10.1016/j.jcss.2011.09.004 Games in Verification.

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

[10] Avrilia Floratou, Sandeep Tata, and Jignesh Patel. 2010. Efficient and Accurate Discovery of Patterns in Sequence Datasets. *Proceedings - International Conference on Data Engineering*, 461–472. https://doi.org/10.1109/ICDE.2010.5447843

[11] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, and Yun Sing Koh. 2017. A Survey of Sequential Pattern Mining. *Data Science and Pattern Recognition* 1, 1 (2017), 54–77.

[12] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Tin Truong-Chi, and Roger Nkambou. 2019. A survey of high utility itemset mining. In *High-Utility Pattern Mining*. Springer, 1–45.

[13] Amol Ghoting and Konstantin Makarychev. 2009. Serial and Parallel Methods for i/o Efficient Suffix Tree Construction. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1559845.1559931

[14] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* 29, 1 (2020), 313–352.

[15] Roberto Grossi, Andrea Pietracaprina, Nadia Pisanti, Geppino Pucci, Eli Upfal, and Fabio Vandin. 2009. MADMX: A Novel Strategy for Maximal Dense Motif Extraction. https://doi.org/10.1007/978-3-642-04241-6_30

[16] Ying Huang, Yingxu Wang, and Yiwang Huang. 2018. Filtering Out Infrequent Events by Expectation from Business Process Event Logs. 374–377. https://doi.org/10.1109/CIS2018.2018.00089

[17] Bokyoung Kang, Dongsoo Kim, and Suk-Ho Kang. 2012. Real-time business process monitoring method for prediction of abnormal termination using KNNI-based LOF prediction. *Expert Systems with Applications* 39, 5 (2012), 6061 – 6068. https://doi.org/10.1016/j.eswa.2011.12.007

[18] Ilya Kolchinsky and Assaf Schuster. 2019. Real-Time Multi-Pattern Detection over Event Streams. 589–606. https://doi.org/10.1145/3299869.3319869

[19] Fabrizio Luccio, Antonio Mesa Enriquez, Pablo Olivares Rieumont, and Linda Pagli. 2001. *Exact Rooted Subtree Matching in Sublinear Time.* Technical Report. TR-01-14, University of Pisa.

[20] Essam Mansour, Amin Allam, Spiros Skiadopoulos, and Panos Kalnis. 2011. ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings. *CoRR* abs/1109.6884 (2011). arXiv:1109.6884 http://arxiv.org/abs/1109.6884

[21] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. 2010. From Regular Expressions to Nested Words: Unifying Languages and Query Execution for Relational and XML Sequences. *PVLDB* 3 (09 2010), 150–161.

[22] Alexandros Nanopoulos, Yannis Manolopoulos, Maciej Zakrzewicz, and Tadeusz Morzy. 2002. Indexing web access-logs for pattern queries. In *Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2002)*. 63–68.

[23] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2004. Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. *IEEE Trans. Knowl. Data Eng.* 16, 11 (2004), 1424–1440.

[24] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. 2001. Optimization of Sequence Queries in Database Systems. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '01)*. Association for Computing Machinery, New York, NY, USA, 71-81. https://doi.org/10.1145/375551.375563

[25] Majed Sahli, Essam Mansour, and Panos Kalnis. 2014. ACME: A scalable parallel system for extracting frequent patterns from a very long sequence. *VLDB J.* 23, 6 (2014), 871–893.

[26] Majed Sahli, Essam Mansour, and Panos Kalnis. 2014. ACME: A scalable parallel system for extracting frequent patterns from a very long sequence. *The VLDB Journal* 23 (12 2014). https://doi.org/10.1007/s00778-014-0370-1

[27] Suhrid Satyal, Ingo Weber, Hye young Paik, Claudio Di Ciccio, and Jan Mendling. 2019. Business process improvement with the AB-BPM methodology. *Information Systems* 84 (2019), 283 – 298. https://doi.org/10.1016/j.is.2018.06.007

[28] Sebastiaan J. van Zelst, Mohammadreza Fani Sani, Alireza Ostovar, Raffaele Conforti, and Marcello La Rosa. 2020. Detection and removal of infrequent behavior from event streams of business processes. *Information Systems* 90 (2020), 101451. https://doi.org/10.1016/j.is.2019.101451 Advances in Information Systems Engineering Best Papers of CAiSE 2018.

[29] J. Wang, C. Li, S. Han, S. Sarkar, and X. Zhou. 2017. Predictive maintenance based on event-log analysis: A case study. *IBM Journal of Research and Development* 61, 1 (2017), 11.

[30] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. *Proceedings of the ACM SIGMOD International Conference on Management of Data* 10, 407–418. https://doi.org/10.1145/1142473.1142520

[31] Jianye Yang, Wenjie Zhang, Shiyu Yang, Ying Zhang, and Xuemin Lin. 2017. TT-Join: Efficient Set Containment Join. In *33rd IEEE International Conference on Data Engineering, ICDE*. 509–520.

[32] Junfu Yin, Zhigang Zheng, and Longbing Cao. 2012. USpan: An efficient algorithm for mining high utility sequential patterns. *KDD 2012* (08 2012). https://doi.org/10.1145/2339530.2339636

[33] Mohammed Javeed Zaki. 2001. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Mach. Learn.* 42, 1/2 (2001), 31–60.

[34] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (06 2014). https://doi.org/10.1145/2588555.2593671