Efficient Discovery of Sequence Outlier Patterns

Lei Cao [1*], Yizhou Yan [2*], Samuel Madden [1], Elke A. Rundensteiner[2], Mathan Gopalsamy[3]

[1]*Massachusetts Institute of Technology, Cambridge, MA USA*

[2]*Worcester Polytechnic Institute Worcester, MA, USA*

[3]*Signify Research, Cambridge, MA USA*

(lcao,madden)@csail.mit.edu, (yyan2,rundenst)@cs.wpi.edu, mathankumar.gopalsamy@signify.com

ABSTRACT

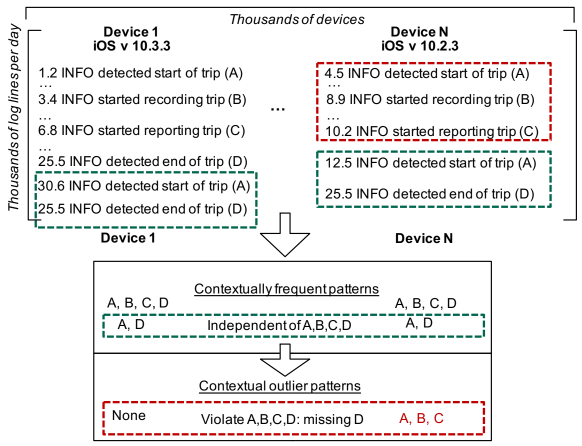
Modern *Internet of Things* (*IoT*) applications generate massive amounts of time-stamped data, much of it in the form of discrete, symbolic sequences. In this work, we present a new system called TOP that deTects Outlier Patterns from these sequences. To solve the fundamental limitation of existing pattern mining semantics that miss outlier patterns hidden inside of larger frequent patterns, TOP offers new pattern semantics based on *contextual patterns* that distinguish the *independent occurrence* of a pattern from its occurrence as part of its super-pattern. We present efficient algorithms for the mining of this new class of contextual patterns. In particular, in contrast to the bottom-up strategy for state-of-the-art pattern mining techniques, our top-down *Reduce* strategy piggy backs pattern detection with the detection of the context in which a pattern occurs. Our approach achieves linear time complexity in the length of the input sequence. Effective optimization techniques such as context-driven search space pruning and inverted index-based outlier pattern detection are also proposed to further speed up contextual pattern mining. Our experimental evaluation demonstrates the effectiveness of TOP at capturing meaningful outlier patterns in several real-world IoT use cases. We also demonstrate the efficiency of TOP, showing it to be up to 2 orders of magnitude faster than adapting state-of-the-art mining to produce this new class of contextual outlier patterns, allowing us to scale outlier pattern mining to large sequence datasets.

PVLDB Reference Format:
Lei Cao, Yizhou Yan, Samuel Madden, Elke A. Rundensteiner, Mathan Gopalsamy. Efficient Discovery of Sequence Outlier Patterns. *PVLDB*, 12(8): 920-932, 2019.
DOI: https://doi.org/10.14778/3324301.3324308

1. INTRODUCTION

Motivation. With the increasing prevalence of sensors, mobile phones, actuators, and RFID tags, *Internet of Things (IoT)* applications generate massive amounts of time-stamped data represented as discrete, symbolic sequences of data. Examples include control signals issued to Internet-connected devices like lights or thermostats, measurements from industrial and medical equipment, or log files from smartphones that record complex behavior of sensor-based applications over time. In these applications, finding *outlier patterns* that represent deviations from normal behavior is critical. Such outlier patterns may indicate devices or agents not performing as expected based on their configurations or operational requirements. To further motivate outlier patterns, we describe two real-world applications that require this functionality:

Example 1. A startup company we work with collects log files from hundreds of thousands of phones running a sensor-based mobile application that records data from users as they drive. Fig. 1 shows examples of log files. After the user starts a trip, the system continuously reports driver's status and is expected to terminate when the trip finishes. This typical system behavior is represented by a frequent sequential pattern $P = \langle StartTrip(A), RecordTrip(B), ReportLoc(C), Terminate(D)\rangle$ as shown in the top portion of Fig. 1. An infrequent pattern $Q = \langle StartTrip(A),\ RecordTrip(B), ReportLoc(C)\rangle$ instead signals an erroneous system behavior, since the *expected* successful termination action is missing. Thus, pattern Q as a *sub-pattern* of P corresponds to an *outlier pattern* that violates the *typical expected* behavior represented by P. In addition to the missing of the expected event, the behavior of the system is also suspicious if the events do not occur in the expected order, for example if the ReportLoc(C) event occurs after Terminate(D) event. Failing to capture such problems hidden in the system can lead to increased power consumption on devices or missed recordings, and thus a poor customer experience and decreased revenue.



```
1.2 INFO detected start of trip (A)
…
3.4 INFO started recording trip (B)
…
6.8 INFO started reporting trip (C)
…
25.5 INFO detected end of trip (D)
30.6 INFO detected start of trip (A)
25.5 INFO detected end of trip (D)
```

```
4.5 INFO detected start of trip (A)
…
8.9 INFO started recording trip (B)
…
10.2 INFO started reporting trip (C)
12.5 INFO detected start of trip (A)
25.5 INFO detected end of trip (D)
```

Figure 1: Log file pattern mining example showing examples of contextual frequent patterns & outliers.

*Equal contribution

Proceedings of the VLDB Endowment, Vol. 12, No. 8
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3324301.3324308

Example 2. We have access to data from a hospital infection control system [31] that tracks healthcare workers (HCWs) for hygiene compliance (for example sanitizing hands and wearing masks) using sensors installed at doorways, beds, and disinfectant dispensers. In this setting, a HCW is required to wash her hands before approaching a patient. This hygiene compliance protocol is modeled by a frequent pattern $P = \langle HandWash(A),\ Enter(B), ApproachPatient(C)\rangle$. An infrequent pattern $Q = \langle Enter(B), ApproachPatient(C)\rangle$ represents a *violation of the hygiene compliance protocol*, namely not performing the *HandWash(A)* action before approaching a patient. Therefore, Q should be captured as an *outlier pattern* that as a sub-pattern of typical pattern P, violates the hygiene compliance protocol represented by P. Missing this type of outlier risks healthcare costs and potentially human life caused by avoidable infection.

Therefore, it is critical to capture the outlier patterns that *violate* the typical patterns modeling the expected behavior of the application or devices, e.g., a sub-pattern or a different event type permutation of a typical pattern.

State-of-the-Art & Challenges. To the best of our knowledge, the problem of detecting outlier patterns from sequence data that violate the typical system behavior has not been solved in the literature, as discussed in our related work (Sec. 7).

Designing an outlier pattern detection approach that is both effective and efficient is challenging, for several reasons. First, we need to detect typical patterns. It may seem that the typical patterns can be captured by directly applying existing frequent pattern mining techniques (e.g. [1]). Unfortunately, relying on the existing semantics [3, 13, 32, 11, 12, 5] to detect outlier patterns is not effective because existing semantics *do not* distinguish between the *independent occurrence* of a pattern Q and its occurrence as part of its super-pattern P. Therefore, any sub-pattern Q of a frequent pattern P is also designated as a frequent pattern by existing semantics, since the frequency of Q will never be smaller than the frequency of its super-pattern P. This is the basis of the well-known Apriori property upon which most pattern mining algorithms are built [10]. Unfortunately, these semantics miss outlier patterns, because as described in our motivating examples above, an independent occurrence of the *sub-pattern* Q of a typical pattern P tends to correspond to an outlier representing anomalous behavior. Thus, to meet this requirement of detecting outlier patterns, new pattern mining semantics are required.

Second, designing an efficient outlier pattern mining technique is challenging. The *pattern-growth strategy* (Growth) that builds patterns from short to long – widely adopted by the pattern mining algorithms [29, 15, 16, 37, 4, 9], prunes unpromising *long* pattern candidates based on the Apriori property. Unfortunately it is not effective in our outlier pattern detection scenario. During the bottom up mining process, Growth produces a large number of short patterns. However, to detect outlier patterns we need to exclude *short* patterns from typical patterns when they occur as part of their super-patterns. Therefore, a post-processing step is required to filter such short patterns. Doing so wastes CPU and memory resources by producing a large number of short patterns that are subsequently discarded. Furthermore, detecting outlier patterns, i.e., violations of the typical patterns, is also challenging because of the enumeration of infrequent patterns, of which there is an overwhelming number in a large dataset, and for which there is no clear property that can be used to prune away unpromising candidates.

Proposed Approach and Contributions. In this work, we describe our system, *TOP*, designed to effectively and efficiently deTect Outlier Patterns. It has been deployed in the data analytics platform of Philips Lighting. TOP features novel *contextual pattern* semantics for the effective detection of abnormal system behavior. New mining strategies are also proposed to mine contextual patterns from sequence data. Our key contributions include:

- We define novel *contextual pattern* semantics. By introducing the concept of a *contextual constraint*, whether an instance of a pattern P is counted for frequent pattern mining purposes depends on the *context* in which P occurs. In particular, it is not counted if it appears in a larger pattern found to be typical or abnormal. This contextual constraint separates out the independent occurrences of a *contextual outlier* (CO) pattern from the occurrences as part of its super-patterns. Thus, a CO pattern will not be misclassified as typical simply because it occurs frequently as part of its super-pattern. This ensures that only the patterns capturing typical system behavior in their own right are considered frequent, so called contextually frequent (CF) patterns.
- Instead of using a two-step bottom-up (Apriori) style solution (Growth), we design a customized top-down pattern mining strategy (Reduce) that directly mines the CF patterns *in one step* (Sec. 4.2). Starting from the longest possible contextual patterns, our Reduce strategy recursively constructs patterns from the longest to the shortest. By this, Reduce effectively avoids the unnecessary generation of the large number of short patterns that violates the contextual constraint. Moreover, it saves the extra overhead introduced by the post-processing step.
- Using an event availability-based pruning strategy in the iterative top-down mining process of Reduce, we turn the contextual constraint into a powerful tool to prune all unpromising CF pattern candidates much more rapidly – than otherwise would have been possible (Sec 4.2.2). Our theoretical analysis shows that our Reduce strategy achieves linear time complexity in the length of the input sequence compared to the quadratic complexity of Growth.
- Our approach detects both CF and CO patterns at the same iteration —- rather than mining the sequences twice for each pattern type. In particular, we show that the candidates of CO can be produced almost *for free* as a by-product of the top-down CF pattern mining process (Sec. 4.3).
- Leveraging the relationships between the CO patterns and the CF patterns potentially violated by CO, we design a customized indexing mechanism to speed up the process of mining CO patterns from the CO candidates.
- We perform empirical studies using a mobile app and a lighting app that demonstrate the effectiveness of our TOP system in capturing anomalous patterns in these real world applications.
- We demonstrate a two orders of magnitude performance gain for our Reduce strategy over the state-of-the-art Growth strategy across a wide range of parameter settings on various datasets with both strategies returning identical results.

2. CONTEXTUAL PATTERN SEMANTICS

2.1 Basic Terminology

Let $S = \langle(e_1, t_1)(e_2, t_2)\ldots, (e_n, t_n)\rangle$ denote a **sequence** generated by one device, with (e_i, t_i) as an **event** of type E_i occurring at time t_i. An example of an event (e_i, t_i) may be a log entry of a certain type or a user interaction like "purchase" or "click". As alternative to the timestamp t_i in an event (e_i, t_i), we reference the relative position of the event in sequence S as (e_i, i). Events in sequence S are ordered by time. A **sequence dataset** D is a set of event sequences S_i, each generated by one device, denoted as $D = \{S_1, S_2, \ldots, S_N\}$.

A **sequence pattern** (or in short, **pattern**) $P = \langle E_1 E_2 \ldots E_m\rangle$ corresponds to an ordered list of event types E_i. An **occurrence** of P in sequence S, denoted by $O_P = \langle(e_1, t_1)\ (e_2, t_2)\ldots$

$(e_m, t_m)\rangle$ is a list of events e_i ordered by time t_i, where $\forall$ (e_i, t_i) $\in O_P$, e_i corresponds to event type $E_i \in P$ and $(e_i, t_i) \in S$.

A pattern $Q = \langle E'_1 E'_2 \dots E'_l \rangle$ is a **sub-pattern** of a pattern $P = \langle E_1 E_2 \dots E_m \rangle$ ($l \leq m$), denoted $Q \sqsubseteq P$, if integers $1 \leq i_1 < i_2 < \cdots < i_l \leq m$ exist such that $E'_1 = E_{i_1}$, $E'_2 = E_{i_2}$, ..., $E'_l = E_{i_l}$. P is also said to be a **super-pattern** of Q. For example, pattern $Q = \langle AC \rangle$ is a sub-pattern of $P = \langle ABC \rangle$.

2.2 Contextual Patterns

We first define the notion of a *contextual constraint*. Based on this concept, we then define the contextually frequent (CF) and contextual outlier (CO) pattern semantics used in TOP. Both CO patterns and CF patterns are called *contextual patterns* in general.

2.2.1 Contextual Constraint

The *contextual constraint* determines whether an instance of a pattern is a valid occurrence of this pattern based on the context in which it occurs. The definition of an outlier pattern or frequent pattern only considers its occurrence satisfying the contextual constraint, so called *contextual outlier* (CO) patterns or *contextually frequent* (CF) patterns.

The *contextual constraint* excludes an occurrence O_Q of pattern Q from Q's support if all events in O_Q are contained in the occurrences of pattern P, where P is frequent and longer than Q. This distinguishes between an independent occurrence O_Q of Q and an occurrence O_Q as a sub-occurrence of P. Intuitively, Q is contextually frequent only when it occurs frequently independent from some frequent super-patterns. This avoids reporting abnormal sub-patterns as typical system behavior. For example, the outlier sub-pattern $Q = \langle ABC \rangle$ in Fig. 1 would not be reported as typical system behavior (i.e., as contextually frequent by itself), because most of its occurrences occur only as sub-occurrences of the frequent super-pattern $P = \langle ABCD \rangle$. Similarly, if an occurrence O_Q of Q is a sub-occurrence of an *outlier* super-pattern P, O_Q should not be counted to the support of Q when mining frequent or outlier patterns.

Difference between contextual constraint and the closed/maximal patterns. As the extensions to the basic frequent pattern semantics, *closed pattern* [13, 32, 35] and *maximal pattern* [11, 12, 26] limit the number of output patterns by applying some hard-coded rules. However, unlike our contextual constraint, neither of these notions separate the independent occurrences of a sub-pattern from the sub-occurrences of its super-pattern.

In particular, the *closed pattern* semantics [13, 32, 35] exclude a frequent pattern from the output if its support is identical to the support of one of its super-patterns. Therefore, even if only one independent occurrence of pattern Q exists, Q will still be reported as frequent, despite the fact that Q might be an outlier pattern violating the typical pattern P. Therefore, the outlier sub-pattern $Q = \langle ABC \rangle$ in the above example would still be erroneously reported as a typical pattern by closed pattern semantics.

The *maximal pattern* semantics [11, 12, 26] keep only the largest frequent patterns, discarding all sub-patterns of P when P is frequent. However, sometimes a sub-pattern of a frequent pattern P may actually be frequent *in its own right*. For example in the mobile app (Fig. 1), the sub-pattern $Q = \langle StartTrip(A), Terminate(D) \rangle$ ($Q \sqsubseteq P$) may also be frequent, because the app may start a trip and immediately terminate it if some condition (e.g., low battery) occurs. Therefore, Q should also be considered typical. Unfortunately, Q is discarded by the maximal pattern. This might lead us to miss the outlier patterns that violate Q. Instead, our contextual constraint enables us to report both a pattern and its sub-pattern as conceptually frequent when the sub-pattern Q also represents typical system behavior in that it frequently occurs independently from any super-pattern P.

The contextual constraint is formalized in Def. 2.1.

DEFINITION 2.1. ***Contextual Constraint.*** *Given a sequence S and all length-M frequent/outlier patterns in S denoted as $\mathbb{P}$, an occurrence $O_Q = \langle (e_1, t_1)\ (e_2, t_2) \dots (e_L, t_L) \rangle$ of a length-L pattern $Q = \langle E_1, E_2, ..., E_L \rangle$ where $L < M$, is said to satisfy the contextual constraint, if there exists an event $e_j \in O_Q$ such that $e_j \notin O_P$, for any $P \in \mathbb{P}$.*

Beyond the contextual constraint, other constraints in the literature continue to be applicable, such as the gap constraint [5].

DEFINITION 2.2. ***Gap Constraint.*** *Given a pattern $P = \langle E_1 E_2 \dots E_m \rangle$ and a sequence gap constraint seqGap, an occurrence $O_P = \langle (e_1, t_1)\ (e_2, t_2) \dots (e_m, t_m) \rangle$ of P has to satisfy the condition: t_m-t_1-$1 \leq$ seqGap.*

The sequence gap constraint accounts for the gap between the first event and the last event. It can be expressed either by the count of the events or by the time difference. O_P is not considered a valid occurrence of P if it does not satisfy the gap constraint. The gap constraint models the timeliness of the system behavior in IoT. In our infection control app. [31], the pattern $\langle HandWash, Enter, ApproachPatient \rangle$ representing the hygiene compliance regulation (Sec. 1) restricts the time between the hand hygiene and approaching patient. Otherwise the hand hygiene behavior of the HCW might not assure the cleanliness required when approaching the patient.

2.2.2 Contextual Pattern Semantics

Next, we formally define our contextually frequent (CF) pattern and contextual outlier (CO) pattern semantics.

First, we define the notion of *contextual support* used in both CF and CO semantics.

DEFINITION 2.3. ***Contextual Support.*** *Given a length-L pattern $Q = \langle E_1, E_2, ..., E_L \rangle$, the* ***contextual support*** *of Q in sequence S denoted as $CSup_S(Q)$ is defined as the size of the occurrence set $\mathbb{O}_Q$ of pattern Q in S. Here each $O_Q \in \mathbb{O}_Q$ satisfies the condition: (1) O_Q satisfies the contextual constraint; (2) $O_Q \cap O'_Q = \emptyset$, $\forall$ $O'_Q \in \mathbb{O}_Q$ and $O'_Q \neq O_Q$.*

By Def. 2.3, given a pattern Q when computing its contextual support in a sequence S ($CSup_S(Q)$), only the occurrences that satisfy the contextual constraint are considered. Further, following the common practice used in sequential pattern mining [23, 27, 30, 34], these occurrences are not allowed to overlap with each other.

DEFINITION 2.4. ***Contextually Frequent (CF) Pattern.*** *Given a minimal support threshold minSup, a pattern $Q = \langle E_1, E_2, ..., E_m \rangle$ is said to be contextually frequent (CF) in sequence S if $CSup_S(Q) \geq minSup$.*

Intuitively, a pattern Q will be a CF pattern in sequence S if Q occurs frequently in S independent of any of its CF super-patterns.

The contextual outliers capture the patterns that occur rarely and violate the typical patterns in the system as formally defined next.

DEFINITION 2.5. ***Contextual Outlier (CO) Pattern.*** *Given a count threshold cntThr, a pattern Q in a sequence S is said to be a contextual outlier w.r.t. a CF pattern P in S, if Q satisfies the following conditions:*

(1) $Q \sqsubset P$; or $\forall$ event type $E_i \in Q$, $E_i \in P$ and $length(Q) = length(P)$ and $Q \neq P$;

(2) $CSup_S(Q) \leq cntThr$;

By Def. 2.5, a pattern Q is a contextual outlier w.r.t. P, if Q is a sub-pattern or a different event type permutation of P (Condition 1) and Q occurs *rarely* in sequence S of D (Condition 2). For example, given a CF pattern $P = \langle ABBC \rangle$ in S, pattern $Q = \langle ABB \rangle$ and pattern $R = \langle ABCB \rangle$ are CO w.r.t. P if Q and R are rare in S. Of course a CO pattern has to satisfy the contextual constraint defined in Definition 2.1.

The CO definition captures sub-pattern outliers that have missing expected events. For example the outlier pattern in the mobile application that misses the expected termination action after finishing a trip, or, in the infection control system the outlier pattern that misses the hand-wash event before the approaching-patient event.

CO also captures the outlier pattern $O = \langle DatabaseUpdate, LocationUpdate \rangle$ in the mobile app, where two threads monitor the driver's location and the database operations separately. Usually the operation of reporting a location update is followed by a database update operation. Therefore $P = \langle LocationUpdate, DatabaseUpdate \rangle$ tends to frequently occur and represent typical system behavior. However, occasionally – although rarely– the database updates occur earlier than the location report due to a scheduling error. This violates the time dependency between location update operation and database update operation represented by pattern P. It is captured by CO semantics, because O is a different event type permutation of typical pattern P.

3. THE OVERALL APPROACH

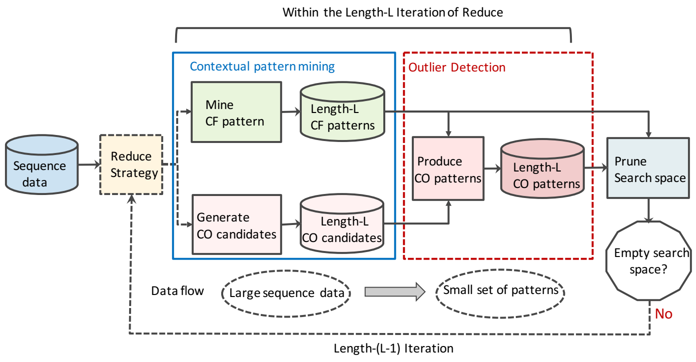


Figure 2: The architecture of TOP

TOP employs a **one-pass** outlier pattern detection strategy that successfully discovers the *CO* patterns by mining the sequence data only once. We now describe the overall structure of TOP.

As shown in Fig. 2, the core of TOP is a top-down contextual pattern mining strategy, called *Reduce*. Starting from the longest possible contextual patterns, Reduce recursively constructs shorter patterns in descending order from the longest to the shortest. In each iteration it ensures that the largest possible patterns are discovered.

First, built upon the Reduce strategy, TOP produces the CF patterns in one step by piggybacking pattern detection with the detection of its context in which a pattern occurs, in contrast to the two-step Growth method. This will be further described in Sec. 4.2. At the same time, TOP produces the infrequent contextual patterns, as the candidates of CO, almost *for free* as a by-product of the top-down CF pattern mining process. This will be further demonstrated in Sec. 4.3. The mining of CF patterns and the generation of CO candidates correspond to the *contextual pattern mining* component in Fig. 2.

Then, the *outlier detection* component produces the CO patterns using the CF patterns and CO candidates as input. The detection of contextual outliers is embedded into the iterative top-down mining process of Reduce. In other words, the length-L CF and CO patterns are mined in the same iteration of Reduce as shown in Fig. 2. Hence, TOP successfully discovers the CO patterns by mining the input sequence only once. The details are presented in Sec. 5.

4. CONTEXTUAL PATTERN MINING

We now present the Reduce strategy for mining contextually frequent (CF) patterns and producing the contextual outlier (CO) candidates from one sequence S. We first introduce how Reduce mines CF patterns. Then we show how to extend Reduce to produce the CO candidates almost for free.

4.1 Challenges with Adopting Traditional Strategies for CF Pattern Problem

The contextual constraint complicates the CF pattern mining process. The contextual support of a short pattern is influenced by the status of patterns longer than it. This contradicts the well-known idea of Apriori, where if a short pattern is infrequent, then its super-patterns are guaranteed to also be infrequent and can be excluded from frequent pattern candidates. Thus, Apriori relies on the status of the short patterns to predict the status of the longer patterns. However, we now have the important observation that in CF semantics the contextual support of a short pattern which determines whether it is a CF pattern cannot be determined without mining longer patterns first.

LEMMA 4.1. *Given a pattern Q and the number of its occurrences in sequence S denoted as $sup_S(Q)$ ($sup_S(Q) \geq minSup$), Q is not a CF pattern if its supper pattern P is a CF pattern, where the contextual support of P $CSup_S(P) > sup_S(Q) - minSup$.*

PROOF. By the definition of CF patterns (Definition 2.4), an occurrence of pattern Q is not counted to its contextual support $CSup_S(Q)$ if it is a sub-occurrence of pattern P, where P is a CF pattern and P is a super-pattern of Q. Therefore, $CSup_S(Q) \leq sup_S(Q) - CSup_S(P)$. Since $CSup_S(P) > sup_S(Q) - minSup$, we get $CSup_S(Q) \leq sup_S(Q) - CSup_S(P) < sup_S(Q) - sup_S(Q) + minSup = minSup$. Therefore, Q is not a CF pattern in this case. ☐

Two-step Apriori Style Solution. One solution to this problem would be to mine the CF patterns in two steps. As shown at the left of Fig. 3, the first step is a traditional Apriori-based mining strategy that finds a set of frequent patterns that are not subject to the contextual constraint. It is a superset of the final contextual frequent pattern set. Since Apriori still holds, the classic pattern-growth strategy widely used in the traditional pattern mining techniques [16, 37, 4] can be used to find these patterns. The second step then filters these frequent patterns that violate the *contextual constraint*. We call this pattern growth-based strategy *Growth*.

Drawbacks of Growth. Although this basic *Growth* approach prunes the search space based on the Apriori property, it has several drawbacks. First, when a large number of long and frequent patterns are generated, it wastes significant CPU and memory resources to generate and then maintain all of the shorter, frequent sub-patterns, most of which are discarded in the second step. Given a length-n pattern, in the worst case it may have to produce and maintain $(2^n - 2)$ sub-patterns. As confirmed in our experiments, this requires a significant amount of memory – sometimes causing an out-of-memory error – when handling long input sequences.

Worst, the backward filtering introduces extra cost associated with recursively updating the support of these shorter patterns and filtering those that do not conform to the contextual constraint.

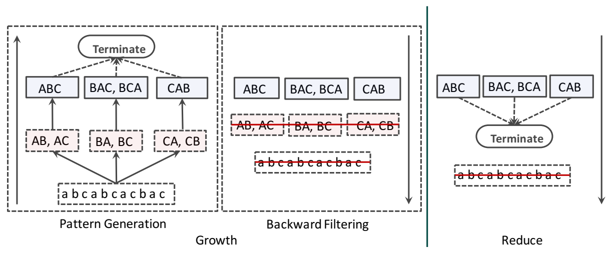


Figure 3: Growth vs Reduce

It may seem that this filtering operation can be supported by traversing backward along the path that grows from the first event in the prefix of P to P. An example is given in Fig. 4. We can traverse from $\langle ABC\rangle$ to $\langle A\rangle$ to adjust the support of the sub-patterns in $\langle ABC\rangle$. However, this method does not work, because this path only contains the sub-patterns of P that have A as prefix. Other sub-patterns of P may exist outside of the path from A. For example in Fig. 4, $\langle BC\rangle$ is a sub-pattern of CF pattern $\langle ABC\rangle$. However, $\langle BC\rangle$ is not in the path from $\langle A\rangle$ to $\langle ABC\rangle$. Therefore, given an occurrence of CF pattern P, the backward filtering has to be conducted by searching for all its sub-patterns and matching each of its occurrences with each occurrence of any of its sub-patterns. This tends to introduce prohibitive costs due to the potentially huge number of sub-patterns.

4.2 Reduce: Reduction-based CF pattern Mining

To address these drawbacks in Sec. 4.1, we develop a top-down pattern mining approach that directly mines the CF patterns in one step. The key innovation of this reduction-based approach, called *Reduce*, is that instead of treating the contextual constraint as a performance bottleneck that requires an expensive post-processing step, Reduce leverages it to effectively prune the search space of pattern mining by ensuring longer patterns are found before mining a shorter one. During the top-down mining process, pruning is continuously applied in each iteration. This way, Reduce constructs short patterns only when necessary. Also, Reduce terminates immediately once no available event exists that can form a valid CF pattern. Both our theoretical analysis and experimental evaluation confirm Reduce's efficiency in processing complex sequence data. It has two steps: *search space construction* and *CF pattern mining* as described in detail below.

Table 1: Search space and Growth & Reduce Strategies

Event	Search space	Growth	Reduce	CF
A	$\langle(a,0)(b,1)(c,2)\rangle$ $\langle(a,4)(b,5)(c,6)(a,7)\rangle$ $\langle(a,7)(c,8)(b,9)\rangle$ $\langle(a,11)(c,12)\rangle$	$\langle A\rangle$ $\langle AB\rangle$ $\langle ABC\rangle$ $\langle AC\rangle$	$\langle ABC\rangle$	$\langle ABC\rangle$
B	$\langle(b,1)(c,2)(a,4)\rangle$ $\langle(b,5)(c,6)(a,7)(c,8)\rangle$ $\langle(b,9)(a,11)(c,12)\rangle$	$\langle B\rangle$ $\langle BA\rangle$ $\langle BAC\rangle$ $\langle BC\rangle$ $\langle BCA\rangle$	$\langle BAC\rangle$ $\langle BCA\rangle$	$\langle BAC\rangle$ $\langle BCA\rangle$
C	$\langle(c,2)(a,4)(b,5)\rangle$ $\langle(c,6)(a,7)(c,8)(b,9)\rangle$ $\langle(c,8)(b,9)(a,11)\rangle$ $\langle(c,12)\rangle$	$\langle CA\rangle$ $\langle CAB\rangle$ $\langle CB\rangle$	$\langle CAB\rangle$	$\langle CAB\rangle$

4.2.1 *Reduce Search Space Construction*

Given a sequence S, we construct a search space $\mathbb{SS}_i$ for each *frequent* event type E_i in S. The search space $\mathbb{SS}_i$ of each E_i is composed of a set of sequence segments (or subsequences). Each subsequence starts with a different E_i type event as prefix. The reasons are twofold. First, using this search space, the largest possible pattern with E_i as prefix can be easily determined.

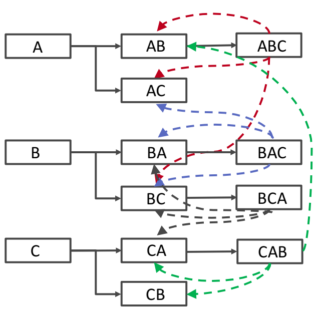


Figure 4: Backward filtering

Finding the largest possible pattern is critical for Reduce. Second, it is for the ease of counting the occurrences of a pattern, as we will show in Lemma 4.2. To illustrate the search space, we use a running example with the input sequence $S = \langle(a,\mathit{0})(b,\mathit{1})(c,\mathit{2})(d,\mathit{3})(a,\mathit{4})(b,\mathit{5})\ (c,\mathit{6})(a,\mathit{7})(c,\mathit{8})\ (b,\mathit{9})\ (e,\mathit{10})\ (a,\mathit{11})\ (c,\mathit{12})\rangle$. The *minSup* and *seqGap* are both set as 2.

First, in one scan of the sequence, all event types with frequency $\geq$ *minSup* are identified as frequent event types. All infrequent event types are filtered from S such as D and E, because they cannot appear in any CF pattern. Next, given a frequent event type E_i, the filtered sequence is divided into multiple subsequences. Each subsequence starts with one type E_i event and stops whenever its length reaches $seqGap + \mathit{2}$ or it hits the end of S. The search spaces for each E_i in the example are shown in the second column of Tab. 1. Note the subsequences could overlap with each other.

Intuitively for each event type E_i, the largest possible pattern with E_i as prefix corresponds to the longest subsequence in $\mathbb{SS}_i$, because no subsequence longer than it would satisfy the *gap constraint* defined in Def. 2.2. In other words its length is at most $seqGap + \mathit{2}$.

Next, we show that each occurrence of P with prefix E_i can be found from one subsequence in search space $\mathbb{SS}_i$ of E_i.

LEMMA 4.2. *Given a sequence S and a pattern candidate P with prefix E_i, any two occurrences O_P^1 and O_P^2 of P in S can be found from two different subsequences S_1 and S_2 in the $\mathbb{SS}_i$ of E_i.*

Proof. First, any occurrence O_P of P in S is guaranteed to be contained in a subsequence S' of $\mathbb{SS}_i$ that starts with the first event of O_P (type E_i event). Otherwise O_P will violate the gap constraint, because the gap between the first event and the last event of O_P is larger than $seqGap$. Therefore no valid occurrence O_P of P will be missed when searching for each occurrence O_P independently in each subsequence in the search space $\mathbb{SS}_i$.

Second, given two occurrences O_P^1 and O_P^2, which by definition are not overlapping, the first event of O_P^1 and O_P^2 must corresponding to two different type E_i events. Since any type E_i event e_i has one distinct subsequence in $\mathbb{SS}_i$ that starts from e_i, O_P^1 and O_P^2 must be from two different subsequences. Lemma 4.2 is proven. ■

Lemma 4.2 not only proves that all occurrences of any given pattern can be discovered from the search spaces, it also inspires an efficient occurrence-based search method. That is, given a candidate pattern P with prefix E_i and the search space $\mathbb{SS}_i$ corresponding to E_i, when searching for the occurrence of O_P in one subsequence S_i, we only have to search for an occurrence that starts from the first event of S_i and stop immediately once we find it.

For example, given a sequence $S = \langle(a,\mathit{0})(b,\mathit{1})(a,2)(b,3)\ (a,\mathit{4})\ (b,\mathit{5})\rangle$, suppose $seqGap$ = 2, then the search space $\mathbb{SS}$ w.r.t. event type A is composed of subsequences $S_1 = \langle(a,0)(b,1)(a,2)(b,3)\rangle$, $S_2 = \langle(a,2)(b,3)(a,4)(b,5)\rangle$, and $S_3 = \langle(a,4)(b,5)\rangle$. Given a pattern candidate AB, it has three occurrences: $\langle(a,0)(b,1)\rangle$, $\langle(a,2)(b,3)\rangle$ and $\langle(a,4)(b,5)\rangle$. Each corresponds to one subsequence in $\mathbb{SS}$. Although S_1 contains two occurrences of AB, the second one $\langle(a,2)(b,3)\rangle$ is not counted. It is captured in subsequence S_2 that starts with event $(a,2)$. Similarly, the occurrence $\langle(a,4)(b,5)\rangle$ in S_2 is ignored. It is found in S_3 that starts with event $(a,4)$.

4.2.2 *The Reduce Strategy*

The Reduce approach features two key ideas, namely *top-down mining* and *event availability-based pruning*.

Top-down Mining Strategy. The top-down mining strategy ensures that in each iteration of the mining process, only the largest CF patterns are generated. Since their contextual supports will not be influenced by any shorter pattern produced later, this saves the post-processing for support adjustment. To achieve this, unlike the traditional Growth strategy which constructs frequent patterns of different prefixes independently, Reduce mines the patterns with different prefixes *simultaneously*. This ensures that the CF patterns with the same length are produced in the same iteration even if they have different prefixes.

Event Availability-based Pruning. Reduce prunes the subsequences in the search space that cannot or do not contain a valid occurrence of a CF pattern. This amounts to an alternative pruning strategy to the Apriori property from traditional bottom up pattern mining. Reduce performs this pruning by dynamically maintaining and updating the "availability" of each event in sequence S, where an event is available if it has not been used by a longer pattern. The event availability is shared by all patterns with different prefixes, which we call *global availability*. In each iteration of the top-down mining process, after the length-L CF patterns are produced, all events utilized by the length-L CF patterns are marked as unavailable. In this way, the subsequences in the search space that do not contain any event that is still "available" can be safely discarded as shown in Lemma 4.3.

LEMMA 4.3. *Given a search space $\mathbb{SS}_i$ of event type E_i, after producing the length-L CF patterns, a subsequence $S_j \in \mathbb{SS}_i$ can be discarded if $\forall$ event $e_x \in S_j$, $e_x \in$ occurrence O_P of P, where P is a CF pattern.*

PROOF. By the definition of contextual constraint (Definition 2.4), an occurrence of a pattern Q is invalid if all events $e_x \in Q$ have been utilized by the patterns P, where length(P) > length(Q). Since all events $e_j \in S_j$ have been used by the length-L_l patterns where $L_l \geq L$, S_j cannot produce any valid occurrence in the mining of length-L_s patterns, where $L_s < L$. Therefore, S_j can be removed in the later iterations. □

This event availability-based pruning significantly reduces the search space for the later evaluation of shorter pattern candidates.

The Overall Reduce Algorithm. Alg. 1 shows the overall process. We explain *Reduce* by our running example with input sequence $S = \langle(a, \mathit{0})(b, \mathit{1})(c, \mathit{2})(d, \mathit{3})\ (a, \mathit{4})(b, \mathit{5})(c, \mathit{6})(a, \mathit{7})$ $(c, \mathit{8})(b, \mathit{9})(e, \mathit{10})(a, \mathit{11})(c, \mathit{12})\rangle$, *minSup* = 2 and *seqGap* = 2 used in the gap constraint (Def. 2.2). The global availability (Line 2) is initialized as [1111111111111].

(1) **Start**. Reduce starts by mining the longest possible patterns, whose lengths are determined as follows. Reduce first computes for each frequent event type $E_i \in \mathbb{E}$ the maximum length of any subsequence in its search space, denoted as $eMax$. Then a global maximum length $gMax$ $max\{eMax|E_i \in \mathbb{E}\}\}$ is computed. $gMax$ is at most equal to $seqGap + \mathit{2}$. This corresponds to the length of the longest possible pattern, because no valid pattern can be longer than the longest subsequence due to the gap constraint (Def. 2.2).

In our example, $gMax = \mathit{4}$, since the longest subsequences are $\langle abca\rangle$, $\langle bcac\rangle$ and $\langle cacb\rangle$ (See Tab. 1).

Algorithm 1 Reduction-based approach

```
 1: function REDUCE(seqGap, minSup, prefixSet)
 2:     globalAvail[1 ... $n$] ← global availability, init all true
 3:     gMax = computeGlobalMaxPatternLength(seqGap,prefixSet)
 4:     currentLen = gMax ← global maximum pattern length
 5:     while currentLen > 0 & $prefixSet$.size() > 0 do
 6:         curFreq = []
 7:         for ∀ prefixSet[$i$] ∈ $prefixSet$ do
 8:             if prefixSet[$i$].maxLen ≥ currentLen then
 9:                 curFreq.add(CONSTRUCTCF(prefixSet[$i$], currentLen, minSup, globalAvail))
10:         for ∀$e_j$ ∈ curFreq do
11:             globalAvail[j]=false
12:         for ∀ prefixSet[$i$] ∈ prefixSet do
13:             UPDATEPREFIX(prefixSet[$i$], globalAvail)
14:             if prefixSet[$i$].SS == null then
15:                 prefixSet.remove(prefixSet[$i$])
16:         freqPatterns.addAll(curFreq)
17:         currentLen = currrentLen-1
18:     return freqPatterns
```

Once $gMax$ is determined, Reduce starts mining the length-$gMax$ patterns (Line 4). Reduce examines whether the longest subsequences can form CF patterns by grouping together the subsequences that are the occurrences of the same pattern and counting the subsequences in each group.

In our example, there exits only one subsequence with length ≥ 4 in the search space of each event type, that is $\langle abca\rangle$ for event type A. Since the support threshold $minSup$ is set to 2, no length-4 CF pattern can be generated.

(2) **Search Space Pruning.** If a length-$gMax$ sequence P is found to be frequent, then the events in each occurrence of P are marked as unavailable in the global availability array by our event availability maintenance strategy. This takes constant time for each event. This operation is conducted only after all length-$gMax$ patterns are processed (Lines 10-11). The search space associated with each prefix is updated based on the latest global availability (Lines 12-13). By Lemma 4.3 the subsequences in the search space that do not contain any event that is still "available" are discarded. Thereafter those prefixes that have an empty search space are removed from the prefix list (Lines 14-15).

(3) **CF Pattern Construction.** After processing the length-$gMax$ patterns, Reduce recursively constructs shorter patterns in a descending order from length-($gMax$-1) to length-1 (Line 17). This mining process terminates when length-1 patterns are generated or no prefix is available in the prefix list (Line 5). The details are shown in Alg. 2 ($constructCF$ subroutine).

Algorithm 2 CF pattern construction for prefix i

```
 1: function CONSTRUCTCF(Object prefix[$i$], int currentLen, int minSup, boolean[] globalAvail)
 2:     subseqs ← HashMap<String, Integer>
 3:     usedPositions ← HashMap<String,BitMap>
 4:     freqSeqs = []
 5:     for seq ∈ prefix[$i$].SS do                          ▷ Iterate each sequence
 6:         Set subs = FINDSUBSEQS(seq, currentLen, globalAvail)
 7:         for $O_P$ ∈ subs do
 8:             if NOTUSED($O_P$, usedPositions.get($P$)) then
 9:                 subseqs.put($P$, subseqs.get($P$)+1)
10:                 SETUSED($O_P$, usedPositions.get($P$))
11:     for $P$ ∈ subseqs do
12:         if subseqs.get($P$) ≥ minSup then
13:             freqSeqs.add($P$)
14:     return freqSeqs
```

The **ConstructCF** subroutine mines length-l CF patterns for a prefix E_i. First, it generates all length-l occurrences from the search space of E_i. Only the subsequences containing at least l

events are considered. Since the subsequences in the search space of E_i are indexed by length, locating subsequences with length more than l can be done in constant time. In the running example, to generate length-3 CF patterns for prefix-$\langle B \rangle$, all three subsequences, namely $\langle bcac \rangle$, $\langle bca \rangle$ and $\langle bac \rangle$ would be considered. The length-4 subsequence $\langle bcac \rangle$ could generate occurrences of the three length-3 sub-patterns $\langle BCA \rangle$, $\langle BCC \rangle$, and $\langle BAC \rangle$. The two length-3 subsequences could generate occurrences of $\langle BCA \rangle$ and $\langle BAC \rangle$ respectively. Each occurrence has to contain the first event of the corresponding subsequence by Lemma 4.2.

Next, *constructCF* evaluates whether each occurrence satisfies the contextual constraint. We have maintained the availability of each event. Therefore, to determine whether an occurrence O_P of pattern P satisfies the contextual constraint, we check whether all events in O_P have been marked as unavailable. If at least one event remains available, then O_P satisfies the contextual constraint.

Further, besides the contextual constraint, given one occurrence we also need to examine if it overlaps with any existing occurrence of the same pattern P, because the subsequences in one search space might overlap with each other. For the efficiency of this examination, given one pattern P, we utilize a bitmap to maintain the event availability specific to pattern P, so called *local availability*. The events used by all previous occurrences of P are marked as unavailable (Line 3). An occurrence O_P is valid only if *all* events in it remain valid (Line 8). The non-overlapping occurrences O_P of pattern P are inserted into a pattern candidate hash map using the pattern as key and the number of occurrences as value (Line 9). If O_P is valid, the local availability bitmap is updated. All events used by O_P are marked as unavailable (Line 10).

After all length-l non-overlapping pattern occurrences that satisfy the contextual constraint have been generated, the CF patterns are discovered by traversing the pattern candidate hash map. A pattern is a CF if its count (value in hash map) is larger or equal to the $minSup$ threshold (Lines 12-13).

For example, for prefix-$\langle B \rangle$ patterns, the supports of $\langle BCA \rangle$ and $\langle BAC \rangle$ are 2. Therefore $\langle BCA \rangle$ and $\langle BAC \rangle$ are CF patterns, while $\langle BCC \rangle$ only has support 1 and is not a CF pattern.

Finally, after all length-l CF patterns w.r.t. all prefixes have been generated, the search space pruning (Step 2 of Alg. 1) is triggered. In our example, besides the two length-3 CF patterns with $\langle B \rangle$ as prefix, prefix $\langle A \rangle$ has one length-3 CF pattern $\langle ABC \rangle$. Prefix $\langle C \rangle$ has one length-3 CF pattern $\langle CAB \rangle$. All events covered by the occurrences of the four length-3 patterns are marked as unavailable in the global availability bitmap. In this case, it is updated to [0001000000100]. Since now all events of the frequent event types are unavailable, the process terminates without attempting to generate length-2 and length-1 pattern. Therefore, *Reduce* avoids generating patterns shorter than 3, in contrast to *Growth* as shown at the right of Fig. 3. None of the patterns produced at the early iteration will be pruned during the later iterations.

4.2.3 Further Optimization of Reduce

We further enhance Reduce with two optimization strategies, called *Start Length Minimization* and *Pattern Candidate Pruning*. Our experiments in Sec. 6.4 confirm the effectiveness of these optimizations in speeding up the basic *Reduce* approach.

Start Length Minimization. As noted in Sec. 4.2.2, the start pattern length $gMax$ determines the number of iterations of the top-down mining process. Therefore minimizing $gMax$ has the potential to significantly improve the efficiency of *Reduce*. Reduce should start with a $gMax$ as close as possible to the actual length of the final longest CF pattern. As shown in Algorithm 1, our basic Reduce approach uses as $gMax$ the length of the longest subsequence in the search spaces of all prefixes. Although it is guaranteed to generate the correct results, $gMax$ is often far from the optimal start length. Our start length minimization optimization solves this problem based on the observation in Lemma 4.4.

LEMMA 4.4. *Given an event type E, the maximum length of the frequent pattern with E as prefix, denoted as M_E, is guaranteed to be no larger than the length of the $minSup$-th longest subsequence in the search space of E, denoted as L_E, that is $M_E \leq L_E$.*

Proof Sketch. This lemma holds because given a length-M_E sequence pattern P under prefix E, P cannot be frequent unless there are at least $minSup$ subsequences equal to or longer than M_E in the search space of E. Since there are at most $minSup$ length-L_E subsequences, no frequent pattern longer than L_E can be formed. ■

L_E tends to be much smaller than the length of the longest subsequence in the search spaces of all prefixes. Therefore replacing $gMax$ with L_E as the start length for the corresponding prefix E helps reduce the start length.

Based on the above observation, we devise a lightweight mechanism to compute a *customized* start length for each prefix E. Specifically, given a prefix E, we first sort all subsequences in its search space in descending order of length. Then the start length of prefix E patterns is set as L_E, where L_E represents the length of its $minSup$-th longest subsequence. In the example shown in Tab. 1, since $minSup = 2$, for each prefix the start length is set as the length of the second longest subsequence in its search space. In this case, the start lengths of prefixes A B, C are all set to 3 instead of $gMax = 4$.

Pattern Candidate Pruning. An event is filtered out from the original input sequence S if its *total frequency* is smaller than $minSup$. This is done because it cannot be part of any frequent pattern. Although this simplistic total frequency based optimization reduces the number of events to be considered in the mining process, it fails to capture other optimization opportunities to further reduce the number of CF pattern candidates.

The insight here is that a stricter filtering criteria can be derived if we scrutinize the count of the events at the subsequence granularity.

LEMMA 4.5. *Given a prefix E_i and another event type E_j, E_j cannot be involved in any length-l frequent pattern with E_i as prefix, if E_j does not appear in at least $minSup$ subsequences in the search space of E_i, where the length of each subsequence has to be at least l.*

Proof Sketch. By Lemma 4.2 each length-x subsequence ($x \geq l$) in the search space of E_i can only form at most one occurrence of a length-l frequent pattern with E_i as prefix. Therefore if type E_j events are not found in at least $minSup$ length-x subsequences contained in the search space of E_i, no length-l CF pattern can be formed that has E_i as prefix and contains E_j. ■

This optimization can be applied in each stage of the CF pattern mining process at each iteration of decreasing length. The subsequence count of each event type can be dynamically updated when refreshing the global availability and pruning the subsequences (Algorithm 1). This keeps the cost of this optimization low.

Further, after the events are pruned, the gap between each adjacent event in an occurrence of a pattern candidate increases. Therefore the gap threshold $seqGap$ (Def. 2.2) becomes more effective at pruning the candidates that violate the gap constraint.

4.2.4 Time Complexity Analysis

Let N be the length of the input sequence S. $M = seqGap + 2$ denotes the maximum length of possible frequent sequences. E

denotes the number of unique events. $minSup$ denotes the minimum support threshold. One length-M occurrence can generate $C_{M-1}^{M-2} = (M-1)$ length-(M-1) candidate sub-patterns. Generating each candidate takes $O(M-1)$. Since the input sequence has no more than N length-M occurrences, the time complexity of generating all length-(M-1) sub-patterns from length-M sequences is $N(M-1)^2$. Similarly, generating length-(M-2) sub-patterns from length-(M-1) sequences takes $N(M-2)^2$. Overall the time complexity of generating all sub-pattern candidates (from length-$M-1$ to length-1) is $\frac{NM(M+1)(2M+1)}{6}$.

Discovering frequent sub-patterns out of these candidates requires one scan of all candidates. So the time complexity depends on the total number of candidate frequent sequences. Given length l, the number of length-l sequence candidates is $O(\frac{N}{minSup})$ in the worst case. Therefore, the total number of candidates from length-M to length-1 pattern is $O(\frac{NM}{minSup})$.

The time complexity of updating the event availability depends on the number of occurrences and the length of each occurrence. Since S cannot have more than N length-M occurrences, event availability update for all length-M sequences takes at most NM. In the worst case event availability update has to be conducted on all occurrences of all patterns from length-M to length-1. In total, it takes $O(\frac{NM(M+1)}{2})$.

In general, the total time complexity of Reduce is $O(\frac{NM(M+1)(2M+1)}{6} + \frac{NM}{minSup} + \frac{NM(M+1)}{2})$. Therefore, it is linear in the length of the input sequence, while the Growth strategy adapted from the literature is shown to have a time complexity quadratic in both the sequence length and the number of event types.

4.3 CO Candidate Generation

In this section we show how to produce the infrequent patterns as the CO candidates from each sequence. By extending the Reduce strategy the CO candidates are produced during the CF pattern mining process almost for free. It avoids the mining of the input sequence twice.

As described in Sec. 4.2, Reduce recursively constructs shorter patterns in descending order from length-(gMax-1) to length-1. In the iteration of mining length-L patterns, after producing the CF patterns, we also maintain *infrequent patterns* with local support no larger than *cntThr* (Def. 2.5). It is straightforward. As shown in Sec. 4.2.2, in each iteration Reduce stores all possible patterns in a hash map with the pattern as the key and the number of subsequences containing the occurrences of this pattern as value. Therefore, the infrequent patterns are *naturally* discovered almost for free when Reduce scans the candidate hash map to find CF patterns. Namely, a pattern is infrequent if its value in the hash map is no larger than the $cntThr$ threshold. Furthermore, as shown in Sec. 4.2.2, when producing the occurrence of each pattern, Reduce already evaluates whether the occurrence satisfies the contextual constraint. Therefore, the captured infrequent patterns must meet the contextual constraint. This way, Reduce not only detects CF patterns but also generates the infrequent CO candidates.

Note this simplistic CO candidate generation method relies on our important observation on the search space of the CO patterns as shown below.

LEMMA 4.6. *Given a sequence S, the search space constructed for mining CF patterns is sufficient to mine all CO patterns.*

PROOF. By the definition of CO (Def. 2.5), a CO pattern Q is a sub-pattern or a different event type permutation of a CF pattern P. Therefore, $\forall$ event type $E_i \in Q$, $E_i \in P$. Therefore, each $E_i \in Q$ is guaranteed to be a frequent event type. Therefore, using the search space constructed only based on the frequent event types will not miss any CO. Lemma 4.6 is proven. $\square$

5. MINING CONTEXTUAL OUTLIER PATTERNS

Next, we present our outlier pattern detection approach, which generates *contextual outlier* (CO) patterns for each sequence S. Leveraging the property of our top-down Reduce strategy, we show that the length-L CO patterns are discovered in the same iteration as mining length-L CF patterns. This effectively solves the complication caused by the interdependence between CF patterns and CO patterns. By the contextual constraint if an occurrence O_Q of Q is a sub-occurrence of an outlier super-pattern P, it should not be counted to the support of Q when mining CF patterns. That is, the mining of the CF patterns also relies on the CO patterns.

LEMMA 5.1. *Given any length-L pattern Q, if Q is an CO, then it can be discovered immediately after Reduce produces the length-L CF patterns.*

PROOF. By the definition of CO (Definition 2.5), a length-L infrequent pattern Q is a candidate CO only if Q is a sub-pattern of CF pattern longer than it or a different permutation of a CF pattern that has the same length with Q. Therefore, once an infrequent length-L pattern Q is acquired, we can immediately determine whether it has a chance to be a CO, because all CF patterns with length $\geq L$ have already been found in the top-down mining process of Reduce. The CF patterns discovered in the future will not turn Q into a CO. Lemma 5.1 is proven. $\square$

Therefore, by Lemma 5.1 the CO pattern mining can be interleaved with each iteration of the CF pattern mining. That is, after the length-L CF patterns are produced, the length-L CO patterns are mined based on the CF patterns produced so far and the length-L CO candidates. Once an infrequent CO candidate is acquired, we determine immediately whether it is a CO pattern. An infrequent length-L CO candidate will be discarded immediately if it is confirmed not to be a CO, since it will not have chance to be a CO anymore in the future iteration. This significantly reduces the number of CO candidates to be maintained in each iteration.

Next, we show how producing length-L CF and CO patterns in one iteration solves the complication caused by the interdependence between CF patterns and CO patterns. After the length-L CO patterns are mined, the events used by the occurrences of the CO patterns are marked as unavailable. Then when mining the length-*(L-1)* contextual patterns in the next iteration, the contextual constraint can be simply evaluated based on the event availability using the strategy described in Sec. 4.2.2. This is because by Definition 2.3 the contextual support of length-*(L-1)* patterns are influenced only by the CF and CO patterns with length $\geq L$, while all events that are used by such CF and CO patterns have already been marked as unavailable.

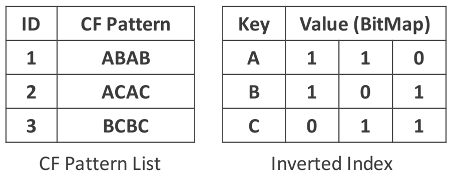
Similarly, the event availability-based pruning (Lemma 4.3, Sec. 4.2.2) can be equally applied here to prune the search space of contextual patterns. The subsequences that do not contain any available event are discarded immediately from the search space.

5.1 Discovering CO with Inverted Index

To discover CO patterns from the CO candidates, we have to examine each CO candidate Q to see whether there is a CF pattern P that is a super-pattern or a different event type permutation of Q. A naive solution will be comparing Q against each CF pattern in the CF pattern list. This tends to be expensive, especially when the number of CO candidates and CF patterns is large.

To accelerate this process, we build an *inverted index* on the existing CF patterns. Given a CO candidate Q, the inverted index assists Reduce to quickly discover a small subset of CF patterns that are sufficient to prove if Q is a CO.

Build Index. We first show how to build an inverted index on the CF patterns stored in the CF pattern list $\mathbb{CF}$. The inverted index is a HashMap with the event type E as key and a bitmap as value. This bitmap records what patterns event type E is involved in. Thus, each bit in it corresponds to one CF pattern. If one type E event is involved in the i-th pattern of the CF pattern list denoted as $\mathbb{CF}[i]$, the i-th bit in the bitmap is set to "1". Otherwise, it is set to "0".



ID	CF Pattern
1	ABAB
2	ACAC
3	BCBC

CF Pattern List

Key	Value (BitMap)		
A	1	1	0
B	1	0	1
C	0	1	1

Inverted Index

Figure 5: Inverted Index on CF Patterns

Fig. 5 shows an example of an inverted CF pattern index. The CF patterns with IDs 1 and 2 both contain "A" event. Therefore, the first and second positions of A's bitmap are set to 1, while the third position corresponding to the CF pattern with ID 3 is set to 0. The bitmaps of B and C are constructed in the same fashion.

Index-based CO candidate Examination. Next, we show how to use the inverted index to efficiently determine whether a pattern Q is a CO. To do this, we first get the bitmaps corresponding to the event types in Q using the inverted index, denoted as $\{B_1, B_2, \ldots, B_m\}$, where B_i ($1 \leq i \leq m$) represents the bitmap of event type E_i in Q. Then we generate a new bitmap B_n by bit operation: $B_n = B_1 \& B_2 \& \ldots \& B_m$. If the i-th bit of B_n $B_n[i]$ is "0", then it is not necessary to examine the corresponding i-th pattern in the CF pattern list $\mathbb{CF}[i]$ when evaluating whether Q is a CO as shown in Lemma 5.2.

LEMMA 5.2. *Given the bitmap $B_n = B_1 \& B_2 \& \ldots \& B_m$, if $B_n[i]$ = 0, then $\mathbb{CF}'$ is sufficient to prove whether Q is a CO, where $\mathbb{CF}'$ = $\mathbb{CF} \setminus \{\mathbb{CF}[i]\}$.*

PROOF. As shown in Def. 2.5, an infrequent pattern Q is a CO only if either it is a sub-pattern or a different permutation of one CF pattern. In both cases, the corresponding CF patterns that Q violate must include all event types that Q contains. Therefore, to determine whether Q is a CO, we only have to examine the CF patterns that contain all event types in Q. Since $B_n[i]$ = 0, $\exists$ $E_j \in Q$: $E_j \notin \mathbb{CF}[i]$. Therefore, it is not necessary to evaluate $\mathbb{CF}[i]$. Lemma 5.2 is proven. ☐

By Lemma 5.2, a CF pattern in the CF pattern list is potentially a super-pattern or a different permutation of Q only if its corresponding bit in B_n is "1". When evaluating whether Q is a CO pattern, only these CF patterns have to be examined. This significantly speeds up the CO pattern detection process.

In the example shown in Fig. 5, given an infrequent pattern $Q = \langle ABB \rangle$, we first get the two bitmaps $B_A = [110]$ and $B_B = [101]$ for event types "A" and "B" from the inverted index. Then bitmap B_n is computed as $B_n = B_A \& B_B = [100]$. Only the first bit is set to "1' in B_n. Therefore, only the first pattern in the CF pattern list, namely $\langle ABAB \rangle$, contains all event types in Q and hence should be contrasted against Q.

6. EXPERIMENTAL EVALUATION

We experiment with both real-world and synthetic datasets. The results of the synthetic data experiments confirm the efficiency of Reduce in handling datasets with a rich variety of characteristics.

Real Datasets. We experiment with two real datasets: **logs** from a mobile app that tracks driver behavior (used in our previous examples) and the **lighting** dataset that contains network messages exchanged between the lighting devices and the servers in a city environment. These datasets are confidential datasets from our industry collaborators. The collaborators are interested in finding anomalies in their systems to capture software bugs and faulty devices.

The **log file** dataset is obtained from 10,000 devices ($|D| =$ 10,000) with 1,790 types of events ($|E| = 1790$). The event types are classified into three categories, namely information, warning, and error. The average length of each sequence is 34,097 with the longest being 100,000 events.

The **lighting** dataset is obtained from 283,144 lighting devices ($|D| = 283,144$) with 13 types of events ($|E| = 13$). The average length of each sequence is 456.

Synthetic datasets. We generated sequence data with various properties to evaluate the efficiency of our *Reduce* strategy. We design a new sequence generator that is able to control the key properties of the generated sequence datasets as listed in Tab. 2.

Table 2: Input Parameters to Sequence Data Generator.

Symbol	Description
$\|D\|$	Number of devices (Number of sequences)
$\|S\|$	Average length of sequences
$\|E\|$	Number of event types
$\|F\|$	Number of frequent patterns
$\|O\|$	Number of outliers
$\|L\|$	Average length of frequent patterns
e	noise rate

Experimental Setup. Since the generation of the contextual patterns for each sequence is independent of other sequences, contextual pattern mining is amenable to parallel processing. Therefore we leverage a distributed computing platform to mine CF patterns. The experiments are run on a Hadoop cluster with one master node and 24 slave nodes. Each node consists of 4 x 4 AMD 3.0GHz cores, 32GB RAM, and 250GB disk. Each node is configured with up to 4 map and 4 reduce tasks running concurrently. Note the reason for using a distributed computing platform to mine CF patterns is simply to speed up the process of our experimental evaluation. It does not effect the performance gain of our proposed strategy over the baseline. All code used in our experiments including the sequence generator is available at GitHub: https://github.com/OutlierDetectionSystem/TOP.

Algorithms. We evaluate (1) *Growth (G)*: the traditional growth-based mining strategy adapted to mine CF patterns (Sec. 4.1); (2) *Reduce (R)*: our proposed reduction-based strategy (Sec. 4.2).

Metrics. First, we evaluate the effectiveness of our contextual pattern semantics in detecting outliers by measuring the number of detected true outliers (*NTO* for short) and precision. Recall is also measured on the lighting dataset. Second, we measure the execution time averaged on each sequence of *Growth* and *Reduce* strategies. The *peak memory* usage is also measured.

Table 3: Effectiveness Evaluation.

Methods	Dataset	Number of true outliers (NTO)	Precision	Recall
TOP	Log file	93	58%	X
Max	Log file	47	38%	X
Closed	Log file	35	29%	X
TOP	Lighting	32	97%	85.4%
Max	Lighting	24	77%	58.5%
Closed	Lighting	19	61%	46.3%

```
(INFO)carrier: XXX
(INFO)server_base_url: XXX
(INFO)aws_region: XXX
(INFO)language: XXX
(INFO)locale: XXX
(INFO)userid: unknown_user
(INFO)log file generation: int
```

(a) Typical Pattern

```
(INFO)carrier: XXX
(INFO)log file generation: int
(INFO)server_base_url: XXX
(INFO)aws_token: XXX
(INFO)language: XXX
(INFO)locale: XXX
(INFO)userid: unknown_user
```

(b) Outlier Pattern

Figure 6: Example of CF & CO Patterns in Log File

6.1 Effectiveness Evaluation

We evaluate the effectiveness of our TOP system at detecting contextual outlier (CO) patterns by comparing against the maximal pattern mining [11, 12] and closed pattern mining [13, 32] semantics denoted as *Max* and *Closed*. We used our definition of COs on the frequent patterns produced by *Max/Closed*. The same sets of parameters are applied in all approaches (Log file data: $minSup$ = 100, $seqGap$ = 8; Lighting data: $minSup$ = 10, $seqGap$ = 8).

NTO and Precision Evaluation. As shown in Table 3, TOP outperforms *Max* and *Closed* in both the number of detected true outliers (NTO) and precision on both datasets. The reason is that our contextual constraint enables TOP to effectively capture the CO outliers, because it never misses an infrequent sub-pattern outlier which violates its typical super-pattern because of the separation of its independent occurrences and the constrained sub-occurrences. This leads to the high NTO of TOP.

However, *Closed* erroneously considers almost all CO patterns as frequent patterns. On the other hand, *Max* blindly discards all patterns if its super-pattern is frequent. Therefore, it tends to miss the typical patterns and in turn the CO patterns that violate the missed typical patterns. Therefore, the NTOs of *Closed* and *Max* are both low.

Further, the contextual constraint also ensures the high precision of TOP for two reasons. First, it does not generate false typical patterns, because it will not consider a pattern to be a CF pattern if it mostly occurs as part of a super-pattern. Second, the detected COs are guaranteed to be independent pattern occurrences. The precision of *Closed* is low, because *Closed* incorrectly reports some of the sub-patterns of the typical patterns as typical, which only occur as part of their super-patterns and hence should be suppressed. The outlier patterns that violate these false typical patterns tend to be false alarms. The precision of *Max* is better than *Closed*, although it is lower than TOP. In *Max* many false alarms are produced from the typical sub-patterns suppressed by their frequent super-patterns, because the events in these sub-patterns are erroneously used to construct infrequent violation outliers.

Compared to *Closed*, *Max* is more effective in capturing CO, because it will not mis-classify a CO as a frequent pattern. Therefore, *Max* has relatively higher NTO than *Closed*.

In particular, for the **log file** dataset, manual analysis by the developers of the logging software revealed that 58% were surprising, or likely indicative of an issue or faulty devices. As an example of one issue that was found, the entries shown in Fig. 6(a) are common on most devices, and happen at initialization time in the displayed order. Surprisingly, one device reported the pattern shown in Fig. 6(b). In this case, further analysis revealed an unexpected anomaly in the logging code itself, which the developers of the logging code subsequently fixed. Several other issues, including application crashes, were also identified via this analysis.

Although in this case TOP has a lower precision compared to the Lighting data, we find this still encouraging given the complex interactions and state transitions represented in these log files. First, in the mobile app multiple threads were performing independent actions writing to the log at the same time, which usually but not always happens in a certain order. Second, versions of the app that ran on only a few devices had a different sequence of log events.

As shown in Table 3, the NTO and precision of Max/Closed are much lower than TOP for the reasons described above.

As for the **Lighting** data, the manual evaluation by domain experts shows that TOP achieves almost perfect results. This is because Lighting contains only 13 event types with relatively regular input sequences. This reduces the chance of getting false outliers. Here, TOP continuously outperforms Max and Closed for the same reasons described above.

Recall Evaluation. We evaluate the Recall on 100 input sequences selected from the lighting dataset. This subset includes all 35 sequences that were found to contain outlier patterns by at least one of the three methods in the above evaluation. Other sequences were randomly picked from the dataset. The ground truth outliers were labeled by non-expert labelers who were given the business rules provided by the domain experts. In total, we identified and then labeled 41 outlier patterns.

As shown in Table 3, our TOP significantly outperforms Max and Closed, because TOP discovers much more true outliers than Max and Closed.

6.2 Efficiency Evaluation on Real Data

We investigate how the input parameters including $minSup$ (Fig. 7) and $seqGap$ (Fig. 8) influence the performance of Reduce using both the log file and Lighting datasets. The parameters are fixed as $minSup = 100\ seqGap = 10$ unless otherwise specified.

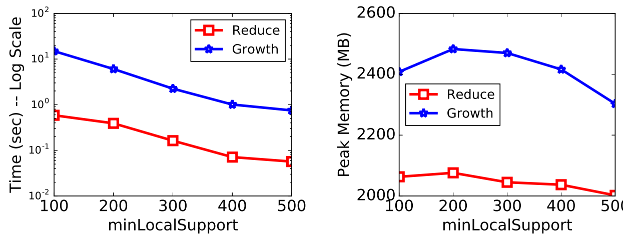
When varying $minSup$, Reduce is at least 13 and 199 times faster than Growth w.r.t. the log file and the Lighting datasets as shown in Figs. 7 and 9. Further, Reduce uses less memory than Growth in all cases. This demonstrates the advantages of the top-down strategy adopted by Reduce. Namely, it avoids producing shorter patterns that eventually get pruned. This saves both the processing time and memory usage.

When varying $seqGap$, Reduce outperforms Growth by up to 29x and 266x in processing time w.r.t. the log file data and the Lighting data as shown in Figs. 8 and 10. As $seqGap$ increases, both Reduce and Growth use more CPU time and memory. The processing time of Reduce increases faster than Growth, because larger $seqGap$ enlarges its search space due to its top-down strategy. Even in the worst case Reduce outperforms Growth by 9x and 68x. Furthermore, when $seqGap$ increases to 12, Growth fails due to out-of memory errors in the Lighting dataset case because of the large number of short patterns produced in the bottom up mining process. In the log file dataset case, the memory usage of Growth increases significantly when $seqGap$ reaches 12 and then fails when $seqGap$ grows.

6.3 Efficiency Evaluation on Synthetic Data

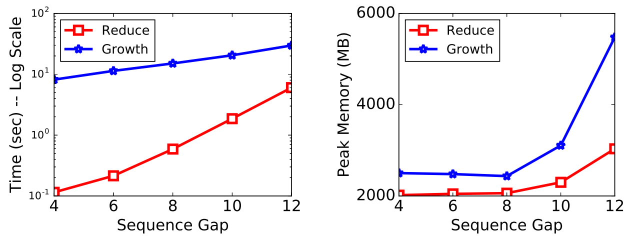
We generate 10 synthetic datasets to evaluate how Growth and Reduce perform on datasets with varying characteristics such as varying the number of event types and the average pattern length. The parameters utilized to generate these synthetic datasets are set to $|D| = 200$, $|S| = 100,000$, $|E| = 2000$, $|F| = 2000$, $|O| = 400$, $|L| = 20$ except for varying the corresponding dataset properties. The input parameters are set to $minSup$ = 20, $seqGap$ = 20, because these parameters are effective in capturing outliers. The results are shown in Fig. 11.

Varying Sequence Lengths. Fig. 11(a) demonstrates the results of varying sequence lengths from 10,000 to 50,000. As shown in Fig. 11(a), Reduce consistently outperforms Growth up to 30x



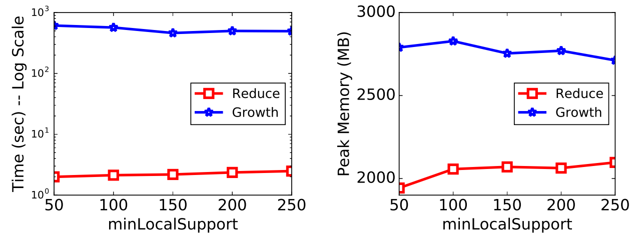
(a) Processing Time (b) Peak Memory Usage

Figure 7: Varying Minimum Support on Logfile Dataset



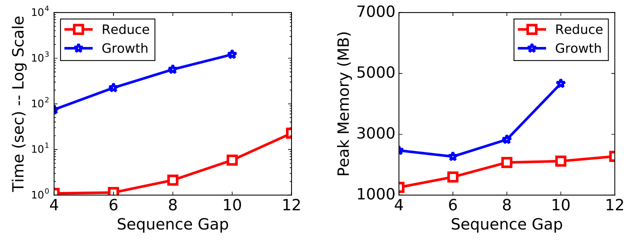
(a) Processing Time (b) Peak Memory Usage

Figure 8: Varying Sequence Gap on Logfile Dataset



(a) Processing Time (b) Peak Memory Usage

Figure 9: Varying Minimum Support on Lighting Dataset



(a) Processing Time (b) Peak Memory Usage

Figure 10: Varying Sequence Gap on Lighting Dataset

(note logarithmic time scale). The longer the sequences are, the more Reduce wins. This confirms that our top-down Reduce strategy performs well when handling long sequences. Even when the sequence length is as a very small value such as 10, Reduce is still about 2x faster than Growth (not shown in the figure).

Varying Number of Event Types. Fig. 11(b) shows the results when varying the number of event types $|E|$ from 1000 to 5000. To ensure each event type can form frequent patterns in each sequence, $|F|$ is set to 5000 and $|L|$ is set to 50. Again Reduce continuously outperforms Growth-based methods by at least 80x in all cases. In particular when $|E|$ increases to the largest number of event types (5000), Reduce is 126x faster than Growth. This shows that Reduce scales better than Growth also in the number of event types.

Varying Pattern Lengths. Fig. 11(c) represents the results when varying average pattern lengths. Reduce outperforms Growth up to 131x. Furthermore, as shown in Fig. 11(c), Reduce is not sensitive to the pattern length. On the contrary, the processing time of Growth increases dramatically as the pattern length gets larger. Worst yet, Growth runs out of memory when the average pattern length is 100. The increasing processing time of Growth results from both its growth-based pattern generation step and its backward filtering step. The pattern generation of Growth is a *recursive process* that continuously grows the patterns from length-1 patterns to the longest possible patterns. As the average pattern length gets larger, a large number of shorter patterns will be generated in the growth process. This substantially increases the processing time. Furthermore, this also increases the costs of backward filtering, since now it has more patterns to examine.

In conclusion, Reduce consistently outperforms Growth by 2 orders of magnitude under a rich variety of data properties.

6.4 Evaluation of Reduce Optimizations

Then, we evaluate the effectiveness of the two optimizations of Reduce, namely *start length minimization* and *pattern candidate pruning*. Due to space constraints we only report the results on the **log file** dataset. Here $seqGap$ is varied, since Reduce is sensitive to $seqGap$ based on our complexity analysis (Sec. 4.2.4).

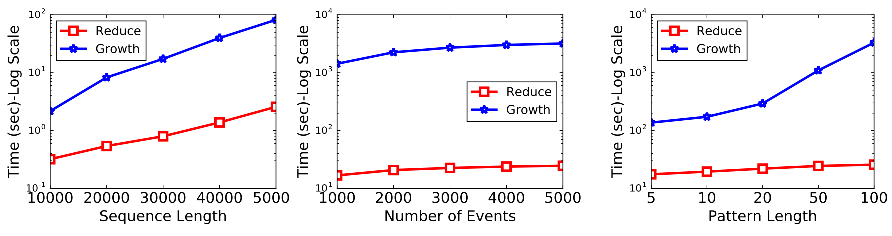
We compare (1) *Both OPT*: full-fledged Reduce with both optimizations; (2) *No OPT*: Reduce without any optimization; (3) *Length Minimization*: Reduce with only start length minimization; (4) *Candidate Pruning*: Reduce with only pattern candidate pruning. As shown in Fig. 12(a), compared to Reduce without any optimization, the start length minimization cuts nearly 10% of the total processing time, while the pattern candidate pruning cuts nearly 20% of the time. The two optimizations together cut the total processing time by up to 30%. Specifically, the start length minimization effectively minimizes the length of the patterns that Reduce starts to mine in the top down process. Therefore it minimizes the search space of Reduce. The candidate pruning strategy improves the performance of Reduce, because it continuously prunes the pattern candidates during the top down mining process.

6.5 Evaluation of Inverted Index.

We compare our inverted index-based CO discovery method with a baseline method that does not use an index when evaluating if a CO candidate violates a CF pattern. Again we measure the average processing time over each input sequence. In the experiment, we vary both the $minSup$ and $seqGap$ parameters. Due to space constraints we only report the results on the **log file** dataset.

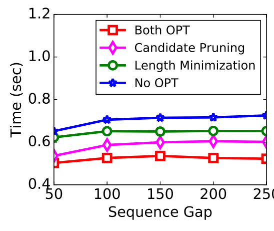
As shown in Fig. 13(a), our inverted index-based approach outperforms the baseline by 9x to 101x. The reason is that with the help of the inverted index our method only needs to compare each CO candidate against a small subset of CF patterns, while the baseline has to compare each CO candidate against all CF patterns. In particular, when varying $minSup$, the smaller the $minSup$ is, the more our index-based method wins as shown in Fig. 13(a). This is because a large number of CF patterns will be generated when $minSup$ is small. For the same reason, the processing times of both methods increase when $minSup$ gets smaller. Fig. 13(b) shows the results when varying $seqGap$. As $seqGap$ increases, the processing times of both methods increase, because more CF patterns will be generated with a larger $seqGap$.

7. RELATED WORK



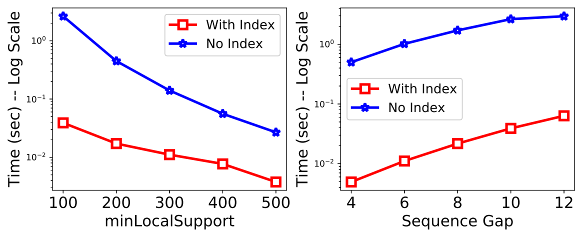
(a) Varying Sequence Length (b) Varying Number of Events (c) Varying Pattern Length

Figure 11: Evaluation of Processing Time on Various Synthetic Datasets



(a) Vary Sequence Gap

Figure 12: Evaluation of Optimizations



(a) Varying Local Support (b) Varying Sequence Gap

Figure 13: Evaluation of Inverted Index.

Frequent Pattern Mining Semantics. Existing semantics [3, 29, 15, 16, 37, 4] determine whether a pattern is frequent purely based on the number of its occurrences. As already noted in Sec. 1 and Sec. 2.2.1, directly applying the existing pattern mining semantics [3, 29, 15, 16, 37, 4, 9, 14, 36, 22, 28, 13, 32, 35, 11, 12, 26] to detect typical patterns does not adequately capture outlier patterns, because they do not distinguish between an independent occurrence of a pattern Q and its occurrence as part of some frequent super-pattern P.

Frequent Pattern Mining Algorithms. Frequent pattern mining was first proposed in [3] to mine typical purchase patterns from a customer transaction dataset. Two Apriori-based mining algorithms, namely AprioriSome and AprioriAll were proposed in [3] to reduce the search space of frequent patterns. Since then, techniques have been proposed such as GSP [29], FreeSpan [15], PrefixSpan [16], SPADE [37], SPAM [4] and their extensions CM-SPADE, CM-SPAM [9] to scale frequent pattern mining to large transaction datasets. In particular, PrefixSpan [16] avoids multiple scans of the whole dataset while still utilizing the pruning capability of the Apriori property [1]. In this work, the *Growth* strategy leverages this idea and decomposes the mining process of our new CF pattern semantics into two steps. However, our experiment shows it performs significantly worse than our *Reduce* strategy.

Top-down Frequent Pattern Mining Approaches. In [25, 18] top-down approaches were presented for frequent pattern mining. Given a dataset with n sequences, they first look at the patterns that occur in all n sequences, and then look at the $n-1$ sequence combinations out of the n sequence to mine the patterns that occur in all sequences of any such combination. This recursive process stops until reaching $minSup$ sequence iteration, where $minSup$ represents the support threshold. The idea stands in contrast to our Reduce strategy which constructs patterns from long to short. Further, this approach is efficient only when the dataset contains a small number of sequences, while our Reduce is shown to be efficient also in large scale sequence data.

Outlier Detection in Sequence Data. Although in [17, 33, 6] the authors proposed methods to detect outliers from symbolic sequence datasets, in their work outliers are defined as input sequences that are different from any other sequence based on distance measures such as edit distance, unlike our outlier patterns that correspond to finding abnormal patterns in each input sequence. In [7], the authors surveyed some methods that identify anomalous subsequences within a long sequence [19, 20] or identify a pattern in a sequence whose frequency of occurrences is abnormal [21, 24]. However, these methods all focus on time series data composed of numerical values instead of symbolic sequence data. Further, adapting these methods to symbolic sequence data cannot solve our problem of detecting infrequent patterns that violate the expected system behavior. First, in [19, 20], the input sequences are first divided into *fixed* length segments. Each segment is assigned an anomaly score based on the similarity to the other segments. Obviously this strategy is not effective in detecting outlier patterns with various lengths. Second, in [21, 24], the anomaly detection problem is defined as: given a short query pattern α, a long test sequence t, and a training set of long sequences S, determine if the frequency of occurrence of α in t is anomalous with respect to frequency of occurrence of α in S. In our work, we instead focus on automatically discovering all outlier patterns without requiring the users to specify candidate outlier patterns. Further, we do not assume a normal training set is given beforehand.

Gap Constraint. Besides the sequence gap constraint (Def 2.2, Sec. 2.2.1) used in our TOP, there are other gap constraints in sequential pattern mining, such as the event gap constraint [5]. They all are applicable in our approach. In addition, gap constraint is also used in time series analysis [2, 8] which different from the sequence data we focus on, is composed of continuous numerical values. In [2], the gap constraint is used when measuring the similarity of two sub-sequences from two different time series. Given two sequences S and T, to determine whether S and T are similar, the small non-matching regions (so called gaps) in S and T are ignored. In addition, *time window* used in Dynamic Time Warping (DTW) [8] also can be considered as one type of gap constraint. These constraints could potentially be adapted to our system.

8. CONCLUSION

In this work we design the TOP system for the effective discovery of abnormal sequence patterns from IoT generated sequence data. TOP features new pattern semantics called *contextual patterns*. It effectively solves the problem of existing pattern mining semantics that tend to miss the abnormal patterns violating the typical patterns. Further, we design novel top-down pattern mining strategy *Reduce* that naturally captures the context in which an outlier pattern occurs. This enables TOP to efficiently discover the outlier patterns by mining the sequence only once in contrast to the traditional pattern growth based mining strategy (*Growth*) that has to mine the sequence twice. Our experimental evaluation with real datasets demonstrates the effectiveness of our new semantics in capturing outlier patterns, and the efficiency of *Reduce* in discovering them.

9. REFERENCES

[1] C. C. Aggarwal and J. Han, editors. *Frequent Pattern Mining*. Springer, 2014.

[2] R. Agrawal, K.-I. Lin, H. S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *VLDB*, pages 490–501, San Francisco, CA, USA, 1995.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, 1995.

[4] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *SIGKDD*, pages 429–435, 2002.

[5] K. Beedkar, K. Berberich, R. Gemulla, and I. Miliaraki. Closing the gap: Sequence mining at scale. *ACM Trans. Database Syst.*, 40(2):8:1–8:44, June 2015.

[6] S. Budalakoti, A. N. Srivastava, and M. E. Otey. Anomaly detection and diagnosis algorithms for discrete symbol sequences with applications to airline safety. *Trans. Sys. Man Cyber Part C*, 39(1):101–113, Jan. 2009.

[7] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 24(5), May 2012.

[8] H. A. Dau, A. J. Bagnall, K. Kamgar, C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, and E. J. Keogh. The UCR time series archive. *CoRR*, abs/1810.07758, 2018.

[9] P. Fournier-Viger, A. Gomariz, M. Campos, and R. Thomas. Fast vertical mining of seq. patterns using co-occurrence information. In *PAKDD*, pages 40–52, 2014.

[10] P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, and Y. S. Koh. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1(1):54–77, 2017.

[11] P. Fournier-Viger, C.-W. Wu, A. Gomariz, and V. S. Tseng. Vmsp: Efficient vertical mining of maximal sequential patterns. In *CAIAC*, pages 83–94, 2014.

[12] P. Fournier-Viger, C.-W. Wu, and V. S. Tseng. Mining maximal sequential patterns without candidate maintenance. In *ADMA*, pages 169–180. Springer, 2013.

[13] A. Gomariz, M. Campos, R. Marin, and B. Goethals. Clasp: An efficient algorithm for mining frequent closed sequences. In *PAKDD*, pages 50–61. Springer, 2013.

[14] K. Gouda, M. Hassaan, and M. J. Zaki. Prism: A primal-encoding approach for freq sequence mining. In *ICDM*, pages 487–492, 2007.

[15] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: frequent pattern-projected seq. pattern mining. In *SIGKDD*, pages 355–359, 2000.

[16] J. Han, J. Pei, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, pages 215–224, 2001.

[17] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, Aug. 1998.

[18] H. Huang, Y. Miao, and J. Shi. Top-down mining of top-k frequent closed patterns from microarray datasets. In *ICISS*, 2013.

[19] E. Keogh, J. Lin, and A. Fu. Hot sax: Efficiently finding the most unusual time series subsequence. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, pages 226–233, 2005.

[20] E. Keogh, J. Lin, S.-H. Lee, and H. V. Herle. Finding the most unusual time series subsequence: Algorithms and applications. *Knowl. Inf. Syst.*, 11(1):1–27, Jan. 2007.

[21] E. Keogh, S. Lonardi, and B. Y.-c. Chiu. Finding surprising patterns in a time series database in linear time and space. In *KDD*, KDD '02, pages 550–556, 2002.

[22] T. Kieu, B. Vo, T. Le, Z.-H. Deng, and B. Le. Mining top-k co-occurrence items with sequential pattern. *Expert Syst. Appl.*, 85(C):123–133, Nov. 2017.

[23] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *KDD*, pages 410–419, 2007.

[24] J. Lin, E. Keogh, A. Fu, and H. Van Herle. Approximations to magic: Finding unusual medical time series. In *Proceedings of the 18th IEEE Symposium on Computer-Based Medical Systems*, CBMS '05, pages 329–334, 2005.

[25] H. Liu, X. Wang, J. He, J. Han, D. Xin, and Z. Shao. Top-down mining of frequent closed patterns from very high dimen. data. *Inf. Sci.*, 179(7):899–924, Mar. 2009.

[26] C. Luo and S. M. Chung. Efficient mining of maximal sequential patterns using multiple samples. In *SDM*, pages 415–426, 2005.

[27] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.

[28] E. Salvemini, F. Fumarola, D. Malerba, and J. Han. Fast sequence mining based on sparse id-lists. ISMIS'11, pages 316–325, 2011.

[29] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. *EDBT*, pages 1–17, 1996.

[30] N. Tatti and J. Vreeken. The long and the short of it: summarising event sequences with serial episodes. In *KDD*, pages 462–470, 2012.

[31] D. Wang, E. A. Rundensteiner, and R. T. Ellison, III. Active complex event processing over event streams. *PVLDB*, 4(10):634–645, 2011.

[32] J. Wang, J. Han, and C. Li. Frequent closed sequence mining without candidate maintenance. *TKDE*, 19(8), 2007.

[33] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IN IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 133–145. IEEE Computer Society, 1999.

[34] C. Wu, Y. Lin, P. S. Yu, and V. S. Tseng. Mining high utility episodes in complex event sequences. In *SIGKDD*, pages 536–544, 2013.

[35] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets. In *SDM*, pages 166–177, 2003.

[36] Z. Yang and M. Kitsuregawa. Lapin-spam: An improved algorithm for mining sequential pattern. ICDE Workshop '05, pages 1222–, 2005.

[37] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine learning*, 42(1):31–60, 2001.