Efficient Discovery of Contextual Sequence Patterns

Lei Cao[1], Yizhou Yan[2], Samuel Madden[1] and Elke A. Rundensteiner[2]
[1]CSAIL, Massachusetts Institute of Technology
[2]Department of Computer Science, Worcester Polytechnic Institute
[1]{lcao,madden}@csail.mit.edu [2]{yyan2,rundenst}@cs.wpi.edu

ABSTRACT

Modern *Internet of Things* (*IoT*) applications generate massive amounts of data, much of it in the form of sequences of readings, events, and log entries. In this paper, we present a new system called TOP to make sense of these sequences by finding Typical and Outlier Patterns. TOP offers new pattern semantics called *contextual patterns*. Specifically, contextual pattern decides whether a pattern is frequent based on the context in which it occurs. It solves the fundamental limitation of existing frequent pattern semantics that tend to mis-classify abnormal patterns as typical patterns when they occur as part of larger patterns. Next, we develop efficient algorithms for mining these new types of contextually frequent patterns and outlier patterns. In particular, we develop a new top-down strategy (*Reduce*), discovering large contextually frequent patterns first. This is in contrast to the bottom-up strategy commonly adopted for classical frequent pattern mining in the literature. Better yet, the outlier candidates are derived naturally as a by-product in the top-down mining process of *Reduce*, out of which the final outlier patterns are produced by a lightweight post-processing that makes only one pass on the outlier candidates. Our experimental evaluation demonstrates the effectiveness of the TOP system at capturing outlier patterns in several real-world IoT applications. We also demonstrate the efficiency of *Reduce*, up to 2 orders of magnitude faster than the classical pattern mining strategy.

1 INTRODUCTION

With the increasing prevalence of sensors, mobile phones, actuators, and RFID tags, *Internet of Things (IoT)* applications generate massive amounts of time-stamped data per day. 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. These applications need to find frequent patterns that represent typical system behavior (e.g., [1, 11]) as well as *outliers* that represent deviations from normal behavior. In these modern applications, *outlier sequence patterns* are important, since such patterns may represent erroneous behavior not captured by simple thresholds or other techniques that examine a single record at a time. Below we describe two real-world applications that motivated our work on both frequent and outlier pattern detection in sequence data.

Example 1. We obtained log files from 10,000 users of a sensor-based mobile application that records data from users as they drive. In these files as shown in the top portion of Fig. 1, the *frequent* pattern $P = \langle StartTrip(A), RecordTrip(B), ReportLoc(C), Terminate(D)\rangle$ represents a typical behavior of the system, namely, after the user starts a trip, the system continues reporting, and then terminates when the trip finishes. Pattern $Q = \langle StartTrip(A), RecordTrip(B), ReportLoc(C)\rangle$ (a sub-pattern of P) represents an erroneous system behavior, since the expected successful termination action (the *Terminate*(D) event) is missing. Thus, pattern Q corresponds to an *outlier pattern* that violates the typical expected behavior represented by P.

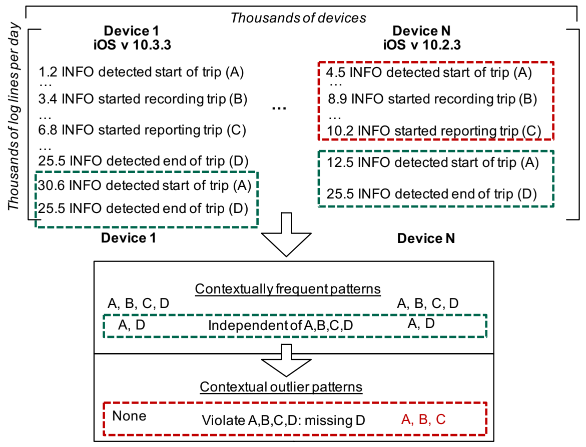

```
1.2 INFO detected start of trip (A)
...
3.4 INFO started recording trip (B)
...
6.8 INFO started reporting trip (C)
...
25.5 INFO detected end of trip (D)
30.6 INFO detected start of trip (A)
25.5 INFO detected end of trip (D)
```
```
4.5 INFO detected start of trip (A)
...
8.9 INFO started recording 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 CF patterns & outliers.

Example 2. Consider a hospital infection control system [20] 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 such healthcare settings, a HCW is required to wash her hands before approaching a patient. This behavior can be modeled as a *frequent* pattern $P = \langle HandWash, Enter, ApproachPatient\rangle$, while pattern $Q = \langle Enter, ApproachPatient\rangle$, a sub-pattern of P, represents a violation of the hygiene compliance protocol, namely not performing the required *HandWash* action before approaching a patient. Thus it should be identified as an *outlier* pattern.

1.1 Limitations of Prior Work

The existing frequent pattern semantics [2, 4, 6–8, 21] and mining algorithms [3, 5, 12, 13, 18, 23] are neither effective nor efficient in discovering typical or outlier patterns in modern applications.

The fundamental limitation of existing semantics [2, 4, 6–8, 21] is that typical patterns are defined purely based on their frequencies. This leads to the well-known Apriori property upon which most pattern mining algorithms are built. With Apriori, since the frequency of Q will never be smaller than the frequency of its super-pattern P, any sub-pattern Q of a frequent pattern P is also guaranteed to be a frequent pattern – representing typical system behavior.

Although these semantics work well in traditional sequence mining settings, where each sequence S in a dataset D represents, for example, a purchase or web surfing behavior from a single person [1], they work less well in the modern applications. In these settings, a pattern P and a sub-pattern Q of P can represent different and possibly contradictory system behavior. In fact, as described in our motivating examples above, a sub-pattern Q of a typical pattern P may correspond to an outlier pattern representing anomalous behavior. Hence, it is not appropriate to designate a sub-pattern Q of a pattern P as typical simply because it occurs frequently as part of P.

Existing **mining techniques** [3, 5, 12, 13, 18, 23] adopt a *pattern-growth strategy* that builds patterns from short to long. Although this strategy effectively uses the Apriori property to prune unpromising *long* pattern candidates, as we will show in our experiments, it is not efficient in meeting the new requirement of modern applications, namely excluding *short* patterns from typical patterns when they occur as part of their super-patterns.

Thus, the new frequent pattern semantics and corresponding mining algorithms will be focus of our work. The limitations of existing semantics will be further illustrated in Sec. 2.2.

1.2 Proposed Approach and Contributions

In this work, we describe our system, *TOP*, which is designed to effectively and efficiently capture the Typical and Outlier Patterns. It has been deployed in the data analytics platform of Philips Lighting Research running at daily base. TOP uses *Contextually Frequent* (CF) pattern semantics and *Contextual Outlier* (CO) pattern semantics to capture the typical and abnormal system behavior. We then design new mining strategies to address the problem of mining CF and CO patterns from sequence data. Our key contributions include:

• Novel CF and CO pattern semantics. In CF pattern semantics, whether a pattern is frequent or not in a given sequence is no longer determined purely by frequency. Instead, it depends on the *context* in which the pattern occurs: an occurrence of pattern P is not counted as an occurrence of any of P's sub-patterns Q if P is a CF pattern. This *contextual constraint* ensures that only patterns that capture typical system behavior in their own right are considered CF patterns. On the other hand, CO pattern semantics not only capture the *infrequent* outlier patterns that violate typical system behavior, but also the *frequent* outlier patterns by considering the global distributional characteristics of the overall sequence dataset. We define CF and CO pattern semantics more formally in Sec. 2.

• Reduce strategy for mining CF patterns. Although CF patterns could be mined by first applying the traditional pattern-growth strategy (*Growth* in short) and then conducting a post-processing filtering step to enforce the contextual constraint, doing so wastes CPU and memory resources by producing a large number of short pattern candidates that are subsequently discarded due to the violation of the contextual constraint. Therefore we design a top-down *reduction-based* strategy called *Reduce* that mines CF patterns in one step. The key idea of *Reduce* is that, instead of treating the contextual constraint as a performance bottleneck that complicates the pattern mining process, we can utilize it to effectively reduce the search space. During the top-down mining process, the pruning is continuously applied in each iteration, such that we always discover the largest possible patterns from a sequence of events. This avoids processing shorter patterns that cannot be CF patterns.

• Lightweight CO pattern detection. We show that CO pattern candidates in each sequence can be naturally derived as a by-product of the top-down CF pattern mining process. The final CO patterns are then produced by applying a lightweight filter as a post-processing step, which can be done with a linear scan on the candidates.

• We perform empirical studies using real world datasets including mobile app data and lighting data to demonstrate the effectiveness of our TOP approach in capturing anomalous patterns.

• We demonstrate a two orders of magnitude performance gain for our Reduce strategy over the classical Growth strategy across a wide range of parameter settings on various datasets – 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) an **event** of type E_i occurring at time t_i. An example of an event e_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 sometimes reference the relative position of the event in sequence S, denoted as (e_i, i). The events in sequence S are ordered by time. A **sequence dataset** is a set of event sequences, each of which is 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 $Q = \langle E'_1 E'_2 \ldots E'_l \rangle$ is a **sub-pattern** of a pattern $P = \langle E_1 E_2 \ldots E_m \rangle$ $(l \leq m)$, denoted $Q \sqsubseteq P$, if integers $1 \leq i_1 < i_2 < \cdots < i_l \leq m$ exist such that $E'_1 = E_{i_1}, E'_2 = E_{i_2}, \ldots, E'_l = E_{i_l}$. Alternately, we say P is a **super-pattern** of Q. For example, pattern $Q = \langle AC \rangle$ is a sub-pattern of $P = \langle ABC \rangle$.

2.2 Limitations of Existing Semantics

As already noted in Sec. 1.1, the semantics of existing frequent pattern mining do not adequately capture either typical or abnormal behavior. Here we further motivate with additional real world examples.

Existing semantics [2, 3, 5, 12, 13, 18, 23] determine whether a pattern is frequent *purely* based on the number of its occurrences. They **do not distinguish** between an *independent occurrence* of a pattern Q and its occurrence as part of some frequent super-pattern P (sub-occurrence of P). Thus, in existing semantics, all sub-patterns of a frequent pattern are also frequent. As noted previously, it is *often misleading to flag* a sub-pattern Q as frequent when in the large fraction of the time it only occurs as a sub-occurrence of a larger pattern P. In fact, an occurrence Q may signal a missing event in P (such as a missing end of trip event in Fig. 1) and thus designate a system error. At first sight, a straightforward solution to this problem perhaps is to keep only the largest frequent patterns, discarding all sub-patterns of P when P is frequent. This is the approach taken in maximum pattern mining [6, 7]. However, sometimes a sub-pattern of a frequent pattern may actually be frequent *in its own right*. For example in the mobile app (Fig. 1), the pattern $Q = \langle \mathit{StartTrip}(A), \mathit{Terminate}(D) \rangle$ $(Q \sqsubseteq P)$ 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. Using maximum patten mining, frequent sub-patterns of P that occur independently and thus represent typical system behavior will be missed. Thus, without taking the context in which a pattern occurs into consideration, existing semantics are not sufficient to handle the intrinsic relationship between the frequent patterns and their corresponding sub-patterns. New semantics must be proposed to effectively capture both typical and abnormal behavior.

2.3 Contextual Patterns

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

2.3.1 Contextual Constraint. The contextual constraint determines whether a sub-pattern is frequent based on the context in which the pattern occurs. A frequent pattern satisfying the contextual constraint is called *contextually frequent*.

The contextual constraint excludes an occurrence O_Q of Q from Q's support if all events in O_Q are contained in the occurrences of patterns P, where P is contextually 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 outside of some frequent super-pattern. 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), because most of its occurrences occur as sub-occurrences of the frequent super-pattern $P = \langle ABCD \rangle$.

Further, unlike the maximum frequent pattern semantics [6, 7], which blindly suppresses all sub-patterns of a frequent pattern, the contextual constraint now enables us to report both a pattern and its sub-pattern as conceptually frequent when the sub-pattern also represents typical system behavior and thus frequently occurs independently. In the mobile application example, the pattern $R = \langle StartTrip, Terminate \rangle$ would be correctly recognized as typical system behavior together with its super-pattern $P = \langle StartTrip, RecordTrip, ReportLoc, Terminate \rangle$. Hence, by separating the independent occurrences and the constrained sub-occurrences, the contextual constraint successfully distinguishes outlier sub-patterns from typical sub-patterns.

2.3.2 Contextually Frequent Pattern Semantics. Next we define the semantics of *contextually frequent (CF) patterns*.

Definition 2.1. **Contextually Frequent (CF) Pattern.** A pattern $Q = \langle E_1, E_2, ..., E_m \rangle$ is said to be *contextually frequent* (CF) in sequence S if $CLocSup_S(Q) \geq minLocalSup$, where $CLocSup_S(Q)$ representing the contextual local support of Q in sequence S is defined as the count of its occurrences $O_Q = \langle (e_1, t_1)\ (e_2, t_2) \dots (e_m, t_m) \rangle$, where each O_Q satisfies the condition: $\exists\ e_j \in O_Q$: $e_j \notin O_P$, $\forall P$ satisfying $CLocSup_S(P) \geq minLocalSup$ & $length(P) > length(Q)$.

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.

Beyond the contextual constraint, other constraints in the literature are orthogonal and are still applicable, i.e., the gap constraint [4] defined below.

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 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 time. 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 hospital infection control application [20], 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.

2.3.3 Contextual Outlier Pattern Semantics. We now introduce our outlier pattern semantics. Given an outlier pattern P, its occurrences have to satisfy the contextual constraint, that is, occurrences independent from its CF super-patterns. Therefore we call this semantics contextual outlier (CO) pattern. The CO pattern semantics include two types of outlier patterns, namely locally frequent outliers (LO) and violation outliers (VO).

The locally frequent outlier (LO) captures the outlier patterns that frequently occur in only a few devices.

Definition 2.3. **Locally Frequent Outliers (LO).** Given a sequence dataset D with n sequences S and a sequence count threshold $seqCnt$, a pattern P is a *locally frequent outlier* if P is a contextually frequent (CF) pattern in fewer than $seqCnt$ sequences $S \in D$.

In practice the input parameter $seqCnt$ should be a small number, such as 5% of the total number of sequences. These outliers typically occur due to system errors on some devices. For example in the mobile app outlier pattern $\langle StartTrip, Init \rangle$ that occurs on some devices due to system crashing will be captured by LO.

The class of infrequent violation outliers (VO) captures the patterns that occur rarely and violate the typical patterns in the system. It is formally defined next.

Definition 2.4. **Infrequent Violation Outliers (VO).** Given a sequence dataset D and a count threshold $cntThr$, a pattern Q in a sequence S is said to be a *violation outlier* w.r.t a CF pattern P in S, if Q satisfies the following conditions:
(1) $\forall$ event type $E_i \in Q$, $E_i \in P$; and $Q \neq P$;
(2) $CLocSup_S(Q) \leq cntThr$;
(3) P is not a locally frequent outlier.

By Def. 2.4, a pattern Q is a violation outlier w.r.t P, if Q is a sub-pattern or a different permutation of P (Condition 1) and Q occurs *rarely* in sequence S of D (Condition 2). Furthermore, the pattern P violated by Q is not a locally frequent outlier pattern (Condition 3). For example, given a CF pattern $P = \langle ABBC \rangle$ in S, pattern $Q = \langle ABB \rangle$ and pattern $R = \langle ABCB \rangle$ are VO w.r.t P if Q and R are rare in S. In practice the input parameter $cntThr$ typically is set as a very small number such as 1 or 2.

VO captures sub-pattern outliers that miss the 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-hygiene event before the approaching-patient event.

VO 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 updating 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 the pattern P and could be captured by the VO semantics.

3 CONTEXTUALLY FREQUENT PATTERN MINING

Now we design efficient algorithms for supporting the new semantics. First, we design a novel *Reduce* strategy to mine the contextually frequent (CF) patterns from each input sequence (Sec. 3). Next, based on the mined CF patterns, we design a lightweight approach to detect contextual outlier (CO) patterns (Sec. 4).

3.1 Adopting Traditional Strategy for CF Pattern

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, in CF semantics, the frequency of a short pattern cannot be determined without mining longer patterns first.

One solution to this problem would be to mine the CF pattern in two steps. The first step is a traditional Apriori-based mining strategy that finds the frequent patterns without the contextual constraint. Since Apriori still holds, a classic pattern-growth strategy such as prefixSpan [13] can be used to find these patterns. The second step then filters these frequent patterns that violate the *contextual constraint*. We call this growth-based strategy *Growth*.

However, 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 frequent patterns and filtering those that do not conform to the contextual constraint.

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. 2. 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. 2, $\langle BC\rangle$ 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.

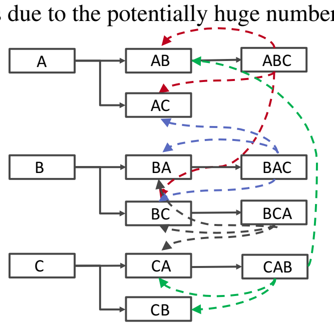


Figure 2: Backward filtering

3.2 The Proposed Reduction-based Approach

To address these drawbacks, we develop a top-down pattern mining approach that directly mines the CF patterns in one step. We refer to this reduction-based approach as *Reduce*. The key innovation of Reduce is that instead of treating the contextual constraint as 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. 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 pattern mining.

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	$\langle(c, 2)(a, 4)(b, 5)\rangle$ $\langle(c, 6)(a, 7)(c, 8)(b, 9)\rangle$ $\langle(c, 8)(b, 9)(a, 11)\rangle$ $\langle(c, 12)\rangle$	$\langle CA\rangle$ $\langle CAB\rangle$ $\langle CB\rangle$	$\langle CAB\rangle$	$\langle CAB\rangle$

3.2.1 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. Finding the largest possible pattern is critical for Reduce. Second, it is for the ease of counting the occurrences of a pattern. It will be shown by Lemma 3.1. To illustrate the search search, we use a running example with the input sequence $S = \langle(a, 0)\ (b, 1)(c, 2)(d, 3)(a, 4)(b, 5)\ (c, 6)(a, 7)(c, 8)\ (b, 9)\ (e, 10)\ (a, 11)\ (c, 12)\rangle$. The *minLocalSup* and *seqGap* are both set as 2.

First, in one scan of the sequence, all event types with frequency $\geq$ *minLocalSup* are identified as frequent event types. All infrequent events 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 $seqGap + 2$ or it hits the end of S. The search spaces for each E_i in the example are shown in the second row of Tab. 1. Note the subsequences could possibly 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 $seqGap + 2$.

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 3.1. *Given a sequence S and a pattern candidate P with prefix E_i, any two distinct 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. Therefore no valid occurrence O_P of P will be missed if searching each occurrence O_P independently from each subsequence in the search space $\mathbb{SS}_i$.

Second, given two occurrence O_P^1 and O_P^2, since they 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 can be found from two different subsequences. Lemma 3.1 is proven. ■

Lemma 3.1 not only proves that all occurrences of any given pattern can be discovered from the search spaces, it also inspires an efficient occurrence 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 search for the occurrence of O_P in one subsequence S_i, we only have to search the occurrence that starts from the first event of S_i and stop immediately once finding one.

For example, given a sequence $S = \langle(a, 0)(b, 1)(a, 2)(b, 3)(a, 4)(b, 5)\rangle$, 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 $(a, 4)$.

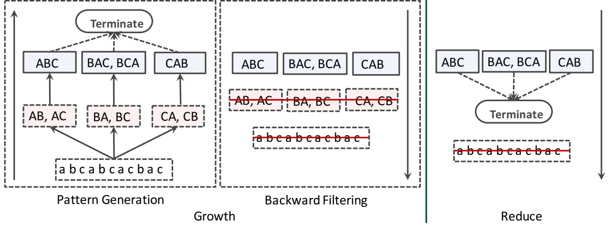


Figure 3: Growth vs Reduce

3.2.2 Reduction-based Pattern Mining

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

The **top-down mining** strategy ensures that in each iteration of the mining process, only the largest CF patterns are generated. Since their 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 all together.

Event availability maintenance dynamically maintains and updates the "availability" of each event in sequence S. The event availability is shared by all patterns with different prefixes, which we call *global availability*. The intuition here is that by the contextual constraint, an occurrence of pattern P is invalid if all of its events have been utilized by the patterns longer than it. Therefore, in the top-down mining process, if we mark all events utilized by the long patterns as unavailable, then the subsequences in the search space that do not contain any event that is still "available" can be discarded. This pruning step significantly reduces the search space for the later evaluation of shorter pattern candidates.

Algorithm 1 shows the overall process. We explain *Reduce* by our running example with input sequence $S = \langle(a, 0)(b, 1)(c, 2)(d, 3)(a, 4)(b, 5)(c, 6)(a, 7)(c, 8)(b, 9)(e, 10)(a, 11)(c, 12)\rangle$, $minLocalSup = 2$ and $seqGap = 2$ used in the gap constraint (Def. 2.2).

Algorithm 1 Reduction-based approach

```
1: globalAvail[1 ... n] ← array of global availability, init all true
2: currentLen = gMax ← global maximum pattern length
3: minLocalSup ← minimum local support value
4: function REDUCE()
5:     while currentLen > 0 & prefixSet.size() > 0 do
6:         curFreqPatterns = []
7:         for ∀ prefixSet[i] ∈ prefixSet do
8:             if prefixSet[i].maxLen ≥ currentLen then
9:                 curFreqPatterns.add(FINDCF(prefixSet[i], currentLen, minLocalSup, globalAvail))
10:        for ∀e_j ∈ curFreqPatterns 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(curFreqPatterns)
17:        currentLen = currrentLen-1
18:    return freqPatterns
```

(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 $seqGap + 2$. It 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, $gMax = 4$, since the longest subsequences are $\langle abca\rangle$, $\langle bcac\rangle$ and $\langle cacb\rangle$ (See Tab. 1).

Once $gMax$ is determined, Reduce starts mining the length-$gMax$ patterns (Line 2). Reduce examines whether the longest subsequences can form CF patterns by simply grouping together the subsequences that are the occurrences of the same pattern and counting the subsequences in each group.

In our example, there exists only one subsequence has length $\geq$ 4 in the search space of each event type. For instance, only

$\langle abca \rangle$ has length ≥ 4 for event type A. Since the support threshold *minLocalSup* 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. It 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). 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 Mining.** 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 Algorithm 2 corresponding to the *FindCF* subroutine.

Algorithm 2 Find frequent pattern for prefix i

```
 1: function FindCF(Object prefix[$i$], int currentLen, int minLocalSup, 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($P$) ≥ minLocalSup then
13:             freqSeqs.add($P$)
14:     return freqSeqs
```

FindCF 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 conducted 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 sub-sequence by Lemma 3.1.

Next, *FindCF* 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 simply 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 *minLocalSup* 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 hence 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 Algorithm 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 ArrayList. In this case, all events in S now are unavailable. Thus the process terminates without attempting to generate length-2 and length-1 frequent patterns. Therefore *Reduce* successfully avoids generating patterns shorter than 3, in contrast to *Growth* as shown in Tab. 1. Significant CPU and memory are saved as confirmed by our experiments (Sec. 5).

3.3 Time Complexity Analysis

In this section, we show that by putting an additional constraint on the formed patterns – called triggering event constraint, our Reduce strategy is able to achieve linear time complexity in the length of the sequence. To the best of our knowledge, this is the *first* linear time solution of frequent pattern mining.

The triggering event constraint is motivated by the observation that in IoT application often a sequence pattern is triggered by its first event, so called triggering event. For example, in the driver mobile app pattern $\langle StartTrip, RecordTrip, ReportLoc, Terminate \rangle$ is triggered by the first event $StartTrip$, while pattern $\langle LocationUpdate, DatabaseUpdate \rangle$ is triggered by $LocationUpdate$ event. Intuitively a sequence pattern tends to contain only one triggering event, since the occurrence of a new triggering event might indicate the start of a new sequence pattern. By this observation, we use triggering event constraint to forbid the triggering event to appear multiple times in a pattern.

Definition 3.2. **Triggering Event Constraint.** Given a pattern $P = \langle E_1 E_2 \ldots E_m \rangle$, an occurrence $O_P = \langle (e_1, t_1)\,(e_2, t_2) \ldots (e_m, t_m) \rangle$ of P in S has to satisfy the condition: $\forall e_j \in O_P$ $(j > 1)$, there does not exist another type E_1 event $(e'_1, t'_1) \in S$ different from the triggering event (e_1, t_1) such that $t_j - t_1 < t_j - t'_1$.

Our Reduce strategy can easily support the triggering event constraint by slightly changing the way of constructing search space. That is, given a prefix E_i, we no longer construct its search space composed of subsequences potentially overlapping with each other. Instead, now the search space is constructed by dividing the input sequence S into none-overlapping subsequences using the prefix E_i event as delimiter. Given the input sequence $S = \langle (a, 0)(b, 1)(c, 2)(d, 3)(a, 4)(b, 5)(c, 6)$

$\langle (a,7)(c,8)(b,9)(e,10)(a,11)(c,12)\rangle$ used in our example, now the search space of A is composed of subsequences $S_1 = \langle (a,0)(b,1)(c,2)\rangle$, $S_2 = \langle (a,4)(b,5)(c,6)\rangle$, $S_3 = \langle (a,7)(c,8)(b,9)\rangle$, and $S_4 = \langle (a,11)(c,12)\rangle$. This ensures that the occurrences of a given pattern P can be discovered independently from the sub-sequences without worrying about the overlapping among the discovered occurrences.

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. *minLocalSup* denotes the minimum local 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 $\frac{NE}{M}$ length-M occurrences, the time complexity of generating all length-(M-1) sub-patterns from length-M sequences is $\frac{NE(M-1)^2}{M}$. Similarly, generating length-(M-2) sub-patterns from length-(M-1) sequences takes $\frac{NE(M-2)^2}{M-1}$. Overall the time complexity of generating all sub-pattern candidates (from length-$M-1$ to length-1) is $O((ln(M) + 1 + \frac{1}{2}M^2 - \frac{3}{2}M)NE)$.

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 $O(\frac{NE}{l \times minLocalSup})$ in the worst case. Therefore the total number of candidates is $O(\frac{NE}{M \times minLocalSup}) + O(\frac{NE}{(M-1) \times minLocalSup}) + \ldots + O(\frac{NE}{1 \times minLocalSup}) = O(\frac{NE(ln(M)+1)}{minLocalSup})$.

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 $\frac{N}{M}$ length-M occurrences for each event type, event availability update for all length-M sequences takes at most NE. 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 $O(NEM)$.

In general, the total time complexity of Reduce is $O((ln(M) + 1 + \frac{1}{2}M^2 - \frac{3}{2}M)NE + \frac{NE(ln(M)+1)}{minLocalSup} + NEM)$. Therefore, it is linear in the length of the input sequence.

4 VO MINER: FINDING OUTLIER PATTERNS

Next, we present our outlier pattern detection approach, which generates *contextual outlier* patterns for each sequence S. It contains two components: a LO (locally frequent outlier) miner and a VO (violation outlier) miner. Both the LO miner and VO miner can be realized by lightweight counting and filtering processes.

The **LO miner** generates locally frequent outliers of each sequence. The LOs are generated from the CF patterns. Whenever some pattern P is recognized as a CF pattern in a sequence S, P is inserted into a CF hash table with P as its key. If P is a new key, then its count (value) is set to 1, otherwise its count is incremented by 1. After all input sequences S have been processed, each bucket in the CF hash table represents a CF pattern P (key) and the number (value) of the sequences S that recognize P as CF. LO patterns can be found by selecting from the hash table all patterns that appear in fewer than *seqCnt* sequences by Def. 2.3. This one pass counting process is linear in the number of CF patterns discovered by *Reduce*.

The **VO miner** is composed of two phases: (1) VO candidate generation and (2) VO pattern generation.

VO Candidate Generation. This phase generates the infrequent violation outlier (VO) candidates in each sequence. It is embedded into the CF pattern mining process of Reduce. As described in Sec. 3.2, 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 discover and maintain the *infrequent patterns* with local support no larger than *cntThr* (Def. 2.4). It is straightforward. As shown in Sec. 3.2.2, 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 can be naturally discovered 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 $cntThr$.

Once an infrequent pattern is acquired, we examine whether it is a potential VO. The key idea here is that by the definition of VO (Def. 2.4), an infrequent pattern P is potentially a VO only if it is a sub-pattern of one CF pattern longer than it or a different permutation of one CF pattern that has the same length with P. Therefore, once an infrequent length-L pattern P is acquired, we can immediately determine whether it has chance to be a VO, because all CF patterns with length $\geq L$ have already been found in the top-down mining process of Reduce. Moreover, an *inverted index* on the existing CF patterns is built to accelerate the examine process.

The inverted index is a HashMap with the event type E as key and a bitmap as value. This bitmap records in what patterns type E event is involved. Therefore, each bit in it corresponds to one CF pattern. Based on the above analysis, given a VO candidate P, the corresponding CF patterns that P violates must include all event types that P contains. Therefore, to determine whether P is a VO candidate, we only have to examine the CF patterns that contain all event types in P. These CF patterns can be efficiently located using the inverted index. That is, we first get the bitmaps corresponding to the event types in P 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 P. Then we generate a new bitmap B_n by bit operation: $B_n = B_1 \& B_2 \& \ldots \& B_m$. If the ith bit of B_n is "1", then the corresponding ith pattern in the CF pattern list is a super-pattern or a different permutation of P.

VO pattern Generation. Next, a filtering step is conducted to select the VO outliers from the candidates. That is, a candidate Q is discarded if the pattern P violated by Q is a LO pattern. The remaining candidates are returned as VO outliers. After storing the final LO patterns in a hash table, given one candidate Q, this filtering can be simply supported by probing its violated CF pattern P in the LO hash table. The complexity of this process is linear in the number of VO candidates.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup and Methodology

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 rich variety of characteristics.

Real Datasets. We experiment with two real datasets: **logs** from a mobile application that tracks driver behavior (used in our previous examples) and the **lighting** dataset that contains network messages exchanged between Philips Hue lighting devices and the servers in a city environment. These datasets are confidential datasets from our industry collaborators. The collaborators are interested in finding the outlier patterns in their systems to capture software bugs and faulty devices. The data will be made publicly available after normalization.

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 datasets with various distributions to evaluate the efficiency of our *Reduce* strategy.

Since the sequence generators utilized in the literature [2, 12, 13] were designed to only generate short transaction data and do not offer control of the number nor the distribution of frequent patterns within the synthetic data, we developed a new sequence data generator. It supports a number of input parameters that allow us 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 CF patterns for each sequence is independent of other sequences, CF pattern mining is amenable for parallel processing. Therefore we leverage distributed computing platform to mine CF patterns. Since the CO pattern mining is lightweight, it is supported on one single machine. 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. All code used in our experiments is available at GitHub: https://github.com/OutlierDetectionSystem/TOP.

Algorithms. We evaluate (1) *Growth (G)*: the traditional growth-based approach adapted to mine CF patterns (Sec. 3.1); (2) *Reduce (R)*: our proposed reduction-based approach (Sec. 3.2).

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 not measured here, because it is impossible for the domain experts to manually find out all true outiers hidden in the data. Second, we measure the execution time averaged on each sequence of *Growth* and *Reduce* strategies. Besides, the *peak memory* usage is also measured.

Table 3: Effectiveness Evaluation.

Methods	Dataset	Number of true outliers (NTO)	Precision
TOP	Log file	93	58%
Max	Log file	47	38%
TOP	Lighting	32	97%
Max	Lighting	24	77%

5.2 Evaluation of Effectiveness

We evaluate the effectiveness of our TOP system at detecting outlier patterns LO and VO by comparing against the maximum frequent pattern concept [6, 7] denoted as *Max*. We apply *Max* to detect outliers by replacing our CF pattern concept used in LO and VO with *Max*. The same sets of parameters are applied to both approaches (Log file data: $minLocalSup = 100$, $seqGap = 8$; Lighting data: $minLocalSup = 10$, $seqGap = 8$).

As shown in Table 3, TOP outperforms Max in both the number of detected true outliers (NTO) and precision on both datasets. The reason is that our CF concept does not miss any pattern that frequently occurs by itself (not as part of a super-pattern), while Max blindly discards all patterns if its super-pattern is frequent. Therefore TOP is able to capture most LO outliers. This leads to high NTO. Furthermore, CF excludes from the VO outlier candidates the patterns that mostly occur as part of a super-pattern. Therefore, TOP does not produce many false VO outliers – hence high precision.

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. 4(a) are common on most devices, and happen at initialization time in the displayed order. Surprisingly, one device reported the pattern shown in Fig. 4(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 application that ran on only a few devices had a different sequence of log events.

As shown in Table 3, the number of true outliers (NTO) found by Max is much lower than TOP, since Max cannot find any outlier pattern P that misses the expected events (sub-pattern outlier) for two reasons. First, without our contextual constraint, any pattern will be considered as frequent if its super-pattern is frequent. Therefore, Max misses the infrequent sub-pattern outliers. Second, if P is locally frequent (LO), it will be suppressed by the frequent super-pattern that it violates by the definition of Max. Its low precision is caused by the false outliers that do not occur independently.

As for the **Lighting** data, 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, because Max fails to capture sub-pattern outliers as described above.

```
(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 4: Example of Typical & Outlier Pattern in Log File

5.3 Evaluation of Efficiency on Real Data

We investigate how the input parameters including *minLocalSup* (Fig. 5) and *seqGap* (Fig. 6) influence the performance of Growth and Reduce using both the log file and Lighting datasets. The parameters are fixed as *minLocalSup = 100 seqGap = 10* unless under variation.

When varying *minLocalSup*, Reduce is at least 13 and 199 times faster than Growth w.r.t. the log file dataset and the Lighting dataset as shown in Figs. 5 and 7. Furthermore, 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 any shorter pattern that eventually gets pruned later. 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. 6 and 8. 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 by Growth 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 down 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.

5.4 Evaluation of Efficiency on Synthetic Dataset

We generate 10 synthetic datasets to evaluate how Growth and Reduce perform on datasets with varying characteristics such as varying 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 varying corresponding dataset properties. The input parameters are set to *minLocalSup* = 20, *seqGap* = 20 and *eventGap* =0, because these parameters are effective in capturing outliers. The results are shown in Fig. 9.

Varying Sequence Lengths. Fig. 9(a) demonstrates the results of varying sequence lengths from 10,000 to 50,000. As shown in Fig. 9(a), Reduce consistently outperforms Growth up to 30x (note logarithmic time scale). The longer the sequences are, the more the Reduce-based approaches win. This confirms that our top-down Reduce strategy performs well when handling long sequences. Moreover, even when the sequence length is as a very small value such as 10, Reduce-based approaches are still about 2x faster than Growth-based approaches (not shown in the figure).

Varying Number of Event Types. Fig. 9(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-based methods continuously outperform Growth-based methods by 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-based methods scale better than Growth-based methods also in the number of event types.

Varying Pattern Lengths. Fig. 9(c) represents the results when varying average pattern lengths. Reduce-based methods outperform Growth up to 131x. Furthermore, as shown in Fig. 9(c), Reduce-based methods are not sensitive to the pattern length. On the contrary, the processing time of Growth-based methods increase dramatically as the pattern length gets larger. Worst yet, the Growth approach without triggering event constraint runs out of memory when the average pattern length is 100. The increasing processing time of Growth-based methods result from both its growth-based pattern generation step and its backward filtering step. The pattern generation of Growth-based approaches are 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.

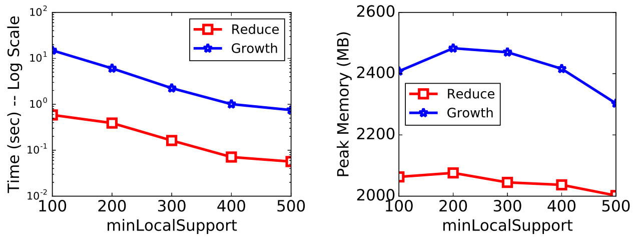
In conclusion, Reduce consistently outperforms Growth by up to 2 orders of magnitude under a rich variety of data characteristics.

6 RELATED WORK

Frequent Pattern Mining Algorithms. Frequent pattern mining was first proposed in [2] to mine typical purchase patterns from a customer transaction dataset. A purchase pattern is called *frequent* if it occurs in more than *support* customer's transaction histories, each of which is considered as one input sequence. Two Apriori-based mining algorithms, namely AprioriSome and AprioriAll, were proposed in [2] to reduce the search space of frequent patterns.

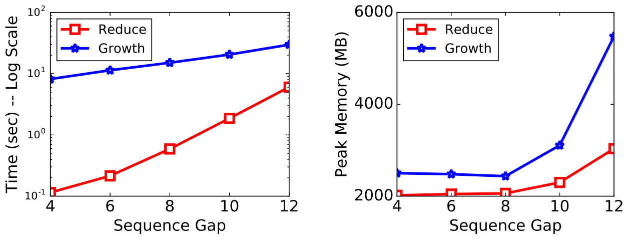
Since then, techniques have been proposed such as GSP [18], FreeSpan [12], PrefixSpan [13], SPADE [23], SPAM [3] and their extensions CM-SPADE, CM-SPAM [5] to scale frequent pattern mining to large transaction datasets. In particular, PrefixSpan [13] avoids multiple scans of the whole dataset while still 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, as confirmed in our experiments, it performs significantly worse than our novel *Reduce* strategy customized to handle complexed IoT sequence datasets.

Extensions of Frequent Pattern Semantics. Different variations of the basic frequent pattern mining semantics [4, 6–8, 21] have been designed to reduced number of reported patterns, in particular *maximum frequent pattern* [6, 7] and *closed frequent pattern* [8, 21]. However, as we have analyzed in Sec. 2.2, maximum frequent patterns are not effective in capturing typical and outlier patterns in IoT applications. The *closed frequent pattern* semantics [8, 21] exclude a frequent pattern from the output if its support is identical to the support of any of its super-patterns. Therefore, even if only one independent occurrence of pattern Q exists, Q will still be reported as frequent. This will miss the opportunity of capturing the independent occurrences of Q as outliers. Overall, the root cause of the



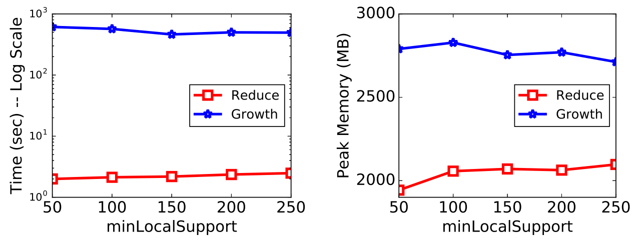
(a) Processing Time (b) Peak Memory Usage

Figure 5: Varying Minimum Local Support on Logfile Dataset.



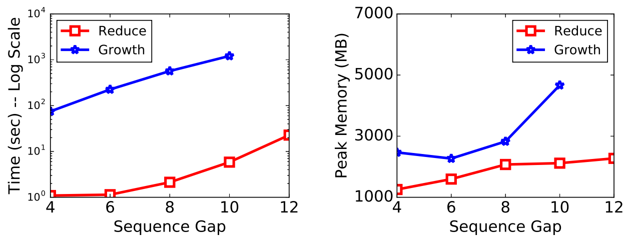
(a) Processing Time (b) Peak Memory Usage

Figure 6: Varying Sequence Gap on Logfile Dataset.



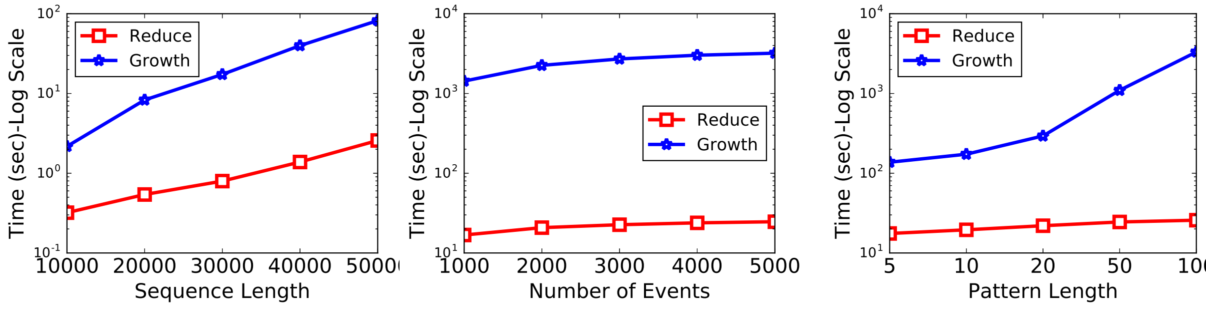
(a) Processing Time (b) Peak Memory Usage

Figure 7: Varying Minimum Local Support on Lighting Dataset.



(a) Processing Time (b) Peak Memory Usage

Figure 8: Varying Sequence Gap on Lighting Dataset.



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

Figure 9: Evaluation of Processing Time on Various Synthetic Datasets

weakness of these prior semantics is that they do not distinguish the independent occurrences of a sub-pattern from the sub-occurrences of its super-pattern. While, in contrast, our contextual pattern semantics successfully solve this problem by leaving the decision up to the individual pattern context – the system status under which the pattern occurs.

Top-down Frequent Pattern Mining Approaches. In [14, 16] 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 c sequence iteration, where c represents the support threshold. The idea stands in contrast to our Reduce strategy which constructs patterns from long to short. Furthermore, this approach works only when the dataset contains a small number of sequences, while our Reduce is shown to be efficient in large scale sequence data.

Episode Mining in Singular Event Sequence. Episode mining studied in [15, 17, 19, 22] is defined as a set of events that frequently occur together in one single sequence. In [15, 17], efficient approaches were developed to count the number of occurrences for the set of candidate episodes specified by the user. Therefore they solve a *counting* problem, while our work focuses on a mining problem without a candidate pattern set given beforehand. In [19], Tatti et al proposed to find a subset of episodes from a given set of episode candidates such that the selected episodes summarize a sequence most succinctly. It is not solving our problem of mining frequent patterns from raw sequence data. The UP-Span algorithm [22] focuses on finding the high utility episodes from a sequence. The utility of each episode is measured by the external and internal utilities of each event provided by the users. Therefore unlike frequent pattern mining, the returned episodes do not reflect their frequencies. Hence UP-Span is not applicable to our problem.

Abnormal Episode Mining. [9, 10] studied the problem of finding abnormal episodes from a set of episode candidates provided by the users. Statistical models were used to learn a frequency cutoff threshold that separates suspicious episodes from normal episodes based on their frequencies. Our TOP system instead automatically discovers the typical and outlier patterns from a set of raw sequences. An abnormal pattern candidate set is not needed.

7 CONCLUSION

In this work we design the TOP system for the effective discovery of both typical and abnormal system behaviors from IoT generated sequence data. TOP features new pattern semantics called *contextual patterns* and a novel pattern mining strategy *Reduce*. Our experimental evaluation with real datasets demonstrates the effectiveness of our new semantics in capturing typical and outlier patterns, and the efficiency of *Reduce* in discovering them.

REFERENCES

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

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] R. Gwadera, M. J. Atallah, and W. Szpankowski. Markov models for identification of significant episodes. In *SDM*, pages 404–414, 2005.

[10] R. Gwadera, M. J. Atallah, and W. Szpankowski. Reliable detection of episodes in event sequences. *Knowl. Inf. Syst.*, 7(4):415–437, 2005.

[11] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15(1):55–86, 2007.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

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

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

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

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

[22] 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.

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