Finding Relevant Patterns in Bursty Sequences

Alexander Lachmann*
RWTH
Aachen, Germany
alexander.lachmann@rwth-aachen.de

Mirek Riedewald
Cornell University
Ithaca, New York
mirek@cs.cornell.edu

ABSTRACT

Sequence data is ubiquitous and finding frequent sequences in a large database is one of the most common problems when analyzing sequence data. Unfortunately many sources of sequence data, e.g., sensor networks for data-driven science, RFID-based supply chain monitoring, and computing system monitoring infrastructure, produce a challenging workload for sequence mining. It is common to find *bursts* of events of the same type. Such bursts result in high mining cost, because input sequences are longer. An even greater challenge is that these bursts tend to produce an overwhelming number of irrelevant repetitive sequence patterns with high support. Simply raising the support threshold is not a solution, because at some point interesting sequences will get eliminated. As an alternative we propose a novel transformation of the input sequences. We show that this transformation has several desirable properties. First, the transformed data can still be mined with existing sequence mining algorithms. Second, for a given support threshold the mining result can often be obtained much faster and it is usually much smaller and easier to interpret. Third, and most importantly, we show that the result sequences retain the important characteristics of the sequences that would have been found in the original (not transformed) data. We validate our technique with an experimental study using synthetic and real data.

Keywords

Frequent sequence mining, data transformation, event stream, bursts, temporal data mining

1. INTRODUCTION

Sequence data is ubiquitous and mining this data to find patterns is a challenging problem for many applications [12]. In this paper we focus on the important problem of *finding frequent subsequences* in a set of given input sequences. Traditionally frequent sequence mining is used to discover purchase patterns in sales transaction data. Consider a customer's purchase history like {chocolate, chips, water} → cheese → {broccoli, carrots}. Here the *set step* {} indicates that products were purchased in the same transaction, while the *sequence step* → indicates that the products on the left were purchased earlier than those on the right. A subsequence like {chocolate, chips} → {broccoli, carrots} could indicate that after indulging on sweets and snacks, the customer feels guilty and purchases healthy vegetables.[1] If many customer sequences contain this subsequence, stores can take advantage of such patterns for targeted advertisement or promotions.

Discovery of common patterns of page visits in Web logs can help in improving the design of a Web site or in deciding what advertisements to present to Web surfers. By finding common sequences of hardware or software related events (errors, warnings, status events) that lead to critical system failures, system administrators can take active measures for re-configuring/re-designing systems or for preventive maintenance. Similarly, there is strong interest in finding frequent patterns in other inherently sequential data like RFID readings in supply chain monitoring and readings from sensors monitoring natural or industrial processes. Last but not least, frequent sequence mining has also been applied to DNA data and medical treatment sequence analysis.

Frequent sequence mining is concerned with finding sequences that are contained in a large fraction of input sequences, i.e., subsequences that have a high *support*. Returning to the purchase analysis example, a sequence pattern is frequent if it occurs in many customer sequences. An input sequence can support a number of subsequences that is exponential in its size. This makes frequent sequence mining for long sequences expensive.

In this paper we concentrate on a problem that is common in all the above mentioned applications concerned with mining of event logs—*bursts of common events*. Consider a large digital printing machine for industrial scale document printing. Complex systems like this continuously produce events reporting status of components (e.g., currents at various electronic components or motors, resets of system components), less severe problems (e.g., paper jams, exceptions reported by firmware, too early or too late arrival of paper at various sensors), or critical errors (transport motor faults, open interlocks during run). If the wrong paper

*Work done while visiting Cornell University.

[1] Notice that transactions in the subsequence do not have to be adjacent to each other in the input sequence; transactions in between can be “skipped over”.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

Original input data	Transformed
167 → 232 → 232 → 167 → 167	$\{167, 232\}$
167 → 232 → 167 → 167 → 167	167 → 232
167 → 232 → 167 → 167 → 232	
167 → 232 → 232 → 232 → 232 → 232 → 232	
167 → 167 → 232 → 232 → 232 → 232	
232 → 167 → 232 → 232 → 232	
And all subsequences of these sequences	

Table 1: Frequent result sequences (real data)

is used or paper feed rolls are dirty, a high rate of paper transport related timeouts might be recorded. A paper jam will typically result in another burst of warning and error messages, and so on. A typical source of event bursts are watchdog processes that repeatedly poll system components and record timeouts.

In general, depending on the state of system components, bursts of certain event types will occur. Similar bursts can also be observed in Web logs (popular topics [8]), sensor networks (typical environmental conditions versus unusual events), and computing system event logs (normal status, overloaded machines, intrusion attacks).

Bursts create two challenges for sequence mining. First, higher event rates produce longer input sequences, which makes mining more expensive. Second, and actually worse, the repeated occurrence of the same event type produces *irrelevant results* that bury the more interesting sequences. The left column in Table 1 shows a typical subset of the result obtained by mining event logs from large digital printing machines. Events 167 and 232 signal that two different electromechanical components are in an error state, and they tend to occur in bursts. In the mining results we can see many combinations of these two events in different orders. Notice that all subsequences of the ones shown are also frequent, but are omitted to avoid clutter.

The problem with results like in the left column in Table 1 is that the different combinations of the common events do not convey much information and make it virtually impossible to find other patterns in the overwhelmingly large result set. One could try to raise the support threshold so that fewer sequences qualify as frequent. However, as we will show, bursts of common events tend to produce uninteresting sequences with high support, and hence other more interesting sequences would be eliminated by higher support thresholds even before many of the repetitive sequences. Alternatively, one could completely remove common events like 167 and 232 from the analysis. However, this is not desirable and would result in significant information loss because such event bursts only occur at certain times. For example, a burst of paper feed delay events might be important in signaling mechanical problems with some printer component and hence should not be ignored.

We propose to transform the input data to eliminate repetitive event patterns created by bursts, but at the same time retain the important structure in the data. Intuitively, we map the individual re-occurrences of an event type to a high-level concept modeling the burst. Then we further transform this mixed sequence of bursts and individual events to a traditional sequence format so that existing sequence mining algorithms can be applied. For the printing system example, the individual occurrences of events 167 and 232 are transformed into a few transactions (sets of events) signaling that "many 167 and 232 events occurred" during the corresponding time period. The mining result for the transformed data is shown in the right column in Table 1. Notice that far fewer sequences are produced, but the essential information that events 167 and 232 occurred is retained.

In particular, we make the following contributions:

- We propose a novel transformation of the input sequences. It first replaces repeated occurrences of the same event type by an object that encodes the notion of a burst. Then it maps a sequence of such bursts and other events to a traditional sequence representation so that existing algorithms for sequence mining can be leveraged.
- We prove that the transformation reduces input sequence length and at the same time retains the most important characteristics of subsequences supported by the original input data.
- We show how we can further reduce the result set for the transformed sequences.
- We show the effectiveness of our technique in a series of experiments with synthetic and real data.

In the remainder of the paper we first introduce important notation and discuss properties of sequence data in Section 2. The input transformation is introduced in Section 3, its properties are analyzed in Section 4. The sequence mining algorithm and extension for reducing result size are discussed in Section 5. In Section 6 we present results of an experimental evaluation. Related work is discussed in Section 7 and we conclude in Section 8.

2. PRELIMINARIES

We first introduce the standard definitions of itemset, sequence, subsequence, sequence database, and support following mostly the notation of [5]. Then we discuss important properties of bursty sequences.

2.1 Notation

Definition 1. (*itemset*, *sequence*) Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items; an itemset is a subset of I. A sequence S is a set of itemsets with timestamps, i.e., $S = \{(s_1, t_1), (s_2, t_2), \ldots, (s_l, t_l)\}$, where $s_j \subseteq I$ $(1 \le j \le l)$ and $t_i < t_j$ for all $1 \le i < j \le l$.

The s_j in sequence S are often referred to as *transactions*. When only the sequence order is important, but not the actual timestamps, we will write S more compactly as $s_1 \to s_2 \to \cdots \to s_l$.

Definition 2. (*length*, *i-length*) The length of a sequence $S = \{(s_1, t_1), (s_2, t_2), \ldots, (s_l, t_l)\}$, denoted $|S|$, is the number of elements in the sequence, i.e., $|S| = l$. The i-length of a sequence is the total number of instances of items in the sequence, i.e., $|s_1| + |s_2| + \cdots + |s_l|$.

Definition 3. (*subsequence*, *super-sequence*) A sequence $A = a_1 \to a_2 \to \cdots \to a_l$, $a_j \subseteq I$ for $1 \le j \le l$, is a subsequence of another sequence $B = b_1 \to b_2 \to \cdots \to b_m$, $b_j \subseteq I$ for $1 \le j \le m$, if there exist integers $1 \le j_1 < j_2 <$

$\cdots < j_l \leq m$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$, …, $a_n \subseteq b_{j_n}$. We write $A \sqsubseteq B$ to denote that A is a subsequence of B, and B is a super-sequence of A.

Notice that the length of a sequence does not depend on the size of the itemsets, e.g., both $a \to b$ and $\{a, c\} \to \{b, d, e\}$ have length 2. However, their i-length is different, 2 and 5, respectively, in the example. For the subsequence relation, the actual timestamps are irrelevant. Only the order of the transactions matters and that some transactions in B contain the corresponding transactions in A.

Definition 4. (*sequence database, support*) A sequence database DB is a set of tuples of the form (cid, S), where cid is a unique identifier, historically referred to as *customer ID*, and S a sequence. A tuple (cid, S) *contains* (or *supports*) a sequence A if A is a subsequence of S. The support of a sequence A in database DB is the number of tuples that contain A, i.e.,

$$\text{support}_{\text{DB}}(A) = |\{(\text{cid}, S)|\ (\text{cid}, S) \in \text{DB} \ \wedge\ A \sqsubseteq S\}|.$$

In the remainder of this article we will omit the database name from the support function and simply write support(A). Note that historically the fact that (cid, S) contains A is also expressed as "customer cid supports sequence A".

The goal of frequent sequence mining *is to find all sequences over itemset I that have a support in database* DB *that is greater than or equal to a user-specific threshold minSupp.*

2.2 Properties of Bursty Sequences

From the definition it follows immediately that sequence support has the monotonicity property. Formally, for sequences A and B, if $A \sqsubseteq B$ then support$(A) \geq$ support(B). This property is heavily exploited for designing efficient algorithms for finding all frequent sequences: Once we know that a given sequence A is not frequent, then none of its super-sequences can be frequent either and the algorithm can prune the search space effectively.

However, the downside of monotonicity is that if some long sequence B is frequent, than all its subsequences are frequent as well. This is particularly problematic for bursty event streams. As the example in Table 1 illustrated, event bursts produce uninteresting repetitive result patterns and increase mining time because finding such long patterns is expensive. The following lemma shows that already bursts of two event types that occur simultaneously, tend to generate long repetitive sequences with high support.

Lemma 1. *Let $A = a_1 \to a_2 \to \cdots \to a_n$ be a sequence of length n where the a_j, $1 \leq j \leq n$, are i.i.d. and each a_j is either i_1 or i_2 with equal probability of 0.5. Then any given sequence of m items, such that $m \leq n$ and each item is either i_1 or i_2, is a subsequence of A with probability at least* $(1 - 0.5^{\lfloor n/m \rfloor})^m$.

Proof. We first prove the lemma for a sequence $B = i_1 \to \cdots \to i_1$ with m occurrences of item i_1.

Consider the first $\lfloor n/m \rfloor$ items of A. Each of these items is either i_1 or i_2 with equal probability. Hence the probability that there is at least one occurrence of i_1 among the first $\lfloor n/m \rfloor$ items is $1 - 0.5^{\lfloor n/m \rfloor}$. We can show similarly, that there is a probability of $1 - 0.5^{\lfloor n/m \rfloor}$ for at least one occurrence of i_1 among the next $\lfloor n/m \rfloor$ items, and so on. In total we therefore obtain a probability of $(1 - 0.5^{\lfloor n/m \rfloor})^m$ of finding at least one instance of i_1 in *each* of the m adjacent subsequences of length $\lfloor n/m \rfloor$.

In general, there could be multiple instances of i_1 in a subsequence of length $\lfloor n/m \rfloor$. Hence $(1 - 0.5^{\lfloor n/m \rfloor})^m$ is a, possibly rather loose, lower bound for the probability of $B = i_1 \to \cdots \to i_1$ being a subsequence of A.

Since i_1 and i_2 have the same probability of occurrence, the above proof can be applied to any sequence of length m that consists of any combination of i_1 and i_2 instances. □

Lemma 1 provides no tight bound, but nevertheless it suffices to illustrate the problem caused by bursts. For example, for $n = 100$ and $m = 10$, the probability is 0.99; for $n = 100$ and $m = 20$ it still is as high as 0.53. This means that a burst of 100 events of type i_1 and i_2 is almost guaranteed to support *all* possible sequences of i_1 and i_2 of length 10, and with high probability it also supports many such sequences of length 20. If i_1 and i_2 are common items (events), then such bursts occur for many input sequences and these repetitive sequence patterns and all their subsequences will have high support. Furthermore, many super-sequences might be frequent as well, creating an overwhelming result set.

This analysis naturally generalizes to bursts containing more than two different items, but the lower bound will be looser the more different items are considered. Obviously, in practice i_1 and i_2 might occur with different probabilities during a burst and occurrences will typically not be independent. In principle it is possible to adjust our analysis for such cases. E.g., different probabilities can be taken into account by partitioning the burst sequence non-uniformly. However, the simple analysis above already illustrates the problem that event bursts tend to support a large number of long and repetitive sequences of common events and we observed such patterns in real data (see Table 1 for a representative example).

3. INPUT TRANSFORMATION

The main idea for transforming the input data is to replace a burst of events of the same type by a single interval covering the time period during which the burst occurred. To be able to leverage existing high-performance algorithms for sequence mining, which cannot handle intervals, we also introduce a mapping from intervals to traditional sequences.

3.1 Transforming Bursts to Intervals

Consider a sequence like $(a, 1), (a, 2), (a, 4), (a, 5), (a, 100)$. If a is a non-critical warning event, users typically do not care about how many a events were created within a short period of time. It is enough to know that "many a events occurred between time 1 and 5". Similarly, for simultaneous bursts of a and b events it would be sufficient to know that such bursts occurred for a certain time period, but the actual number and ordering of the a and b events is not important.

We model this abstract notion of a burst by supporting events with *duration.* In the above example, we can replace the original sequence of a events by two intervals: $(a, 1, 5)$ and $(a, 100, 100)$. This idea is formalized as follows.

Definition 5. (*τ-linkable transactions*) Let τ be a user specified non-negative number, called *merge threshold.*

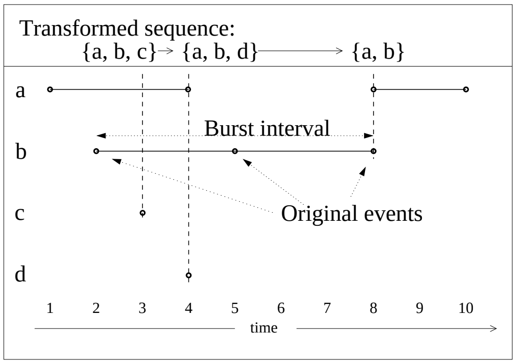

Figure 1: Transformation of item-sequence

Transactions (s_1, t_1) and (s_2, t_2) are τ-linkable if (1) s_1 and s_2 contain exactly one item, (2) $s_1 = s_2$, and (3) $t_2 - t_1 \leq \tau$.

Definition 6. (τ*-maximal interval set* for homogeneous singleton sequence) For a sequence $A = \{(i, t_1), (i, t_2), \ldots, (i, t_j)\}$ of singleton transactions containing only item $i \in I$, the τ-maximal interval set, $\text{intervals}_\tau(A)$, is the set of intervals obtained by merging all τ-linkable transactions, i.e.,

$$\begin{aligned}\text{intervals}_\tau(A) &= \{(i, t_k, t_l) |\ 1 \leq k \leq l \leq j \\ &\quad \wedge \forall k \leq m < l : t_{m+1} - t_m \leq \tau \\ &\quad \wedge t_k - t_{k-1} > \tau \ \wedge \ t_{l+1} - t_l > \tau\}.\end{aligned}$$

Definition 7. (i*-projected sequence*) For an item $i \in I$ and a sequence $S = \{(s_1, t_1), (s_2, t_2), \ldots, (s_l, t_l)\}$, the corresponding i-projected sequence $S(i)$ is defined as

$$S(i) = \{(i, t) |\ \exists (s, t) \in S : i \in s\}.$$

Definition 8. (τ*-maximal interval set* for arbitrary sequence) Let $S = \{(s_1, t_1), (s_2, t_2), \ldots, (s_l, t_l)\}$ be a sequence. The corresponding τ-maximal interval set, $\text{intervals}_\tau(S)$, is the set of intervals obtained by merging all τ-linkable transactions, i.e.,

$$\text{intervals}_\tau(S) = \bigcup_{i \in I} \text{intervals}_\tau(S(i)).$$

Example 1. Consider input sequence $S = \{(a, 1), (b, 2), (c, 3), (\{a, d\}, 4), (b, 5), (\{a, b\}, 8), (a, 10)\}$ and merge threshold $\tau = 3$. The corresponding a-projected sequence is $S(a) = \{(a, 1), (a, 4), (a, 8), (a, 10)\}$. For the τ-maximal interval set for $S(a)$ we obtain $\text{intervals}_\tau(S(a)) = \{(a, 1, 4), (a, 8, 10)\}$. Similarly, we obtain $\text{intervals}_\tau(S(b)) = \{(b, 2, 8)\}$, $\text{intervals}_\tau(S(c)) = \{(c, 3, 3)\}$, and $\text{intervals}_\tau(S(d)) = \{(d, 4, 4)\}$. Hence $\text{intervals}_\tau(S) = \{(a, 1, 4), (a, 8, 10), (b, 2, 8), (c, 3, 3), (d, 4, 4)\}$. Notice that for example intervals $(a, 1, 4)$ and $(b, 2, 8)$ overlap and $(a, 1, 4)$ contains $(c, 3, 3)$. The items and intervals are shown in the lower section of Figure 1.

3.2 From Intervals to Sequences

The intervals in a τ-maximal interval set can overlap, hence this set does not define a sequence in the traditional sense. Existing algorithms for frequent sequence mining therefore cannot be applied to this set. One option would be to design new algorithms for interval data. This is an interesting direction for future work. For this article we chose to map the interval data to a sequence so that the wealth of existing algorithms and experience for sequence mining can be leveraged.

The mapping from (possibly overlapping) intervals to sequences follows the intuitive interpretation that an interval (i, t_k, t_l) signals that "event i repeatedly occurs between time t_k and t_l". If two intervals (i_1, t_{k_1}, t_{l_1}) and (i_2, t_{k_2}, t_{l_2}) are disjoint, then the corresponding bursts happen one strictly before the other. If the intervals intersect, then both bursts happen simultaneously. We capture this interpretation by creating itemsets for such periods of overlapping time intervals.

Let $\mathcal{T}(S)$ denote the set of all times in $\text{intervals}_\tau(S)$ at which either an interval starts or ends, i.e.,

$$\mathcal{T}(S) = \{t |\ \exists (i, t_k, t_l) \in \text{intervals}_\tau(S) \ \wedge \ (t = t_k \vee t = t_l)\}.$$

Furthermore, let $\text{start}(t)$ and $\text{end}(t)$ denote the set of intervals from $\text{intervals}_\tau(S)$ that start, respectively end, at time t. Similarly, $\text{contain}(t)$ is the set of intervals from $\text{intervals}_\tau(S)$ that start strictly before time t and end at time t or later. We can now formally define the transformation from intervals to a sequence.

Definition 9. (τ*-transformed sequence*) For a given sequence S, let $\mathcal{T}(S) = \{t_1, t_2, \ldots, t_k\}$. The τ-transformed sequence, $\text{transf}_\tau(S)$ is defined as

$$\begin{aligned}\text{transf}_\tau(S) &= \{(\sigma, t_i) |\ t_i \in \mathcal{T}(S) \wedge \text{start}(t_i) \neq \emptyset \\ &\quad \wedge (\text{end}(t_i) \neq \emptyset \vee \text{start}(t_{i+1}) = \emptyset) \\ &\quad \wedge \sigma = \text{contain}(t_i) \cup \text{start}(t_i)\}\end{aligned}$$

Intuitively, the definition ensures that we only generate transactions that are "locally maximal", i.e., no transaction is the subset or superset of the next following transaction in the sequence. In particular, we only generate a transaction for time instants t_i where new intervals start and where either some intervals end as well, or where no intervals start at the next time instant t_{i+1} in $\mathcal{T}(S)$ (and hence some interval has to end at t_{i+1}, because otherwise that time instant would not be in $\mathcal{T}(S)$).

To illustrate this transformation, we continue Example 1 (see Figure 1). For the example, $\mathcal{T}(S) = \{1, 2, 3, 4, 8, 10\}$. Now we derive the transactions for each time in $\mathcal{T}(S)$. At time 1, $\text{start}(1) = \{(a, 1, 4)\}$, but $\text{end}(1) = \emptyset$ and $\text{start}(2) \neq \emptyset$, therefore no transaction is created for time 1. Similarly, at time 2 we have $\text{start}(2) = \{(b, 2, 8)\}$, but $\text{end}(2) = \emptyset$ and $\text{start}(3) \neq \emptyset$, and therefore $\text{transf}_\tau(S)$ also does not contain a transaction for time 2. Then at time 3, we have $\text{start}(3) = \{(c, 3, 3)\}$ and $\text{end}(3) = \{(c, 3, 3)\}$. According to Definition 9, we therefore add transaction $(\{a, b, c\}, 3)$ to $\text{transf}_\tau(S)$. For time 4 we similarly add transaction $(\{a, b, d\}, 4)$. Notice how c and d are still in separate transactions, even though they both are contained in the same a and b interval. Then for time 8, we add $(\{a, b\}, 8)$, because $\text{start}(8) = \{(a, 8, 10)\}$, $\text{end}(8) \neq \emptyset$, and $\text{contain}(8) = (b, 2, 8)$. At time 10 no new transaction is generated, because $\text{start}(10) = \emptyset$. In summary, the final transformed sequence is $\text{transf}_\tau(S) = \{(\{a, b, c\}, 3), (\{a, b, d\}, 4), (\{a, b\}, 8)\}$. Notice how the conditions in Definition 9 eliminate transactions like $(a, 1)$, $(\{a, b\}, 2)$, and $(a, 10)$, because they are contained in $(\{a, b, c\}, 3)$ and $(\{a, b\}, 8)$, respectively.

The transformed sequence captures very naturally the notion of a burst as an ongoing occurrence of an event, or a set of simultaneously occurring events. Uninteresting order information, here for the common a and b events, is eliminated.

4. TRANSFORMATION PROPERTIES

We show that the proposed transformation reduces input length and still preserves the important structural information of the original data. Then we analyze how the result of frequent sequence mining is affected.

4.1 Length Reduction

Theorem 1. *The τ-transformed sequence* $\text{transf}_\tau(S)$ *has at most as many transactions as the original sequence S, i.e., $|\text{transf}_\tau(S)| \leq |S|$.*

Proof. Definition 9 guarantees that the transformed sequence can only have transactions for times t at which an interval in $\text{intervals}_\tau(S)$ starts. Intervals can only start at time t if there was a transaction with timestamp t in the original input data. Hence $\text{transf}_\tau(S)$ can contain transactions only for those times t at which S had a transaction as well. □

In general, the larger τ, the more the individual events get merged into longer intervals, thus reducing the number of transactions in the transformed sequence. However, as more intervals overlap, the transactions in the transformed sequence tend to contain more items than those in the original sequence. In the running example, $\text{transf}_\tau(S)$ contains the 3-item transaction $(\{a,b,c\},3)$, while S only contained 1- and 2-item transactions. This implies that the transformation, while guaranteeing to never increase the length, it might possibly increase the *i-length* of a sequence. For most bursty sequences, including real data, we observed a significant reduction in both length and in i-length for a wide variety of merge threshold settings.

Notice also that the transformation can be done very efficiently (see Section 5). A user therefore can try different values for τ and choose the one that results in good reduction in i-length as well. If no appropriate τ can be found, then the sequences are not bursty enough for the transformation to pay off. In this case the algorithm falls back to mining the original data and the transformation would not be applied.

It is also fairly easy to modify the transformation process so that it uses different merge thresholds for different event types and even for the same event type during different time periods. This way one can more aggressively merge event types that tend to occur in larger bursts at certain times, while leaving event types with short bursts unmerged. Definition 9 applies to any set of intervals, no matter if the same τ was used for all events at all times or not. Developing such adaptive strategies for setting τ is part of our future work.

4.2 Structure Preservation

We first introduce additional notation to simplify the discussion.

Definition 10. Let $\{a_1, a_2, \ldots, a_n\} \subseteq I$ be a transaction in a given sequence. We will also use "$\leftrightarrow$" to denote that two items are in the same transaction, i.e., $\{a_1, a_2, \ldots, a_n\} = a_1 \leftrightarrow a_2 \leftrightarrow \cdots \leftrightarrow a_n$.

Definition 11. For a given sequence S, adjacent items a and b in the sequence have relation $a \rightsquigarrow b$ if and only if either $a \rightarrow b$ or $a \leftrightarrow b$.

Consider a sequence $\{i_1, i_2, i_3\} \rightarrow \{i_4, i_5\}$. We can equivalently write this sequence as $i_1 \leftrightarrow i_2 \leftrightarrow i_3 \rightarrow i_4 \leftrightarrow i_5$. This sequence therefore follows the general pattern $i_1 \rightsquigarrow i_2 \rightsquigarrow i_3 \rightsquigarrow i_4 \rightsquigarrow i_5$.

Theorem 2. *Let $S = \{(s_1,t_1),(s_2,t_2),\ldots,(s_l,t_l)\}$ and* $\text{transf}_\tau(S)$ *be a given original sequence and its corresponding transformed sequence for some $\tau \geq 0$ as defined before. Furthermore, let*

$$A = a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k$$

be a sequence where $a_i \in I$ for $1 \leq i \leq k$ and $\rightsquigarrow_i \in \{\leftrightarrow, \rightarrow\}$ for $1 \leq i < k$. If A is a subsequence of S, then $\hat{A}$ is a subsequence of $\text{transf}_\tau(S)$ *where*

$$\hat{A} = a_1 \rightsquigarrow'_1 a_2 \rightsquigarrow'_2 \cdots \rightsquigarrow'_{k-1} a_k$$

$$\wedge\ \forall 1 \leq i < k : \rightsquigarrow'_i \in \{\leftrightarrow, \rightarrow\}\ \wedge\ ((\rightsquigarrow_i \equiv \leftrightarrow) \Rightarrow (\rightsquigarrow'_i \equiv \leftrightarrow)).$$

Intuitively, the theorem states that if A is a subsequence of the original input sequence, then we can find a subsequence of the transformed input sequence that preserves the important characteristics of A. In particular, all items that occur in A will also occur in the same order in $\hat{A}$ and the $\leftrightarrow$ relation will be preserved. In fact, the only possible change between A and $\hat{A}$ is that some of the sequence steps ($\rightarrow$) in A might have turned into set steps ($\leftrightarrow$) in $\hat{A}$. For example, $A = \{a,b\} \rightarrow c = a \leftrightarrow b \rightarrow c$ might turn into $\hat{A} = a \leftrightarrow b \leftrightarrow c = \{a,b,c\}$, but not into $a \rightarrow b \rightarrow c$.

We will now prove the theorem. The proof makes use of the following lemma.

Lemma 2. *Let $S = \{(s_1,t_1),(s_2,t_2),\ldots,(s_l,t_l)\}$ and* $\text{transf}_\tau(S)$ *be a given original sequence and its corresponding transformed sequence for some $\tau \geq 0$ as defined before. Then the following holds:*

$$\forall a \in I : (\exists (s,t) \in S : a \in s)\ \Rightarrow\ (\exists (\sigma, t') \in \text{transf}_\tau(S)$$

where $a \in \sigma$ and $\exists (a, t_l, t_u) \in \text{intervals}_\tau(S) : t_l \leq t, t' \leq t_u$).

Proof. (Lemma 2) The lemma states that for each item a occurring in a transaction at some time t in the original input sequence, there is a transaction in the transformed sequence that also contains the item and whose timestamp is "approximately" t. Here "approximately" means that if the a instance at time t in the original data is merged into a larger interval with other a instances, then the timestamp in the transformed data could be any time within this interval. See Figure 2 for an illustration. There item a_k with timestamp t_k is contained in an interval starting at time $t_{l(k)}$ and ending at time $t_{u(k)}$. As we show below, this interval will create a transaction in the transformed data at some time between $t_{l(k)}$ and $t_{u(k)}$, and a_k belongs to this transaction.

To prove this lemma, notice that the occurrence of a in a transaction with timestamp t in the original sequence implies that there exists an interval $(a, t_l, t_u) \in \text{intervals}_\tau(S)$ with $t_l \leq t \leq t_u$. This follows from Definitions 6, 7, and 8.

Now we need to show that this interval creates a transaction (σ, t') in $\text{transf}_\tau(S)$ such that $a \in \sigma$ and $t_l \leq t' \leq t_u$. This follows from Definition 9: Without loss of generality, let $\{t_l, t_{l+1}, \ldots, t_u\}$ be the set of timestamps from $\mathcal{T}(S)$

that fall into the range between t_l and t_u. Since (a, t_l, t_u) starts and ends at time t_l and t_u, respectively, we have (1) $(a, t_l, t_u) \in \text{start}(t_l)$, (2) $(a, t_l, t_u) \in \text{end}(t_u)$, and (3) $\forall t \in \{t_{l+1}, \ldots, t_u\} : (a, t_l, t_u) \in \text{contain}(t)$.

Now assume for contradiction that $\text{transf}_\tau(S)$ does not contain any transaction that contains a and has timestamp t' with $t_l \leq t' \leq t_u$. For time t_l we know that $\text{start}(t_l) \neq \emptyset$. According to Definition 9, there is a transaction (σ, t_l) with $a \in \sigma$ in $\text{transf}_\tau(S)$, unless both $\text{end}(t_l) = \emptyset$ and $\text{start}(t_{l+1}) \neq \emptyset$. Hence for the initial assumption that there is no such transaction at time t_l to be satisfied, for the next time t_{l+1} it has to hold that $\text{end}(t_{l+1}) = \emptyset$ and $\text{start}(t_{l+2}) \neq \emptyset$. The same argument can now in turn be applied to t_{l+2} and so on. In the last step of this chain of implications we reach time t_u, for which it also has to hold that $\text{start}(t_u) \neq \emptyset$. However, since also $\text{end}(t_u) \neq \emptyset$ and $a \in \text{contain}(t_u)$, there will be a transaction (σ, t_u) with $a \in \sigma$, contradicting the assumption. □

Proof. (Theorem 2) The proof of Theorem 2 proceeds by induction on the length of A. We prove the following, slightly stronger, statement: *For a subsequence $A = a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k$ of S where a_k has timestamp t_k, there is a corresponding sequence $\hat{A} = a_1 \rightsquigarrow'_1 a_2 \rightsquigarrow'_2 \cdots \rightsquigarrow'_{k-1} a_k$ in $\text{transf}_\tau(S)$ that satisfies the following properties: (1) $a_k \in \sigma$ for some transaction $(\sigma, t'_k) \in \text{transf}_\tau(S)$, (2) $t_{l(k)} \leq t'_k \leq t_{u(k)}$, where $t_{l(k)}$ and $t_{u(k)}$ are the lower and upper endpoint, respectively, of an interval $(a_k, t_{l(k)}, t_{u(k)})$ in $\text{intervals}_\tau(S)$ that satisfies $t_{l(k)} \leq t_k \leq t_{u(k)}$, and (3) if there are multiple transactions $(\sigma, t'_k) \in \text{intervals}_\tau(S)$ with properties (1) and (2), then the a_k instance from A is* mapped *to the transaction with the lowest timestamp value.*

Stated differently, we show that all items in the original sequence A can be mapped to their corresponding transactions in the transformed sequence. And this mapping preserves the timestamps sufficiently so that no item in A would be "skipped" in the transformed data.

Base case: Let $A = a_1$ be a subsequence of S for some a_1 with timestamp t_1 in S. From Lemma 2 follows directly that there exists a transaction (σ, t'_1) in $\text{transf}_\tau(S)$ such that $a \in \sigma$ and $t_{l(1)} \leq t, t' \leq t_{u(1)}$, where $t_{l(1)}$ and $t_{u(1)}$ are the endpoints of some interval $(a_1, t_{l(1)}, t_{u(1)}) \in \text{intervals}_\tau(S)$.

Induction step: Assuming the hypothesis is true for $a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k$, we have to show it also holds for $a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k \rightsquigarrow_k a_{k+1}$.

For the following discussion, Figure 2 provides an illustration. Notice that the figure indirectly implies $a_k \neq a_{k+1}$. However, the proof also applies to $a_k = a_{k+1}$. Similarly, the figure shows only one of several possible spatial relationships between the interval for a_k and the interval for a_{k+1}. The proof also applies to other spatial relationships between these intervals.

Consider the two intervals in $\text{intervals}_\tau(S)$ that contain (a_k, t_k) and (a_{k+1}, t_{k+1}). Let these intervals be $(a_k, t_{l(k)}, t_{u(k)})$, $t_{l(k)} \leq t_k \leq t_{u(k)}$, and $(a_{k+1}, t_{l(k+1)}, t_{u(k+1)})$, $t_{l(k+1)} \leq t_{k+1} \leq t_{u(k+1)}$, respectively. Recall that Lemma 2 guarantees that there exist transactions (σ_k, t'_k) and (σ_{k+1}, t'_{k+1}) in the transformed data such that $a_k \in \sigma_k$, $a_{k+1} \in \sigma_{k+1}$, $t_{l(k)} \leq t'_k \leq t_{u(k)}$, and $t_{l(k+1)} \leq t'_{k+1} \leq t_{u(k+1)}$. Since $t_k \leq t_{k+1}$, the interval endpoints can only have the following order relationships:

1. $t_{l(k)} \leq t_{u(k)} < t_{l(k+1)} \leq t_{u(k+1)}$
2. $t_{l(k)} \leq t_{l(k+1)} \leq t_{u(k)} \leq t_{u(k+1)}$
3. $t_{l(k)} \leq t_{l(k+1)} \leq t_{u(k+1)} \leq t_{u(k)}$
4. $t_{l(k+1)} \leq t_{l(k)} \leq t_{u(k)} \leq t_{u(k+1)}$
5. $t_{l(k+1)} \leq t_{l(k)} \leq t_{u(k+1)} \leq t_{u(k)}$

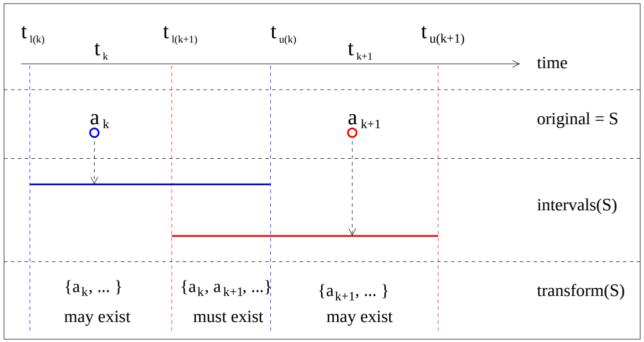

Figure 2: Items $a_k, a_{k+1} \in S$, their corresponding intervals and transactions in the transformed data

We now show for each of these cases that the induction hypothesis is satisfied for $k + 1$.

For the first interval relationship, $t_{l(k)} \leq t'_k \leq t_{u(k)}$ and $t_{l(k+1)} \leq t'_{k+1} \leq t_{u(k+1)}$ implies $t'_k < t'_{k+1}$. Hence the transformed data has the following structure: $\text{transf}_\tau(S) = \cdots \rightarrow (\sigma_k, t'_k) \rightarrow \cdots \rightarrow (\sigma_{k+1}, t'_{k+1}) \rightarrow \cdots$. (Timestamps are shown in the transactions for convenience.) Hence there is a sequence relationship between $a_k \in \sigma_k$ and $a_{k+1} \in \sigma_{k+1}$ and therefore the transformed data must support subsequence $a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k \rightarrow a_{k+1}$. We map a_{k+1} to (σ_{k+1}, t'_{k+1}); if there are multiple such transactions in the transformed data, we map to the one with the smallest timestamp t'_{k+1}.

For the second interval relationship, the intervals overlap between $t_{l(k+1)}$ and $t_{u(k)}$. This is the case illustrated in Figure 2. It is fairly straightforward to show that this intersection implies that the transformed sequence must contain a transaction (γ, t) with the following properties: (1) $a_k \in \gamma$, (2) $a_{k+1} \in \gamma$, and (3) $t_{l(k+1)} \leq t \leq t_{u(k)}$. (The proof is similar to the one for Lemma 2 and therefore omitted.) Now there are two cases to consider.

Case 1: a_k was mapped to a transaction in the transformed data with timestamp $t'_k < t_{l(k+1)}$, i.e., strictly before the intersection with the interval containing a_{k+1}. Then the transformed sequence has the following structure: $\text{transf}_\tau(S) = \cdots \rightarrow (\sigma_k, t'_k) \rightarrow \cdots \rightarrow (\gamma, t) \rightarrow \cdots$, where (σ_k, t'_k) is the earlier transaction containing a_k, which a_k was mapped to. In this case the transformed sequence supports $a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k \rightarrow a_{k+1}$, and we map a_{k+1} to transaction (γ, t) (actually the earliest of these transactions, if there are multiple such transactions containing a_{k+1} with timestamp in the range from $t_{l(k+1)}$ to $t_{u(k)}$).

Case 2: a_k was mapped to a transaction in the transformed data with timestamp $t_{l(k+1)} \leq t'_k \leq t_{u(k)}$, i.e., a time covered by the interval $(a_{k+1}, t_{l(k+1)}, t_{u(k+1)})$. (Notice that it cannot be mapped to a later time, because transaction (γ, t) as defined above contains a_k and we mapped a_k to the transaction with the earliest time.) Definition 9 guarantees that any transaction generated with timestamp t in the range $t_{l(k+1)} \leq t \leq t_{u(k)}$ will contain both a_k and a_{k+1}, because both intervals cover this time range and hence the items are in either start(t) or in contain(t). This in turn implies that the transformed sequence must support $a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k \leftrightarrow a_{k+1}$. We map a_{k+1} to transaction (γ, t) (actually the earliest of these transactions, if there are multiple such transactions containing a_{k+1} with timestamp in the range from $t_{l(k+1)}$ to $t_{u(k)}$).

The proofs for the remaining interval relationships 3–5 are similar, and therefore omitted due to space constraints. □

4.3 Discussion of Theorem 2

Theorem 2 might appear counter-intuitive. We transform the original data to remove repeated event occurrences during bursts, hence it seems impossible that $\hat{A}$ can preserve all of A's items. The "magic" lies in the set relation ↔. Consider the example in Table 1. Sequence $167 \rightarrow 232 \rightarrow 232 \rightarrow 167 \rightarrow 167$ is preserved as $167 \leftrightarrow 232 \leftrightarrow 232 \leftrightarrow 167 \leftrightarrow 167$, which is identical to $\{167, 232\}$—exactly what we set out to achieve. The other sequences are preserved similarly as most of the sequence steps are replaced by set steps.

In general, Theorem 2 guarantees that if there is a subsequence with certain items in the original data, then there is a subsequence with these same items also in the transformed data. However, depending on the merge threshold τ, *repeated* occurrences of items in the original sequence A might disappear due to the transformation.

An important question for frequent sequence mining is if some sequence A is frequent in the original database, will an approximate version of it, e.g., some instance of $\hat{A}$, be also frequent in the transformed database?

Unfortunately, Theorem 2 is not strong enough to guarantee this desirable property. Consider a frequent sequence $a \rightarrow b$ in the original database. As the proof for Theorem 2 showed, depending on the relationship between the intervals for a and b, $a \rightarrow b$ might either be preserved as $a \rightarrow b$ or it might turn into $a \leftrightarrow b = \{a, b\}$ as a result of the transformation. Assume $a \rightarrow b$ had support of 100 in the original data and the support threshold is $minSupp = 90$. Out of the 100 supporting sequences, the transformation might result in a scenario where 50 of the input sequences still support $a \rightarrow b$, while the other 50 now support $a \leftrightarrow b$ instead. In this case, neither $a \rightarrow b$ nor $a \leftrightarrow b$ would be found as a frequent sequence in the transformed data.

We refer to this problem as *fragmentation of support*. There are several ways of dealing with it.

Option 1: We do nothing about it. Notice that while support from a pattern with many sequence steps is usually reduced by the transformation, support for patterns with set steps actually tends to increase. More precisely, a sequence A with n sequence steps (→) distributes its support over up to 2^n different corresponding sequence pattern instances $\hat{A}$. (Each of these patterns $\hat{A}$ is obtained by replacing one or more of the sequence steps in A by set steps.) On the other hand, a sequence $\hat{A}$ with m set steps (↔) receives additional support from up to 2^m different corresponding original sequence patterns A. (Each of these patterns A is obtained by replacing one or more of the set steps in $\hat{A}$ by sequence steps.) Hence overall there is a re-distribution of support, and in some cases the transformation reveals new important patterns.

Option 2: To avoid fragmenting the support, we can modify the definition of support and let a transaction with k items support any sequence that is a permutation of these k items. For example, $\{a, b\}$ would support both $a \rightarrow b$ and $b \rightarrow a$. This way it is guaranteed that if $A = a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k$ is frequent in the original database, then there *exists* an instance of pattern $\hat{A} = a_1 \rightsquigarrow'_1 a_2 \rightsquigarrow'_2 \cdots \rightsquigarrow'_{k-1} a_k$ for some assignment of the different $\rightsquigarrow'$ to either → or ↔, that is frequent in the transformed database.

Option 3: We can lower the support threshold to guarantee that if $A = a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k$ is frequent in the original database, then there exists an $\hat{A} = a_1 \rightsquigarrow'_1 a_2 \rightsquigarrow'_2 \cdots \rightsquigarrow'_{k-1} a_k$ for some assignment of the different $\rightsquigarrow'$ to either → or ↔, that is frequent in the transformed database. For a sequence A with support equal to support(A) with n sequence steps (→), it is sufficient to choose a lower minimum support threshold of $minSupp/2^n$ to guarantee this property. Intuitively, this is true because the support of A only gets distributed over the different sequences $\hat{A}$ that are obtained by replacing some sequence steps in A by set steps. Since there are at most 2^n such different patterns $\hat{A}$, it follows from the generalized pigeonhole principle that at least one of them has to receive support of at least support$(A)/2^n$ after data transformation. Hence, if support$(A) > minSupp$, then for this sequence pattern we have support$(\hat{A}) \geq$ support$(A)/2^n > minSupp/2^n$ and it would therefore be frequent for the modified support threshold.

We propose to use option 1 for several reasons. First, our experiments indicate that usually some approximate version $\hat{A}$ of A is preserved, i.e., fragmentation of support does not appear to be a significant problem in practice. Second, our experiments also show that lowering the support threshold typically leads to a large increase in total runtime and in the number of frequent patterns found. This often more than offsets the runtime and result size improvements of the data transformation, making option 3 an unattractive choice. Third, option 2 will have a similar effect like option 3. During the mining process, for a transaction with n items, all possible $n!$ orderings need to be examined when searching for sequence patterns. This can dramatically increase mining time as well as result size. A detailed study of these tradeoffs is beyond the scope of this paper.

5. ALGORITHM AND EXTENSIONS

The algorithm for mining databases with bursty sequences first transforms the original sequences, then it runs a traditional sequence mining algorithm on the transformed data. The data transformation can be done in time linear in the input size, therefore its cost is usually negligible compared to the mining cost.

For ease of presentation, we describe the data transformation algorithm as if it had two different steps. In the first step, it sequentially reads each sequence in the database in increasing order of transaction timestamps, converting it into the corresponding set of intervals.[2] This is straightforward, and the result is a set of intervals ordered by their starting and then ending timestamps. In the second step, the algorithm scans this database of interval sets and creates the corresponding transactions as follows. It advances from one timestamp t in $\mathcal{T}(S)$, the set of all interval endpoints, to the next. For time t, first all intervals starting at time t are added to the current transaction. As long as

[2] Sequence data in practice usually is recorded in timestamp order. If this is not the case, an additional sorting step would have to be added to the transformation algorithm.

Algorithm 1 Transformation algorithm

```
trans = $\emptyset$; result = $\emptyset$
addingphase = true
for all time in set of transaction timestamps in input sequence in increasing order do
  items = getEvents(time)
  if items $\neq \emptyset$ then
    trans = trans $\cup$ items
    addingphase = true
  items = getIntervalsEnding(time)
  if items $\neq \emptyset$ then
    if addingphase then
      result = result $\cup$ trans
      addingphase = false
    trans = trans $-$ items
return result
```

there are no intervals ending, more items will be added to this transaction for later timestamps. This is the so-called *adding phase* of the transformation. As soon as an interval endpoint is encountered, the algorithm changes from adding phase to *removing phase*. On this phase change, the transaction is written to the output. The algorithm will then stay in the removing phase until another interval starts, at which point it switches back to the adding phase. If at some time t there are both intervals starting and ending, then the algorithm first processes the starting intervals. This also handles correctly point-intervals, i.e., those where start and end time are the same. It is easy to show that this algorithm implements Definition 9.

The algorithm as described above uses two scans, one of the original sequences and one of the interval sets, to transform the data. Both steps can actually be combined into a single pass over the original input sequences. For this the algorithm has to "look ahead" in the item sequence to be able to find the correct ending times of the intervals. When processing an item at time t in the original sequence, it is sufficient to look ahead up to $t + \tau$ to know if the item belongs to an interval ending at time t or not. The algorithm is shown in Algorithm 1.

Example 2. Consider intervals $(a, 1, 4)$, $(a, 8, 10)$, $(b, 2, 8)$, $(c, 3, 3)$ and $(d, 4, 4)$ as before (see Figure 1). The output of the transformation will be the new customer sequence $\{a, b, c\} \rightarrow \{a, b, d\} \rightarrow \{a, b\}$. The dotted lines in Figure 1 mark the times where a new transaction is generated.

5.1 Removing Redundancy

In this subsection we discuss an extension of the mining process to further remove redundant structure in result sequences and also address redundancy that might be created by the transformation. As we will show, Theorems 1 and 2 still hold.

5.1.1 Formalizing the Notion of Redundancy

Consider again Example 1 as illustrated in Figure 1. The transformed sequence contains transactions $(\{a, b, c\}, 3)$ and $(\{a, b, d\}, 4)$. These transactions support sequence $\{a, b\} \rightarrow a$, where both instances of a originate from the same burst interval. This is a redundant and hence undesirable "re-use" of the same burst interval for item a. On the other hand, sequence $\{a, b\} \rightarrow \{a, d\}$ is also supported by the same transactions, but it is not redundant. The crucial difference is the additional item d in the second result transaction. Even though the burst interval for a is still the same, the new d event creates the non-redundant information that the $\{a, b\}$ burst is followed by d, which occurs simultaneously with the a burst.

In general, redundancy can occur when some intervals are contained in others. "Re-use" of the same burst without adding new items is undesirable, as it re-introduces some uninteresting repetitive patterns that we set out to eliminate with the transformation. (Notice that other result patterns like $\{a, b\} \rightarrow \{a, b\}$ would have the same problem.) We therefore would like to remove such sequences from the mining result.

To be able to decide if a result sequence like $\{a, b\} \rightarrow a$ is redundant, we need information about the underlying intervals that created the supporting transactions. To distinguish the intervals, we add a unique ID to each interval as follows: The interval for item a with the lowest start time has ID $a(1)$, the interval for a with the second-lowest start time has ID $a(2)$, and so on. IDs for the other items are assigned in the same way. We can then create the *augmented transformed sequence* for a given original input sequence by using the unique interval IDs instead of the items. For Example 1, the augmented transformed sequence is $\{(\{a(1), b(1), c(1)\}, 3), (\{a(1), b(1), d(1)\}, 4), (\{a(2), b(1)\}, 8)\}$. With the added interval IDs, redundancy for sequence $\{a, b\} \rightarrow a$ can be easily detected. From the first two transactions, we obtain $\{a(1), b(1)\} \rightarrow a(1)$, which is redundant. On the other hand, from the first and the third transaction, we obtain $\{a(1), b(1)\} \rightarrow a(2)$, which is a non-redundant result, because the two transactions are supported by different a-bursts.

To formalize this idea of redundancy, we introduce the concept of a *witness assignment* for a result transaction.

Definition 12. (*witness assignment*) Let $\text{transf}_\tau(S) = \{(\sigma_1, t_1), (\sigma_2, t_2), \ldots, (\sigma_m, t_m)\}$ be a transformed sequence as defined before, where the σ's are itemsets and the t's are timestamps. Furthermore, let $P = \pi_1 \rightarrow \pi_2 \rightarrow \cdots \rightarrow \pi_p$ be some sequence with $\pi_i \subseteq I$ for all i. A function $W : \{\pi_1, \ldots, \pi_p\} \rightarrow \{\sigma_1, \ldots, \sigma_m\}$ is a witness assignment if

1. $\forall 1 \leq i \leq p : \exists 1 \leq j_i \leq m : W(\pi_i) = \sigma_{j_i}$ such that
2. $\pi_i \subseteq \sigma_{j_i}$ and
3. $\forall 1 \leq i < k \leq p : 1 \leq j_i < j_k \leq m$.

A witness assignment intuitively maps each transaction in a result sequence to the corresponding transaction in the transformed input sequence by which it is supported. The conditions ensure that the mapping is total, the witness transaction indeed supports the corresponding result transaction, and the mapping respects transaction order.

Definition 13. (*non-redundant* witness assignment) Let $\text{transf}_\tau(S)$ and P as in Definition 12. Furthermore, let the corresponding augmented transformed sequence for $\text{transf}_\tau(S)$ be $\text{augm}(\text{transf}_\tau(S)) = \{(\sigma'_1, t_1), (\sigma'_2, t_2), \ldots, (\sigma'_m, t_m)\}$, where each σ'_j is obtained from σ_j by replacing all items with their corresponding interval IDs. A witness assignment W is non-redundant, if

$$\forall 1 \leq i \leq p : \forall 1 \leq k < i :$$
$$(W(\pi_i) = \sigma_{j_i} \wedge W(\pi_k) = \sigma_{j_k}) \quad \Rightarrow \quad \pi_i \cap \sigma'_{j_i} \not\subseteq \sigma'_{j_k}$$

To illustrate the last two definitions, we return to our running example. The transformed sequence is $\mathrm{transf}_\tau(S) = \{(\{a,b,c\},3),(\{a,b,d\},4),(\{a,b\},8)\}$; the corresponding augmented sequence is $\{(\{a(1),b(1),c(1)\},3),(\{a(1),b(1),d(1)\},4),(\{a(2),b(1)\},8)\}$. Now consider result sequence $\{a,b\} \rightarrow a$ and the following two alternatives for witness assignments:

1. $W_1(\{a,b\}) = \{a,b,c\}$ and $W_1(a) = \{a,b,d\}$.
2. $W_1(\{a,b\}) = \{a,b,c\}$ and $W_1(a) = \{a,b\}$.

For the first assignment, $a \cap \{a(1),b(1),d(1)\} = \{a(1)\}$ and since $\{a(1)\} \subseteq \{a(1),b(1),c(1)\}$, this assignment is redundant. For the second assignment, we have $a \cap \{a(2),b(1)\} = \{a(2)\}$. And since $\{a(2)\} \not\subseteq \{a(1),b(1),c(1)\}$, this assignment is non-redundant.

Notice that in Definition 13 we slightly abuse notation by overloading the $\cap$ operator. It first computes the intersection by ignoring the additional interval ID information, but after computing the intersection it adds the interval IDs back to the result. Hence in the example we have $a \cap \{a(1),b(1),d(1)\} = \{a(1)\}$ and $a \cap \{a(2),b(1)\} = \{a(2)\}$.

5.1.2 Efficient Algorithm

Having formalized the notion of redundancy in mining results after transformation, we propose an efficient algorithm for removing redundant result sequences. The algorithm is based on the following lemma.

Lemma 3. *Let* $\mathrm{transf}_\tau(S)$, $\mathrm{augm}(\mathrm{transf}_\tau(S))$, *and* P *be defined as in Definition 13. A witness assignment* W *is non-redundant, if and only if*

$$\forall 1 \le i \le p :$$
$$(W(\pi_i) = \sigma_{j_i} \wedge W(\pi_{i-1}) = \sigma_{j_{i-1}}) \quad \Rightarrow \quad \pi_i \cap \sigma'_{j_i} \not\subseteq \sigma'_{j_{i-1}}$$

The lemma can be proved by using the property that an interval that starts at some time $k \le i-1 < i$ and ends after time i, will also contain timestamp $i-1$. The lemma intuitively states that to test if witness σ_{j_i} of result transaction π_i is redundant, we do not need to test the witnesses of *all* previous result transactions. All we need to do is test the *preceding* transaction π_{i-1} to see if all burst intervals used for support of π_i are already contained in the augmented witness of π_{i-1}. This reduces algorithm complexity from depending on $|P|^2$ to being linear in $|P|$.

Algorithm 2 for finding a non-redundant witness assignment follows directly from the above lemma. It maps each result transaction to the earliest possible witness that is non-redundant. The algorithm finds a non-redundant witness assignment if and only if such an assignment exists, and returns `true` if and only if such an assignment was found.

Algorithm 2 is used to "correct" the support count for result sequences. It is applied after the initial mining result has been found on the transformed data. The complete algorithm for finding frequent sequences in bursty data now has the following steps:

1. Transform all input sequences using Algorithm 1.
2. Find all frequent sequences in the transformed data, using an existing frequent sequence mining algorithm.
3. For each result sequence, compute its non-redundant support as the number of transformed input sequences that have a non-redundant witness assignment for this result sequence (using Algorithm 2); eliminate result sequences whose non-redundant support is below the support threshold.

Algorithm 2 Existence of Non-Redundant Witness Assignment

```
$P = \pi_1 \rightarrow \pi_2 \rightarrow \cdots \rightarrow \pi_p$
$\mathrm{transf}_\tau(S) = \{(\sigma_1,t_1),(\sigma_2,t_2),\ldots,(\sigma_m,t_m)\}$
$j = 0$
witness $= \emptyset$
for $i = 1$ to $p$ do
  accept $=$ false
  while $j < m$ do
    $j++$
    if $\pi_i \subseteq \sigma_j$ and $\pi_i \cap \sigma'_j \not\subseteq$ witness then
      witness $= \sigma_j$
      accept $=$ true
      break while
return accept
```

An obvious question is if we can inline the non-redundant support counting into the mining process. Apart from having to modify existing algorithms, the main problem is that non-redundant support does not have the *monotonicity property.* For example, $a(1) \rightarrow \{a(1),b(1)\}$ non-redundantly supports $a \rightarrow \{a,b\}$, but it does not non-redundantly support $a \rightarrow a$.

5.1.3 Preserving Transformation Properties

Adding the redundancy removal step does not affect the validity of Theorems 1 and 2 (see Section 4). Theorem 1 is obviously unaffected. For Theorem 2 we need to show that for each pattern $A = a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k$ in the original data, at least one of the corresponding $\hat{A} = a_1 \rightsquigarrow'_1 a_2 \rightsquigarrow'_2 \cdots \rightsquigarrow'_{k-1} a_k$ is preserved even after removing redundancy. Stated differently, we need to show that not all possible instances of $\hat{A}$ will be eliminated due to redundancy.

Thanks to Lemma 3, this proof is surprisingly simple. It is based on the following observation. Let π_{i-1}, π_i, $\sigma_{j_{i-1}}$ and σ_{j_i} be the result transactions and their corresponding redundant witnesses, i.e., $\pi_i \cap \sigma'_{j_i} \subseteq \sigma'_{j_{i-1}}$. Since $\pi_i \subseteq \sigma_{j_i}$ (witness property), this implies $\pi_i \subseteq \sigma_{j_{i-1}}$. Together with $\pi_{i-1} \subseteq \sigma_{j_{i-1}}$ (witness property), this gives us $\pi_{i-1} \cup \pi_i \subseteq \sigma_{j_{i-1}}$. Stated differently, if there is a pair of adjacent result transactions that violates the non-redundancy property, then the union of these transactions is supported by the witness of the earlier one. This way we can "eliminate" such transaction pairs with redundant support by replacing $\pi_{i-1} \rightarrow \pi_i$ with $\pi_{i-1} \cup \pi_i$, which is the same as $\pi_{i-1} \leftrightarrow \pi_i$. If we start out with some result sequence $\hat{A}$, then it is not difficult to show that repeated combination of adjacent transactions by the union operation will always result in a pattern that also corresponds to an instance of $\hat{A}$, just with more set steps $\leftrightarrow$ that have replaced some of the sequence steps $\rightarrow$.

6. EXPERIMENTS

We study the effects of the proposed data transformation for synthetic and real data, using option 1 for dealing with support fragmentation (see Section 4.3). All experiments are performed on a Pentium 4 PC with 3.80GHz and 2 GB

Parameter	Dense	Sparse
sequence length	200	200
# items	50	100
HL	100	20
LL	500	50

Table 2: Default parameters, synthetic data

of RAM running Windows XP. To mine the frequent sequences, we use the PrefixSpan algorithm that is available in the *IlliMine system package*. PrefixSpan [10] is implemented in C++, our transformation and post processing algorithms are implemented in Java.

We do not report the runtime of the data transformation algorithm. It is a little higher than the cost of a single scan of the input sequences. In all our experiments this cost was negligible compared to the cost of finding the frequent subsequences.

6.1 Synthetic Data

Our synthetic data generator creates customer sequences of desired lengths and can choose events from a pool of different items. To simulate bursts caused by the changing state of a machine, items can be in one of two different phases—a phase where they occur with high frequency and a phase with low frequency. These phases alternate for each item. Events are generated by a Poisson process, whose mean interarrival time determines the average frequency of the items. State changes from high to low frequency and vice versa are controlled by another Poisson process with an independently selectable mean interarrival time.

A synthetic data set is described by four parameters $Hx_1\_Lx_2\_HLx_3\_LLx_4$. Hx_1 is the mean interarrival time for an item in its high frequency phase, Lx_2 is the mean interarrival time of the item during the low frequency phase. The two parameters HLx_3 and LLx_4 determine the average length of the high and low frequency phases, respectively. For example, for parameters $H5\_L20\_HL100\_LL400$ an item has a mean interarrival time of 5 during the high frequency phase, and of 20 during the low frequency phase. The average length of the high and low frequency phases is 100 and 400, respectively.

We studied the impact of the data transformation for various parameter settings. In general we observed that the number of frequent transactions found can be reduced considerably, while significantly speeding up the computation.

In the remainder of this subsection we discuss representative results of our experimental analysis. We include two different types of synthetic data: *sparse* and *dense*. The density of a sequence often results in very different behavior of the mining algorithms. While sparse data sets are usually mined with a low support threshold, dense data has to be mined with a higher support threshold. Dense data is generated by drawing items from a small pool of possible items, while for sparse data a larger item pool is used. The default parameters are shown in Table 2.

Figures 3, 4, and 5 show typical results for dense data. They were obtained for $H2\_L40$ (other parameters at their default). In the transformed data, significantly fewer frequent sequences are contained. With decreasing minimum support threshold, the number of frequent sequences increases rapidly for the original data, but increases only slowly for the transformed data. The computation time for these data sets is strongly dependent on the number of frequent sequences found as can be seen in Figure 4. The same general observations can be made for the sparse data as shown in Figures 6, 7, and 8 for $H2\_L1000$; however notice the different support threshold.

For the sparse data set, the transformation reduces the number of possible combinations in the data sequences even more aggressively, eliminating many long sequences with “medium” support. Decreasing the support threshold therefore leads to a comparably low increase in the number of frequent sequences found for the transformed data. For the original data the number of frequent sequences grows exponentially.

6.2 Real Data

We obtained a number of proprietary datasets containing event logs generated by large printing machines. There are hundreds of machines with hundreds to hundreds of thousands of events recorded for each machine. Frequent sequence mining is used to find those event sequences that signal the occurrence of severe faults. For each severe fault type X, an input sequence database is obtained by considering the sequences of non-X events occurring between all pairs of consecutive X events for all printing machines that had X events. There are dozens of severe faults of interest, resulting in dozens of unique real datasets. We present representative results for one fault type.

In Figure 9 the decrease of the input data size is shown when the transformation is applied to the original data. The data size drops rapidly for merge thresholds between 0 and 10 seconds. For larger merge thresholds the graph takes a slower descent and converges to a data size of about 1750 kilobytes. This indicates the presence of many bursts with a variety of interarrival times.

Figure 10 shows the number of frequent sequences found when using a merge threshold of 80 seconds. The original dataset supports significantly more sequences than the transformed one. For a minimum support of 0.13, there are more than twice as many frequent sequences compared to the transformed data. With decreasing minimum support, this ratio increases even further. For a minimum support of 0.07 there are four times as many frequent sequences as in the transformed data. Filtering out redundant result sequences, as discussed in Section 5.1, reduces the number of sequences in the transformed result even further.

However, the redundancy is not very high, because for this specific case there are not many burst intervals that contain other intervals. For other fault types, minimum support, and higher merge thresholds, more redundancy was removed. Figure 11 shows the effect of choosing a larger merge threshold for the same data set and for minimum support 0.1. Larger merge thresholds create longer intervals, hence redundancy will be more likely. This leads to a situation where the result size for the transformed data at some point starts increasing, even though more and more individual events become absorbed by burst intervals. This increase is due to redundancy, as the “filtered” line indicates, which shows the result size after removing redundant result sequences.

6.3 Preserving Relevant Results

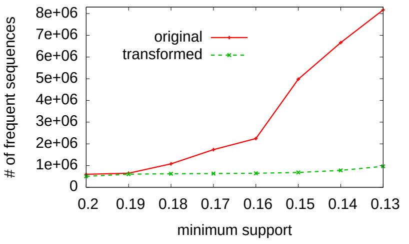

Figure 3: Dense Data, $H2\_L40$

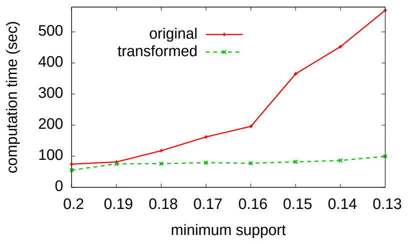

Figure 4: Dense Data, $H2\_L40$

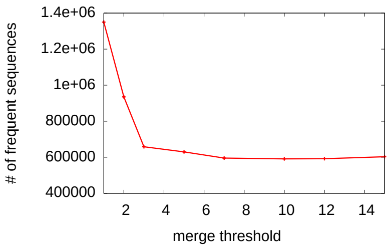

Figure 5: Dense Data, $H2\_L40$

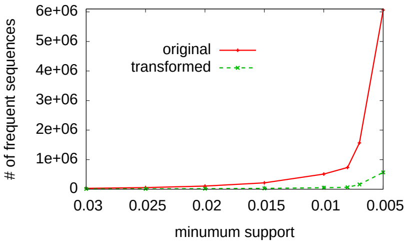

Figure 6: Sparse Data, $H2\_L1000$

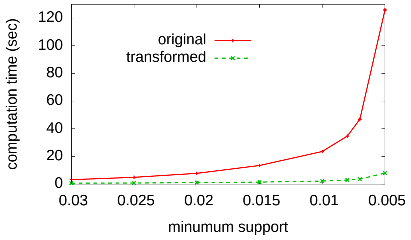

Figure 7: Sparse Data, $H2\_L1000$

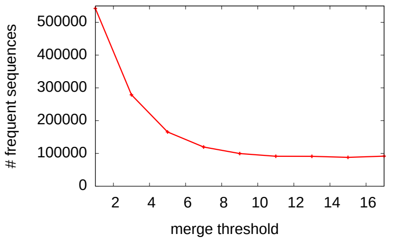

Figure 8: Sparse Data, $H2\_L1000$

We discussed in Section 4.3 that our transformation does not guarantee that some version $\hat{A}$ of a frequent pattern A in the original data will be preserved. In general, only a domain expert can determine if the transformation indeed only eliminated irrelevant patterns, but preserved the most important patterns.

As an objective measure of how well important patterns are preserved, we propose the following approach. For a sequence pattern $A = a_1 \rightsquigarrow_1 a_2 \rightsquigarrow_2 \cdots \rightsquigarrow_{k-1} a_k$, we refer to the set $\{a_1, a_2, \ldots, a_k\}$ (without duplicates!) as its *contained itemset*. We determine the number of different contained itemsets for the frequent sequences found for the original data. Then we determine how many of them are also among the contained itemsets for the frequent sequences found for the transformed data. The ratio between this number for the transformed data divided by the number for the original data indicates how well important non-redundant pattern structure was preserved—a ratio close to 100% indicates good preservation, a ratio close to zero that nothing was preserved. This ratio is a good measure for two reasons. First, there is no redundancy in the contained itemset (it keeps only the "distinct" items occurring in A), and hence all irrelevant structure caused by bursts is definitely eliminated. Second, even though fine-grained sequence structure is lost in the contained itemset, all items that are relevant in the sequence pattern are preserved. This often provides valuable information when studying causes of severe faults.

We computed the contained itemset size ratios for different severe faults for various combinations of merge thresholds between 1 minute and 1 day and different support thresholds. In all cases, we observed a contained itemset ratio between 89% and 97%. An analysis of result samples revealed that many results looked similar to the example in Table 1, i.e., irrelevant structure was removed. This provides strong evidence, showing that across a variety of parameter settings, for almost all frequent patterns in the original data, at least one approximate version of the pattern was preserved.

7. RELATED WORK

The problem of mining sequential patterns was first introduced by Agrawal and Srikant [2] and is strongly related to mining of association rules [1]. There the Apriori algorithm was first introduced whose monotonicity based pruning inspired many sequence mining algorithms.

Early sequence mining algorithms relied on a breadth-first technique that in theory allows for optimal pruning of the search space, which can dramatically improve performance for large data sets [11, 16, 15]. More recent algorithms use different strategies to traverse the search space. FreeSpan, PrefixSpan and Spam [7, 10, 3] rely on depth-first traversal.

Other research is concerned with the extension of the applicability of existing techniques and with further improved performance of existing methods [6, 4, 14]. Another important research field in sequence mining is to improve the result, e.g., by finding only maximal or closed sequences. Here the main focus is on reduction of the final result set [9, 13, 17]. In PlanMine, sequences are mined that lead to a positive outcome of a plan. The number of sequences can be reduced by removing those that also lead to negative outcomes of a plan. This technique, however, cannot be applied to many datasets because of the lack of such "negative information".

Closed sequence mining [13], approaches used in PlanMine [17], and other existing techniques for mining sequences are orthogonal to our work. Since our transformation produces sequence data, these techniques can be applied to the transformed data as well.

For itemset mining, [18] addresses mining cost for the explosive number of frequent itemset patterns due to the monotonicity of frequency.

8. CONCLUSIONS

We proposed a novel technique that allows efficient mining of bursty sequences. The general idea is to merge items of the same type when they are in a chronologically short distance from each other. The item-intervals generated this

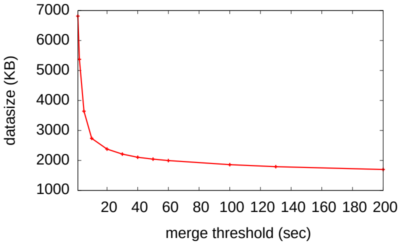

Figure 9: Real Data

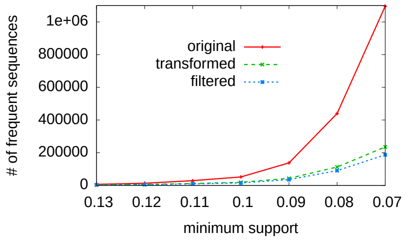

Figure 10: Real Data

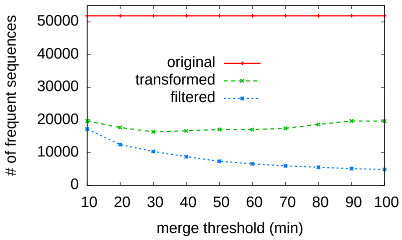

Figure 11: Real Data

way can be transformed back to a sequence database. This allows the use of existing sequence mining algorithms on the transformed data.

We proved that the transformation reduces input sequence length and preserves important structure of individual patterns. We also discussed the problem of fragmentation of support and how it might cause loss of important frequent patterns. Our experiments show that significant reduction in mining time and result size are achieved, while still preserving important frequent patterns. In general fragmentation of support was not a problem in practice, supporting our decision for option 1 (see Section 4.3).

Our work on transforming input sequences raises interesting questions for future work. First, instead of mapping burst intervals back to traditional sequences, it might be more efficient to mine sequences of intervals directly. Second, the problem of mining very long sequences has still many open questions. Transforming the original sequences in different ways can be the answer and our approach is merely a first step in this direction.

9. ACKNOWLEDGEMENTS

The authors would like to thank Tracy Thieret for many insightful discussions and the reviewers for their constructive comments.

This material is based upon work supported by the National Science Foundation under Grant No. 0748626, by the AFOSR under Award No. FA9550-06-1-0111, by the U.S. Department of Homeland Security under Grant Award Number 5-36423.5750, and by a gift from Xerox Corporation. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

10. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[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 *ACM SIGKDD*, pages 429–435, 2002.

[4] A. Demiriz and M. J. Zaki. webSPADE: A parallel sequence mining algorithm to analyze the web log data, 2002.

[5] G. Dong and J. Pei. *Sequence Data Mining (Advances in Database Systems)*. Springer-Verlag New York, 2007.

[6] M. El-Sayed, C. Ruiz, and E. A. Rundensteiner. Fs-miner: Efficient and incremental mining of frequent sequence patterns in web logs. In *Workshop on Web Information and Data Management (WIDM)*, 2004.

[7] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *ACM SIGKDD*, 2000.

[8] J. Kleinberg. Bursty and hierarchical structure in streams. In *ACM SIGKDD*, pages 91–101, 2002.

[9] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, pages 398–416, 1999.

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

[11] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT*, pages 3–17, 1996.

[12] W. Wang and J. Yang. *Mining Sequential Patterns from Large Data Sets*. Springer-Verlag New York, 2005.

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

[14] Z. Yang and M. Kitsuregawa. LAPIN-SPAM: An improved algorithm for mining sequential pattern. In *ICDE Workshops*, 2005.

[15] M. J. Zaki. Sequence mining in categorical domains: Incorporating constraints. In *CIKM*, pages 422–429, 2000.

[16] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1-2):31–60, 2001.

[17] M. J. Zaki, N. Lesh, and M. Ogihara. PlanMine: Sequence mining for plan failures. In *ACM SIGKDD*, pages 369–373, 1998.

[18] F. Zhu, X. Yan, J. Han, P. S. Yu, and H. Cheng. Mining colossal frequent patterns by core pattern fusion. In *ICDE*, 2007.