LASH: Large-Scale Sequence Mining with Hierarchies

Kaustubh Beedkar
Data and Web Science Group
University of Mannheim
Germany
kbeedkar@uni-mannheim.de

Rainer Gemulla
Data and Web Science Group
University of Mannheim
Germany
rgemulla@uni-mannheim.de

ABSTRACT

We propose LASH, a scalable, distributed algorithm for mining sequential patterns in the presence of hierarchies. LASH takes as input a collection of sequences, each composed of items from some application-specific vocabulary. In contrast to traditional approaches to sequence mining, the items in the vocabulary are arranged in a hierarchy: both input sequences and sequential patterns may consist of items from different levels of the hierarchy. Such hierarchies naturally occur in a number of applications including mining natural-language text, customer transactions, error logs, or event sequences. LASH is the first parallel algorithm for mining frequent sequences with hierarchies; it is designed to scale to very large datasets. At its heart, LASH partitions the data using a novel, hierarchy-aware variant of item-based partitioning and subsequently mines each partition independently and in parallel using a customized mining algorithm called pivot sequence miner. LASH is amenable to a MapReduce implementation; we propose effective and efficient algorithms for both the construction and the actual mining of partitions. Our experimental study on large real-world datasets suggest good scalability and run-time efficiency.

1. INTRODUCTION

Sequential pattern mining is a fundamental tool in data mining and has been studied extensively in the literature [8, 20, 23, 26, 31]. Applications include market-basket analysis [26, 27], web usage mining [28], machine translation [19], language modeling [9], or information extraction [21]. In general, the goal of sequence mining is to mine interesting (e.g., frequent) sequential patterns from a potentially large collection of input sequences.

In most of the applications mentioned above, the individual items of the input sequences are naturally arranged in a hierarchy. For example, the individual words in a text document can be arranged in a *syntactic hierarchy*: words (e.g., "lives") generalize to their lemma ("live"), which in turn generalize to their respective part-of-speech tag ("verb"). Products in sequences of customer transactions also form a natural *product hierarchy*, e.g. "Canon EOS 70D" may generalize to "digital camera", which generalizes to "photography", which in turn generalizes to "electronics". As a final example, entities such as persons can be arranged in *semantic hierarchies*; e.g., "Angela Merkel" may generalize to "politician," "person," "entity." Depending on the application, the hierarchy may exhibit different properties; e.g., it may be flat or deep, or it may have low or high fan-out. Hierarchies are sometimes inherent to the application (e.g., hierarchies of directories or web pages) or they are constructed in a manual or automatic way (e.g., product hierarchies).

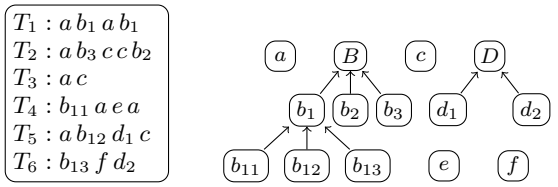
In this paper, we consider a generalized form of frequent sequence mining—which we refer to as *generalized sequence mining* (GSM)—in which the item hierarchies are specifically taken into account. In particular, the items in both input sequences and sequential patterns may belong to different levels of the item hierarchy. This generalization allows us to find sequences that would otherwise be hidden. For example, in the context of text mining, such patterns include generalized n-grams (the ADJ house) or typed relational patterns (PERSON lives in CITY). In both cases, the patterns do not actually occur in the input data, but are useful for language modeling [1, 15, 18, 30] or information extraction tasks [7, 21, 22]. Hierarchies can also be exploited when mining market-basket data [26, 27]—e.g., users may first buy some camera, then some photography book, and finally some flash—or in the context of web-usage mining [13, 17].

The problem of mining frequent sequences with hierarchies has been studied in the literature. A well-known approach [26] to deal with hierarchies is to make use of a mining algorithm that takes as input sequences of *itemsets* (as opposed to sequences of items). The hierarchy is then encoded into itemsets by replacing each item ("lives") by an itemset consisting of the item and its parents ({"lives", "live", "VERB"}); pruning or post-processing techniques are used to output consistent generalized patterns. In practice, the sequence databases can be very large; e.g., consider a document collection with millions of documents or a web site with millions of users. Miliaraki et al. [20] recently proposed a scalable, distributed sequence mining algorithm called MG-FSM, which can handle databases with billions of sequences. However, MG-FSM cannot handle hierarchies. In fact, the problem of how to scale frequent sequence mining with hierarchies to large databases has not been studied in the literature.

In this paper, we propose LASH[1], the first scalable, general-purpose algorithm for mining frequent sequences with hierarchies. LASH is inspired by MG-FSM: it first partitions the data and subsequently mines each partition independently and in parallel. Key ingredients to the scalability of LASH are (i) a novel, hierarchy-aware variant of item-based partitioning, (ii) optimized partition construction techniques, and (iii) an efficient special-purpose algorithm called *pivot sequence miner* (PSM) for mining each partition.

To judge the effectiveness of our methods, we implemented LASH

[1] LArge-scale Sequence mining with Hierarchies



(a) Example database (b) Vocabulary and hierarchy

Figure 1: A sequence database and its vocabulary

using MapReduce and performed an experimental study on large real-world datasets including natural-language text and product sequences. Our results suggest that LASH has good scalability and run-time efficiency. LASH also supports sequence mining without hierarchies and was the best-performing method for this task in our experiments.

The remainder of this paper is organized as follows. In Sec. 2, we formally define the problem of generalized sequence mining. In Sec. 3, we give an overview of LASH and alternative baseline algorithms. In Sec. 4, we describe the partition construction step of LASH in more detail. Algorithms for mining each partition are discussed in Sec. 5. Sec. 6 describes our experimental study and results. We discuss related work in Sec. 7 and conclude in Sec. 8.

2. PRELIMINARIES

We start with a formal definition of the GSM problem and related concepts. Our notation and terminology closely follows standards in the sequence mining literature and deviates only when necessary.

Denote by *sequence database* $\mathcal{D} = \{ T_1, T_2, \cdots, T_{|D|} \}$ a multiset of sequences, where each *sequence* $T = t_1 t_2 \cdots t_n$ is composed of items from a *vocabulary* $\mathcal{W} = \{ w_1, w_2, \cdots, w_{|\mathcal{W}|} \}$. We denote by $|T|$ the length of sequence T and by $\mathcal{W}^+$ the set of all non-empty sequences that can be formed from items in $\mathcal{W}$. An example sequence database consisting of six sequences is shown in Figure 1(a).

Hierarchies. In GSM, the vocabulary is arranged in a hierarchy, i.e., each item has zero or more children and at most one parent.[2] Fig. 1(b) shows a simple example hierarchy that we will use throughout this paper. If an item v is an ancestor of some item u, we say that v is "more general" than u; e.g., in our example hierarchy, B is more general than b_1. We distinguish *leaf items* (most specific, no descendants), *root items* (most general, no ancestors), and *intermediate items*. In a hierarchy about music, for example, the song "Blue Monday" may be a leaf item, its parent "pop song" an intermediate item, which in turn may have as parent the root item "music". For two items $u, v \in \mathcal{W}$, we say that u *directly generalizes* to v if u is more specialized than v; i.e., if u is a child of v in the hierarchy, which we denote by $u \rightarrow v$. Denote by $\rightarrow^*$ the symmetric transitive closure of $\rightarrow$. In our example, we have $b_{11} \rightarrow b_1$, $b_1 \rightarrow B$ and consequently $b_{11} \rightarrow^* B$.

Generalized sequences. We extend relation $\rightarrow$ to sequences in a natural way. In particular, we say that sequence $T = t_1 \cdots t_n$ directly generalizes to sequence $S = s_1 \cdots s_{n'}$, denoted $T \rightarrow S$, if $n = n'$ and there exists an index $j \in [n]$ such that $t_j \rightarrow s_j$ and $t_i = s_i$ for $i \neq j$. In our example, sequence $T_1 = ab_1ab_1$ satisfies $T_1 \rightarrow aBab_1$, $T_1 \rightarrow ab_1aB$, and $T_1 \rightarrow^* aBaB$ (the most general form of T_1). Note that we do not place any limitation on the set of items that occur in database $\mathcal{D}$; each input sequence may be composed of items from arbitrary levels of the hierarchy.

Subsequences. GSM ultimately aims to mine frequent generalizations of subsequences of the sequences in $\mathcal{D}$. In practice, it is often useful to focus on subsequences that are contiguous or "close". For example, n-gram mining aims to find consecutive subsequences in text. Similarly, when mining web usage data or any form of log files, sequences of items that are close may be more insightful than sequences of far-away items. We model the notion of closeness with a *gap constraint* $\gamma \geq 0$ and consider two items as close if at most γ items occur in between. Formally, a sequence $S = s_1 s_2 \cdots s_n$ is said to be a *subsequence* of $T = t_1 t_2 \cdots t_m$, denoted $S \subseteq_\gamma T$, if there exists integers $1 \leq i_1 < \cdots < i_n \leq m$ such that $s_j = t_{i_j}$ and $0 \leq i_{j+1} - i_j - 1 \leq \gamma$. We write $S \subseteq T$ if S is an unconstrained subsequence of T, i.e., $S \subseteq_\infty T$. For sequence $T_5 = ab_{12}d_1c$ of our example database, we have $a \subset_0 T_5$, $ab_{12} \subset_0 T_5$ and $ad_1c \subset_1 T_5$, but $b_{12}a \not\subset T_5$ and $ad_1c \not\subset_0 T_5$.

Generalized subsequences. Combining generalizations and subsequences, we say that $S = s_1 s_2 \cdots s_n$ is a *generalized subsequence* of $T = t_1 t_2 \cdots t_m$, denoted $S \sqsubseteq_\gamma T$, if there exists integers $1 \leq i_1 \leq \cdots \leq i_n \leq m$ such that $t_{i_j} \rightarrow^* s_j$ (recall that $t_{i_j} \rightarrow^* s_j$ includes the case $t_{i_j} = s_j$) and $0 \leq i_{j+1} - i_j - 1 \leq \gamma$. For example, we have $ad_1 \sqsubseteq_1 T_5$ and $aD \sqsubseteq_1 T_5$ (even though D does not occur in T_5). Note that if S is a subsequence of T, then S is also a generalized subsequence of T; the opposite may or may not hold.

Support. Denote by

$$\mathrm{Sup}_\gamma(S, \mathcal{D}) = \{ T \in \mathcal{D} : S \sqsubseteq_\gamma T \},$$

the *support set* of sequence S in the database $\mathcal{D}$, i.e., the multiset of input sequences in which S occurs directly or in specialized form. In our example database, we have $\mathrm{Sup}_0(aBc, \mathcal{D}) = \{ T_2 \}$ and $\mathrm{Sup}_1(aBc, \mathcal{D}) = \{ T_2, T_5 \}$. Denote by $f_\gamma(S, \mathcal{D}) = |\mathrm{Sup}_\gamma(S, \mathcal{D})|$ the *frequency* (or *support*) of S; e.g., $f_0(aBc, \mathcal{D}) = 1$ and $f_1(aBc, \mathcal{D}) = 2$. We say that sequence S is *frequent* in $\mathcal{D}$ if its frequency passes a *support threshold* $\sigma > 0$, i.e., $f_\gamma(S, \mathcal{D}) \geq \sigma$.

Problem statement. We are now ready to formally define the GSM problem:

> Denote by $\sigma > 0$ a minimum support threshold, by $\gamma \geq 0$ a maximum-gap constraint, and by $\lambda \geq 2$ a maximum-length constraint. The GSM problem is to find all frequent generalized sequences S, $2 \leq |S| \leq \lambda$, along with their frequencies $f_\gamma(S, \mathcal{D})$ ($\geq \sigma$).

Note that we exclude frequent items in our problem statement; these items can easily be determined (and are, in fact, also determined by our LASH algorithm). In our ongoing example and for $\sigma = 2$, $\gamma = 1$ and $\lambda = 3$, we obtain (sequence, frequency)-pairs: $(aa, 2)$, $(ab_1, 2)$, $(b_1a, 2)$, $(aB, 3)$, $(Ba, 2)$, $(aBc, 2)$, $(Bc, 2)$, $(ac, 2)$, $(b_1D, 2)$, and $(BD, 2)$. Observe that b_1D is frequent even though it does not occur in the database and none of its specializations are frequent. Thus GSM can detect non-obvious patterns in the data.

Discussion. The GSM problem as stated above asks for *all* sequences that frequently occur (directly or in specialized form) in the database. Depending on the dataset, the set of frequent sequences can be very large and partly redundant. In the example above, for instance, the fact that b_1D is frequent implies that BD must also be frequent. In this case, the frequencies match; in general, they can be different (e.g., aB has higher frequency ab_1). The potentially large number of output sequences is acceptable for applications that focus on exploration (like the Google n-gram viewer [1] or Netspeak [2]) or use frequent sequences as input to

[2] In this paper, we assume that the item hierarchy forms a forest. In some applications, this assumption may be violated and the hierarchy may instead form a directed acyclic graph; our methods can be extended to deal with such hierarchies as well.

further automated tasks (e.g., as features in a learning system). In some applications, the set of output sequences needs to be further restricted (e.g., using maximality or closedness constraints); we do not consider such restrictions in this paper.

3. DISTRIBUTED MINING

In what follows, we first discuss a set of baseline algorithms for solving the GSM problem in a distributed fashion and describe their advantages and drawbacks. We then propose LASH, a scalable distributed algorithm that alleviates the drawbacks of the baseline approaches.

3.1 MapReduce

All algorithms are described in terms of a MapReduce framework [11], but can also be implemented using other parallel systems. MapReduce is a popular framework for distributed data processing on clusters of commodity hardware. It operates on key-value pairs and allows programmers to express their problem in terms of a *map* and a *reduce* function. Key-value pairs emitted by the map function are partitioned by key, sorted, and fed into the reduce function. An additional *combine* function can be used to pre-aggregate the output of the map function and increase efficiency. The MapReduce runtime takes care of execution and transparently handles failures in the cluster. While originally proprietary, open-source implementations of MapReduce, most notably Apache Hadoop, are available and have gained wide-spread adoption.

3.2 Naïve Approach

A naïve approach to GSM is to first generate each generalized subsequence of each input sequence and to subsequently count the global frequency of each such subsequence. This approach can be implemented in MapReduce in a way similar to "word counting". In more detail, denote by

$$G_\lambda(T) = \{\, S \mid S \sqsubseteq_\gamma T, 1 < |S| \leq \lambda \,\}$$

the set of generalized subsequences of T that match the length and gap constraints (we write $G(T)$ when λ is clear from the context). For example, for transaction $T_4 = b_{11}aea$ and $\gamma = 1$ and $\lambda = 3$, we obtain

$$G_3(T_4) = \{\, b_{11}a, b_{11}e, ae, aa, ea, b_{11}ae, b_{11}aa, b_{11}ea, aea,\\ b_1a, b_1e, b_1ae, b_1aa, b_1ea, Ba, Be, Bae, Baa, Bea \,\},$$

where the first line lists subsequences and the second line their generalizations. To implement the naïve approach in MapReduce, we map over input sequences and, for each input sequence T, we output each element $S \in G(T)$ (as key). In the reduce function, we count for each generalized subsequence S how often it occurred in the data and output S if $f_\gamma(S) \geq \sigma$.

The key advantage of the naïve algorithm is its simplicity. The key disadvantage, however, is that it creates excessive amounts of intermediate data and is thus generally inefficient (cf. $G_3(T_4)$ above). Denote by δ the maximum depth of the item hierarchy and set $l = |T|$. For $\gamma = 0$, naïve outputs $O(l\delta^\lambda)$ generalized subsequences per input sequence, i.e., it is exponential in λ and polynomial in δ; this number is infeasibly large in all but the most simple cases. When $\gamma, \lambda \geq l$, the situation becomes even more severe and naïve outputs $O((\delta+1)^l)$ generalized subsequences per input sequence.

3.3 Semi-Naïve Approach

To reduce the number of subsequences generated by the naïve approach, we can make use of item frequencies to prune the set $G(T)$ of generated subsequences. We refer to this improvement as the semi-naïve approach.

The semi-naïve approach makes use of a *generalized f-list*, which contains each frequent item w along with its frequency $f_0(w, \mathcal{D})$. Note that the generalized f-list is hierarchy-aware, i.e., the frequency of each item $w \in \mathcal{W}$ is given by the number of input sequences that contain w or any of its descendants. In other words, item w is *frequent* if $f_0(w, \mathcal{D}) \geq \sigma$; otherwise w is *infrequent*. For our example database and $\sigma = 2$, the generalized f-list is shown in the top-left corner of Fig. 2; it is also used by our LASH algorithm.

The generalized f-list can be computed efficiently in a single MapReduce job. Denote by $G_1(T) = \{\, w' \mid w \in T, w \rightarrow^* w' \,\}$ the set of items that appear in T along with their generalizations. For example,

$$G_1(T_4) = \{\, b_{11}, a, e, a, b_1, B \,\}.$$

Note that $G_1(T)$ has size $O(l\delta)$, where as before $l = |T|$, and is thus linear in l and δ. To obtain the generalized f-list, we map over each $T \in \mathcal{D}$ and output each item in $G_1(T)$ along with an associated frequency of 1. The reducer sums up the frequencies for each item w to obtain $f_0(w, \mathcal{D})$.

The semi-naïve algorithm computes the set of frequent generalized sequences in a second MapReduce job. It uses the generalized f-list to reduce the number of generalized subsequences emitted by the map function of the naïve algorithm; the reduce function remains unmodified and counts frequencies. The semi-naïve algorithm outputs only the subsequences $S \in G(T)$ of input sequence T that do not contain any infrequent item (see below). For example, the semi-naïve algorithms emits for transaction $T_4 = b_{11}aea$, $\gamma = 1$, and $\lambda = 3$ the generalized subsequences

$$aa, b_1a, b_1aa, Ba, Baa.$$

Compared to the set $G_3(T_4)$ output by the naïve algorithm, the output size is reduced by a factor of more than 3.

The correctness of the semi-naïve algorithm stems from the following lemma, which implies that frequent sequences cannot contain infrequent items:

LEMMA 1. (SUPPORT MONOTONICITY). *For any pair of generalized sequences S_1 and S_2 such that $S_1 \sqsubseteq_\gamma S_2$, we have $\mathrm{Sup}_\gamma(S_1, \mathcal{D}) \supseteq \mathrm{Sup}_\gamma(S_2)$ and consequently $f_\gamma(S_1, \mathcal{D}) \geq f_\gamma(S_2, \mathcal{D})$.*

This lemma is a straightforward generalization of the Apriori principle [6]; we omit the proof here.

The map phase (of the second job) can be implemented efficiently by first generalizing each item of T to its closest frequent ancestor (if existent). If an item has no frequent ancestor, we replace it by a special *blank symbol*, denoted by "␣". For example, for $T_4 = b_{11}aea$ and $\sigma = 2$ (see Fig. 2), we obtain $T_4' = b_1a␣a$; here a is frequent, b_{11} is infrequent but has frequent parent b_1, and e is infrequent and has no frequent ancestor. We then enumerate and emit all sequences in $G_\lambda(T_4')$ that do not contain a blank symbol. As will become evident later, the generalization of infrequent items is a concept that we also make use of in LASH (although in a slightly different way).

The semi-naïve algorithm is more efficient than the naïve algorithm if many items are infrequent; i.e., when σ is set to a high value. In the worst case, however, all items are frequent and the semi-naïve algorithm reduces to the naïve algorithm (with the additional overhead of computing of the generalized f-list).

3.4 Overview of LASH

The key idea of our LASH algorithm is to partition the set of sequential patterns using a *hierarchy-aware variant* of item-based

Algorithm 1 Partitioning and mining phase of LASH

Require: $\mathcal{D}, \mathcal{W}, \rightarrow, \sigma, \gamma, \lambda$
1: MAP(T)
2: **for all** $w \in G_1(T)$ with $\text{Sup}(w, \mathcal{D}) \geq \sigma$ **do**
3: Construct $\mathcal{P}_w(T)$ *// Sec. 4*
4: Emit $(w, \mathcal{P}_w(T))$
5: **end for**
6:
7: REDUCE($w, \mathcal{P}_w$)
8: Compute the set $G_{\sigma,\gamma,\lambda}(w, \mathcal{P}_w)$ of the locally-frequent pivot sequences *// Sec. 5*
9: **for all** $S \in G_{w,\sigma,\gamma,\lambda}(\mathcal{P}_w)$ **do**
10: Emit $(S, f_\gamma(S, \mathcal{P}_w))$
11: **end for**

partitioning. Item-based partitioning is a well-known concept in pattern mining; it is used, for example, in the FP-growth algorithm [12] for frequent itemset mining. LASH is inspired by the MG-FSM algorithm [20], which uses item-based partitioning to obtain a scalable sequence mining algorithm. In contrast to MG-FSM, LASH supports hierarchies and exploits them whenever possible.

LASH creates a partition $\mathcal{P}_w$ for every frequent item $w \in \mathcal{W}$ and then mines frequent sequences in each partition independently. We subsequently refer to item w as the *pivot item* of partition $\mathcal{P}_w$. LASH is divided into a preprocessing phase, a partitioning phase, and a mining phase.

Preprocessing. In the preprocessing phase, LASH computes the item frequencies to obtain a generalized f-list (as in Sec. 3.3) and a total order $<$ on $\mathcal{W}$. The total order determines the partitioning used in the later phases; frequent items will be "small". In particular, for any pair of items $w_1, w_2 \in \mathcal{W}$, we set $w_1 < w_2$ if $f_0(w_1, \mathcal{D}) > f_0(w_2, \mathcal{D})$. Ties are handled in a hierarchy-aware form: if $f_0(w_1, \mathcal{D}) = f_0(w_2, \mathcal{D})$ and w_1 occurs at a higher level of the item hierarchy, we set $w_1 < w_2$; the remaining ties are broken arbitrarily. This particular order ensures that $w_2 \rightarrow w_1$ implies $w_1 < w_2$. Fig. 2 shows the generalized f-list of our example database for $\sigma = 2$. Here items are ordered from small to large; i.e., we have $a < B < b_1 < c < D$. Note that item frequencies and total order can be reused when LASH is run with different parameters (only the generalized f-list needs to be adapted).

Partitioning and mining phase. The partitioning and mining phases of LASH are performed in a single MapReduce job as outlined in Alg. 1. LASH generates a partition $\mathcal{P}_w$ for each frequent item w; in our running example, the five partitions $\mathcal{P}_a$, $\mathcal{P}_B$, $\mathcal{P}_{b_1}$, $\mathcal{P}_c$, and $\mathcal{P}_D$ are created. From partition $\mathcal{P}_w$, we mine all generalized sequences that contain w but no larger item (according to $<$). For example, from partition $\mathcal{P}_a$, we mine sequences that consists of a only (aa, aaa, ...) and from partition $\mathcal{P}_B$ sequences that contain B and optionally a (BB, aB, ...). More formally, denote by $p(S) = \max_{w \in S}(S)$ the *pivot item* of sequence S; e.g., we have $p(aBcB) = c$ with our example order. If S is frequent, it will be mined from partition $\mathcal{P}_{p(S)}$ (and no other partition).

The partitioning phase is carried out in the map function, which as before maps over each input sequence T. For each frequent item $w \in G_1(T)$, we construct a "rewritten" sequence $\mathcal{P}_w(T)$ and output it with reduce key w. Note that if w is frequent and one of its descendants occurs in T, we create $\mathcal{P}_w(T)$ even if $w \notin T$. A simple and correct approach to compute $\mathcal{P}_w(T)$ is to set $\mathcal{P}_w(T) = T$. A key ingredient of LASH is to use rewrites that compress T as much as possible while maintaining correctness; we discuss such rewrites in Sec. 4.

The mining phase is carried out in the reduce function. The MapReduce framework automatically constructs partitions

$$\mathcal{P}_w = \biguplus_{T \in \mathcal{D}} \{ \mathcal{P}_w(T) \}.$$

Each reduce function then runs a customized GSM algorithm on its partition $\mathcal{P}_w$; partitions are processed independently and in parallel. The GSM algorithm is provided with the parameters w, σ, γ, and λ and produces the set $G_{\sigma,\gamma,\lambda}(w, \mathcal{P}_w)$ of *locally-frequent pivot sequences* such that, for each $S \in G_{\sigma,\gamma,\lambda}(w, \mathcal{P}_w)$, S is frequent, $p(S) = w$ and $2 \leq |S| \leq \lambda$. This local mining step can be performed using an arbitrary GSM algorithm (which produces a superset of $G_{\sigma,\gamma,\lambda}(w, \mathcal{P}_w)$) followed by a filtering step. As we will see in Sec. 5, LASH proceeds more efficiently by using PSM, a special-purpose miner that directly produces $G_{\sigma,\gamma,\lambda}(w, \mathcal{P}_w)$.

Discussion. The key difference between LASH and the naïve and semi-naïve algorithm is the use of item-based partitioning (LASH) versus the use of sequence partitioning (naïve and semi-naïve). The advantage of item-based partitioning is that the amount of data communicated from map to the reduce phase can be significantly lowered by the use of good rewrite techniques. Moreover, the reduce functions can directly leverage state-of-the-art sequential GSM algorithms; we discuss such algorithms in Sec. 5.

4. PARTITION CONSTRUCTION

We now discuss partition construction and, in particular, our rewrite techniques in more detail. As stated above, a simple way to construct $\mathcal{P}_w(T)$ is to set $\mathcal{P}_w(T) = T$. For our example database ($\sigma = 2$), we obtain for pivot B the partition

$$\mathcal{P}_B = \{ \, a\, b_1\, a\, b_1,\ a\, b_3\, c\, c\, b_2,\ b_{11}\, a\, e\, a,\ a\, b_{12}\, d_1\, c,\ b_{13}\, f\, d_2 \, \} \quad (1)$$

Using such a partitioning strategy is inefficient due to the following reasons: (1) *skew*: partitions of highly frequent items will contain many more sequences than partitions of less frequent items, (2) *redundant computation*: a large number of duplicate sequences are mined at multiple partitions (e.g., sequence aBc will be mined in partitions $\mathcal{P}_a$, $\mathcal{P}_B$, $\mathcal{P}_{b_1}$ and $\mathcal{P}_c$ but output only in partition $\mathcal{P}_c$), and (3) *high communication cost*: each input sequence T is replicated $|G_1(T)|$ times, which results in substantial communication cost.

In what follows, we propose rewrite techniques for constructing $\mathcal{P}_w(T)$ with the aim to overcome the above mentioned shortcomings. We refer to these rewrites as *reductions* (since they ultimately reduce the length of T).

4.1 Generalized w-Equivalency

We first establish the notion of *generalized w-equivalency*, which is an important criterion for the correctness of LASH. In particular, LASH is guaranteed to produce correct results if for all frequent items w, partition $\mathcal{P}_w$ and database $\mathcal{D}$ are w-equivalent.

Extending our running notation, denote by

$$G_{w,\lambda}(T) = \{ \, S \mid S \sqsubseteq_\gamma T, 2 \leq |S| \leq \lambda, p(S) = w \, \} \quad (2)$$

the set of generalized subsequences S of T that (1) satisfy the length and gap constraints and (2) have pivot item w. Note that we suppress the dependence of $G_{w,\lambda}(T)$ on γ for brevity. We refer to each sequence in $G_{w,\lambda}(T)$ as *pivot sequence*. For our example and for $\sigma = 2$ and $\gamma = 1$ (which we use from now on), we obtain

$$G_{b_1,2}(T_1) = \{ \, ab_1, b_1a, b_1b_1, b_1B, Bb_1 \, \}. \quad (3)$$

Note that $BB \notin G_{b_1,2}(T_1)$ since each pivot sequence must contain at least one pivot (and $p(BB) = B \neq b_1$).

We say that two sequences T and T' are w-equivalent, if

$$G_{w,\lambda}(T) = G_{w,\lambda}(T'),$$

i.e., they both generate the same set of pivot sequences. For example,

$$G_{B,2}(T_2) = G_{B,2}(a\,b_3\,c\,c\,b_1) = \{\,aB\,\} = G_{B,2}(aB).$$

LASH produces correct results if $\mathcal{P}_w(T)$ is w-equivalent to T. To see this, denote by

$$G_{w,\lambda}(\mathcal{D}) = \biguplus_{T\in\mathcal{D}} G_{w,\lambda}(T)$$

the multiset of pivot sequences generated from $\mathcal{D}$. Now observe that if $\mathcal{P}_w(T)$ is w-equivalent to T, then $G_{w,\lambda}(\mathcal{D}) = G_{w,\lambda}(\mathcal{P}_w)$; we then say that databases $\mathcal{D}$ and $\mathcal{P}_w$ are w-equivalent. Both databases then agree on the multiset of pivot sequences and, consequently, on their frequencies. Thus for every S with $p(S) = w$ and $2 \leq |S| \leq \lambda$, we have $f_\gamma(S,\mathcal{D}) = f_\gamma(S,\mathcal{P}_w)$. Since these are precisely the sequences that LASH mines and retains from $\mathcal{P}_w$ in the mining phase, correctness follows. Note that two databases can be w-equivalent but disagree on a frequency of any non-pivot sequence; e.g., $\mathcal{D}$ and $\mathcal{P}_B$ may be B-equivalent but disagree on the frequency of B itself (5 versus 4 in our example). In particular, the frequency of any non-pivot sequence can be equal, lower, or higher in $\mathcal{D}$ than in $\mathcal{P}_B$ without affecting correctness. The above discussion leads to the following lemma:

LEMMA 2. *If $\mathcal{D}$ and $\mathcal{P}_w$ are w-equivalent w.r.t. λ and γ, then $f_\gamma(S,\mathcal{D}) = f_\gamma(S,\mathcal{P}_w)$ for all S satisfying $p(S) = w$ and $2 \leq |S| \leq \lambda$.*

Our notion of w-equivalency is a generalization of the corresponding notion of [20], which also gives a more formal treatment and proof of correctness. The key difference in LASH is the correct treatment of hierarchies.

4.2 w-Generalization

From the discussion above, we conclude that we can rewrite each input sequence T into any w-equivalent sequence $T' = \mathcal{P}_w(T)$ in Line 3 of Alg. 1. Our main goal is to make T' as small as possible; this decreases both communication cost, computational cost, and (as will become evident later) skew.

Fix some pivot w and let $T = t_1 t_2 \ldots t_l$. The first and perhaps most important of our rewrites is called w-generalization, which tries to rewrite T such that only "relevant" items remain. In particular, we say that an item is *w-relevant* if $w' \leq w$; otherwise it is w-irrelevant. Similarly, index i is w-relevant if and only if t_i is w-relevant. For example, in sequence $T_2 = ab_3ccb_2$ only index 1 is B-relevant.

The key insight of w-generalization is that any generalized subsequence of T that contains an irrelevant item cannot be a pivot sequence (since the pivot is smaller than any irrelevant item by definition). Ideally, we would like to simply drop all irrelevant items from T; unfortunately, such an approach may lead to incorrect results since (1) if we drop irrelevant items, we cannot guarantee that the gap constraint remains satisfied and (2) generalizations of irrelevant items may be relevant and thus be part of a pivot sequence. To illustrate the violation of the gap constraint, suppose that we dropped cc from $T_2 = ab_3ccb_2$ to obtain $T_2' = ab_3b_2$. Then $BB \in G_{B,2}(T_2')$ but $BB \notin G_{B,2}(T_2')$ for $\gamma = 1$. To illustrate the second point, suppose that we drop all irrelevant items from T_2 to obtain a. We then miss pivot sequence $aB \sqsubseteq_1 T_2$ since $aB \not\sqsubseteq_1 a$.

Instead of dropping irrelevant items, w-generalization replaces irrelevant items by other items. There are two cases: (1) If index i is irrelevant and item t_i does not have an ancestor $w' < w$, we replace t_i by the special blank symbol $\textvisiblespace$, where $w < \textvisiblespace$ for all $w \in \mathcal{W}$. The blank symbol acts as a placeholder and is needed to handle gap constraints. (2) Index i is irrelevant but has an ancestor that is smaller than the pivot. Let w' be the largest such ancestor. We then replace t_i by w'. This step is similar to the generalization performed by the semi-naïve algorithm. It is more effective, however, since it generalizes all items that are less frequent than the pivot, whereas the semi-naïve algorithm only generalizes infrequent items before applying the naïve algorithm. Continuing our example with $T_2 = ab_3ccb_2$ with pivot B, indexes 3 and 4 are irrelevant and replaced by blanks (since c does not have an ancestor that is more frequent than B), whereas indexes 2 and 5 are irrelevant and replaced by B (the largest sufficiently frequent ancestor of both b_3 and b_2). We thus obtain $T_2' = aB\textvisiblespace\textvisiblespace B$.

At first glance, it seems as if w-generalization does not help: T_2 and T_2' have exactly the same length. However, we argue that the use of T_2' leads to substantially lower cost. First, we can represent blanks more compactly than irrelevant items; e.g., by using run-length encoding ($T_2' = aB\textvisiblespace^2 B$) and/or variable-length encoding (few bits for blanks). Second, for similar reasons, we can represent smaller, generalized items more compactly than large items. Third, w-generalization enables the use of other effective rewrite techniques; see Sec. 4.3. Finally, w-generalization (as well as some of the other rewrites) makes sequences more uniform. If two sequences agree on their w-generalization, they can be "aggregated"; see the discussion in Sec. 4.4.

The correctness of w-generalization is captured in the following lemma.

LEMMA 3. *Let $T = t_1 t_2 \cdots t_n$ and denote by $T' = t_1' t_2' \cdots t_n'$ the w-generalization of T. Then T and T' are w-equivalent.*

PROOF. We have to show that $G_{w,\lambda}(T) = G_{w,\lambda}(T')$. Let $S = s_1 \ldots s_k \in G_{w,\lambda}(T)$. By definition, sequence S is a generalized subsequence of T, $p(S) = w$, and $s_j \leq w$ for $1 \leq j \leq k$. Thus there exists a set of indexes $1 \leq i_1 < \ldots < i_k \leq n$ such that $t_{i_j} \rightarrow^* s_j$ and $i_{j+1} - i_j - 1 \leq \gamma$. We claim that $t'_{i_j} \rightarrow^* s_j$ so that $S \in G_{w,\lambda}(T')$. There are two cases. If $t_{i_j} \leq w$, w-generalization does not modify index t_j so that $t'_{i_j} = t_j \rightarrow^* t_j$. Otherwise, if $t_{i_j} > w$, w-generalization replaces t_{i_j} by the largest ancestor t'_{i_j} that is smaller then the pivot. Since $t_{i_j} \rightarrow^* s_j$ and $s_j \leq w$, we conclude that $t_{i_j} \rightarrow^* t'_{i_j} \rightarrow^* s_j$ holds as well. Putting everything together, we obtain $G_{w,\lambda}(T') \subseteq G_{w,\lambda}(T')$.

It remains to show that $G_{w,\lambda}(T') \supseteq G_{w,\lambda}(T)$. This can be shown using the property that whenever $S \in G_{w,\lambda}(T')$, then $\textvisiblespace \notin S$. The proof is similar to the one above and omitted here. □

4.3 Other Rewrites

LASH performs a number of additional rewrites, all of which aim to reduce the length of the sequence. In contrast to w-generalization, these rewrites closely resemble the rewrites of MG-FSM [20]; we summarize them here and point out minor differences.

The first rewrite removes items that are unreachable in that they are "far away" from a *pivot index*. Let $T = t_1 t_2 \cdots t_l$. In what follows, we assume that T has already been w-generalized; then t_i is a pivot index if and only if $t_i = w$. In practice, w-generalization and unreachability reduction are performed jointly and are thus slightly more complex. Consider for example the sequence $T = ab_1acd_1ad_2cfb_2c$, pivot D, the hierarchy of Fig. 1(b), and the item order of Fig. 2. We obtain $T' = ab_1acDaDc\textvisiblespace Bc$ by D-generalization; thus indexes 5 and 7 are pivot indexes. We then compute the left and right distances to a pivot, as well as the minimum distance. The left/right distance of an index is the size of the minimum set of increasing/decreasing indexes from a pivot index to the target index; only indexes that do not correspond to a blank

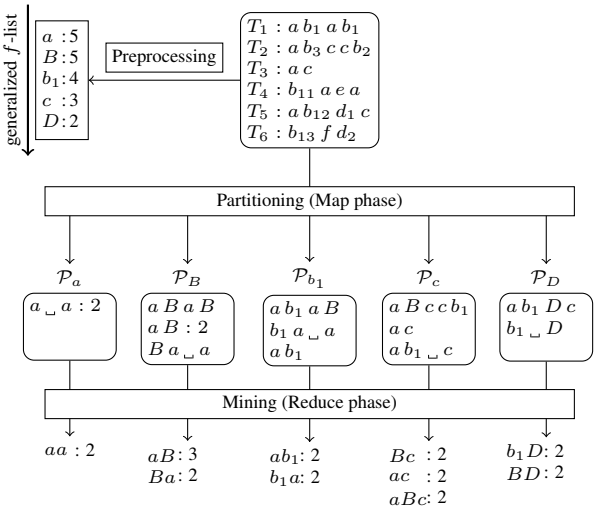


Figure 2: Preprocessing, partitioning and mining phases of LASH for $\sigma = 2$, $\gamma = 1$ and $\lambda = 3$.

as well as the target index are allowed and subsequent indexes must satisfy the gap constraint (at most γ items in between). For $\gamma = 1$, we obtain:

i	1	2	3	4	**5**	6	**7**	8	9	10	11
t'_i	a	b_1	a	c	D	a	D	c	␣	B	c
left	-	-	-	-	1	2	1	2	2	3	4
right	3	3	2	2	1	2	1	-	-	-	-
minimum	3	3	2	2	1	2	1	2	2	3	4

Here "-" corresponds to infinite distance. The left pivot distance of index 11, for example, is determined by the index sequence 7, 8, 10, 11 (length 4); the sequence 7, 9, 11 is not allowed since index 9 corresponds to a blank. As argued in [20], indexes that have distance larger than λ are unreachable and the corresponding items can be removed safely. For $\lambda = 2$, we obtain the reduced sequence $acDaDc_␣$; for $\lambda = 3$, we obtain $ab_1acDaDc_␣B$.

We also make use of a few other reductions of MG-FSM, which also apply to our generalized setting. First, we remove isolated pivot items, i.e., pivot items that do not have a non-blank item close by (within distance γ). We also remove leading and trailing blanks and replace any sequence of more than $\gamma + 1$ blanks by exactly $\gamma + 1$ blanks.

4.4 Putting Everything Together

We perform the above mentioned rewrites efficiently as follows. We first scan the sequence from right to left and, for each index, perform w-generalization and compute its left distance. We then scan the sequence from left to right, compute the right and pivot distance of each index, remove unreachable indexes, and remove blanks as described above. The computational complexity for rewriting an input sequence of length l given a pivot is $O(l)$. Since an input sequence has at most δl pivot items, the overall computational complexity is $O(\delta l^2)$. Moreover, we output $O(\delta l)$ rewritten sequences of length at most l for all choices of γ and λ. Thus the communication complexity of LASH is polynomial, whereas the communication complexity of the naïve and semi-naïve approaches can be exponential ($O((\delta + 1)^l)$). Moreover, our experiments suggest that LASH often performs much better than what could be expected from the above worst-case analysis.

The partitions generated by LASH for our example database are given in Fig. 2. Recall the partition $\mathcal{P}_B$ from Eq. (1). Using our rewrites, we obtain

$$\mathcal{P}_B = \{\, a\,B\,a\,B,\ a\,B,\ B\,a\,␣\,a,\ a\,B \,\}$$

which is significantly smaller. Observe that sequence aB occurs twice. We use combine functionality of Hadoop to aggregate such duplicated sequences. We also perform aggregation in the reduce function before starting the actual mining phase. Continuing the example, the final partition $\mathcal{P}_B$ is given by

$$\mathcal{P}_B = \{\, a\,B\,a\,B : 1,\ a\,B : 2,\ B\,a\,␣\,a : 1 \,\}.$$

Aggregation of duplicated sequences saves communication cost and reduces the computational cost of the GSM algorithm run in the mining phase.

5. SEQUENTIAL MINING

We now discuss methods to mine each partition locally. We first describe how existing sequential algorithms can be adapted to handle hierarchies efficiently. These approaches mine all locally-frequent sequences and must be combined with a filtering step to restrict output to pivot sequences. To avoid this overhead, we propose a more efficient, special-purpose sequence miner that mines pivot sequences directly; we refer to this algorithm as pivot sequence miner (PSM).

5.1 Sequential GSM Algorithms

Most existing sequence miners make use of either breadth-first search (BFS) or depth-first search (DFS). We briefly describe how to extend both search techniques with hierarchies.

BFS with hierarchies. Methods based on BFS use a level-wise approach, i.e., they first compute sequences of length 1, then length 2, and so on. Here we describe how to extend SPADE [31] to mine generalized sequences. SPADE employs the *candidate-generation-and-test* framework of [26]: It makes use of the frequent sequences of length l to generate candidate sequences of length $l+1$ that are potentially frequent, and then determines which of these candidates are actually frequent. The latter step is performed efficiently by making of vertical representation of the sequence database, i.e., an inverted index which maps each length-l sequence S to a *posting list* consisting of the set of input sequences in which S occurs as well as the corresponding positions. To obtain the frequency of a sequence of length $l + 1$, SPADE intersects the posting lists of its length-l prefix and suffix. We adapt SPADE as follows: We first scan each sequence $T \in \mathcal{P}_w$ to create a posting list for each frequent length-2 sequence. In particular, we add sequence T to the posting list of each element of $S \in G_2(T)$. Note that $G_2(T)$ consists of the 2-sequences that occur in S as well as all of their generalizations; this makes our approach hierarchy-aware. Consider for example input sequence $T = cab_1D$, the hierarchy of Fig. 1(b), and $\gamma = 1$. Then $G_2(T) = \{\, ca, cb_1, cB, ab_1, aB, aD, b_1D, BD \,\}$ so that we add T to 8 posting lists. The construction of the 2-sequence index is the only difference to SPADE, i.e., we now proceed with SPADE's level-wise approach unmodified. For example, when sequence ca and aD are frequent, we generate candidate sequence caD and obtain its frequency by intersecting the posting lists of ca and aD.

DFS with hierarchies. An alternative to BFS is to use a DFS approach such as the *pattern-growth* framework of the PrefixSpan [23]. Pattern-growth approaches start with the empty sequence and recursively expand a frequent sequence S (of length l) to generate all frequent sequences with prefix S (of length $l + 1$). To make this expansion step efficient, PrefixSpan maintains a projected database, which consists of the set of input sequences in

which S occurs. In each expansion step, PrefixSpan scans the projected database to determine the frequent items occurring to the right of S; we refer to this step as a *right-expansion* (RE). To adapt PrefixSpan to mine generalized sequences, we replace the projected database by the support set $\mathcal{D}_s$, which consists all input sequences in which S *or a specialization of* S occurs. When we right-expand S, the set of right items for transaction $T \in \mathcal{D}_s$ is given by $\mathcal{W}_S^{\text{right}}(T) = \{ w' \mid Sw' \sqsubseteq_\gamma T \}$, i.e., we look for occurrences of S or a specialization of S, and then consider the items to the right along with their generalizations. For our example sequence $T = cab_1D$ with $\gamma = 1$, we have $\mathcal{W}_{ca}^{\text{right}}(T) = \{ b_1, B, D \}$ and $\mathcal{W}_{cB}^{\text{right}} = \{ D \}$. Right-expansion is performed by scanning $\mathcal{D}_S$ and computing the set $\mathcal{W}_S^{\text{right}} = \bigcup_{T \in \mathcal{D}_S} \{\mathcal{W}_S^{\text{right}}(T)\}$ of right items along with their frequencies. For each frequent item $w' \in \mathcal{W}_S^{\text{right}}$, we output Sw' and recursively grow Sw'.

Overhead. In the context of LASH, the approaches described above have substantial computational overhead: They compute and output all frequent sequences, whether or not these sequences are pivot sequences (i.e., $p(S) = w$). To see this, consider pivot D, the hierarchy of Fig. 1(b), and example partition

$$\mathcal{P}_D = \{ aDDa, c\,a\,b_1D, c\,a_\sqcup DB, B_\sqcup aDb_1c \}, \qquad (4)$$

for $\sigma = 2$, $\gamma = 1$ and $\lambda = 4$. Both BFS and DFS methods will produce sequences such as ca, ab_1, and aB, neither of which contain pivot D and thus need to be filtered out by LASH. Unfortunately, neither BFS nor DFS can be readily extended to avoid enumerating non-pivot sequences. This is because short non-pivot sequence might contribute to longer pivot sequences. In BFS, we obtain frequent pivot sequence caD from ca (a non-pivot sequence) and aD (a pivot sequence). Similarly, DFS obtains caD by expanding the non-pivot sequence ca. This costly computation of non-pivot sequences cannot be avoided without sacrificing correctness. Note that both approaches also compute frequent sequences that do not contribute to a pivot sequence later on (e.g., sequence aB).

5.2 Pivot Sequence Miner

In what follows, we propose PSM, an effective and efficient algorithm that significantly reduces the computational cost of mining each partition. In contrast to the methods discussed above, PSM restricts its search space to only pivot sequences and is thus customized to LASH. We also describe optimizations that further improve the performance of PSM. Note that our PSM algorithm is aware of hierarchies, but it is nevertheless also beneficial for "flat" sequence mining without hierarchies; see Sec. 6.3.

Algorithm. The key goal of PSM is to only enumerate pivot sequences. PSM is based on the hierarchy-aware version of PrefixSpan described above, but, in contrast, starts with the pivot w (instead of the empty sequence) and extends to the left and to the right (instead of just to the right). Since PSM starts with the pivot, every intermediate sequence will be a pivot sequence. The PSM algorithm is shown as Alg. 2. We assume that for all $T \in \mathcal{P}_w$, $p(T) = w$; this property is ensured by w-generalization (Sec. 4.2).

PSM starts with $S = w$ (pivot item) and determines the support set $\mathcal{D}_w$ (line 1); under our assumptions, $\mathcal{D}_w = \mathcal{P}_w$ so that nothing needs to be done. We then perform a series of right-expansions almost as described above (lines 2 and 13); the only difference is that we do not right-expand with the pivot item (cf. line 11). After the right-expansions are completed, we have produced all frequent pivot sequences that start with the pivot item (and do not contain another occurrence of the pivot item).

Fig. 3 illustrates PSM on the partition of Eq. (4) with pivot D. Solid nodes represent frequent sequences; dotted nodes represent infrequent sequences that are explored by PSM. Each edge corresponds to an expansion and is labeled with its type (RE=right expansion, LE=left expansion) and order of expansion. We start with sequence D and perform the first right-expansion (RE 1) to obtain sequences Da, Db_1, DB, and Dc from which only DB turns out to be frequent. We then right-expand DB to obtain DBc (RE 2), which is infrequent. At this point, no more right-expansions are performed and we return to the pivot.

Algorithm 2 Mining pivot sequences

```
Require: P_w, W, →, σ, γ, λ
 1: S ← w, D_S ← Sup_γ(S, P_w)
 2: EXPAND(S, D_S, right)
 3: EXPAND(S, D_S, left)
 4:
 5: EXPAND(S, D_S, dir)
 6: if |S| = λ then
 7:    return
 8: else
 9:    Scan D_S to compute W_S^dir
10:    if dir = right then
11:       for all w' ∈ W_S^dir \ { w } with f_γ(Sw', P_w) ≥ σ do
12:          Output (Sw', f_γ(Sw', P_w))
13:          EXPAND(Sw', D_Sw', right)
14:       end for
15:    end if
16:    if dir = left then
17:       for all w' ∈ W_S^dir with f_γ(w'S, P_w) ≥ σ do
18:          Output (w'S, f_γ(w'S, P_w))
19:          EXPAND(w'S, D_w'S, right)
20:          EXPAND(w'S, D_w'S, left)
21:       end for
22:    end if
23: end if
```

After the pivot has been right-expanded, PSM performs a *left-expansion* of the pivot w (line 3), producing sequences of form $w'w$. Left-expansions are symmetrical to right-expansions, but we expand S to the left by computing the set of items $\mathcal{W}_S^{\text{left}} = \bigcup_{T \in \mathcal{D}_S} \{\mathcal{W}_S^{\text{left}}(T)\}$, where $\mathcal{W}_S^{\text{left}}(T) = \{ w' \mid w'S \sqsubseteq_\gamma T \}$. In our example, we obtain frequent sequence aD and some infrequent sequences (LE 3). We now perform a sequence of right-expansions on aD (RE 4 and RE 5, line 19 of Alg.2). Note that PSM never left-expands a sequence that is a result of a right-expansion. Once all right-expansions of aD have been computed, we left-expand it (LE 6, line 20) and proceed recursively as above.

Correctness. PSM enumerates each frequent pivot sequence exactly once; there are no duplicates and no missed sequences. To see this, consider an arbitrary pivot sequence S of length at least 2 (with pivot w). Then there is a unique decomposition

$$S = S_l w S_r$$

such that $w \notin S_r$. We refer to S_l as the *prefix* of S (i.e., the part of S occurring to the left of the (last) pivot) and S_r as the *suffix* (to the right). For example, sequence caD with pivot D has prefix $S_l = ca$ and suffix $S_r = \epsilon$. Note that the decomposition is unique because $w \notin S_r$; e.g., $S = aDDa$ uniquely decomposes into $S_l = aD$ and $S_r = a$. This is the reason why we do not right-expand with the pivot (line 11). PSM generates S from D by first performing left-expansions until $S_l w$ is obtained, and then a series of right-expansions to obtain $S_l w S_r$. Fig. 3 shows a number of examples; e.g., sequence $caDB$ is obtained by expansions LE 3, LE 6, and RE 7. If PSM were to perform left-expansions after a right expansion, then $caDB$ would also be obtained from

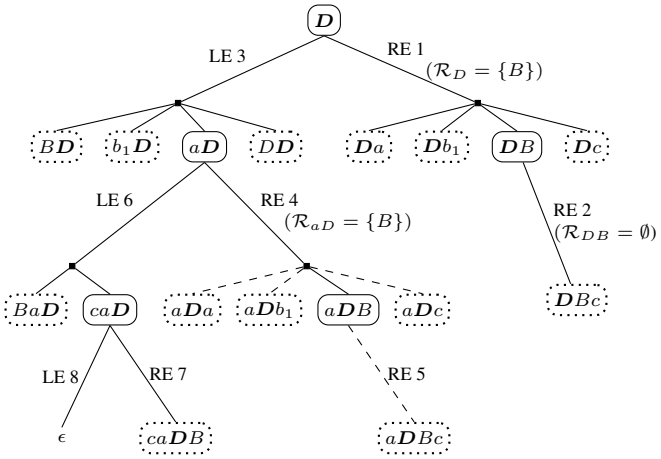


Figure 3: Pivot sequence enumeration for partition $\mathcal{P}_D$ for $\sigma = 2$, $\gamma = 1$ and $\lambda = 4$.

a left-expansion of aDB (obtained from RE 4). PSM avoids such duplicates.

Indexing right-expansions. We now describe an optimization technique which further reduces the search space. The key idea is to store information of right-expansions to make future right-expansions more efficient. To see why this may help, consider RE 1 and RE 4 in Fig. 3. From RE 1, we know that Da is infrequent. Thus, when performing RE 4, we do not need to consider sequence aDa since it must also be infrequent (Lemma. 1). In general, if Sw' is an infrequent right-expansion of S, then $w''Sw'$ will be an infrequent right-expansion of $w''S$.

We make use of this observation as follows. Whenever we perform a right-expansion of some sequence S, we store in an index the set $\mathcal{R}_S$ of the resulting frequent expansion items. In our example, we have $\mathcal{R}_D = \{ B \}$ from RE 1 since DB is the only frequent right-expansion of D. We subsequently use the information about $\mathcal{R}_S$ as follows. Whenever we perform a right-expansion of some sequence S_lS, we restrict the set of expansion items to $\mathcal{R}_S$. In our example, when expanding aD in RE 4, we only consider expansion item B (since $\mathcal{R}_D = \{ B \}$). For all other items, neither counting nor support set computation is performed; these items are shown in nodes connected with dashed lines in Fig. 3. If $\mathcal{R}_S$ is empty, no right-expansions need to be performed and we do not scan the database. This happens for the sequence aDB in our example; since we obtain $\mathcal{R}_{DB} = \emptyset$ from RE 2, we do not perform RE 5.

Our choice of indexing only right-expansions is tailored to the order in which PSM explores pivot sequences. For example, consider LE 3 in Fig. 3. Information about frequent left-expansions for $S = D$ (i.e., aD) will not be of any use, since during the traversal, we will never left-expand any sequence of the form SS_r (such as DB; recall that PSM never left-expands a sequence that is a result of a right-expansion). Therefore, we only index right-expansions to prune search space. To save memory, our actual implementation unions the indexes of each level of each series of right expansions (i.e., we maintain one index for the frequent items that occur directly after S, one index for the items that occur two items after S, and so on).

Analysis. In what follows, we study the worst-case size of the search space of the PSM algorithm and compare it to the one of the BFS and DFS approaches. Let us assume that a database (or partition) has k distinct items and that each sequence in the database has length λ. Further assume that all possible sequences of length up to λ are frequent in the database; there are $\sum_{l=1}^{\lambda} k^l$ such sequences. Both BFS and DFS will first produce all of these sequences, but in the context of LASH, subsequently only output the ones that contain the pivot item. There are $\sum_{l=1}^{\lambda}(k-1)^l$ sequences that do not contain the pivot; these are produced unnecessarily. In contrast, PSM only explores pivot sequences, of which there are $\sum_{l=1}^{\lambda} k^l - \sum_{l=1}^{\lambda}(k-1)^l$. Thus PSM explores a fraction of

$$1 - \frac{\sum_{l=1}^{\lambda}(k-1)^l}{\sum_{l=1}^{\lambda} k^l} \ll 1$$

of the sequences explored by BFS or DFS methods. For example, if $k = 100{,}000$ and $\lambda = 5$, PSM explores 0.005% of the search space of BFS or DFS.

In practice, the worst-case rarely occurs, of course. To shed more light on the relationship between PSM and DFS, consider our running example and suppose that we used DFS. In a first step, DFS computes all (item, frequency)-pairs, of which there are five: $(a, 2)$, $(b_1, 2)$, $(B, 3)$, $(c, 3)$ and $(D, 4)$. For each so-found frequent sequence, DFS recursively makes a right-expansions to compute longer frequent sequences. In our running example, DFS ultimately computes 17 length-2 sequences (but outputs only the frequent ones): $(aD, 4)$, $(ab_1, 2)$, $(aB, 2)$, $(aa, 1)$, $(b_1D, 1)$, $(b_1c, 1)$, $(BD, 1)$, $(Ba, 1)$, $(Bc, 1)$, $(ca, 2)$, $(cb_1, 1)$, $(cB, 1)$, $(DD, 1)$, $(Da, 1)$, $(DB, 2)$, $(Db_1, 1)$, $(Dc, 1)$. Similarly, DFS computes 13 length-3 sequences and two length-4 sequences. The total size of the search space of DFS is thus 37. On the other hand, PSM only explores 13 sequential patterns; these are shown by the nodes connected with solid lines in Fig. 3. Thus PSM explores only roughly one third of the search space of DFS in our example.

6. EXPERIMENTS

We now present results of our experimental study using two large real-world datasets in the contexts of generalized n-gram mining from textual data and customer behavior mining from product sequences. In particular, we compared LASH to the naïve and the semi-naïve algorithms, evaluated the efficiency of the PSM algorithm for mining each partition, and studied the scalability of LASH. We also studied the effect of different parameters—i.e., support (σ), gap (γ) and length (λ)—and how different types of hierarchies affect the performance of LASH.

We found that LASH outperformed the naïve and semi-naïve algorithms by multiple orders of magnitude. For mining partitions locally, the PSM algorithm was more efficient and faster than the BFS and DFS algorithms. Our scalability experiments suggest that LASH scales linearly as we add more compute nodes and/or increase input dataset size.

6.1 Experimental Setup

Implementation and cluster. We implemented LASH, the semi-naïve and naïve methods in Java (JDK 1.7). We represent items by assigning integers item ids according to the order $<$ obtained from the generalized f-list. Thus, highly frequent items are assigned smaller integer ids. We represent sequences as arrays of item ids and compress the data transmitted between the map and reduce phase using variable-length integer encoding. All experiments were run on a local Hadoop cluster consisting of eleven Dell PowerEdge R720 computers, each with 64GB of main memory, eight 2TB SAS 7200 RPM hard disks and two Intel Xeon E5-2640 6-core CPUs. Debian Linux (kernel version 3.2.48.1.amd64-smp) was used as an operating system. The machines in the cluster are connected via 10 GBit Ethernet. We use the Cloudera cdh3u6 distribution of Hadoop 0.20.2 running on Oracle Java 1.7.0_25. One

Dataset	Sequences	Avg length	Max length	Total items	Unique items
NYT	49,605,960	21.1	15,199	1,047,419,137	2,763,301
AMZN	6,643,666	4.5	25,630	29,667,966	2,374,096

Table 1: Dataset characteristics

Dataset	Hierarchy	Total items	Leaf items	Root items	Intermediate items	Levels	Avg.fan-out	Max.fan-out
NYT	L	2,910,327	407,806	2,502,521	0	2	2.7	36
	P	2,617,581	2,617,559	22	0	2	124,645.6	1,828,130
	LP	2,910,347	2,763,300	22	147,025	3	19.8	1,822,454
	CLP	2,970,092	2,763,300	22	206,770	4	14.4	1,822,454
AMZN	h2	2,374,147	2,371,524	2,623	0	2	48,398.4	904,162
	h3	2,374,509	2,371,536	2,630	343	3	6,050.7	332,723
	h4	2,376,539	2,371,670	2,633	2,236	4	1,038.9	332,723
	h8	2,387,422	2,373,158	2,634	11,630	8	204.2	332,723

Table 2: Hierarchy characteristics

machine acted as a Hadoop master node; the other ten machines acted as worker nodes. The maximum number of concurrent map or reduce tasks was set to 8 per worker node. All tasks are launched with 4 GB heap space.

Datasets. Statistics of the datasets and hierarchies used in our experiments are summarized in Tab. 1 and Tab. 2 respectively. We used two real-world datasets: New York Times corpus (NYT) [3] for mining generalized n-grams and Amazon product reviews dataset (AMZN) [5] for mining product sequences.

The NYT dataset consists of roughly 50M sentences from 1.8 million articles published during 1987 and 2007. We treat each sentence as an input sequence with each word (token) as an item. We generated a syntactic hierarchy by annotating the each word with its lemma and part-of-speech tag using the Stanford CoreNLP parser [4] and also annotated each word with its lower-case form (if different that its surface form). In our syntactic hierarchy, a word appearing in a sentence can generalize to its lowercase form, which generalizes to its lemma, which in turn generalizes to its part-of-speech tag. For example, the word "Changing" $\rightarrow$ "changing" $\rightarrow$ "change" $\rightarrow$ "VERB". We generated four variants of this hierarchy: NYT-L (word $\rightarrow$ lemma), NYT-P (word $\rightarrow$ pos), NYT-LP (word $\rightarrow$ lemma $\rightarrow$ pos) and NYT-CLP (word $\rightarrow$ case $\rightarrow$ lemma $\rightarrow$ pos). Note that the surface form of many words appearing the input sequences is same as their lowercase or lemma; this naturally creates a hierarchy in which items appearing in the input sequences come from different levels.

The AMZN dataset consists over 35 million reviews from over 6 million users spanning from 1995 to 2013. To generate product sequences, we identified a user sessions by grouping the reviews by user and sorting each so-obtained sequence by timestamp. We used the Amazon product hierarchy, in which, for example, the book "For Whom the Bell Tolls" $\rightarrow$ "Classics" $\rightarrow$ "Literature & Fiction" $\rightarrow$ "Books". We also considered different hierarchy types of varying depths (2-8) by varying the number of intermediate categories a product is assigned to.

Measures. In the following experiments, we report the performance measure as total time elapsed between launching a task and receiving the final result. We break down this time into time taken by the map phase, shuffle phase and the reduce phase. Since these phases overlap in a MapReduce job, we report the time elapsed until finishing of each phase. We also report the bytes transferred as the total data transferred between map and reduce task as obtained from Hadoop's `MAP_OUTPUT_BYTES` counter. All measurements reported are based on average of three independent runs and were performed with exclusive access to the machines.

6.2 Overall Runtime

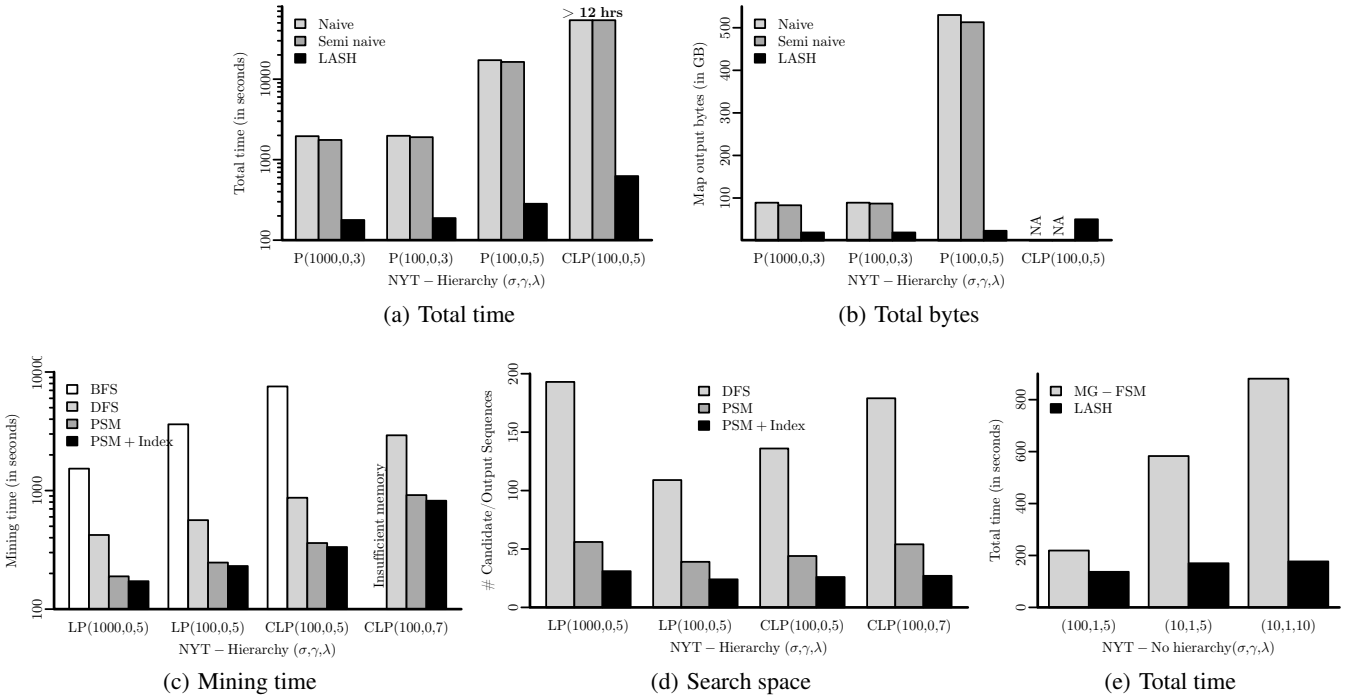
We initially evaluated the performance of LASH generalized n-gram mining (i.e., $\gamma = 0$) and compared it with the naïve and semi-naïve methods (discussed in Sec. 3) using the NYT-P dataset having two levels of hierarchy. For this dataset, we mined generalized n-grams with three different parameter settings of increasing difficulty w.r.t. to output size. The results are plotted using a log-scale in Fig. 4(a). With $\sigma = 1000$, $\lambda = 3$ and $\sigma = 100$, $\lambda = 3$, LASH obtained a speedup of around $10\times$. Further, LASH achieves a speed up of more than $50\times$ for the setting with $\sigma = 100$, $\lambda = 5$.

For the entire NYT-CLP dataset having four levels of hierarchy, the naïve and semi-naïve algorithms were unable to handle the combinatorial blowup of the search space and were aborted after 12 hours. On the other hand, LASH required a little over 600 seconds only. Also, as shown in Fig. 4(b) the total bytes transferred between the map and reduce phase is significantly less for LASH.

6.3 Local Mining

In our next set of experiments, we studied the efficiency of the PSM algorithm by running LASH with sequential GSM algorithms of Sec. 5.1. We used the NYT dataset and performed runs with different settings of increasing difficulty w.r.t. output size. The results are shown in Fig. 4(c) using log-scale. Since our choice of sequential mining approaches only affect the reduce phase, we report the mining time as time elapsed between the end of last reduce task and end of first sort task.

Compared to BFS, with the LP hierarchy (three levels) and the parameters $\sigma = 1000$, $\lambda = 5$, PSM was $9\times$ faster. As we decreased the value of σ to 100, PSM was $15\times$ faster and up to $22\times$ faster with the full CLP hierarchy. For the setting CLP ($\sigma = 100$, $\lambda = 7$), BFS reported insufficient memory and terminated. On comparison to DFS, PSM was $2.5\times$ to $3.5\times$ faster for these settings. The efficiency of PSM stems from an optimized search space exploration of pivot sequences w.r.t. the partitions being mined. Since PSM uses a customized depth-first search, we also compared the number of candidate sequences generated per output sequence by DFS and PSM. As observed from Fig. 4(d), PSM explores a much smaller fraction of the search space.



(a) Total time (b) Total bytes

(c) Mining time (d) Search space (e) Total time

Figure 4: Performance of distributed (a,b,e) and sequential algorithms (c,d).

We also evaluated our optimization technique by studying PSM's performance with indexing (PSM+Index). We observed a trade-off between the construction cost and the benefit of indexing. The runtimes improved by 100s with increase in the values of λ and levels of hierarchy. We also observed that in all the cases, our indexing significantly pruned a lot of search space up to 2× (see Fig. 4(d)).

Lastly, we compared LASH with MG-FSM [20], a state-of-the-art frequent sequence miner. As MG-FSM does not support hierarchies, we ran LASH on the data without using any hierarchies.[3] We used the the NYT dataset with 3 different parameter settings of increasing difficulty. We report the total runtime in Fig. 4(e). We observed an overall speedup of 2× to 5× which essentially stems from using the PSM algorithm for mining partitions where as MG-FSM uses standard BFS approach for mining each partition. Our results indicate that LASH is the best-performing method for mining sequences without hierarchies as well.

6.4 Effect of Parameters

In this group of experiments, we studied how the performance is affected by different parameters σ, γ and λ. We used the AMZN-h8 dataset with full 8 levels of the hierarchy and fixed the parameters to $\sigma = 100$, $\gamma = 1$ and $\lambda = 5$.

We first studied how the minimum support σ affects the performance by varying its value from 10 to 10,000. The results are shown in Fig. 5(a). The time taken by the map phase which consist of rewriting input sequences for each partition decreases as we increase the support. Recall that, our rewrites are independent of σ (see discussion in Sec. 4.4); however, σ has an indirect effect. At higher supports, fewer items from the lower levels of the hierarchy are frequent so that the effective depth of the hierarchy is reduced. Since our rewrites depends on this depth, the time per rewrite decreases as the support threshold is increased. The reduce time decreases as well since mining becomes cheaper at higher supports.

Second, we varied the value of maximum gap γ from 0 to 3. As we can see in Fig. 5(b), the impact on map times was not significant as the cost of rewriting is largely independent of γ. However, it had a significant impact on the reduce times as as the search space during mining significantly increases with γ.

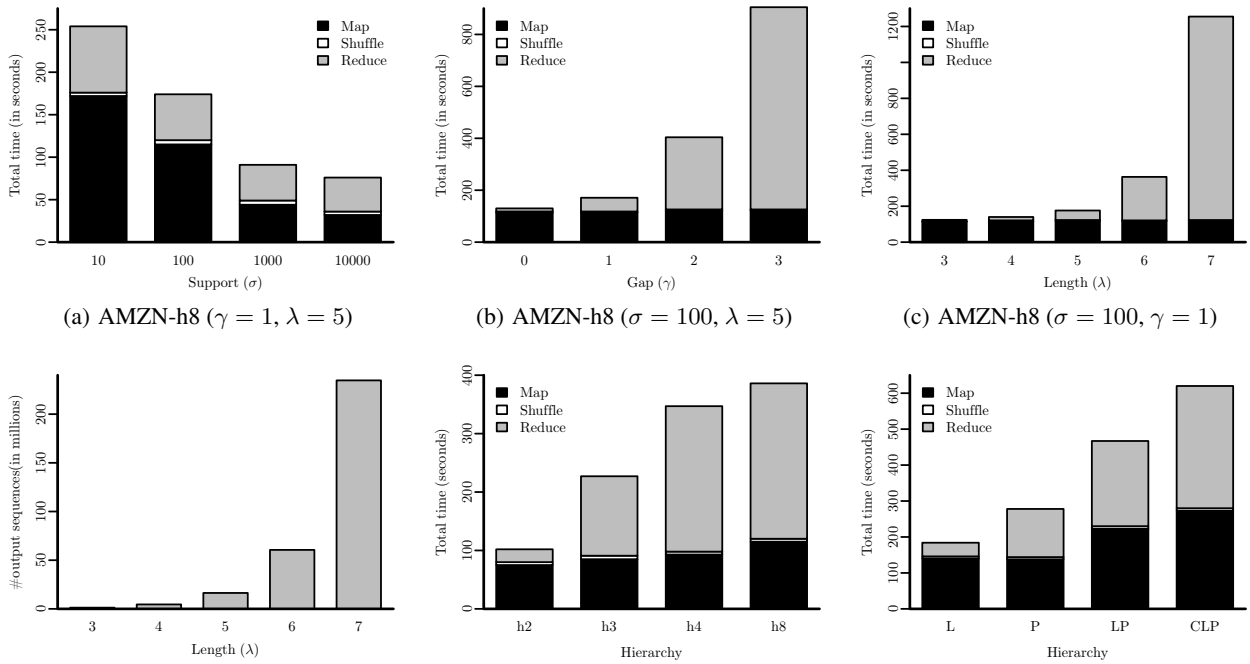
Lastly, we evaluated how maximum length λ effects the performance of LASH by varying its value from 3 to 7. The results are shown in Fig. 5(c). We observed that λ had very little impact on the map time. The reduce time increases significantly as we increase λ since mining becomes more expensive. In fact, the output size increases as we increase λ. Fig. 5(d) shows that output size and reduce times are proportional.

6.5 Effect of Hierarchies

In this group of experiments, we studied how different types of hierarchies affect the performance of LASH. We used the AMZN and NYT datasets. Our results are shown in Fig. 5(e) and Fig. 5(f) respectively.

For the AMZN dataset (Fig. 5(e)), we fixed the parameters $\sigma = 100$, $\gamma = 2$, $\lambda = 5$ and varied the hierarchy levels from 2 to 8. The map times slightly increases with an increase in levels, even though the support is fixed. This is because rewriting each sequence depends on the hierarchy depth. The reduce times increase significantly since an increase in hierarchy levels in turn increases the number of intermediate items (see Tab. 2). This makes mining more expensive: a partition needs to be created and mined for each intermediate item and the mining time of a partition also depends on the depth of the hierarchy. The effect in reduce times is less pronounced when the using the full hierarchy (8 levels) compared to 4 levels because most products in the Amazon product hierarchy have no more than 4 parent categories.

[3] In this setting, LASH is equivalent to MG-FSM with its local miner replaced by PSM.



(a) AMZN-h8 ($\gamma = 1$, $\lambda = 5$) (b) AMZN-h8 ($\sigma = 100$, $\lambda = 5$) (c) AMZN-h8 ($\sigma = 100$, $\gamma = 1$)

(d) AMZN-h8 ($\sigma = 100$, $\gamma = 1$) (e) AMZN ($\sigma = 100$, $\gamma = 2$, $\lambda = 5$) (f) NYT ($\sigma = 100$, $\lambda = 5$)

Figure 5: Effect of parameters (a-d) and hierarchies (e-f).

For the NYT dataset (Fig. 5(f)), we set $\sigma = 100$, $\lambda = 5$, and considered four variants of the syntactic hierarchy (see Sec. 6.1). NYT-L and NYT-P both have two levels but show a significant difference in reduce times. This is because the NYT-L hierarchy has many roots with low fan-out, whereas the NYT-P hierarchy has few roots with high fan-out. Mining the latter hierarchy is more expensive, partly due to the high frequency of the root items, partly due to larger output size. We also observed that adding more levels to the hierarchy (NYT-LP and NYT-CLP) significantly increases both the map and the reduce times.

6.6 Scalability

In our final group of experiments, we studied the scalability of LASH as we add more compute nodes and/or increase the input data size. We used the NYT dataset with full CLP hierarchy and set the parameters $\sigma = 100$ and $\lambda = 5$.

We first investigated the performance of LASH as we vary the input data size. To this end, from the NYT dataset, we extracted datasets that contain a random 25%-, 50%- and 75%-sample of the input sequences. The results are shown in Fig. 6(a). We observed that LASH is robust in handling increasing amounts of data with both map and reduce times increasing linearly as we add more data.

We evaluated strong scalability by running LASH on a fixed dataset (100% NYT-CLP) and varying the amount of parallel work by using 2, 4 and 8 compute nodes. We observed from Fig. 6(b) that LASH exhibits good linear scalability with both map and reduce times decreasing equally as we increase the number of compute nodes.

We also evaluated weak scalability for LASH, in which we increase the input data size as we add more compute nodes. In particular, we simultaneously increased the size of the input data (25%, 50% and 100% of NYT-CLP) and number of compute nodes (2, 4 and 8). As observed from Fig. 6(c), LASH exhibits good weak scalability. Note that the total time ideally remains constant as we double both computational resources and input dataset. In practice, however, the number of output sequences increases by a factor of more than 2 when doubling the input data. We thus observe a slight increase in the runtimes. In this particular case, the number of output sequences increased from 43M (25% of input) to 99M (50% of input) to 220M (100% of input), which is a factor of 2.2×.

Hence, LASH provides an efficient partitioning and mining technique for highly parallelized generalized frequent sequence mining.

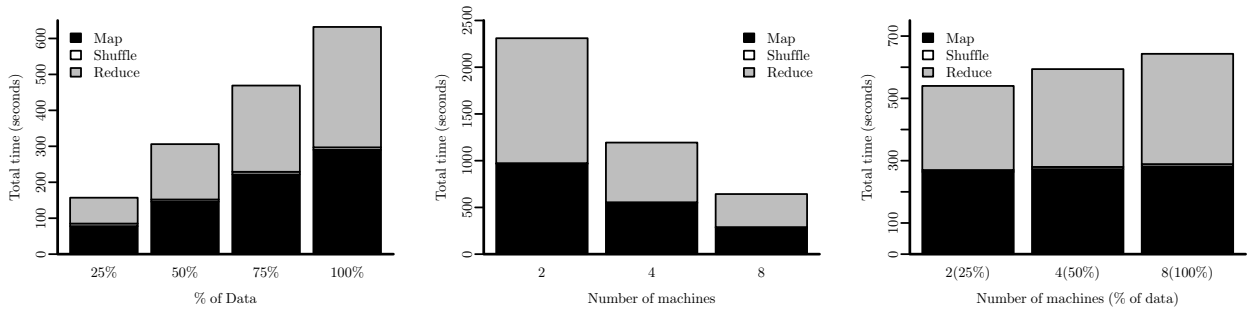
Dataset		Non-trivial (%)	Closed (%)	Maximal (%)
	Hierarchy			
NYT ($\sigma = 100$, $\lambda = 5$)	P	75.47	89.08	31.92
	LP	73.47	50.38	10.11
	CLP	70.26	35.42	6.06
	Min sup (σ)			
AMZN-h8 ($\gamma = 1$, $\lambda = 5$)	10,000	100	100	21.56
	1,000	99.78	85.79	14.50
	100	97.38	64.86	10.06

Table 3: Output Statistics

6.7 Output Statistics

We computed a number of statistics of the set of generalized subsequences that we mined from our datasets; the results are shown in Tab. 3.

First, we computed the percentage of *non-trivial* output sequences to judge whether generalized sequence mining is beneficial. We say that an output sequence is *trivial*, if it can be generated from the output of a standard sequence miner (which ignores hierarchies) by generalizing items. For example, non-trivial sequences on the



(a) NYT-CLP ($\sigma = 100$, $\lambda = 5$) (b) NYT-CLP ($\sigma = 100$, $\lambda = 5$) (c) NYT-CLP ($\sigma = 100$, $\lambda = 5$)

Figure 6: Scalability results

NYT-CLP dataset ($\sigma = 100$) include: "NOUN lives in NOUN", "NOUN works at NOUN" and "the ADJ Book"; no specializations of these patterns were frequent in the input data. For the NYT and AMZN datasets, we observed that more than 70% and 95%, resp., of the sequences were non-trivial.

Recall the discussion at the end of Sec. 2, in which we argue that GSM may produce "redundant" (but nevertheless potentially useful) sequences. To see how many redundant sequence are mined, we computed the number of closed and maximal subsequences. In the context of GSM, a frequent sequence S is *maximal* if every supersequence $S' \sqsupseteq_0 S$ is infrequent, and closed if every supersequence has a different frequency. In Tab. 3, we observe that adding more levels to the hierarchy or lowering the support increases the fraction of redundant patterns, but that nevertheless a large number of patterns is non-redundant. To the best of our knowledge, direct mining of maximal or closed sequences in the context of hierarchies has not been studied in the literature. Our results indicate that such methods are a promising direction for future work.

7. RELATED WORK

We now discuss, how our work relates to existing literature, which can be broadly categorized into:

Sequential pattern mining. The problem of mining sequential patterns was introduced in the seminal work by Agarwal et al. [6]. The proposed APRIORI algorithm is based on a candidate-generation-and-test approach that repeatedly scans the input data to generate and count candidate sequential patterns and prunes the infrequent ones. Apriori's successor, the GSP algorithm [26], iteratively generates candidate l-sequences by joining frequent $(l-1)$-sequences and prunes the infrequent ones by scanning the input database. SPADE [31] uses a vertical representation of the sequence database which can be seen as an inverted index that maps sequences to their corresponding offsets in input transactions. It uses either breadth-first or depth-first traversal of the sequence lattice to determine frequent patterns. The BFS approach (Sec. 5.1) adapts these algorithms to efficiently handle hierarchies.

In contrast to the candidate-generation-and-test approach, Han et al. proposed the FP-Growth algorithm [12] based on the pattern-growth principle. It uses an item-based partitioning of the output search space. Based on similar ideas, the PrefixSpan algorithm [23] uses a suffix-based partitioning of the output space and uses database projections to mine frequent patterns. SPAM [8], which is similar to SPADE, uses an internal bitmap structure for database representation and employs a pattern-growth approach to mine frequent sequential patterns. The BIDE [29] and the Gap-BIDE [16] algorithms mine frequent closed subsequences; they use a depth-first strategy to enumerate closed sequences. The DFS (Sec. 5.1) and PSM (Sec. 5.2) algorithms are pattern-growth approaches.

Hierarchies in sequential pattern mining. Agarwal et al. [26] proposed the use of extended sequences to incorporate hierarchies into the mining process. In this approach, each item in a sequence is replaced by an itemset containing the item and all its ancestors. Generalized sequence mining using extended sequences is inefficient as it increases the size of the sequence database by a factor of roughly the depth of the hierarchy. Hierarchies have also been explored in context of multi-dimensional sequential pattern mining. To this end, Plantevit et al. proposed the HYPE algorithm [25] and the M^3SP algorithm [24] as its successor. In their approach, they prune hierarchies by only considering maf-sequences, which are pairs of items (each belonging to a dimension) that are maximal (i.e., each specialization is infrequent). Subsequently, they use SPADE to generate frequent sequences. A known limitation of their approach is that they do not mine all frequent sequences. Chen et al. [10] sketched the idea of fuzzy multi-level sequential patterns. They consider hierarchies in which an item can have more than one parent with different degrees of confidence and use a GSP-like approach to mine such patterns. Huang [14] later presented a divide-and-conquer strategy based on the pattern-growth approaches to mine such fuzzy multi-level patterns. Both approaches encode hierarchy information in each item, which is reminiscent of extended sequences and are outperformed by GSP [14].

Parallel and distributed mining. Our work is most closely related to the MG-FSM algorithm [20] by Miliaraki et al. and shares the same design philosophy. However, their approach is not applicable in our setting since MG-FSM's partitioning and mining techniques cannot handle hierarchies. LASH extends the partition construction framework of MG-FSM to efficiently handle hierarchies and employs PSM, a more efficient local mining algorithm. On datasets without hierarchies, on which both MG-FSM and LASH can be applied, LASH is more efficient due to PSM.

8. CONCLUSION

We proposed LASH, an algorithm for mining frequent sequences in presence of hierarchies. To the best of our knowledge, LASH is the first distributed, scalable algorithm for mining such generalized sequences. LASH uses a novel, hierarchy-aware form of item-based partitioning, optimized partition construction techniques, and an efficient, special-purpose algorithm for mining each partition independently and in parallel. Our experimental study indicates that LASH is efficient, scales to large real-world datasets, and is multiple orders of magnitude faster than existing baseline methods.

9. REFERENCES

[1] Google n-Grams. https://books.google.com/ngrams/.

[2] Netspeak. http://www.netspeak.org/.

[3] The New York Times annotated corpus. http://catalog.ldc.upenn.edu/LDC2008T19.

[4] Stanford coreNLP parser. http://nlp.stanford.edu/software/corenlp.shtml.

[5] Web data: Amazon reviews. http://snap.stanford.edu/data/web-Amazon.html.

[6] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, 1995.

[7] L. V. Q. Anh and M. Gertz. Mining spatio-temporal patterns in the presence of concept hierarchies. In *ICDMW*, 2012.

[8] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *SIGKDD*, 2002.

[9] S. Bergsma, D. Lin, and R. Goebel. Web-scale n-gram models for lexical disambiguation. In *IJCAI*, volume 9, pages 1507–1512, 2009.

[10] Y.-L. Chen and T. C.-K. Huang. A novel knowledge discovering model for mining fuzzy multi-level sequential patterns in sequence databases. *Data Knowl. Eng.*, 66(3):349–367, Sept. 2008.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[12] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[13] L. Hollink, P. Mika, and R. Blanco. Web usage mining with semantic analysis. In *WWW*, 2013.

[14] T. C.-K. Huang. Developing an efficient knowledge discovering model for mining fuzzy multi-level sequential patterns in sequence databases. *Fuzzy Sets Syst.*, 160(23):3359–3381, 2009.

[15] H. Jang and J. Mostow. Inferring selectional preferences from part-of-speech n-grams. In *EACL*, 2012.

[16] C. Li and J. Wang. Efficiently mining closed subsequences with gap constraints. In *SDM*, 2008.

[17] Z. Liao, D. Jiang, E. Chen, J. Pei, H. Cao, and H. Li. Mining concept sequences from large-scale search logs for context-aware query suggestion. *ACM Trans. Intell. Syst. Technol.*, 3(1):17:1–17:40, 2011.

[18] Y. Lin, J.-B. Michel, E. L. Aiden, J. Orwant, W. Brockman, and S. Petrov. Syntactic annotations for the google books ngram corpus. In *ACL*, 2012.

[19] A. Lopez. Statistical machine translation. *ACM Comput. Surv.*, 40(3), 2008.

[20] I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos. Mind the gap: Large-scale frequent sequence mining. In *SIGMOD*, 2013.

[21] N. Nakashole, M. Theobald, and G. Weikum. Scalable knowledge harvesting with high precision and high recall. In *WSDM*, 2011.

[22] N. Nakashole, G. Weikum, and F. Suchanek. Patty: A taxonomy of relational patterns with semantic types. In *EMNLP-CoNLL*, 2012.

[23] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. PrefixSpan: Mining sequential patterns by prefix-projected growth. In *ICDE*, 2001.

[24] M. Plantevit, A. Laurent, D. Laurent, M. Teisseire, and Y. W. Choong. Mining multidimensional and multilevel sequential patterns. *TKDD*, 4(1):4:1–4:37, 2010.

[25] M. Plantevit, A. Laurent, and M. Teisseire. Hype: Mining hierarchical sequential patterns. In *DOLAP*, 2006.

[26] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EBDT*, 1996.

[27] R. Srikant and R. Agrawal. Mining generalized association rules. In *VLDB*, pages 407–419, 1995.

[28] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explor. Newsl.*, 1(2):12–23, Jan. 2000.

[29] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *ICDE*, 2004.

[30] W. Wang and D. Vergyri. The use of word n-grams and parts of speech for hierarchical cluster language modeling. In *ICASSP*, 2006.

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