Mind the Gap: Large-Scale Frequent Sequence Mining

Iris Miliaraki Klaus Berberich Rainer Gemulla Spyros Zoupanos

{miliaraki, kberberi, rgemulla, zoupanos}@mpi-inf.mpg.de
Max Planck Institute for Informatics
Saarbrücken, Germany

ABSTRACT

Frequent sequence mining is one of the fundamental building blocks in data mining. While the problem has been extensively studied, few of the available techniques are sufficiently scalable to handle datasets with billions of sequences; such large-scale datasets arise, for instance, in text mining and session analysis. In this paper, we propose MG-FSM, a scalable algorithm for frequent sequence mining on MapReduce. MG-FSM can handle so-called "gap constraints", which can be used to limit the output to a controlled set of frequent sequences. At its heart, MG-FSM partitions the input database in a way that allows us to mine each partition independently using any existing frequent sequence mining algorithm. We introduce the notion of w-equivalency, which is a generalization of the notion of a "projected database" used by many frequent pattern mining algorithms. We also present a number of optimization techniques that minimize partition size, and therefore computational and communication costs, while still maintaining correctness. Our experimental study in the context of text mining suggests that MG-FSM is significantly more efficient and scalable than alternative approaches.

Categories and Subject Descriptors

H.2.8 [**Database management**]: Database Applications—*Data Mining*

Keywords

data mining, frequent sequence mining, MapReduce

1. INTRODUCTION

Frequent sequence mining (FSM) is a fundamental component in a number of important data mining tasks. In text mining, for example, frequent sequences can be used to construct statistical language models for machine translation [17], information retrieval [32], information extraction [26], or spam detection [14]. Word associations have also been applied to relation extraction [20]. In Web usage mining and session analysis [24], frequent sequences describe common behavior across users (e.g., frequent sequences of page visits). In these and similar applications, instances of FSM can get very large and may involve billions of sequences. For example, Microsoft provides access to an n-gram collection based on hundreds of billions of web pages and Google published a corpus of more than 1 billion n-grams. Similarly, in web companies with millions of users, the amount of usage data can be substantial. At such massive scales, distributed and scalable FSM algorithms are essential.

Given a collection of *input sequences* of items, the goal of FSM is to find all subsequences that "appear" in sufficiently many input sequences. In text mining, for example, each input sequence corresponds to a document (or a sentence) and each item to a word in the document. The definition of "appears" is application-dependent; e.g., the goal of *n-gram mining* is to find frequent *consecutive* word sequences of length n, whereas the goal of *word association mining* is to find combinations of words that frequently appear in close proximity (but not necessarily consecutively). As another example, in session analysis, input sequences correspond to user sessions and items to user actions (often with an additional time stamp). Depending on the application, we may be interested in sequences of either consecutive actions or non-consecutive actions that are sufficiently close (e.g., few actions in between or temporally close). This requirement of closeness is addressed by *gap-constrained frequent sequence mining* [23], in which FSM is parameterized with a *maximum-gap parameter* γ. Informally, for a given input sequence, we consider only subsequences that can be generated without skipping more than γ consecutive items. We obtain n-gram mining for $\gamma = 0$, word association mining for (say) $\gamma = 5$, and unconstrained FSM [31] for $\gamma = \infty$. Temporal gaps—such as "at most one minute" for session analysis—can also be handled.

In this paper, we propose a scalable, distributed (i.e., shared nothing) FSM algorithm called MG-FSM. Although scalable algorithms exist for n-gram mining [13, 3], MG-FSM is the first distributed algorithm that supports general gap constraints. MG-FSM targets MapReduce [8]—which constitutes a natural environment for text mining and analysis of user access logs—but is also amenable to other distributed programming environments. At a high-level, MG-FSM carefully partitions and rewrites the set of input sequences in such a way that each partition can be mined independently and in parallel. Once partitions have been constructed, an arbitrary gap-constrained FSM algorithm can be used to mine each partition; no post-processing of results across partitions is needed.

MG-FSM extends the notion of item-based partitioning, which underlies a number of frequent pattern mining algorithms, including FP-growth [12] as well as the distributed algorithms of [5, 7] for frequent itemset mining, to gap-constrained frequent sequence mining. In more detail, we first develop a basic partitioning scheme that ensures correctness but allows for *flexible* partition construction. This flexibility is captured in our notion of w-equivalency, which generalizes the concept of a "projected database" used by many FSM algorithms. We also propose a number of novel optimization techniques that aim to reduce computational and communication costs, including *minimization* (prunes entire sequences), *reduction* (shortens long sequences), *separation* (splits long sequences), *aggregation* (of repeated sequences), and *light-weight compression*. Our experiments, in which we mine databases with more than 1 billion sequences, suggest that MG-FSM is multiple orders of magnitude faster than baseline algorithms for general gap-constrained FSM and is competitive to state-of-the-art algorithms for n-gram mining.

2. PRELIMINARIES

2.1 Problem Statement

A *sequence database* $\mathscr{D} = \{ S_1, \ldots, S_D \}$ is a multiset of *input sequences*.[1] A *sequence* is an ordered list of *items* from some *dictionary* $\Sigma = \{ w_1, \ldots, w_{|\Sigma|} \}$. We write $S = s_1 s_2 \cdots s_{|S|}$ to denote a sequence of *length* $|S|$, where $s_i \in \Sigma$ for $1 \leq i \leq |S|$. Denote by Σ^+ the set of all non-empty sequences constructed with items from Σ. In what follows, we will often use the symbol T to refer to an input sequence in the database and symbol S to refer to an arbitrary sequence.

Denote by $\gamma \geq 0$ a *maximum-gap parameter*. We say that S is a γ*-subsequence* of T, denoted $S \subseteq_\gamma T$, when S is a subsequence of T and there is a gap of at most γ between consecutive items selected from T; standard n-grams correspond to 0-subsequences. Formally, $S \subseteq_\gamma T$ if and only if there exist indexes $i_1 < \ldots < i_n$ such that (i) $S_k = T_{i_k}$ for $1 \leq k \leq n$, and (ii) $i_{k+1} - i_k - 1 \leq \gamma$ for $1 \leq k < n$. For example, if $T = abcd$, $S_1 = acd$ and $S_2 = bc$, then $S_1 \subseteq_1 T$ (but $S_1 \not\subseteq_0 T$) and $S_2 \subseteq_0 T$.

The γ*-support* $\mathrm{Sup}_\gamma(S, \mathscr{D})$ of S in database $\mathscr{D}$ is given by the multiset

$$\mathrm{Sup}_\gamma(S, \mathscr{D}) = \{ T \in \mathscr{D} : S \subseteq_\gamma T \}.$$

Denote by $f_\gamma(S, \mathscr{D}) = |\mathrm{Sup}_\gamma(S, \mathscr{D})|$ the *frequency* of sequence S. Our measure of frequency corresponds to the notion of *document frequency* in text mining, i.e., we count the number of input sequences (documents) in which S occurs (as opposed to the total number of occurrences of S). For $\sigma > 0$, we say that sequence S is (σ, γ)-frequent if $f_\gamma(S, \mathscr{D}) \geq \sigma$. The gap-constrained frequent sequence mining problem considered in this paper is as follows:

> Given a *support threshold* $\sigma \geq 1$, a *maximum-gap parameter* $\gamma \geq 0$, and a *length threshold* $\lambda \geq 2$, find the set $F_{\sigma,\gamma,\lambda}(\mathscr{D})$ of all (σ, γ)-frequent sequences in $\mathscr{D}$ of length at most λ. For each such sequence, also compute its frequency $f_\gamma(S, \mathscr{D})$.

[1]We indicate both sets and multisets using $\{ \}$; the appropriate type is always clear from the context. The operators $\uplus$, $\Cap$, and $\setminus^+$ correspond to multiset union, multiset intersection, and multiset difference.

For database $\mathscr{D} = \{ abcaaabc, abcbbabc, abcccabc \}$, we obtain $F_{3,0,2}(\mathscr{D}) = \{ a, b, c, ab, bc \}$, $F_{3,1,2}(\mathscr{D}) = \{ a, b, c, ab, ac, bc \}$, and $F_{3,2,2}(\mathscr{D}) = \{ a, b, c, ab, ac, bc, ca \}$.

2.2 MapReduce

MapReduce, developed by Dean and Ghemawat [8] at Google, 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 input 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.

2.3 Naïve Approach

A naïve approach to gap-constrained FSM in MapReduce modifies WordCount, which determines how often every word occurs in a document collection and is often used to explain how MapReduce works, as follows. In the map function, which is invoked on each input sequence, we emit all distinct γ-subsequences of length at most λ that occur in the input sequence. In the reduce function, we count how often every subsequence S has occurred, thus determining $f_\gamma(S, \mathscr{D})$, and emit it if frequent. The method is generally inefficient and not scalable since it creates and communicates large intermediate data: For example, if $|S| = n$ and $\lambda \geq n$, the naïve approach emits per input sequence $O(n^2)$ key-value pairs for $\gamma = 0$ and $O(2^n)$ key-value pairs for $\gamma \geq n$.

3. THE MG-FSM ALGORITHM

The idea behind item-based partitioning is to create one *partition* $\mathscr{P}_w$ for every σ-frequent item $w \in \Sigma$; we refer to w as the *pivot item* of partition $\mathscr{P}_w$. In the context of frequent itemset mining, item-based partitioning is exploited in the well-known FP-growth algorithm [12] as well as the distributed frequent itemset miners of [5, 16]. In our setting of frequent sequence mining, partition $\mathscr{P}_w$ is itself a sequence database—also called *projected database*—and captures relevant information about (some) frequent sequences containing pivot w. We first describe the MG-FSM algorithm in general. MG-FSM is based on the notion of w-equivalency, which we introduce in Sec. 3.3. In particular, w-equivalency is a necessary and sufficient condition for the correctness of MG-FSM, which we establish in Sec. 3.4.

3.1 Algorithm Overview

MG-FSM is divided into a preprocessing phase, a partitioning phase, and a mining phase; all phases are fully parallelized. In the *preprocessing phase*, we gather basic statistics about the data. In the *partitioning phase*, we construct w-equivalent partitions for all frequent items w in Σ. Each of these partitions is mined independently and in parallel in the *mining phase* using an FSM algorithm of choice. The final output is obtained by filtering the output of the FSM algorithm locally at each partition. MG-FSM is given as Alg. 1; the notation is described below and in Sec. 3.3.

Preprocessing. In the preprocessing phase, we compute the frequency of each item $w \in \Sigma$ and construct the set

Algorithm 1 The MG-FSM algorithm

```
Require: Sequence database $\mathscr{D}$, $\sigma$, $\gamma$, $\lambda$, f-list $F_{\sigma,0,1}(\mathscr{D})$
 1: MAP($T$):
 2: for all distinct $w \in T$ s.t. $w \in F_{\sigma,0,1}(\mathscr{D})$ do
 3:     Construct a sequence database $\mathscr{P}_w(T)$ that is
        $(w,\gamma,\lambda)$-equivalent to $\{ T \}$
 4:     For each $S \in \mathscr{P}_w(T)$, output $(w, S)$
 5: end for
 6:
 7: REDUCE($w$, $\mathscr{P}_w$):
 8: $F_{\sigma,\gamma,\lambda}(\mathscr{P}_w) \leftarrow \mathrm{FSM}_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$
 9: for all $S \in F_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$ do
10:     if $p(S) = w$ and $S \neq w$ then
11:         Output $(S, f_\gamma(S, \mathscr{P}_w))$
12:     end if
13: end for
```

$F_{\sigma,0,1}(\mathscr{D})$ of frequent items, commonly called *f-list*. This can be done efficiently in a single MapReduce job (by running a version of WordCount that ignores repeated occurrences of items within an input sequence). At this point, we can already output the set of frequent sequences of length 1; we subsequently focus on sequences of length 2 and above.

In MG-FSM, we use the f-list to establish a total order $<$ on Σ: Set $w < w'$ if $f_0(w, \mathscr{D}) > f_0(w', \mathscr{D})$; ties are broken arbitrarily. Thus "small" items are highly frequent, whereas "large" items occur rarely. Write $S \leq w$ if $w' \leq w$ for all $w' \in S$ and denote by $\Sigma^+_{\leq w} = \{ S \in \Sigma^+ : w \in S, S \leq w \}$ the set of all sequences that contain w but no items larger than w. Finally, denote by $p(S) = \min_{w \in S}(S \leq w)$ the *pivot item* of sequence S, i.e., the largest item in S. Note that $p(S) = w \iff w \in S \wedge S \leq w \iff S \in \Sigma^+_{\leq w}$. For example, when $S = abc$, then $S \leq c$ and $p(S) = c$; here, as well as in all subsequent examples, we assume order $a < b < c < \ldots$.

Partitioning phase (map). The partitioning and mining phases of MG-FSM are performed in a single MapReduce job. We construct partitions $\mathscr{P}_w$ in the map phase: For each distinct item w in each input sequence $T \in \mathscr{D}$, we compute a small sequence database $\mathscr{P}_w(T)$ and output each of its sequences with reduce key w. We require $\mathscr{P}_w(T)$ to be "(w, γ, λ)-equivalent" to T, see Sec. 3.3. For now, assume that $\mathscr{P}_w(T) = \{ T \}$; a key contribution of this paper lies in the refined construction of $\mathscr{P}_w(T)$.

Mining phase (reduce). The input to the mining phase, which is run in the reduce step, is given by

$$\mathscr{P}_w = \biguplus_{T \in \mathscr{D}, w \in T} \mathscr{P}_w(T),$$

which is automatically constructed by the MapReduce framework. We run an arbitrary FSM algorithm with parameters σ, γ, and λ on $\mathscr{P}_w$—denoted $\mathrm{FSM}_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$ in Alg. 1—to obtain the frequent sequences $F_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$ as well as their frequencies. Since every frequent sequence may be generated at multiple partitions, we filter the output of the FSM algorithm in order to produce each frequent sequence exactly once. In particular, we output sequence S at partition $\mathscr{P}_{p(S)}$, i.e., at the partition corresponding to its largest item. Observe that with our choice of $\mathscr{P}_w(T) = \{ T \}$, $f_\gamma(S, \mathscr{P}_w) = f_\gamma(S, \mathscr{D})$ for all sequences S such that $w \in S$, so that MG-FSM produces the correct result.

MG-FSM is reminiscent of the distributed frequent itemset mining algorithms of [5, 16]; the key difference lies in partition construction (line 3 of Alg. 1), i.e., in our notion of w-equivalency.

3.2 Example

Consider the database

$$\mathscr{D}_{\text{ex}} = \{ acb, dacbd, dacbddbca, bd, bcaddbd, addcd \} \quad (1)$$

and pivot c. Under our running assumption, we set $\mathscr{P}_c(T) = \{ T \}$ if $c \in T$ and $\mathscr{P}_c(T) = \emptyset$ otherwise. We thus obtain

$$\mathscr{P}_c = \{ acb, dacbd, dacbddbca, bcaddbd, addcd \}. \quad (2)$$

With such a partitioning, $\mathscr{P}_c$ is large so that there is substantial communication cost. Moreover, the FSM algorithm run on $\mathscr{P}_c$ in the mining phase produces a large number of sequences that do not pass the subsequent filter. For example, $F_{1,1,3}(\mathscr{P}_c)$ contains sequences da, dab, add and so on, all of which are filtered out (and, in fact, also produced at partition $\mathscr{P}_d$). Such redundant computation is wasteful in terms of computational costs. In what follows, we introduce the concept of w-equivalency, which will allow us to significantly reduce both communication and computational costs.

3.3 w-Equivalency

As mentioned above, w-equivalency is a necessary and sufficient condition for the correctness of MG-FSM; the flexibility implied by w-equivalency forms the basis of our partition construction algorithms of Sec. 4.

Say that a sequence S is a *pivot sequence* w.r.t. $w \in \Sigma$ if $p(S) = w$ and $2 \leq |S| \leq \lambda$. Denote by

$$G_{w,\gamma,\lambda}(T) = [F_{1,\gamma,\lambda}(\{ T \}) \cap \Sigma^+_{\leq w}] \setminus \{ w \}$$

the *set* of pivot sequences that occur in T, i.e., are γ-subsequences of T. If $S \in G_{w,\gamma,\lambda}(T)$, we say that T (w, γ, λ)-generates (or simply w-generates) S. For example,

$$G_{c,1,2}(acbfdeacfc) = \{ ac, cb, cc \}.$$

Recall that our choice of $<$ is based on the f-list, which ultimately aims to reduce variance in partition sizes: Partitions corresponding to highly frequent items are affected by many input sequences, but each input sequence generates few pivot sequences (e.g., a does not generate any pivot sequence in the previous example). In contrast, partitions corresponding to less frequent pivot items are affected by few input sequences, but each input sequence generates many pivot sequences (e.g., f).

We also extend the above definition to sequence databases as follows:

$$G_{w,\gamma,\lambda}(\mathscr{D}) = \biguplus_{T \in \mathscr{D}} G_{w,\gamma,\lambda}(T). \quad (3)$$

Note that $G_{w,\gamma,\lambda}(T)$ is a set, whereas $G_{w,\gamma,\lambda}(\mathscr{D})$ is a multiset. This difference is a consequence of our use of document frequency, i.e., we generate each subsequence at most once per input sequence but potentially many times per sequence database. We are now ready to define w-equivalency.

Definition 1. *Two sequence databases $\mathscr{D}$ and $\mathscr{P}_w$ are (w, γ, λ)-equivalent (or simply w-equivalent) if and only if*

$$G_{w,\gamma,\lambda}(\mathscr{D}) = G_{w,\gamma,\lambda}(\mathscr{P}_w).$$

Both databases thus generate the same (multiset of) pivot sequences.

Continuing the example of Sec. 3.2, observe that $\mathscr{P}_c$ as given in Eq. (2) is $(c, 1, 3)$-equivalent to $\mathscr{D}$ (see the discussion at the end of this section). However, so is partition

$$\mathscr{P}'_c = \{acb, acb, acb, bca, bca\}, \tag{4}$$

which is significantly smaller and contains many repeated sequences. In Sec. 4, we present a number of rewriting techniques that ultimately produce $\mathscr{P}'_c$ as given above.

The following lemma establishes that a w-equivalent database retains the frequency (w.r.t. γ) of pivot sequences; this property is exploited by our MG-FSM algorithm.

LEMMA 1. *If $\mathscr{D}$ and $\mathscr{P}_w$ are (w, γ, λ)-equivalent, then*

$$f_\gamma(S, \mathscr{P}_w) = f_\gamma(S, \mathscr{D})$$

for all $S \in \Sigma_w^+$ such that $2 \leq |S| \leq \lambda$.

PROOF. Denote by $f(T, \mathscr{A})$ the frequency of T in multiset $\mathscr{A}$. Note that f counts input sequences, whereas f_γ counts γ-subsequences. Pick an arbitrary pivot sequence S, i.e., $p(S) = w$ and $2 \leq |S| \leq \lambda$. Since $\mathscr{P}_w$ is (w, γ, λ)-equivalent to $\mathscr{D}$, we have

$$\begin{aligned}
&G_{w,\gamma,\lambda}(\mathscr{P}_w) = G_{w,\gamma,\lambda}(\mathscr{D}) \\
\Longrightarrow\ & f(S, G_{w,\gamma,\lambda}(\mathscr{P}_w)) = f(S, G_{w,\gamma,\lambda}(\mathscr{D})) \\
\Longleftrightarrow\ & |\{\, T \in \mathscr{P}_w : S \in G_{w,\gamma,\lambda}(T) \,\}| \\
&\quad = |\{\, T \in \mathscr{D} : S \in G_{w,\gamma,\lambda}(T) \,\}| \\
\Longleftrightarrow\ & |\{\, T \in \mathscr{P}_w : S \subseteq_\gamma T \,\}| = |\{\, T \in \mathscr{D} : S \subseteq_\gamma T \,\}| \\
\Longleftrightarrow\ & f_\gamma(S, \mathscr{P}_w) = f_\gamma(S, \mathscr{D}),
\end{aligned}$$

where applied the definition of $G_{w,\gamma,\lambda}(\mathscr{D})$ and the fact that $S \subseteq_\gamma T$ if and only if $S \in G_{w,\gamma,\lambda}(T)$ for our choice of S. □

Observe that the frequency $f_\gamma(S, \mathscr{P}_w)$ of any non-pivot sequence S does not affect w-equivalency and can thus be arbitrary and, in particular, *larger* than $f_\gamma(S, \mathscr{D})$. As we will see, this gives us more flexibility for constructing partitions while still maintaining correctness. Also note that a partition can be w-equivalent to $\mathscr{D}$ for more than one item w. The perhaps simplest, non-trivial partitioning that is w-equivalent to $\mathscr{D}$ is given by $\mathscr{P}_w = \{\, T \in \mathscr{D} : w \in T \,\}$, which corresponds to the partitioning used in the beginning of this section. It is easy to see that there is an infinite number of sequence databases $\mathscr{P}_w$ such that $\mathscr{P}_w$ is (w, γ, λ)-equivalent to $\mathscr{D}$. All these databases agree on the multiset $G_{w,\gamma,\lambda}(\mathscr{P}_w)$ of generated pivot sequences.

3.4 Correctness of MG-FSM

The following theorem establishes the correctness of MG-FSM.

THEOREM 1. *MG-FSM outputs each frequent sequence $S \in F_{\sigma,\gamma,\lambda}(\mathscr{D})$ exactly once and with frequency $f_\gamma(S, \mathscr{D})$. No other sequences are output.*

PROOF. We first show that if MG-FSM outputs a sequence S, it does so exactly once. If $|S| = 1$, S is output in the preprocessing phase but not in the mining phase (due to line 10 of Alg. 1). If $|S| > 1$, S is output at partition $\mathscr{P}_{p(S)}$ in the mining phase (passes line 10) but at no other partitions (does not pass line 10).

Now fix some $S \in F_{\sigma,\gamma,\lambda}(\mathscr{D})$. We show that MG-FSM outputs S with correct frequency. If $|S| = 1$, then S occurs in the f-list and is output with correct frequency in the preprocessing phase. Assume $|S| > 1$ and set $w = p(S)$. We claim that S is output with correct frequency at partition $\mathscr{P}_w$ during the reduce phase of Alg. 1. First, observe that $\mathscr{P}_w$ is w-equivalent to $\mathscr{D}$ since

$$\begin{aligned}
G_{w,\gamma,\lambda}(\mathscr{P}_w) &= \biguplus_{T \in \mathscr{D}} G_{w,\gamma,\lambda}(\mathscr{P}_w(T)) = \biguplus_{T \in \mathscr{D}, w \in T} G_{w,\gamma,\lambda}(T) \\
&= \biguplus_{T \in \mathscr{D}} G_{w,\gamma,\lambda}(T) = G_{w,\gamma,\lambda}(\mathscr{D}).
\end{aligned}$$

Here the first equality follows from Eq. (3) and line 4 of Alg. 1, the second equality from the definition of w-equivalency and line 3, and the third equality from the fact that $G_{w,\gamma,\lambda}(T) = \emptyset$ if $w \notin T$. From Lemma 1, we immediately obtain $f_\gamma(S, \mathscr{P}_w) = f_\gamma(S, \mathscr{D})$. Since therefore $S \in F_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$, S is found by the FSM algorithm run in line 8 of Alg. 1 and the assertion follows.

Now fix some $S \notin F_{\sigma,\gamma,\lambda}(\mathscr{D})$. We show that MG-FSM does not output S. If $|S| = 1$, then $S = w$ and $f_\gamma(w) < \sigma$ so that S is neither output in the preprocessing phase (not σ-frequent) nor in the mining phase (filtered out in line 10). If $|S| > \lambda$, we also do not output S since it is too long to be produced by $\text{FSM}_{\sigma,\gamma,\lambda}$ in line 8 of Alg. 1. Finally, if $2 \leq |S| \leq \lambda$, then S could potentially be output at partition $\mathscr{P}_w$, where $w = p(S)$. However, by the arguments above, we have $f_\gamma(S, \mathscr{P}_w) = f_\gamma(S, \mathscr{D}) < \sigma$, so that $S \notin \text{FSM}_{\sigma,\gamma,\lambda}(\mathscr{P}_w)$. □

4. PARTITION CONSTRUCTION

Recall that MG-FSM rewrites each input sequence T into a small sequence database $\mathscr{P}_w(T)$ for each $w \in T$. We have shown that MG-FSM produces correct results if $\mathscr{P}_w(T)$ is w-equivalent to T, i.e., $G_{w,\gamma,\lambda}(T) = G_{w,\gamma,\lambda}(\mathscr{P}_w(T))$. In this section, we propose rewriting methods that aim to minimize the overall size of $\mathscr{P}_w(T)$. In fact, the smaller $\mathscr{P}_w(T)$, the less data needs to be communicated between the map and reduce phases of MG-FSM, and the less work needs to be performed by the FSM algorithm in the mining phase.

To see the need for rewriting, assume that we simply set $\mathscr{P}_w(T) = \{\, T \,\}$ as before. Such an approach is impractical for a number of reasons. First, input sequence T is replicated to d partitions, where d corresponds to the number of distinct items in T; this is wasteful in terms of communication cost. Second, every frequent sequence $S \in F_{\sigma,\gamma,\lambda}(\mathscr{D})$ will be computed multiple times: If S contains d distinct items, it is first computed by the FSM algorithm at each of the d corresponding partitions but then output at partition $\mathscr{P}_{p(S)}$ only; this is wasteful in terms of computational costs. Finally, the choice of $\mathscr{P}_w(T) = \{\, T \,\}$ leads to highly imbalanced partition sizes: Partitions corresponding to frequent items are large (since these items occur in many input sequences), whereas partitions corresponding to less frequent items will be smaller (since these items occur in less input sequences).

To get some insight into potential rewrites, consider input sequence $T = cbdbc$. Each of the following sequence databases is $(c, 0, 2)$-equivalent to T: $\mathscr{P}_1 = \{\, cbdbc \,\}$, $\mathscr{P}_2 = \{\, cbbc \,\}$, $\mathscr{P}_3 = \{\, cbc \,\}$, and $\mathscr{P}_4 = \{\, cb, bc \,\}$. Note that in $\mathscr{P}_4$, the frequencies of both b and c increased by one; our notion of w-equivalency allows for such cases. It is not obvious which of these databases is best overall. On the one hand, $\mathscr{P}_3$ appears to be preferable to $\mathscr{P}_1$ and $\mathscr{P}_2$ since it contains less items. On the other hand, the maximum sequence length $\mathscr{P}_4$ is smaller than the one of $\mathscr{P}_3$ (2 vs. 3).

In what follows, we propose a number of properties that are useful in partition construction: minimality, irreducibility, and inseparability. Since it is computationally expensive to satisfy all of these properties, we give efficient rewriting algorithms that satisfy weaker, practical versions.

4.1 Minimality

In this and subsequent sections, we assume that we are given an input sequence T and aim to produce a w-equivalent sequence database $\mathscr{P}_w(T)$. Unless otherwise stated, our running assumption is that $\mathscr{P}_w(T)$ is w-equivalent to T.

Minimality, as defined below, ensures that $\mathscr{P}_w(T)$ contains no *irrelevant sequences*, i.e., sequences that do not generate a pivot sequence.

Definition 2 (Minimality).
A sequence database $\mathscr{P}_w(T)$ is (w, γ, λ)-minimal *if*

$$G_{w,\gamma,\lambda}(S) \neq \emptyset \quad \textit{for all } S \in \mathscr{P}_w(T).$$

Clearly, any sequence $S \in \mathscr{P}_w(T)$ for which $G_{w,\gamma,\lambda}(S) = \emptyset$ does not contribute to w-equivalency; we can thus safely prune such irrelevant sequences. Minimality also allows us to prune entire input sequences, even if they contain the pivot. Consider for example input sequence $T = addcd$ with pivot c. For $\gamma = 1$ (and any $\lambda \geq 2$), T does not generate any pivot sequences so that we can set $\mathscr{P}_c(T) = \emptyset$. Note that, for this reason, the choice of $\mathscr{P}_w(T) = \{ T \}$ does not guarantee minimality. For any $\gamma > 1$, T does generate pivot sequence ac so that $\mathscr{P}_c(T)$ becomes non-empty.

In general, we prune sequences that either do not contain the pivot or in which each occurrence of a pivot is surrounded by sufficiently many irrelevant items, i.e., items larger than the pivot. In particular, $G_{w,\lambda,\gamma}(T) \neq \emptyset$ if and only if there is at least one occurrence of some item $w' \leq w$ within distance $\gamma + 1$ of an occurrence of pivot w. Length parameter λ does not influence minimality. If a sequence is irrelevant for some choice of γ, it is also irrelevant for all $\gamma' < \gamma$; the opposite does not hold in general. Thus minimality pruning is most effective when γ is small.

4.2 Irreducibility

Irreducibility is one of the main concepts employed by MG-FSM. We say that a sequence is irreducible if there is no shorter way to write it.

Definition 3 (Irreducibility).
A sequence S is (w, γ, λ)-irreducible *if there exists no sequence S' with length $|S'| < |S|$ such that*

$$G_{w,\gamma,\lambda}(S) = G_{w,\gamma,\lambda}(S').$$

If such a sequence S' exists, we say that S *reduces* to S'. Moreover, we say that $\mathscr{P}_w(T)$ is irreducible if all sequences $S \in \mathscr{P}_w(T)$ are irreducible. Consider for example the sequences $S = acdeb$ and $S' = acb$ and pivot c. Here S' is obtained from S by removing all *irrelevant items*, i.e., all items larger than the pivot. Then S' is a $(c, 2, 2)$-reduction of S; it is not, however, a $(c, 1, 2)$-reduction of S. This is because $cb \in G_{c,1,2}(S')$ but $cb \notin G_{c,1,2}(S)$. Thus, perhaps contrary to expectation, we cannot simply remove all irrelevant items to obtain a reduced sequence: Whether or not an irrelevant item can be dropped depends on the particular choice of γ and λ. Note that the shortest $(c, 1, 2)$-reduction of S is given by ac.

We can reduce sequences in more sophisticated ways than by simply removing irrelevant items. For example, $S = cbac$ can be $(c, 0, 2)$-reduced to acb, but it cannot be $(c, 0, 2)$-reduced to any sequence $S' \subset_\infty S$. Thus reduction can be non-monotonic, i.e., may require reordering of items. As an additional example, consider the sequence $S = acadac$, which $(c, 0, 2)$-generates sequence ac twice. Since repeated generations do not affect w-equivalency, we can reduce S to aca. Both detection of non-monotonic reductions and (exhaustive) detection of repeatedly generated pivot sequences appear computationally challenging. Since we perform sequence reduction for every input sequence, we need it to be extremely efficient. We thus make use of a weaker form of irreducibility, which does not consider such sophisticated rewrites.

To avoid confusion with repeated items, we explain our reduction techniques using indexes instead of items. Let $T = s_1 \cdots s_l$ and consider pivot w. We say that index i is *w-relevant* if s_i is w-relevant, i.e., $s_i \leq w$; otherwise index i is w-irrelevant. In sequence $abddc$, for example, indexes 3 and 4 are c-irrelevant. Since irrelevant indexes do not contribute to a pivot sequence, it suffices to keep track of the fact that an index i is irrelevant, i.e., we do not need to retain value s_i. We thus replace all items at irrelevant indexes by a special *blank symbol*, denoted "␣"; in what follows, we assume that $w <$ ␣ for all $w \in \Sigma$. Using blanks, sequence $abddc$ is written as ab␣␣c (for pivot c). As discussed in Sec. 5, our use of a blank symbol is helpful in an implementation of MG-FSM since it enables effective compression of irrelevant items (e.g., $abddc$ can be written as ab␣^2c).

In what follows, we describe a number of reductions that reduce T by removing items or blanks (while still maintaining correctness). Our first reduction, termed *unreachability reduction*, removes unreachable items, i.e., items that are "far away" from any pivot. Fix an input sequence $T = s_1 \cdots s_l$ and pick any index $1 \leq i \leq |T|$. To determine whether index i is unreachable, we consider the "distance" of i to its surrounding pivots. Suppose that there is a pivot at an index $i' < i$, i.e., $s_{i'} = w$, and denote by i_{prev} the largest such index. Informally, the *left distance* $l_{w,\gamma,\lambda}(i \mid T)$ of index i is given by the smallest number of items that we need to "step onto" when moving from i_{prev} to i via (1) relevant items and (2) by skipping at most γ items in each step. If no such path exists or if its length is larger than λ, we set $l_{w,\gamma,\lambda}(i \mid T) = \infty$. Similarly, we define the *right distance* $r_{w,\gamma,\lambda}(i \mid T)$ of index i as the distance to the closest pivot to the right of index i. A formal definition of left and right distances is given in App. A. For example, we obtain the following left and right distances for pivot c, $\gamma = 1$, and $\lambda = 4$:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
s_i	***c***	***a***	␣	***b***	***a***	***b***	␣	***a***	␣	***c***	␣	␣	***a***	␣
$l_{c,1,4}$	1	2	(2)	3	4	4				1	(2)	(2)		
$r_{c,1,4}$	1			4	4	3	(3)	2	(2)	1				

Here blank entries indicate infinite distance and entries in parentheses indicate irrelevant items (which cannot be "stepped onto"). For example, the closest pivot to the left of index 5 occurs at index 1. We can reach index 5 from index 1 via the items at the four indexes $1 \cdot 2 \cdot 4 \cdot 5$ (but not via $1 \cdot 3 \cdot 5$ because index 3 is irrelevant); thus the left distance of index 5 is four.

Definition 4 (Reachability). *Let $T = s_1 \cdots s_l$. Index i is (w,γ,λ)-reachable if*

$$\min\{\, l_{w,\gamma,\lambda}(i \mid T), r_{w,\gamma,\lambda}(i \mid T)\,\} \le \lambda.$$

Continuing the example above, indexes 1–12 are $(c,1,4)$-reachable, indexes 1–4 and 6–12 are $(c,1,3)$-reachable, and indexes 1–3 and 8–12 are $(c,1,2)$-reachable.

Lemma 2 (Unreachability reduction). *Let $T = s_1 \cdots s_l$ and denote by I the set of all (w,γ,λ)-unreachable indexes in T. Then*

$$G_{w,\gamma,\lambda}(T) = G_{w,\gamma,\lambda}(T_{-I}),$$

where T_{-I} is obtained by removing the items at indexes I from T.

A proof of this lemma is given in App. A. The lemma asserts that, in fact, we can simply remove all unreachable indexes at once. In our example, we obtain $ca\_a\_c\_\_$ for $\lambda = 2$ (removing indexes 4–7, 13, and 14) and $ca\_bb\_a\_c\_\_$ for $\lambda = 3$ (removing indexes 5, 13, and 14).

The computational cost of our unreachability reduction mainly depends on the time required to compute left and right distances. This can be done efficiently as follows: We compute the left distances in a forward scan of T, and right distances in a subsequent backward scan. During each scan, we keep track of the position i and distance d of the most recently processed relevant item; we also keep track of the index i' of the most recently processed relevant item at distance $k-1$. This information is sufficient to compute the distance of the currently processed item. With this approach, computation of distances takes time linear in $|T|$.

Reconsider the sequence $T = ca\_a\_c\_\_$ from above. Clearly, the two blanks at the end do not carry useful information so that we would like to reduce T to ca_a_c. The following lemma asserts that we can do so, i.e., we can drop prefixes and suffixes of irrelevant items.

Lemma 3 (Prefix/suffix reduction).

$$G_{w,\gamma,\lambda}(\_^{l_1} T \_^{l_2}) = G_{w,\gamma,\lambda}(T).$$

Prefix/suffix reduction is particularly effective in conjunction with unreachability reduction. In fact, irrelevant items that are not part of a prefix or suffix in T can become so after removing unreachable items. In our ongoing example, this is the case for the irrelevant items at indexes 11 and 12, which became part of the suffix only after the removal of indexes 13 and 14. In fact, if T contains exactly one pivot, and T' is obtained from T by an unreachability reduction followed by a prefix/suffix reduction, one can show that $|T'| \le 2(\gamma+1)\lambda$, a quantity that is independent of $|T|$.

The following lemma asserts that we can shrink long sequences of blanks.

Lemma 4 (Blank reduction). *For $k > \gamma + 1$,*

$$G_{w,\gamma,\lambda}(T_1 \_^{k} T_2) = G_{w,\gamma,\lambda}(T_1 \_^{\gamma+1} T_2).$$

Thus every sequence of $k > \gamma + 1$ blanks can be replaced by exactly $\gamma + 1$ blanks. Note that blank reduction can be effective only when T contains multiple occurrences of the pivot (since otherwise T does not contain more than γ blanks after our unreachability and prefix/suffix reductions).

The above reductions are not complete, i.e., they do not necessarily produce irreducible sequences. For example, sequence ca_a_c is $(c,1,2)$-reducible to ca_ac (and, in fact, to aca); this reduction is not covered by our techniques. In our experiments, however, we found that the simple reduction techniques described above already lead to a significant reduction of partition sizes.

4.3 Aggregation

Reconsider the example database $\mathscr{D}_{\text{ex}}$ given in Eq. (1). If we apply all the reduction techniques above, we obtain

$$\mathscr{P}''_c = \{\, acb, acb, acb\_\_bca, bca \,\}$$

for pivot c, $\sigma = 1$, $\gamma = 1$, and $\lambda = 3$. Observe that sequence acb is repeated in $\mathscr{P}''_c$, even though $\mathscr{D}_{\text{ex}}$ does not contain any repeated sequences. To reduce communication cost, we can aggregate such repeated sequences and represent them using (sequence, frequency)-pairs. Thus, we obtain

$$\mathscr{P}'''_c = \{\, (acb, 2), (acb\_\_bca, 1), (bca, 1) \,\}.$$

Compression of repeated sequences can be performed efficiently by exploiting the *combine* functionality of MapReduce. When the FSM algorithm run in the mining phase is able to exploit frequency information, computational costs and memory consumption may also reduce; see Sec. 5 for details.

4.4 Inseparability

Recall the set $\mathscr{P}''_c$ above. We show below that sequence $acb\_\_bca \in \mathscr{P}''_c$ can be "split" into two sequences acb and bca without sacrificing $(c,1,3)$-equivalency to $\mathscr{D}_{\text{ex}}$. Such sequence splitting, which we refer to as *separation*, is meant to increase the effectiveness of aggregation. In fact, if we perform the split above and aggregate, we obtain partition $\{\, (acb, 3), (bca, 2) \,\}$, which is compact and constitutes an aggregated version of the partition of Eq. (4) that we promised to obtain in Sec. 3.3.

Definition 5 (Separability). *An input sequence T is weakly (w,γ,λ)-separable if there exist sequences T_1 and T_2 such that $G_{w,\gamma,\lambda}(T) = G_{w,\gamma,\lambda}(\{\, T_1, T_2 \,\})$, $G_{w,\gamma,\lambda}(T_1) \ne \emptyset$, and $G_{w,\gamma,\lambda}(T_2) \ne \emptyset$; otherwise it is weakly (w,γ,λ)-inseparable. T is strongly (w,γ,λ)-separable (or simply (w,γ,λ)-separable) if additionally $|T_1| + |T_2| \le |T|$.*

Note that separation is possible only because we allow for an *increase* of frequencies on non-pivot sequences in $\mathscr{P}_w(T)$. If a sequence is w-separable, we can safely write it in terms of multiple shorter sequences, which we refer to as *splits*. As indicated above, both strong and weak separation improve the effectiveness of aggregation, and strong separation additionally reduces the overall partition size.

Revisiting $S = acb\_\_bca$, we observe that S is $(c,1,3)$-separable into splits $\{\, acb, bca \,\}$. In general, one can test for weak separability as follows: Construct the set $G_{w,\gamma,\lambda}(S)$ and create a graph (V,E), where $V = G_{w,\gamma,\lambda}(S)$ and edge $(S_1, S_2) \in E$ if there exists an item $w' \in \Sigma$ and a sequence S' of form ww' or $w'w$ such that $S' \subseteq_0 S_1$ and $S' \subseteq_0 S_2$. If the resulting graph is connected, S is not (even weakly) (w,γ,λ)-separable. Intuitively, this is because any input sequence that generates S_i, $i \in \{\,1,2\,\}$, will also generate S'. Since S' is a pivot sequence, however, we must not generate it in more than one split, which implies that S_1 and S_2 must be generated by the same split. In our example, we have $G_{c,1,3}(S) = \{\, ac, acb, cb, bc, bca, ca \,\}$; the corresponding graph has two connected components so that S is $(c,1,3)$-separable. As a final remark, one can show that any sequence S can be separated into k splits, where k is the number of connected components in the graph corresponding to the pivot sequences generated by S.

As with reduction, there are quite sophisticated cases of separable sequences. As a pathological example, it is possible that an irreducible sequence is weakly separable only into irreducible sequences of larger length. Consider, for example, sequence $T = abc$, pivot c, $\gamma = 1$, and $\lambda = 3$. T is irreducible and can be separated into splits $\{ ac, a␣bc \}$, in which each sequence is again irreducible. Separations such as this one appear counterproductive. Moreover, the weak separation detection method outlined above is too expensive in practice since it generates $G_{w,\gamma,\lambda}(S)$. In what follows, we present a simple separation technique, called *blank separation*, that is efficient and handles cases such as sequence $acb␣␣bca$ discussed above.

Assume S is of form $S_1␣^{k_1}S_2␣^{k_1}\cdots␣^{k_{n-1}}S_n$, where $k_i > \gamma$ for $1 \leq k < n$, i.e., consists of subsequences separated by sufficiently many blanks. Our blank separation technique first breaks up S into the set $B(S) = \{ S_1, \ldots, S_n \}$; irrelevant subsequences (i.e., subsequences that do not generate a pivot sequence) are not included into $B(S)$. For example, $B(acb␣␣bca) = \{ acb, bca \}$ for $\gamma = 1$. Since the S_i are separated by at least $\gamma + 1$ gaps in S, every pivot sequence in S is generated by one or more of the S_i (i.e., it does not span multiple S_i's). Thus $D(G_{w,\gamma,\lambda}(B(S))) = G_{w,\gamma,\lambda}(S)$, where $D(A)$ denotes the set of distinct sequences in multiset A. We now consider the S_i as potential splits and proceed as above: We first construct a graph $G = (V, E)$, where $V = D(B(S))$ and $(S_i, S_j) \in E$, $i \neq j$, if S_i and S_j generate a joint pivot sequence.[2] Denote by $B_1, \ldots, B_k$ the connected components of G. We output a single sequence S'_i for component B_i by *stitching* the subsequences in B_i with sufficiently many blanks:

$$S'_i = B_{i,1}␣^{\gamma+1}B_{i,2}␣^{\gamma+1}\cdots␣^{\gamma+1}B_{i,|B_i|},$$

where $B_{i,j}$ refers to the j-th sequence in B_i. In our ongoing example, we have $B_1 = \{ acb \}$, $B_2 = \{ bca \}$ and thus $S'_1 = acb$ and $S'_2 = bca$. As another example, consider the sequence $bcb␣bca␣ac␣bca␣c$ for $\gamma = 0$ and $\lambda = 3$. Then $B_1 = \{ bcb, bca \}$ and $B_2 = \{ ac \}$, and thus $S'_1 = \{ bcb␣bca \}$ and $S'_2 = \{ ac \}$. Thus blank separation is able to detect (to some extent) repeated subsequences as well as irrelevant subsequences.

In combination with our reduction techniques, blank separation is only effective if S contains more than one occurrence of a pivot. This is because our reduction techniques ensure that otherwise S does not contain more than γ consecutive blanks. On datasets in which items are rarely repeated, blank separation is therefore not effective. For example, we found that in our experiments on text data (where each sequence corresponded to a single sentence), the overall effect of blank separation was marginal.

4.5 Summary

To summarize, we obtain $\mathscr{P}_w(T)$ from input sequence T as follows. We first check whether T is relevant for pivot w using the minimality test described in Sec. 4.1. If not, we set $\mathscr{P}_w(T) = \emptyset$. Otherwise, we run a backward scan of T to obtain the right distances of all indexes. We then perform a forward scan of T in which we simultaneously (1) compute the left distances, (2) perform unreachability reduction, (3) replace irrelevant items by blanks, (4) perform prefix/suffix and blank reduction, and (5) break up the sequence into subsequences separated by $\gamma + 1$ gaps. These subsequences are then fed into our blank separation method, which ultimately outputs $\mathscr{P}_w(T)$.

[2]This can be tested efficiently by considering only the left and right $(\gamma+1)$-neighborhood of each occurrence of a pivot.

5. IMPLEMENTATION

In this section, we give some guidance into how to implement MG-FSM efficiently. The source code of our implementation is available at [19].

Compression. Careful compression of all intermediate sequences significantly boosted performance in our experiments. In particular, we assign an integer item identifier to each item and represent sequences compactly as arrays of item identifiers. We compress each such array using variable-byte encoding [29]. To make most effective use of this technique, we order item identifiers by descending item frequency (as obtained from the f-list). Moreover, we replace irrelevant items by blanks (identifier -1) and use a form of run-length encoding [29] to represent consecutive sequences of blanks (e.g., ␣␣ by -2).

Grouping partitions. Instead of creating a single partition for each distinct pivot, we greedily combine multiple pivots into a single partition such that each partition contains $\approx m$ sequences or more (e.g., $m = 10\,000$). Grouping is performed by scanning the f-list in descending order of frequency and combining items until their aggregate frequency exceeds m.

Aggregation. To implement aggregation (Sec. 4), we make use of both the combine function and the secondary sort facility of Hadoop. Recall Alg. 1, in which we output pairs of form (w, S), where w is a pivot and S is a sequence. In our actual implementation, we output (sequence, frequency)-pairs, i.e., pairs of form $(w \cdot S,\ 1)$ instead. We customize the partitioning function of MapReduce to ensure that w is used as partitioning key as before. This representation allows us to use the combine function to aggregate multiple occurrences of the same sequence locally. We perform the final aggregation in the reduce function before invoking our FSM method (exploiting secondary sort to avoid buffering).

Frequent Sequence Mining. MG-FSM can use any existing FSM method to mine one of its partitions. In our implementation, we use a method inspired by GSP [23] and SPADE [31]. Like the former, our method is based on a level-wise approach and generates frequent k-sequences from the frequent $(k-1)$-sequences; like the latter, our method operates on what is essentially an inverted index for sequences. Initially, while reading the sequences of a partition to be processed, our method builds an inverted index that maps 2-sequences to their offsets in the input sequences. We can then emit all frequent 2-sequences and remove infrequent ones from the index. Afterwards, our method iteratively combines frequent $(k-1)$-sequences by intersecting the corresponding inverted index lists to construct an inverted index for k-sequences. Frequent k-sequences can be emitted immediately once they have been determined. The inverted index for $(k-1)$-sequences can be deleted at the end of each iteration. Our implementation is aware of the aggregation optimization described in Sec. 4, i.e., it can operate directly on input sequences along with their aggregate frequency. Although Java-based, our implementation avoids

	ClueWeb	New York Times
Average length	19	19
Maximum length	20 993	21 174
Total sequences	1 135 036 279	53 137 507
Total items	21 565 723 440	1 051 435 745
Distinct items	7 361 754	1 577 233
Total bytes	66 181 963 922	3 087 605 146

Table 1: Dataset characteristics

object orientation to the extent possible. Inverted index lists, for instance, are encoded compactly as byte arrays using the compression techniques described above.

6. EXPERIMENTAL EVALUATION

We conducted an extensive experimental study in the context of text mining on large real-word datasets. In particular, we investigated the effectiveness of the various partition construction techniques of Sec. 4, studied the impact of parameters σ, γ, and λ, compared MG-FSM to both the naïve algorithm and a state-of-the-art n-gram miner, and evaluated weak and strong scalability of MG-FSM. We found that most of our optimizations for partition construction were effective; a notable exception was blank separation, which did not provide substantial efficiency gains on our textual datasets (see the discussion at the end of Sec. 4.4). MG-FSM outperformed the naïve approach by multiple orders of magnitude and was competitive to state of the art n-gram miners. Our scalability experiments suggest that MG-FSM scales well as we add more compute nodes and/or increase the dataset size.

6.1 Experimental Setup

We implemented MG-FSM [19] as well as the Naïve method of Sec. 2.3 in Java. We also obtained a Java implementation of Suffix-σ [3], a state-of-the-art n-gram miner, from its authors. Unless stated otherwise, we performed all of our rewriting steps for MG-FSM.

Hadoop cluster. We ran our experiments on a local cluster consisting of ten Dell PowerEdge R720 computers connected using a 10 GBit Ethernet connection. Each machine has 64GB of main memory, eight 2TB SAS 7200 RPM hard disks, and two Intel Xeon E5-2640 6-core CPUs. All machines ran Debian Linux (kernel version 3.2.21.1.amd64-smp), Oracle Java 1.6.0_31, and use the Cloudera cdh3u0 distribution of Hadoop 0.20.2. One machine acted as the Hadoop master node, while the other nine machines acted as worker nodes. The maximum number of concurrent map or reduce tasks was set to 10 per worker node. All tasks launched with 4 GB heap space.

Datasets. We used two real-world datasets for our experiments, see Table 1. The first dataset was the New York Times corpus (NYT) [27], which consists of over 1.8 million newspaper articles published between 1987 and 2007. The second dataset was a subset of ClueWeb [6] (CW), which consists of 50 million English pages; this well-defined subset is commonly referred to as ClueWeb09-T09B and also used as a TREC Web track dataset. We performed sentence detection using Apache OpenNLP and applied boilerplate detection and removal as described in [15]. Both datasets were represented compactly as sequences of item identifiers as described in Sec. 5.

Methodology. We used the following measures in our evaluation. First, we measured the total time elapsed between launching a method and receiving the result (all methods use a single MapReduce job). To provide more insight into potential bottlenecks, we also broke down total time into time spent for the map phase, shuffle phase, and reduce phase. Since these phases are overlapping in MapReduce, we report the elapsed time until finishing each phase; e.g., the time until the last map job finishes. Second, we measured the total number of bytes received by reducers. Note that when a combiner is used, this number is usually smaller than the total number of bytes emitted by the mappers. All measurements were averaged over three independent runs.

6.2 Partition Construction

We first evaluated the effectiveness of the compression and rewriting techniques of MG-FSM for partition construction. We used the following settings: *basic* (irrelevant items were replaced by blanks), *compressed* (consecutive blanks were compressed and leading/trailing blanks are removed), *reduced* (unreachable items were removed), *aggregated* (identical sequences were aggregated), and *separated* (blank separation was performed). We applied these optimizations in a stacked manner; e.g., when using the aggregation optimization, we also removed unreachable items and compressed consecutive blanks.

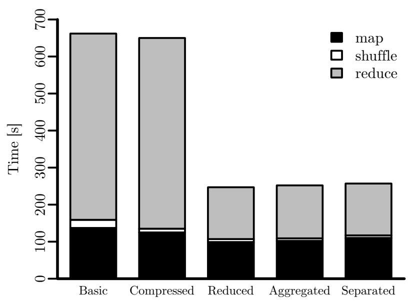
We used both NYT and CW to explore the different rewriting techniques. The results are shown in Figs. 1(a), 1(b), and 1(c), which also give the parameter settings. As can be seen, the removal of unreachable items resulted in a significant runtime improvement on both datasets, reducing the total time by a factor of up to 6 (for CW). For the smaller NYT dataset, map tasks are relatively inexpensive throughout and our techniques mainly reduce the runtime of the much more costly reduce operations. For CW, both map and reduce runtimes are significantly reduced, the former mainly due to the large reduction in transferred bytes (Fig. 1(c)). Aggregation is effective for CW, reducing the total number of bytes received by the reducers by more than 70 GB (28%). We also ran Naïve for NYT dataset (not shown); the algorithm finished after 225 minutes. In contrast, MG-FSM completes after 4 minutes and is thus more than 50 times faster.

6.3 Mining n-Grams

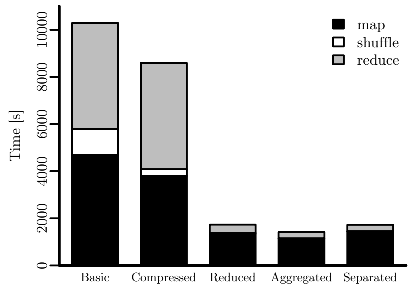
In our next experiments, we investigated the performance of MG-FSM n-gram mining ($\gamma = 0$) and compared it against both the Naïve method from Sec. 2 and a state-of-the-art approach called Suffix-σ [3]. Since Naïve could not handle the CW dataset in reasonable time, we focused on the NYT dataset. We ran sequence mining in three different configurations of increasing output size; the results are shown in Figs. 1(d) and 1(e) (log-scale). For the "easier" settings ($\sigma = 100, \lambda = 5$ and $\sigma = 10, \lambda = 5$), MG-FSM achieved an order of magnitude faster performance than Naïve. For the "harder" setting ($\sigma = 10$, $\lambda = 50$), MG-FSM was two orders of magnitudes faster than Naïve. MG-FSM also outperformed Suffix-σ in all settings (up to a factor of 1.6x faster). The total bytes transferred between the map and reduce phases is depicted in Fig. 1(e); it is smallest for MG-FSM.

6.4 Impact of Parameter Settings

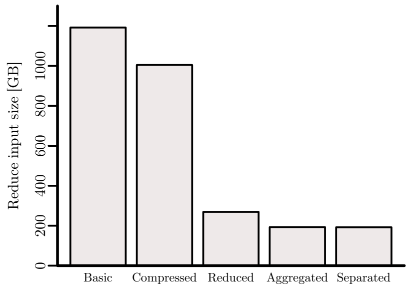
We studied the performance of MG-FSM as we varied the minimum support σ, the maximum gap γ, and the maximum



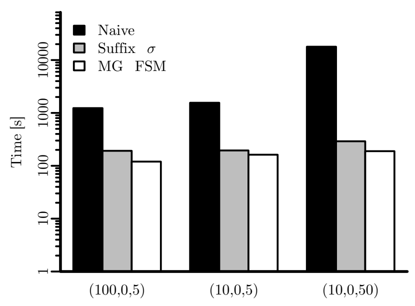
(a) NYT ($\sigma = 100, \gamma = 1, \lambda = 5$)



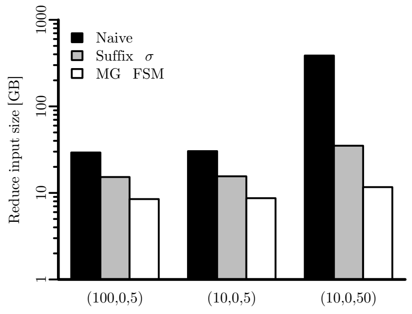
(b) CW ($\sigma = 1\,000, \gamma = 0, \lambda = 5$)



(c) CW ($\sigma = 1\,000, \gamma = 0, \lambda = 5$)



(d) NYT (σ, γ, λ)



(e) NYT (σ, γ, λ)

Figure 1: Impact of partition construction (a–c) and performance for n-gram mining (d–e)

length λ. We use the NYT dataset throughout and set the default values to $\sigma = 100$, $\gamma = 1$, and $\lambda = 5$.

We first studied how the minimum support σ affects performance by increasing its value from 10 to 10 000. The results are shown in Fig. 2(a). The map phase, which performs the rewriting of the input sequences, took roughly the same time for all different values of σ. This time mainly consists of the cost of scanning the input sequences, which is independent of σ. The reduce time, however, dropped significantly as we increased the minimum support, mainly because the mining cost and output size reduced (the slowest reduce task took 31s for $\sigma = 10\,000$). Due to the relatively large fraction of time spent in the map phase for large values of σ, we do not expect any significant runtime improvements if we increased σ further.

Second, we increased the maximum gap γ from 0 to 4. As we can see in Fig. 2(b), γ strongly affected reduce time, while the impact on map time was again not significant. This was partly due to the larger number of bytes received by the reducers (see Fig. 2(c)) and also because mining becomes harder when the output size increases: The total number of frequent sequences increased from 1 985 702 for $\gamma = 0$ to 51 166 966 for $\gamma = 4$.

Finally, we studied how the maximum length λ affects MG-FSM varying its value from 5 to 20. As Fig. 2(d) shows, λ had little effect on the map operations. Reduce time increased with increasing values of λ. This effect was less pronounced for larger values of λ (say, $\lambda \geq 10$) because there are fewer sequences of at least that length in the data (the average input sequence length is 19; see Table 1).

6.5 Scalability

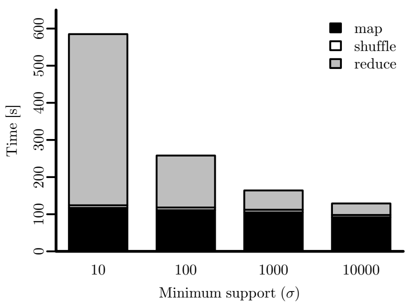
In our final set of experiments, we explored the scalability of MG-FSM. To evaluate strong scalability, we ran MG-FSM on a fixed dataset using 2, 4, and 8 worker nodes ($\sigma = 1\,000, \gamma = 1, \lambda = 5$). In order to finish the experiment in reasonable time on 2 nodes, we used a 50% sample of CW (consisting of more than half a billion input sequences). Our results are shown in Fig. 2(e). MG-FSM exhibited linear scalability as we increased the number of available machines, managing to equally decrease the times for the map and reduce tasks. The ability of MG-FSM to scale up can be credited to the large number of partitions that can be processed and mined independently.

We also performed a weak scalability experiment for MG-FSM, in which we simultaneously increased the number of available machines (2, 4, 8) and the size of the sequence database (25%, 50%, 100% of CW). In the ideal case, the total time remains constant as we scale out. As Fig. 2(f) shows, this is almost true, but we observe a small increase in runtime on 8 worker nodes (around 20%). This is because doubling the size of the input sequence database can increase the number of output sequences by a factor larger than 2. In this specific case, 50% of ClueWeb generated 6M frequent sequences, whereas the full corpus generated 13.5M frequent sequences (an 2.25x increase).

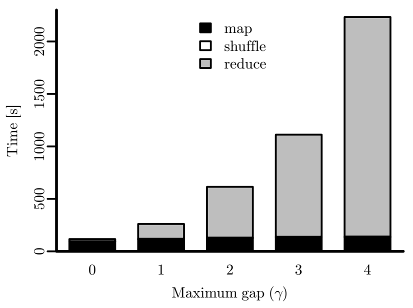
7. RELATED WORK

We now relate the ideas put forward in this paper to existing prior work. Prior approaches can be coarsely categorized with respect to the type of pattern being mined (frequent itemsets, frequent sequences, or application-specific special cases such as n-grams) and according to their parallelization (sequential, shared-memory parallel, shared-nothing parallel, or MapReduce).

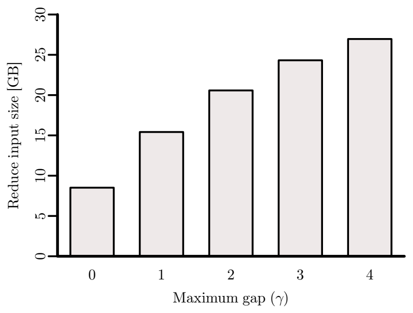
Sequential approaches to frequent itemset mining fall into two families. Candidate generation and pruning methods



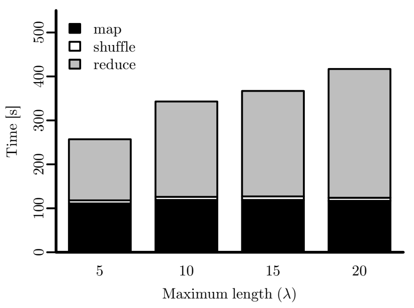
(a) NYT ($\gamma = 1, \lambda = 5$)



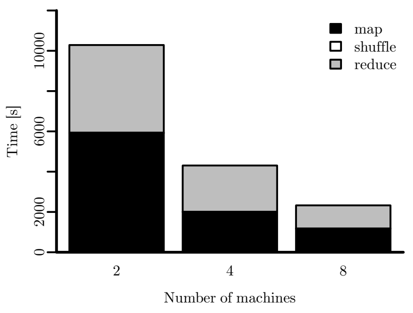
(b) NYT ($\sigma = 100, \lambda = 5$)



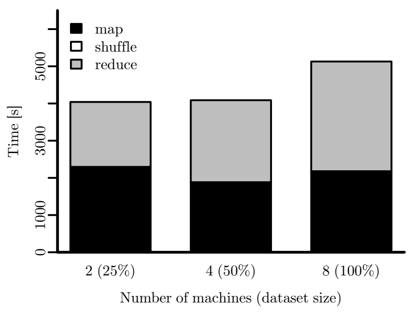
(c) NYT ($\sigma = 100, \lambda = 5$)



(d) NYT ($\sigma = 100, \gamma = 1$)



(e) 50% of CW ($\sigma = 1\,000, \gamma = 1, \lambda = 5$)



(f) CW ($\sigma = 1\,000, \gamma = 1, \lambda = 5$)

Figure 2: Impact of parameter settings (a–d) and scalability results (e–f)

such as Apriori [2] repeatedly scan the input data to count and prune candidate itemsets of increasing cardinality. Pattern-growth approaches, in contrast, scan the input data only once and construct a compact representation of it. FP-growth [12], as one such method, canonicalizes transactions by ordering items therein according to their support and represents the input data as a compact augmented prefix tree. Frequent itemsets can then be determined efficiently by traversing this so-called FP-tree. Our approach uses a similar item-based partitioning of the output space as FP-growth. Moreover, to cope with input data that exceeds the available main memory, FP-growth works with small projected databases, which contain all items required for one of the output partitions. Adaptations of these ideas have also been used for frequent sequence mining. GSP [23] generates candidate k-sequences by joining frequent $(k-1)$-sequences and prunes them by means of scanning the input data. FreeSpan [11] can be seen as an early adaptation of FP-growth to mine frequent sequences. PrefixSpan [22], its successor, uses a pattern-growth approach based on database projections, but employs a suffix-based partitioning of the output space. SPADE [31] assumes an alternative vertical representation of the input data, which can be understood as an inverted index that maintains for each item the list of transactions containing the item, and traverses the (conceptual) lattice of all sequences in breadth-first or depth-first order. Frequent episode mining [18], as a related yet slightly different problem, determines sequences that occur frequently within a single transaction. We refer to Han et al. [10] for a more detailed discussion of sequential approaches to frequent pattern mining.

Parallel approaches to frequent itemset and sequence mining have been proposed for different machine models. For frequent itemset mining in parallel shared-memory architectures, Parthasarathy et al. [21] describe an approach that aims for access locality when generating candidates in parallel. Zaki [30] investigates how SPADE can be parallelized by distributing data and/or work among machines. Buehrer et al. [5], targeting parallel distributed-memory architectures, exploit the item-based partitioning of FP-Growth to have different machines operate on partial aggressively pruned copies of the global FP-tree. Guralnik et al. [9] examine how the projection-based pattern-growth approach from [1], which is similar to PrefixSpan, can be parallelized by distributing data and/or work among machines. For the special case of closed sequences, Cong et al. [7] describe a parallel distributed-memory variant of BIDE [28]. They partition the sequence database based on frequent 1-sequences; a partition contains all suffix projections of sequences that contain the corresponding frequent 1-sequence. Only little work has targeted MapReduce as a model of computation. Li et al. [16] describe PFP, another adaptation of FP-Growth using item-based partitioning that is focused on finding the k most frequent itemsets.

Given the important role of n-grams in natural language processing and information retrieval, it is not surprising that several solutions exist for this specific special case of frequent sequence mining. SRILM [25] is one of the best-known toolkits to compute and work with n-gram statistics for document collections of modest size. Brants et al. [4] describe how large-scale statistical language models are trained at Google. To compute counts of n-grams having length five or less, they use a simple extension of WordCount in MapReduce (along the lines of the naïve approach of Sec. 2.3). Huston et al. [13] develop distributed methods to build an inverted index for n-grams that occur more than once in the document collection. Most recently, Berberich and Bedathur [3] described Suffix-σ, which we compared to in our experiments. The algorithm operates on suffixes, akin to [7], and runs in a single MapReduce job.

None of the existing work provides a satisfactory solution to general frequent sequence mining in MapReduce. While earlier parallel approaches [5, 9] also use an item-based partitioning of the output space, they rely on database projections to distribute data, which are less flexible than our partition construction techniques. Previous approaches typically have been evaluated on small-scale and/or synthetic datasets, whereas our experiments are based on more than 1 billion real-world input sequences.

8. CONCLUSIONS

We proposed MG-FSM, a scalable algorithm for gap-constrained frequent sequence mining in MapReduce. MG-FSM partitions the input database into a set of partitions that can be mined efficiently, independently, and in parallel. Our partitioning is based on a novel notion of w-equivalency, which generalizes the concept of a "projected database" used in many frequent pattern mining algorithms. Scalability is obtained due to a number of novel optimization techniques, including unreachability reduction, prefix/suffix reduction, blank reduction, blank separation, aggregation, and light-weight compression. Our experiments suggest that MG-FSM is orders of magnitudes more efficient and scalable than baseline algorithms for gap-constrained frequent sequence mining, and competitive to state-of-the-art algorithms for distributed n-gram mining (an important instance of gap-constrained FSM). For example, in our experiments, MG-FSM mined more than 1 billion input sequences for n-grams in less than half an hour on nine worker machines.

9. REFERENCES

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *J. Parallel Distrib. Comput.*, 61(3):350–371, 2001.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

[3] K. Berberich and S. Bedathur. Computing n-gram statistics in MapReduce. In *EDBT*, pages 101–112, 2013.

[4] T. Brants, A. C. Popat, P. Xu, F. J. Och, J. Dean, and G. Inc. Large language models in machine translation. In *EMNLP*, pages 858–867, 2007.

[5] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Toward terabyte pattern mining: An architecture-conscious solution. In *PPoPP*, pages 2–12, 2007.

[6] ClueWeb09 dataset. `lemurproject.org/clueweb09/`.

[7] S. Cong, J. Han, and D. Padua. Parallel mining of closed sequential patterns. In *KDD*, pages 562–567, 2005.

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

[9] V. Guralnik and G. Karypis. Parallel tree-projection-based sequence mining algorithms. *Parallel Computing*, 30(4):443–472, 2004.

[10] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: Current status and future directions. *Data Mining and Knowledge Discovery*, 15:55–86, 2007.

[11] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. FreeSpan: Frequent pattern-projected sequential pattern mining. In *KDD*, pages 355–359, 2000.

[12] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1):53–87, 2004.

[13] S. Huston, A. Moffat, and W. B. Croft. Efficient indexing of repeated n-grams. In *WSDM*, pages 127–136, 2011.

[14] R. Kant, S. H. Sengamedu, and K. S. Kumar. Comment spam detection by sequence mining. In *WSDM*, pages 183–192, 2012.

[15] C. Kohlschütter, P. Fankhauser, and W. Nejdl. Boilerplate detection using shallow text features. In *WSDM*, pages 441–450, 2010.

[16] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. PFP: Parallel FP-growth for query recommendation. In *RecSys*, pages 107–114, 2008.

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

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

[19] MG-FSM source code. `http://www.mpi-inf.mpg.de/departments/d5/software/mg-fsm/`.

[20] N. Nakashole, M. Theobald, and G. Weikum. Scalable knowledge harvesting with high precision and high recall. In *WSDM*, pages 227–236, 2011.

[21] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, 3:1–29, 2001. 10.1007/PL00011656.

[22] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The PrefixSpan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16:1424–1440, 2004.

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

[24] 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.

[25] A. Stolcke. SRILM - an extensible language modeling toolkit. In *Interspeech*, 2002.

[26] N. Tandon, G. de Melo, and G. Weikum. Deriving a Web-scale common sense fact database. In *AAAI*, 2011.

[27] The New York Times annotated corpus. `corpus.nytimes.com`.

[28] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *ICDE*, pages 79–90, 2004.

[29] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes: Compressing and indexing documents and images*. Morgan Kaufmann, 2nd edition, 1999.

[30] M. J. Zaki. Parallel sequence mining on shared-memory machines. In *Journal of Parallel and Distributed Computing*, pages 401–426, 2001.

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

[32] C. Zhai. Statistical language models for information retrieval a critical review. *Found. Trends Inf. Retr.*, 2:137–213, March 2008.

APPENDIX

A. PROOF OF LEMMA 2

We start by relating sequences of indexes of an input sequence T to the pivot sequences generated by T.

DEFINITION 6 (REACHABLE INDEX SEQUENCE). *Let $T = s_1 \cdots s_l$. We say that a sequence $I = i_1 i_2 \cdots i_n$ of increasing indexes is a* (w,γ,λ)-reachable index sequence *for T if $i_{k+1} - i_k \leq \gamma + 1$ for $1 \leq k < n$, $s_{i_k} \leq w$ for $1 < k < n$, $s_{i_k} = w$ for at least one $1 \leq k \leq n$, and $2 \leq n \leq \lambda$. Denote by $T(I) = s_{i_1} \cdots s_{i_n}$ the sequence generated by I, by $I_{w,\gamma,\lambda}(T)$ the set of all (w,γ,λ)-reachable index sequences for T, and by $I_{w,\gamma,\lambda}(S \mid T) \subseteq I_{w,\gamma,\lambda}(T)$ the set of (w,γ,λ)-reachable index sequences for T that generate S.*

This definition matches the definition of γ-subsequences of length at most λ but allows the first and last item to be irrelevant. In fact, $S \in G_{w,\gamma,\lambda}(T)$ if and only if both $S \leq w$ and $I_{w,\gamma,\lambda}(S \mid T)$ is non-empty. Consider for example the input sequence $T = aadc$. We have $G_{c,2,3}(T) = \{ ac, aac \}$, $I_{c,2,3}(T) = \{ 1 \cdot 4,\ 2 \cdot 4,\ 3 \cdot 4,\ 1 \cdot 2 \cdot 4 \}$, $I_{c,2,3}(ac \mid T) = \{ 1 \cdot 4,\ 2 \cdot 4 \}$ and $I_{c,2,3}(aac \mid T) = \{ 1 \cdot 2 \cdot 4 \}$.

We make use of a generalized distance definition in our proof of the correctness of the unreachability reduction.

DEFINITION 7 (DISTANCE). *Let $T = s_1 \cdots s_l$. Let $1 \leq i, j \leq n$ and set $l = \min\{ i, j \}$ and $r = \max\{ i, j \}$. The* (w,γ,λ)-distance *$d_{w,\gamma,\lambda}(i, j \mid T)$ between i and j is given by*

$$\min \{ |I| : I = i_1 \cdots i_n \in I_{w,\gamma,\lambda}(T), i_1 = l, i_n = r \} \cup \{ \infty \}.$$

Note that $d_{w,\gamma,\lambda}(i, j \mid T) \in \{ 1, 2, \ldots, \lambda, \infty \}$. Intuitively, the distance is equivalent to the smallest number of items that we need to "step onto" when moving from i to j via relevant items, by skipping at most γ items in each step, and by stepping onto at least one pivot item; it is infinite if there is no such path of length at most λ. We can now define left and right distances formally.

DEFINITION 8 (LEFT DISTANCE). *Let $T = s_1 \cdots s_l$. Fix some $1 \leq i \leq l$ and denote by $i_{prev} < i$ the largest index such that $s_{i_{prev}} = w$, if such an index exists. The* (w,γ,λ)-left distance *of index i is defined as $l_{w,\gamma,\lambda}(i \mid T) = d_{w,\gamma,\lambda}(i_{prev}, i \mid T)$ if i_{prev} exists; otherwise $l_{w,\gamma,\lambda}(i \mid T) = \infty$.*

We define the (w,γ,λ)-*right distance* $r_{w,\gamma,\lambda}(i \mid T)$ similarly w.r.t. the closest pivot to the right of index i. The following lemma captures most of the proof of the correctness of unreachability reduction.

LEMMA 5. *Let $T = s_1 \cdots s_l$ and let $1 \leq k \leq l$ be a (w,γ,λ)-unreachable index. Then*

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

where T' is obtained by removing index k from T. Moreover, for $1 \leq i \leq j \leq n$, $i \neq k$, $j \neq k$, we have

$$d_{w,\gamma,\lambda}(i, j \mid T) = d_{w,\gamma,\lambda}(i', j' \mid T'), \tag{5}$$

where $i' = i$ if $i < k$ and $i' = i - 1$ if $i > k$ (similarly j').

The first part of the lemma states that we can safely remove a single unreachable item from T. The second part states that all distances between remaining indexes are unaffected. Thus if an index i in T is unreachable in T, the corresponding index i' in T' will also be unreachable. We can thus remove all unreachable items in one go, which proves Lemma 2.

PROOF. For brevity, we drop subscript (w,γ,λ) from our notation.

First observe that if k is unreachable, we have for all indexes $i_- < k$ and $i_+ > k$, $d(i^-, k \mid T) = \infty$ and $d(k, i^+ \mid T) = \infty$, which implies $d(i_-, i_+ \mid T) = \infty$. Our definition of distance thus implies that there is no reachable index sequence of T that "crosses" k, i.e., simultaneously contains indexes less than k and indexes larger than k. Now pick any sequence $S \in G(T)$ and any of its index sequences $I \in I(S \mid T)$. If I consists only of indexes smaller than k, then $T'(I) = T(I) = S$. Otherwise, I consists only of indexes larger than k. Then $T'(I') = T(I) = S$, where I' is obtained from I by decrementing every index by one. Thus $S \in G(T)$ implies $S \in G(T')$.

We now show that no additional sequences are generated from T'. Suppose to the contrary that there exists a sequence S of length at most λ that is generated from T' but not from T. Then $I(S \mid T) = \emptyset$ but $I(S \mid T') \neq \emptyset$. Pick any $I' \in I(S \mid T')$, denote by i'_- and i'_+ the smallest and largest index in I', respectively. We must have $i'_- < k$ and $i'_+ \geq k$; otherwise the arguments above imply that T would have also generated S. Since $I' \in I(S \mid T')$, we obtain $d(i'_-, i'_+ \mid T') \leq \lambda$. Assume for now that Eq. (5) asserted above indeed holds. Then $d(i'_-, i'_+ \mid T') = d(i_-, i_+ \mid T)$, where $i_- = i'_-$ and $i_+ = i'_+ + 1$ denote the corresponding indexes in T. Since we showed above that $d(i_-, i_+ \mid T) = \infty$, we conclude that $d(i'_-, i'_+ \mid T') = \infty > \lambda$, a contradiction. Thus $S \in G(T')$ implies $S \in G(T)$.

It remains to show that Eq. (5) holds. Observe that the distance between two indexes i and j depends only on the set $\{ v : i \leq v \leq j \}$ of in-between indexes and, in particular, depends on s_v only through the *properties* of s_v, i.e., whether or not s_k is relevant and whether or not it is a pivot. Pick any $i_- \neq k$ and $i_+ > i_-$, $i_+ \neq k$, and denote by i'_- and i'_+ the corresponding indexes in T'. We need to show that

$$d(i_-, i_+ \mid T) = d(i'_-, i'_+ \mid T'). \tag{6}$$

If $i_+ < k$ or $i_- > k$, $T(i_- \cdots i_+) = T'(i'_- \cdots i'_+)$ and Eq. (6) follows immediately. Otherwise, we have $i^- < k$ and $i^+ > k$ and $d(i^-, i^+ \mid T) = \infty$. First, $d(i'_-, k \mid T') = \infty$ since (1) $T(i_- \cdots [k-1]) = T'(i'_- \cdots [k-1])$, (2) any difference between $d(i_-, k \mid T)$ and $d(i'_-, k \mid T')$ thus depends on the k-th item in each input sequence, (3) the k-th item is neither a pivot in T nor in T' (otherwise index k would be reachable in T) so that $d(i'_-, k \mid T') = d(i_-, k \mid T)$, and (4) $d(i_-, k \mid T) = \infty$. Using similar arguments, we can show that $d(k, i'_+ \mid T') = \infty$ and therefore $d(i'_-, i'_+ \mid T') = \infty$ as desired. □