Distributed frequent sequence mining with declarative subsequence constraints

Master's thesis

presented by
Florian Alexander Renz-Wieland

submitted to the
Data and Web Science Group
University of Mannheim

supervised by
Prof. Dr. Rainer Gemulla
Dr. Kaustubh Beedkar

examined by
Prof. Dr. Rainer Gemulla
Prof. Dr. Simone Ponzetto

to obtain the degree
Master of Science in Business Informatics

April 2017

Contents

List of Figures

List of Tables

List of Definitions

List of Algorithms

List of Abbreviations

DFA Deterministic Finite Automaton

FSM Frequent Sequence Mining

FST Finite State Transducer

NFA Nondeterministic Finite Automaton

Abstract

Frequent sequence mining extracts frequently occurring patterns from sequential data. Some algorithms allow users to specify constraints to control which sequences are of interest. Each algorithm usually supports a particular subset of available constraints. Recently, based on regular expressions, a declarative approach to specifying many of these constraints in a unified way was proposed.

Scalable algorithms are essential to mine large datasets efficiently. Such algorithms exist for particular subsets of constraints. However, no scalable algorithm with support for many constraints has been put forward yet.

We propose the distributed two-stage algorithm DDIN based on item-based partitioning. It processes input sequences in parallel and constructs partitions that can mine for frequent sequences in parallel. We use nondeterministic finite automata as an efficient representation for the intermediary sequences we send to the partitions.

Our experimental evaluation on four real-world datasets suggests that DDIN outperforms naive approaches for declarative constraints, is competitive to a state-of-the-art algorithm for traditional constraints, offers linear scalability, and makes it possible to mine datasets that cannot be mined efficiently using sequential algorithms.

Chapter 1

Introduction

Sequences are present in many datasets. For example, a sequence can be a succession of words in sentences, nucleotides in DNA molecules, products purchased by customers in an online shop, or subpages visited by the user of a website. For example, the customer of an online store might purchase an *EOS 5D* digital camera followed by a *LP-E6* battery pack and a *600EX-RT* flash. We refer to the elements of a sequence as *items*.

Often, it is of interest to find frequently occurring patterns in these sequences. Linguists might be interested in frequently used word combinations or grammatical structures. Biologists might want to find common nucleotide sequences related to certain genetic features. Retailers might ask which products customers frequently buy in succession. User interface designers can use common click patterns on a specific website to improve the navigation of the site.

This problem of finding frequently occurring sequences is commonly referred to as *frequent sequence mining* (FSM). Given a collection of input sequences, the goal of FSM is to find subsequences that occur in more input sequences than a given threshold. Such a subsequence is then considered a *frequent sequence*.

Often, one is interested in general patterns. For example, only a couple of customers might purchase the *EOS 5D* digital camera and the *600EX-RT* flash in succession. However, the purchase of a camera followed by the purchase of a flash might be quite common. An *item hierarchy* allows to find such patterns. In a hierarchy, an item can *generalize* to a more general notion. Typically, these relations are of the *is-a* type. On product data, such an item hierarchy can be a product categorization. Such that *EOS 5D* is a *camera*, which in turn is an *electronic device*.

In textual data, one can build hierarchies by using grammatical information or external knowledge about known entities. For example, the word *works* stems from *work*, which is a *verb*. The entity *Albert Einstein* is a *scientist*, which is a *person*. For many applications, it is useful to have multiple generalizations for a single item. *Albert Einstein* is a *scientist*, but also a *father*.

Typically, FSM produces many frequent sequences as output. Often, many of these found sequences are not of interest for the specific application. Therefore, soon after the initial proposal of unconstrained FSM (Agrawal and Srikant, 1995), researchers have developed ways to constrain the output of FSM. Srikant and Agrawal (1996) propose *maximum length* and *maximum gap* constraints, such that only subsequences with a length up to a specified threshold are considered and only a specified number of items of the

input sequence can be skipped to produce subsequences. Srikant et al. (1997) propose *item constraints* in combination with hierarchies such that subsequences are only considered if they contain a specified item or an item that generalizes to a specified item.

Regular expressions have been studied as a tool to flexibly specify a number of constraints (Garofalakis et al., 1999; Pei et al., 2002; Albert-Lorincz and Boulicaut, 2003; Trasarti et al., 2008). They are suitable to express constraints because they offer both a simple syntax and sufficient expressive power to specify a wide range of constraints (Garofalakis et al., 1999).

Incorporating constraints has two main advantages (Garofalakis et al., 1999). First, they limit the number of found sequences. Eliminating uninteresting sequences from the result makes the output easier to interpret for users. Second, if the mining algorithm takes the constraints into account early on in the mining process, it can avoid computation for sequences that are not relevant to the user. Garofalakis et al. (1999) show that this can significantly lower the amount of necessary computation for selective constraints. Pei et al. (2002) show that some constraints can be integrated into the mining process easier than others.

Beedkar and Gemulla (2016a) propose a *pattern expression language* that extends previous regular-expression based approaches and unifies many previously studied constraints into one framework. In contrast to previous regular expression constraints, which are applied directly to subsequences and have no support for hierarchies, *pattern expressions* are evaluated on the input sequence, and they support hierarchies.

Beedkar and Gemulla show that many traditional constraints can be modeled using this pattern expression language. Beyond that, the language enables new ways to constrain FSM. As the pattern expression language makes it possible to specify a wide range of subsequence constraints using a high-level logic language, we refer to these constraints as *declarative subsequence constraints*.

Distributed and scalable algorithms, which can run the mining on multiple computers in parallel, are crucial for mining frequent sequences from large datasets. Some of today's datasets might not fit on the hard drive of a single machine. For other FSM problems, the computational power of one machine might not be sufficient to find frequent sequences in a reasonable time.

Parallel algorithms have been proposed early on for FSM without subsequence constraints or support for hierarchies (Zaki, 2001; Guralnik and Karypis, 2004). More recently, distributed algorithms with support for maximum length and maximum gap constraints have been proposed, first without support for hierarchies (Miliaraki et al., 2013) and later with support for hierarchies (Beedkar and Gemulla, 2015). To the best of our knowledge, no distributed algorithms have yet been proposed for FSM with regular expression constraints or declarative constraints.

A straightforward distributed algorithm for declarative constraints can be developed from the sequential DESQ-COUNT algorithm (Beedkar and Gemulla, 2016a). Its major shortcoming is a computational blow-up for subsequence constraints that produce many intermediary subsequences, which we call *candidate sequences*. We discuss this algorithm and its deficiencies in Section 3.1. The goal of this thesis is to develop an algorithm that overcomes these limitations.

Problem statement. Develop a distributed algorithm for FSM with declarative subsequence constraints. The algorithm should perform efficiently for constraints that produce many candidate sequences.

We propose a two-stage distributed algorithm. The algorithm first processes input sequences in parallel and extracts relevant information. Then, the computers of the distributed system exchange data to create a number of independent partitions. In the second stage, these partitions are processed in parallel again to produce the set of frequent sequences. To construct the partitions, we use a hierarchy-aware variant of item-based partitioning. To the partitions, we send the candidate sequences generated by an input sequence, which we encode as a *nondeterministic finite automaton* (NFA). For local mining at the partitions, we adapt implement a pattern-growth approach based on DESQ-DFS (Beedkar and Gemulla, 2016a).

We proceed as follows. First, Chapter 2 gives a formal definition of the FSM problem with declarative subsequence constraints, introduces a computational model for these constraints, and discusses sequential mining algorithms for the problem. It further reviews distributed approaches to FSM with traditional constraints. In Chapter 3, we consider the parts of the proposed algorithm in more detail. After covering a first naive approach to distribution in Section 3.1 and giving a high-level overview of the proposed algorithm in Section 3.2, we consider how to divide up the problem into multiple smaller ones in Section 3.3 and what information to send between the computers of the distributed system in Section 3.4. We further discuss how to compactly represent candidate sequences for communication, how to generate such a representation efficiently, and how to serialize the representation in Section 3.5, Section 3.6, and Section 3.7, respectively. We present our algorithm for local mining in Section 3.8. Finally, in our experimental evaluation in Section 4, we examine the performance of the proposed algorithm.

Chapter 2

Preliminaries

In this section, we first define the FSM problem in the context of declarative subsequence constraints and introduce a model to express and compute these constraints. We then look at sequential mining algorithms for this problem and review existing distributed algorithms for similar problems.

2.1 FSM with declarative subsequence constraints

Sequential pattern mining is a problem in data mining that is concerned with finding 'interesting' subsequences from a database of input sequences. In sequential pattern mining, different notions exist for what a sequence is and when a sequence is 'interesting'. In literature, many approaches study the general case of sequences of itemsets, where a sequence is formed by a succession of itemsets. In this thesis, we focus on the special case where sequences are formed of individual items instead of itemsets. This notion is relevant for many applications, for example when working with textual data, DNA molecules, event logs, or user sessions. Further, to measure the 'interestingness' of sequences, we focus on how frequent the sequence is in the input data. We refer to this variant of sequential pattern mining as FSM. In the following, after introducing necessary basic notation, we formally define this problem.

Frequent sequence mining is closely related to *frequent itemset mining*, which is also called *association rule mining* and was first introduced by Agrawal et al. (1993). Instead of working with sequences as input, frequent itemset mining works on sets of items and extracts combinations of patterns that frequently co-occur in the input itemsets.

2.1.1 Basic concepts of FSM

Sequence database. We call a collection of input sequences the *sequence database*. Each input sequence is an ordered succession of *items*. These items stem from one *item vocabulary* Σ. We denote a sequence T of length $|T|$ as $T = \langle t_1 t_2 ... t_{|T|} \rangle$, where each item stems from the vocabulary: $t_i \in \Sigma$. For the sequence database $\mathcal{D}$ we denote $\mathcal{D} = \{T_1, T_2, ..., T_{|\mathcal{D}|}\}$. Figure 2.1(a) depicts an example sequence database $\mathcal{D}_{ex}$, which contains six input sequences and 11 distinct items.

Our example consists of generic items such as a_1, c, and B. These placeholder items can stand for items of any type. One typical application of sequence mining is text processing,

$T_1 : \langle a_1 c b_{12} e \rangle$

$T_2 : \langle b_2 a_2 b_{12} e \rangle$

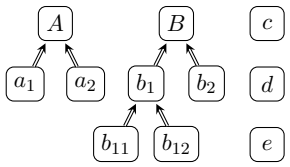
$T_3 : \langle a_2 d a_1 a_2 b_{11} e \rangle$

$T_4 : \langle a_1 d B e \rangle$

$T_5 : \langle e a_1 b_2 e \rangle$

$T_6 : \langle a_1 c a_1 a_1 b_2 e a_1 c b_2 e d a_1 \rangle$

(a) Sequence database $\mathcal{D}_{ex}$



(b) Item hierarchy $\mathcal{H}_{ex}$

	Item	Frequency
(1)	A	6
(2)	B	6
(3)	e	6
(4)	a_1	5
(5)	b_1	3
(6)	b_2	3
(7)	d	3
(8)	a_2	2
(9)	b_{12}	2
(10)	c	2
(11)	b_{11}	1

(c) f-list for $\mathcal{D}_{ex}$ with $\mathcal{H}_{ex}$

Figure 2.1: Example dataset: sequence database, item hierarchy, and f-list

where one item corresponds to a word and a sequence corresponds to either a sentence or a document. The sequence database is then a collection of sentences or a collection of documents.

Another application is customer purchase analysis. Here, items correspond to products and a sequence is a succession of items bought by one customer. The sequence database consists of successions of products purchased by different customers.

Item hierarchy. An *item hierarchy* $\mathcal{H}$ defines how items of Σ can be generalized. An item can generalize to zero or more other items from Σ. Figure 2.1(b) presents an example hierarchy $\mathcal{H}_{ex}$ for our example sequence database $\mathcal{D}_{ex}$. In the example, item a_1 generalizes to item A, as does item a_2. Items b_{11} and b_{12} generalize to item b_1, which in turn generalizes to item B. Item b_2 also generalizes to item B. All other items in the vocabulary do not generalize to another item.

We write $\Rightarrow$ if an item directly generalizes to an item, such as $b_{11} \Rightarrow b_1$ and $b_1 \Rightarrow B$. We further write $\Rightarrow^*$ if an item generalizes either directly or indirectly — with other items as intermediary generalization steps — to another item. For example, $b_{11} \Rightarrow^* B$ and $b_{11} \Rightarrow^* b_1$. For easier notation, we specify that an item indirectly generalizes to itself, that is, $b_{11} \Rightarrow^* b_{11}$.

Ancestors and descendants. We introduce two further notations for generalization. Informally, the *ancestors* of item ω are all items that can be reached from ω by following generalizations. Formally:

$$\text{anc}(\omega) = \{\omega' \mid \omega \Rightarrow^* \omega'\}.$$

Note that the set of ancestors includes the item itself, that is, $\omega \in \text{anc}(\omega)$. Referring to the item hierarchy depicted in Figure 2.1(b), some examples are $\text{anc}(b_{12}) = \{b_{12}, b_1, B\}$, $\text{anc}(b_1) = \{b_1, B\}$, and $\text{anc}(e) = \{e\}$.

Analogously, we define the set of *descendants*. Informally, an item ω' is a descendant of ω if one can generalize to ω when starting at ω':

$$\text{desc}(\omega) = \{\omega' \mid \omega' \Rightarrow^* \omega\}.$$

Again, the set includes the item itself. For example, $\text{desc}(A) = \{a_1, a_2, A\}$, $\text{desc}(b_1) = \{b_1, b_{11}, b_{12}\}$, and $\text{desc}(e) = \{e\}$.

Subsequences. Let $S = \langle s_1 s_2 ... s_{|S|} \rangle$ and $T = \langle t_1 t_2 ... t_{|T|} \rangle$ be two sequences, consisting of items from Σ. We define S to be a *generalized subsequence* of T if S can be produced from T by generalizing and deleting items. We use the term *subsequence* as a synonym for generalized subsequence in the following. We write this as $S \sqsubseteq T$. Formally, $S \sqsubseteq T$ if and only if there exist integers $1 \leq i_1 < i_2 < ... < i_{|S|} \leq |T|$ such that $t_{i_1} \Rightarrow^* s_1, t_{i_2} \Rightarrow^* s_2, ..., t_{i_{|S|}} \Rightarrow^* s_{|S|}$. One can see how one can derive several generalized subsequences by deletions and generalizations in the following example for input sequence T_1. We start on the right with T_1 and generalize or delete items in every step towards the left:

$$\langle B \rangle \sqsubseteq \langle AB \rangle \sqsubseteq \langle Ab_1 \rangle \sqsubseteq \langle Acb_1 \rangle \sqsubseteq \langle a_1 cb_1 \rangle \sqsubseteq \langle a_1 cb_1 e \rangle \sqsubseteq \langle a_1 cb_{12} e \rangle = T_1.$$

All these sequences are subsequences of T_1. Sequence $\langle AB \rangle$, for example, is a subsequence of T_1. In total, with our example hierarchy $\mathcal{H}$, input sequence T_1 has 47 subsequences. Some other examples are $\langle Ae \rangle \sqsubseteq T_1$, $\langle a_1 b_1 \rangle \sqsubseteq T_1$. However, $\langle eA \rangle$ and $\langle cb_{11} \rangle$ are not: $\langle eA \rangle \not\sqsubseteq T_1$ and $\langle cb_{11} \rangle \not\sqsubseteq T_1$.

The maximum number of subsequences of a sequence T is exponential in the length of the sequence. Let n be the length of the sequence and for now, we ignore generalization. Then the number of subsequences is $2^n - 1$. It is easy to see this when considering that we can either include or exclude each item of sequence T. This gives two options per item, so with n items we have 2^n options. If one excludes every item of the sequence, we obtain the empty sequence, which we do not consider a subsequence for our purposes, therefore $2^n - 1$.

Now we consider generalization. The number of subsequences depends on the depth of the hierarchy. So assume we can generalize each item of the sequence to four further items. That gives six options per input item: exclude the item, include the item, or generalize it to one of the four other items. So with a hierarchy of four, an input sequence T can potentially produce $(2 + 4)^n - 1$ subsequences. For a sequence of length $n = 10$, that are about 60 million possible subsequences.

2.1.2 Problem definition

Algorithms for sequential pattern mining typically offer means to constrain the number of subsequences that should be considered for the mining. Srikant and Agrawal (1996) suggest *sliding windows* and *time constraints*. Since then, other types of constraints have been introduced. Mooney and Roddick (2013) give an overview over suggested constraints. Typically, an FSM algorithm supports only a subset of the available constraints.

Beedkar and Gemulla (2016a) aim to unify most of the proposed constraints into one framework. To do this, they propose *subsequence predicates*.

A subsequence predicate constrains the set of subsequences of an input sequence that is considered for the mining. Essentially, it divides the set of subsequences of an input sequence into two parts. One part conforms with the predicate and is considered for the mining. We call this part the *candidate sequences* generated by this input sequence. The other part of the subsequences is discarded. Let π be a subsequence predicate. We denote by $S \sqsubseteq_\pi T$ that S is a subsequence of T with respect to subsequence predicate π. We say that T π-generates S.

Definition 1 (Candidate sequences) *We depict as candidate sequences of a sequence T with respect to subsequence constraint π all subsequences S of T that conform with π.*

We write

$$C'(T, \pi) = \{S \mid S \sqsubseteq_\pi T\}.$$

As discussed, input sequence T_1 has 47 subsequences. However, we might be interested only in the ones that start with item A, so we can restrict the set of subsequences to the ones that start with A. That reduces the number of subsequences we consider to 16.

We aim to find generalized subsequences that occur in multiple input sequences. For this, we define the support of a sequence. This definition then allows defining the FSM problem with support for an item hierarchy and a subsequence predicate.

Definition 2 (Support of a sequence) *Given a sequence database $\mathcal{D}$, an item hierarchy $\mathcal{H}$, and a subsequence predicate π, the π-support of a sequence S is the set of input sequences that π-generate S:*

$$Sup_{\mathcal{D},\mathcal{H},\pi}(S) = \{T \in \mathcal{D} \mid S \in C'(T, \pi)\}$$

Definition 3 (FSM with item hierarchy and subsequence predicate) *Given a sequence database $\mathcal{D}$, an item hierarchy $\mathcal{H}$, a subsequence predicate π, and a minimum support threshold $\sigma > 0$, find all (π,σ)-frequent sequences and output each of them along with its frequency.*
A sequence S is (π,σ)-frequent if at least σ input sequences π-generate it. That is, if

$$|Sup_{\mathcal{D},\mathcal{H},\pi}(S)| \geq \sigma.$$

The frequency of a sequence S is $|Sup_{\mathcal{D},\mathcal{H},\pi}(S)|$.

Subsequence predicates give a flexible way to constrain the set of subsequences. As an example, consider analyzing the products purchased in an online shop for electronics. To improve recommendations, shop owners might be interested in the types of products purchased after digital cameras. Alternatively, when analyzing a large text corpus, one might be interested in the adjectives that are commonly used around a set of known entities, for example around the names of a country's former presidents. Subsequence predicates in combination with hierarchies allow such mining tasks.

2.2 A computational model for subsequence constraints

We now look at how one can implement the abstract notion of a subsequence predicate. We first present a language to express these predicates and then introduce a model to efficiently compute candidate sequences from an input sequence.

2.2.1 Pattern expressions

To implement subsequence predicates, Beedkar and Gemulla (2016a) introduce a *pattern expression language*. The language is based on regular expressions. One *pattern expression* models one subsequence predicate. Of the set of all subsequences, only subsequences that conform with the given pattern expression are considered. For this work, we do not introduce the full pattern expression language. Instead, we focus on the aspects critical to this work.

Table 2.1: Item expressions, adapted from Beedkar and Gemulla (2016a)

Item expression	Matches	Outputs
ω	$\omega' \in \text{desc}(\omega)$	–
(ω)	$\omega' \in \text{desc}(\omega)$	ω'
$\omega^\uparrow$	$\omega' \in \text{desc}(\omega)$	–
$(\omega^\uparrow)$	$\omega' \in \text{desc}(\omega)$	$\text{anc}(\omega') \cap desc(\omega)$
$(\omega^\uparrow_=)$	$\omega' \in \text{desc}(\omega)$	ω
.	$\omega' \in \Sigma$	–
(.)	$\omega' \in \Sigma$	ω'
$(.^\uparrow)$	$\omega' \in \Sigma$	$\text{anc}(\omega')$

To understand the pattern expression language, we first look at an example pattern expression and the candidate sequences it produces. We then explain how the elements of the language work. Let us consider pattern expression π_1 given by

$$\pi_1 = A([c|d][A^\uparrow|B^\uparrow]^+e).$$

A pattern expression is run on each input sequence and, for each input sequence, produces zero or more candidate sequences. On input sequence T_1 of our running example, this pattern expression produces the following candidate sequences:

$$C'(T_1, \pi_1) = \{\langle cb_{12}e\rangle, \langle cb_1e\rangle, \langle cBe\rangle\} \text{ from } T_1 = \langle a_1cb_{12}e\rangle.$$

Other subsequences of T_1, such as $\langle a_1b_{12}e\rangle$ or $\langle ce\rangle$, are not produced. From a similar sequence $T_1' = \langle cb_{12}e\rangle$ and input sequences T_2 and T_4 of our running example, the pattern expression produces the candidate sequences

$$\begin{aligned} C'(T_1', \pi_1) &= \{\} \text{ from } T_1' = \langle cb_{12}e\rangle, \\ C'(T_2, \pi_1) &= \{\} \text{ from } T_2 = \langle b_2a_2b_{12}e\rangle, \text{ and} \\ C'(T_4', \pi_1) &= \{\langle dBe\rangle\} \text{ from } T_4 = \langle a_1dBe\rangle. \end{aligned}$$

The building blocks of a pattern expression are *item expressions*. In our example, these item expressions are A, c, d, $A^\uparrow$, $B^\uparrow$, and e. The item expressions can be combined in various ways that are known from regular expressions. For example, item expressions can be concatenated to match a succession of input items, such as ab_2, or alternated, such as $[c|d]$, to use either of the two item expressions. Typical quantifications are also available. For example, the $^+$ means that an item expression can be applied one or multiple times.

In regular expressions, an item A in the pattern expression would typically match an A in the input sequence. We work with item hierarchies: an item expression A can match multiple items. Typically, an item expression matches all its descendants. So in our example, the A in the pattern expressions would match any input item A, a_1, or a_2. For example, in input sequence T_1, item a_1 is matched by the A in the pattern expression.

Item expressions outside parentheses do not produce any *output items* and therefore do not contribute items to the candidate sequences. We say these item expressions are *uncaptured*. They are used to match context and can be seen as a way to delete the matched

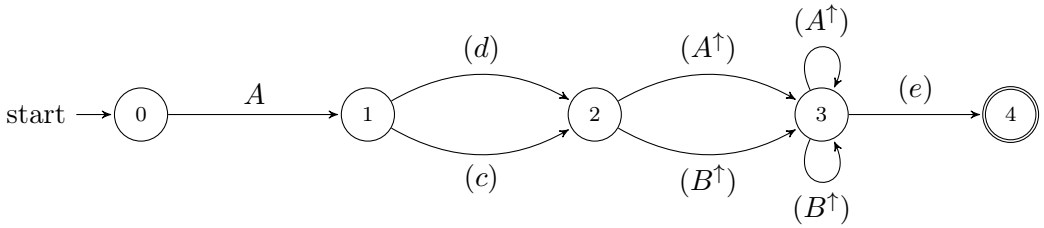


Figure 2.2: FST for pattern expression $\pi_1 = A([c|d][A^{\uparrow}|B^{\uparrow}]^{+}e)$. Transitions are labeled with item expressions, which determine the accepted input items and the produced output items as defined in Table 2.1.

item from the subsequences. In our example, the first A is not enclosed in parentheses and therefore does not contribute any items to the candidate sequences. However, if there is no item to match an uncaptured item expression in the input sequence, the pattern expression does not match the input sequence and no candidate sequence is produced. This is the case for T_1'. It contains no item to match the uncaptured A part of the pattern expression.

Captured item expressions inside parentheses contribute output items to one or more candidate sequences. This behavior can be modified with the upwards arrow. Without a consecutive upwards arrow, an item expression produces the item it matched. For example, if an item expression A matches an input item a_1, it produces output item a_1. If the item expression is followed by an upwards arrow, generalizations of the matched item are produced as well, but only up to the item specified in the item expression. For example, for input sequence T_1, item expression $B^{\uparrow}$ produces items b_{12}, b_1, and B from matched input item b_{12}. An overview of these options for item expressions is given in Table 2.1. A further modifier, the subscript equality sign $_{=}$, specifies that the item in the item expression is output instead of the matched item. As in regular expressions, a dot can be used to match any item.

2.2.2 Finite state transducers

To generate candidate sequences efficiently, Beedkar and Gemulla (2016a) introduce a restricted form of *finite state transducers* (FST) as a computational model. An FST is a finite-state machine. In contrast to other finite-state machines such as an NFA or a *deterministic finite automaton* (DFA), an FST generates an output string while reading the input string (Mohri, 1997). We omit a formal introduction of FSTs. Beedkar and Gemulla (2016a) give a formal introduction of FSTs for pattern expressions.

Beedkar and Gemulla (2016a) propose methods to construct an FST for a given pattern expression, such that, when reading a given input sequence as the input string, the FST produces exactly the set of frequent candidate sequences for that input sequence as output. Figure 2.2 presents an FST constructed from example pattern expression π_1. For every input sequence T, this FST produces exactly the set of candidate sequences $C'(T, \pi_1)$. We mark the initial state of the FST with an ingoing arrow and an accepting state of the FST with a double outline.

The transitions of the FST are labeled with item expressions. Each transition matches input items and produces output items according to this item expression. If a transition

Table 2.2: The accepting path for input sequence T_1 through the FST for pattern expression π_1

From → to states	Transition	Input item	Output item(s)
$0 \rightarrow 1$	A	a_1	–
$1 \rightarrow 2$	(c)	c	c
$2 \rightarrow 3$	$(B^{\uparrow})$	b_{12}	b_{12}, b_1, B
$3 \rightarrow 4$	(e)	e	e

is labeled with a captured item expression, it produces one ore more output items. For example, the (c) transition from state 1 to state 2 matches any item $\omega' \in \text{desc}(c)$ and produces that item ω'. The $(B^{\uparrow})$ transition from state 2 to state 3 matches any item $\omega' \in \text{desc}(B) = \{b_{11}, b_{12}, b_1, b_2, B\}$ and produces all elements $o \in \text{anc}(\omega') \cap \text{desc}(B)$. Consequently, if the matched item ω is b_1 with $\text{anc}(b_1) = \{b_1, B\}$, the transition produces items b_1 and B, as $\{b_1, B\} \cap \{b_{11}, b_{12}, b_1, b_2, B\} = \{b_1, B\}$.

The FST is run on each input sequence. The FST matches an input sequence T if, starting from the initial state, we can take one consecutive input item ω from T per transition and arrive at an accepting state. More formally, it matches an input sequence $T = \langle \omega_1 \omega_2 ... \omega_n \rangle$ if there is a succession of transitions $u_1 u_2 ... u_n$, such that: transition u_i accepts item ω_i for $1 \leq i \leq n$, transition u_1 originates in an initial state, transition u_n ends in an accepting state, and transition u_i originates in the target state of transition u_{i-1} for $2 \leq i \leq n$. We call such a succession of transitions an *accepting path* through the FST and say the FST for pattern expression π matches T. There can be multiple accepting paths through an FST for one input sequence.

Throughout this thesis, we use example pattern expression $\pi_1 = A([c|d][A^{\uparrow}|B^{\uparrow}]^{+}e)$. We allow this pattern expression to match anywhere in an input sequence, without the need to read the entire input sequence. That is, formally, we use pattern expression $\pi_{1\text{f}}$.

$$\pi_{1\text{f}} = .^{*}\ A([c|d][A^{\uparrow}|B^{\uparrow}]^{+}e)\ .^{*}$$

We omit the surrounding expression $.^{*}$ for better readability.

For example, for input sequence $T_1 = \langle a_1 c b_{12} e \rangle$, there is one accepting path through the FST: starting from state 0, it takes transition A to state 1 with input item a_1, transition (c) to state 2 with input c, transition $(B^{\uparrow})$ to state 3 with input b_{12}, and transition (e) to state 4 with input e. Table 2.2 depicts this path along with the produced output items.

An accepting path produces a set of candidate sequences. Each transition of the accepting path produces either no output item (if uncaptured), or a group of output items, which consists of one or more items. The path produces all sequences that can be created by picking one item from each non-empty group of output items and arranging them in order. If there are multiple accepting paths through the FST, the set of all candidate sequences is constructed by the union of the candidate sequences of all accepting paths.

For example, the accepting path depicted in Table 2.2 generates the sets of output items $\{\}$, $\{c\}$, $\{b_{12}, b_1, B\}$, $\{e\}$, in this order. By picking one item from each non-empty set and arranging them in order, we can create the sequences $\langle c b_{12} e \rangle$, $\langle c b_1 e \rangle$, and $\langle c B e \rangle$.

If the last set of items would contain another hypothetical item E, we could additionally create sequences $\langle cb_{12}E\rangle$, $\langle cb_1E\rangle$, and $\langle cBE\rangle$.

2.3 Sequential algorithms for FSM with declarative subsequence constraints

In this section, we briefly present three sequential algorithms for mining frequent sequences in the presence of declarative subsequence constraints. We start with a naive approach and a variation of the naive approach, DESQ-COUNT. Naive approaches are efficient for selective pattern expressions, which generate only few candidate sequences. We then discuss a more involved approach, DESQ-DFS, which is also applicable to less selective patterns. The algorithms have been proposed by Beedkar and Gemulla (2016a). They are sequential algorithms, so they run on one core of one machine, without parallel execution.

2.3.1 Naive

As a naive approach, one can proceed as follows: For each input sequence, generate all candidate sequences. Then count how often each candidate sequence has been generated and output the ones that have been generated at least σ times.

This approach works efficiently for pattern expressions that generate a small number of candidate sequences. We call such pattern expressions *selective*. For these pattern expressions, it is possible to generate all candidate sequences and count their occurrences in a reasonable time.

For pattern expressions that produce many candidate sequences, this approach is inefficient or even infeasible. An input sequence can have exponentially many subsequences. An *unselective* pattern expression excludes only some of these subsequences. Consequently, producing and storing all candidate sequences can take exponential time and space. An exponential increase in time can make the algorithm run long. An exponential increase in space can make the algorithm infeasible if it needs more space than the executing machine can offer.

2.3.2 DESQ-COUNT

DESQ-COUNT is mostly identical to the naive approach but improves on it in one way. That is, it also produces candidate sequences and counts their occurrence. However, instead of producing all candidate sequences of each input sequence, it produces only a subset of candidate sequences. It omits a particular group of candidate sequences, for which it can be sure that they will not be a frequent sequence.

To do this, DESQ-COUNT makes use of one key observation: if a candidate sequence contains any item ω that itself 'occurs' in less than σ input sequences, this candidate sequence cannot occur in more than σ input sequences. In our running example, item b_{11} occurs only in input sequence T_3 and no other input sequence. Therefore, any candidate sequence containing b_{11} cannot be a frequent sequence for any $\sigma \geq 2$. The notion of in how many input sequence an item ω 'occurs' we define as *item frequency*. However, we work with an item hierarchy, which allows an item to produce generalized items as candidate sequences. We need to take this into account for our definition of item frequency.

Definition 4 (Item frequency) *The item frequency of an item $\omega \in \Sigma$ is given by the number of input sequences that contain item ω or any of its descendants. We write*

$$f(\omega) = |\{T \in \mathcal{D} \mid \omega' \in T \text{ and } \omega' \in desc(\omega)\}|.$$

We say that an item ω is frequent if $f(\omega) \geq \sigma$. Otherwise, we say the item is infrequent.

For each input sequence, DESQ-COUNT produces only the candidate sequences that consist only of frequent items: the set of *frequent candidate sequences*.

Definition 5 (Frequent candidate sequences) *The set of frequent candidate sequences of an input sequence T with respect to a pattern expression π_1 is a subset of all candidate sequences of T. The subset contains the candidate sequences that contain only frequent items. We write*

$$C(T, \pi) = \{S \in C'(T, \pi) \mid f(\omega) \geq \sigma \text{ for all } \omega \in S\}.$$

Note that $C(T, \pi)$ depends on σ. We omit this dependency from our notation for better readability.

In contrast, the item frequency $f(\omega)$ for a single item is independent of minimum support σ and pattern expression π. Consequently, it can be precomputed. DESQ-COUNT computes a list of item frequencies $f(\omega)$ for all items $\omega \in \Sigma$ before the mining is run. This list is called the *f-list*. Figure 2.1(c) depicts the f-list for example sequence database $\mathcal{D}_{ex}$, considering item hierarchy $\mathcal{H}_{ex}$.

Let us consider an example. For input sequence $T_3 = \langle a_2 d a_1 a_2 b_{11} e \rangle$ of our example sequence database, the set of all candidate sequences for pattern expression π_1 is as follows:

$$\begin{aligned} C'(T_3, \pi_1) = \{&\langle da_1a_2b_{11}e\rangle, \langle da_1a_2b_1e\rangle, \langle da_1a_2Be\rangle, \langle da_1Ab_{11}e\rangle, \langle da_1Ab_1e\rangle, \langle da_1ABe\rangle, \\ &\langle dAa_2b_{11}e\rangle, \langle dAa_2b_1e\rangle, \langle dAa_2Be\rangle, \langle dAAb_{11}e\rangle, \langle dAAb_1e\rangle, \langle dAABe\rangle\}. \end{aligned}$$

When setting $\sigma = 2$, item b_{11} is infrequent: $f(b_{11}) = 1 < 2 = \sigma$. Therefore, no sequences containing b_{11} can be frequent. So they are not included in the set of frequent candidate sequences:

$$\begin{aligned} C(T_3, \pi_1) = \{&\langle da_1a_2b_1e\rangle, \langle da_1a_2Be\rangle, \langle da_1Ab_1e\rangle, \langle da_1ABe\rangle, \\ &\langle dAa_2b_1e\rangle, \langle dAa_2Be\rangle, \langle dAAb_1e\rangle, \langle dAABe\rangle\}. \end{aligned}$$

The elimination of infrequent items is an adaptation of the more general *support anti-monotonicity* property, which is used in many frequent itemset mining and sequential pattern mining algorithms. The more general case says that any subsequence of a frequent sequence must be frequent. This property holds for unconstrained FSM and certain traditional constraints. For the setting of declarative constraints, this property might hold for some particular pattern expression, but does not hold in general. Therefore, DESQ-COUNT considers only single item frequencies.

To recap, DESQ-COUNT uses an FST for the given pattern expression and the given σ to directly produce the set of frequent candidate sequences $C(T, \pi)$ for all input sequences $T \in \mathcal{D}$. It then aggregates all produced candidate sequences by count and outputs the ones that have been produced from more than σ input sequences.

Beedkar and Gemulla show that eliminating candidate sequences that contain infrequent items can significantly reduce the number of candidate sequences. The actual effect depends mostly on the value chosen for σ. The larger σ, the more items are infrequent, and consequently, the more candidate sequences can be eliminated.

However, for many pattern expressions, DESQ-COUNT still produces many candidate sequences and consequently is, as the naive approach, inefficient or infeasible for these settings.

2.3.3 DESQ-DFS

To address this issue, Beedkar and Gemulla (2016a) introduced a more involved algorithm, DESQ-DFS. They show that DESQ-DFS can be more than an order of magnitude faster than DESQ-COUNT for unselective pattern expressions. For selective pattern expressions, DESQ-DFS has performance comparable with DESQ-COUNT. We briefly present the core idea of DESQ-DFS here.

DESQ-DFS adapts the *pattern-growth* approach of PrefixSpan (Pei et al., 2001). A key goal of pattern-growth is to prevent the generation of candidate sequences that will turn out not to be a frequent sequence. The earlier in the mining process one can decide that a sequence will turn out to be infrequent, the more unnecessary computation can be avoided.

The pattern-growth approach generates frequent sequences by growing a sequence item by item. It starts with an empty sequence and expands the sequence by repeatedly appending single frequent items. As items are only appended, the growing sequence is often called the *prefix*. After each expansion, pattern-growth checks whether the current prefix S can lead to a frequent sequence. If it finds that S cannot lead to a frequent sequence, it does not expand S further. Instead, it removes the last appended item and tries other expansions.

To efficiently check whether the current prefix S can lead to a frequent sequence, pattern-growth methods typically build an *inverted index* for every prefix. For a prefix S, the data structure holds a list of input sequences that can generate S. If the list contains less than σ input sequences, expansion can be stopped. When a new prefix $S\omega$ is created by appending ω to the current prefix S, the new inverted index for $S\omega$ is constructed by filtering input sequences in the inverted index of S.

Originally, pattern-growth works directly on input sequences. DESQ-DFS incorporates the FST for the given pattern expression into the mining process. In the inverted index for a prefix S, DESQ-DFS stores a list of input sequences, from which the FST can produce the prefix. It additionally stores two further details in the index. It stores the state the FST is in after producing prefix S. Also, it stores the position of the next input item in the input sequence.

By checking the frequency of each prefix and stopping expansion of infrequent prefixes early, DESQ-DFS does not produce the large number of candidate sequences DESQ-COUNT produces. Therefore, in contrast to DESQ-COUNT, it performs well also for pattern expressions that produce many candidate sequences per input sequence.

2.4 Apache Hadoop and Apache Spark

MapReduce is a programming model and a runtime system originally developed at Google (Dean and Ghemawat, 2004). Originally a proprietary system, the MapReduce programming model has been widely adopted and was implemented in many systems, most commonly known in the popular open-source project *Apache Hadoop*[1].

The MapReduce programming model aims to facilitate distributed data processing on a cluster of network-connected computers, which are typically called the *nodes* of the cluster. Typically, data for MapReduce computational tasks is distributed among these nodes. Drawing from functional programming, MapReduce allows applications to define *map* and *reduce* functions. These functions are executed in parallel on parts of the data. Map functions execute steps on data records in parallel, reduce functions combine parts of the data, which makes communication between the nodes necessary. The run time system takes care of load balancing, redistributes data between consecutive map and reduce calls, and handles node failures.

Recent versions of Apache Hadoop implement the MapReduce programming model, the distributed file system HDFS (Shvachko et al., 2010), and a run time system with a resource management platform YARN (Vavilapalli et al., 2013).

Apache Spark (Zaharia et al., 2010) is an open-source data flow engine that builds on the MapReduce programming model. It aims to make MapReduce more applicable for iterative tasks and simplifies the programming interface. Its key modification to Hadoop is that it seeks to keep intermediary results in memory whenever possible. For some tasks, this modification can lead to significant speed-ups (Zaharia et al., 2010).

We use Apache Spark as computational model for our implementation because it is widely adopted and because it offers several additional high-level functions that facilitate the development of distributed applications compared to Hadoop MapReduce. For example, such functions make it convenient to pre-process datasets, which makes our implementation comfortable to use for others.

2.5 Related work on distributed FSM

Sequential Pattern Mining is an active field of research, with many proposed algorithms. Mabroukeh and Ezeife (2010) and Mooney and Roddick (2013) give an overview of the field. Many of these algorithms run only sequentially and, consequently, are of limited use for large datasets. To mine large datasets efficiently, parallel algorithms have been developed for shared-memory architectures (Zaki, 2001) and distributed memory architectures (Guralnik et al., 2001; Guralnik and Karypis, 2004), but without support for constraints or hierarchies.

More recently, algorithms for specific applications of sequential pattern mining (Zihayat et al., 2016) and algorithms for parallel frequent itemset mining (Qiu et al., 2014) have targeted Apache Spark as a computational platform. Apache Spark's integrated machine learning library MLLIB (Meng et al., 2016) features a distributed version of PrefixSpan (Pei et al., 2001) for distributed FSM, but also without hierarchies or subsequence constraints.

[1] `https://hadoop.apache.org`

Most related to our work is a group of distributed sequential pattern mining algorithms targeted towards the MapReduce programming model: Suffix-σ (Berberich and Bedathur, 2013), MG-FSM (Miliaraki et al., 2013; Beedkar et al., 2015), and LASH (Beedkar and Gemulla, 2015).

Suffix-σ is an algorithm developed for one specific case of FSM: mining of frequent n-grams. An n-gram is a consecutive subsequence consisting of n items. Suffix-σ can mine all n-grams up to a specified length in one map and one reduce step. This is achieved by careful partitioning and, compared to other methods that Berberich and Bedathur study, which execute multiple map and reduce steps in succession, decreases run times significantly.

MG-FSM is a scalable distributed algorithm for FSM. It is also developed for the MapReduce programming model and also runs in one map and one reduce step. It allows for gap and length constraints. In the map step, MG-FSM processes input sequences in parallel. It then groups input sequences into partitions such that partitions can mine for frequent sequences independently. To group the extracted information, MG-FSM uses *item-based partitioning*.

Item-based partitioning divides the space of possible frequent sequences by one particular item in each sequence. The idea was introduced by Han et al. (2000) for frequent itemset mining and is used in several algorithms for parallel frequent itemset mining (Buehrer et al., 2007; Cong et al., 2005; Li et al., 2008). The partitioning in MG-FSM creates one partition $\mathcal{P}_\omega$ for each frequent item ω in the vocabulary. This item ω is called the *pivot item* of a partition.

At each partition, subsequences with 'largest' item ω are mined and output. This notion of the 'largest' item is based on the total order established by the f-list. This f-list based ordering is also used in FP-Growth (Han et al., 2000) and PFP, a parallel version of FP-Growth (Li et al., 2008). As an input sequence can have multiple subsequences and these subsequences might have different 'largest' items, an input sequence might be required for the mining process at different partitions. An input sequence is sent to all partitions where it contributes at least one relevant subsequence.

To reduce the communication between the nodes of the cluster, MG-FSM introduces a couple of techniques. First, it only sends an input sequence to partitions for which it can produce a relevant subsequence. Further, MG-FSM proposes a number of rewrites for the sent input sequences. It eliminates parts of input sequences that are not relevant for a specific partition. Parts that are irrelevant but cannot be omitted completely are rewritten so they can be sent more concisely.

To achieve correctness for these input sequence rewrites, MG-FSM introduces the notion of ω-equivalency. Let T be an input sequence that produces a certain set of subsequences relevant for partition $\mathcal{P}_\omega$. A rewritten input sequence T' is ω-equivalent to the original input sequence T if it produces the equivalent set of subsequences relevant for partition $\mathcal{P}_\omega$. Consequently, T' can be sent to the partition instead of T without sacrificing correctness. If T' is more compact than T, this reduces the amount of communication between the nodes.

LASH (Beedkar and Gemulla, 2015) extends MG-FSM with hierarchies. An item is allowed to generalize to another item in the item hierarchy. As MG-FSM, it supports maximum gap and maximum length parameters. LASH also runs in a single map and reduce step and adapts the item-based partitioning scheme. It introduces a hierarchy-

aware version of item-based partitioning, which creates partitions for generalized items as well. It adapts the notion of ω-equivalency to work with hierarchies.

LASH further introduces a local mining algorithm that is designed specifically for the item-based partitioning scheme. The algorithm is based on the pattern-growth approach. At each partition $\mathcal{P}_\omega$, it makes sure to produce only frequent sequences that are relevant for pivot item ω. LASH starts pattern-growth by setting the growing sequence to the pivot item ω. It then expands the sequence to both the left and the right by appending items. This makes sure that the mining process does not generate frequent sequences that do not contain the pivot item and are therefore irrelevant for this partition.

To the best of our knowledge, no distributed or parallel algorithm has yet been proposed for FSM with subsequence constraints based on regular expressions or pattern expressions.

Chapter 3

Distributed FSM with declarative subsequence constraints

3.1 A naive approach: distributed DESQ-COUNT

Before we develop a more efficient algorithm in the following sections, we first use this section to discuss a naive approach to the problem and its limitations.

3.1.1 Algorithm

For the sequential algorithm DESQ-COUNT proposed by Beedkar and Gemulla (2016a), there exists a natural distributed version. One can express this distributed version using one *map* step, one *reduce* step, and one *filter* step in Apache Spark or the MapReduce framework in general. We assume that each of the nodes of the cluster holds a part of the input sequences. We further assume that single item frequencies are known before the algorithm is run. That is, we send a copy of the f-list to each node before we start the mining. The mining algorithm proceeds as follows:

1. In the map step, we produce the set of frequent candidate sequences $C(T, \pi)$ for each input sequence in parallel.

2. We aggregate the produced candidate sequences by count. For that, it is necessary to collect all occurrences of one candidate sequence at one node. This can be done efficiently in a reduce step.

3. We output the candidate sequences that have a count larger or equal σ. This can be done in a filter step.

We now discuss each step in more detail.

First, in the map step, one uses the FST to produce the frequent candidate sequences for every input sequence. This works as it does in sequential DESQ-COUNT, only that we generate the frequent candidate sequences in parallel: the generation of the frequent candidate sequences depends only on the FST and σ and can consequently be done independently for each input sequence. By passing the FST to all participating nodes in the cluster, each node can generate the candidate sequences for its share of the sequence database. As in the sequential version of DESQ-COUNT, it is not necessary to produce any

candidate sequences that contain an infrequent item, because these candidate sequences cannot be globally frequent.

Second, we aggregate the produced candidate sequences by count. To do this, all counts of one candidate sequence have to be sent to one node. For example, assume a candidate sequence C_1 is produced from 200 input sequences at node 1 and from 150 input sequences at node 2; and σ is set to 300. To decide that C_1 is frequent, one node needs to know about all occurrences of C_1, for example, node 1. Each candidate sequence C is assigned to one node of the cluster and information about all occurrences of that candidate sequence C is sent to this node. As we merely count the number of occurrences of a candidate sequence, the counts can be aggregated locally before they are sent to the responsible node. For example, we can send information (C_1, 200) from node 1 to node 2. Node 2 then sums up the two counts to a total count of 350. In Apache Spark, an adequate reduce function takes care of this aggregation.

Third, with the total count for a candidate sequence, a node can decide whether a candidate sequence is frequent and should, therefore, be output, or infrequent and, therefore, should not be output. This can be expressed as a filter step, in which all candidate sequences with a total count smaller than σ are not output. For the example candidate sequence C_1, node 1 decides that, with a total count of 350, C_1 is (π,σ)-frequent and therefore outputs the sequence with its total count 350.

3.1.2 Limitations

As the sequential version, the distributed version of DESQ-COUNT works well for pattern expressions that produce a manageable number of candidate sequences for each input sequence.

However, the distributed version of DESQ-COUNT has the same limitation as the sequential variant. It generates and communicates all frequent candidate sequences. For pattern expressions that are not selective, this can exponentially increase the amount of memory or disk space to hold the candidate sequences. Sending an exponential number of candidate sequences and aggregating each of them by count can lead to slow run times.

Let us consider a worst case scenario: a pattern expression that generates all possible subsequences. As discussed before, when assuming that each item in the sequence can generalize to four other items and n being the length of the input sequence, the maximum number of subsequences is $(2 + 4)^n - 1$. Assume we have an input sequence of length $n = 200$, which is not untypical – some input sequences in the real-world datasets we use in our experiments are longer than 20 000 items. For $n = 200$, there is a theoretical maximum of 4.26×10^{155} possible subsequences. If we assume an average length of 100 for the subsequences and that we store each item in a subsequence as a 4-byte integer, we would need $((2 + 4)^{200} - 1) \times 100 \times 4 \approx 1.7073 \times 10^{158}$ bytes to store all candidate sequences. That is approximately 1.5528×10^{146} terabytes – much more than one machine can typically hold.

This is a worst case scenario. The purpose of allowing pattern expressions is to be more selective than allowing all subsequences. Also, the actual number of subsequences is usually lower than the theoretical maximum, as items might reoccur within a sequence and, therefore, do not create distinct subsequences. Also, items in the hierarchy might

reoccur, and not all item hierarchies have four generalizations for each item. On the other side, real world datasets can contain input sequences longer than 200 items.

In our experiments, generating all candidate sequences lead to a significant increase in run time also for pattern expressions that produce fewer candidate sequences than the theoretical maximum. For their sequential algorithms, Beedkar and Gemulla (2016b) show that DESQ-DFS, which does not produce all candidate sequences, can run up to 22 times faster than DESQ-COUNT for such pattern expressions. To mine frequent sequences for these pattern expressions efficiently in the distributed setting, we need a different strategy for distributing the work across nodes of the cluster. We discuss a framework for such an algorithm in the following section and describe its components in more detail in the sections that follow.

3.2 Overview of the proposed algorithm

The key challenge for a parallel algorithm is to decompose a large computational problem into smaller problems that can be solved independently. In a distributed algorithm, the smaller problems are processed on physically separate computers.

In the previous section, we have seen one approach to doing this. The distributed version of DESQ-COUNT splits the work of generating candidate sequences, exchanges information between the nodes, and then aggregates and filters independently again. We have also seen the limitations of this approach. In this section, we lay out the framework for a distributed algorithm that can overcome these limitations for many pattern expressions. Our method carefully partitions the work such that most of the work can run in parallel. The approach does not require to produce all candidate sequences. In this section, we give a general intuition of how the approach works before we look at each part of the approach in more detail in the upcoming sections.

For this approach to work, we need a way to group candidate sequences. The key requirement is still that all occurrences of one candidate sequence C_1 have to be sent to one node, such that this node can decide whether candidate sequence C_1 occurs more than σ times in the sequence database $\mathcal{D}$. For now, assume that we have a way to do this: suppose that we have a scheme to assign one candidate sequence to one *partition*. Many candidate sequences can belong to one partition, but one candidate sequence belongs to exactly one partition. We discuss the scheme in more detail in Section 3.3.

A partition is assigned to one node of the cluster. Typically, in our scheme, the number of partitions is larger than the number of nodes, so each node is assigned multiple partitions. The nodes can then process the assigned partitions independently and in parallel.

Each node needs to receive the candidate sequences from all input sequences that contribute towards its partitions. We discussed before that one input sequence T can produce no or multiple candidate sequences. Therefore, one input sequence can contribute information to zero or more partitions.

Our algorithm is executed in two main stages, with a communication stage in-between:

1. In parallel at each node, run the FST on each input sequence and determine for which partitions the input sequence produces candidate sequences.

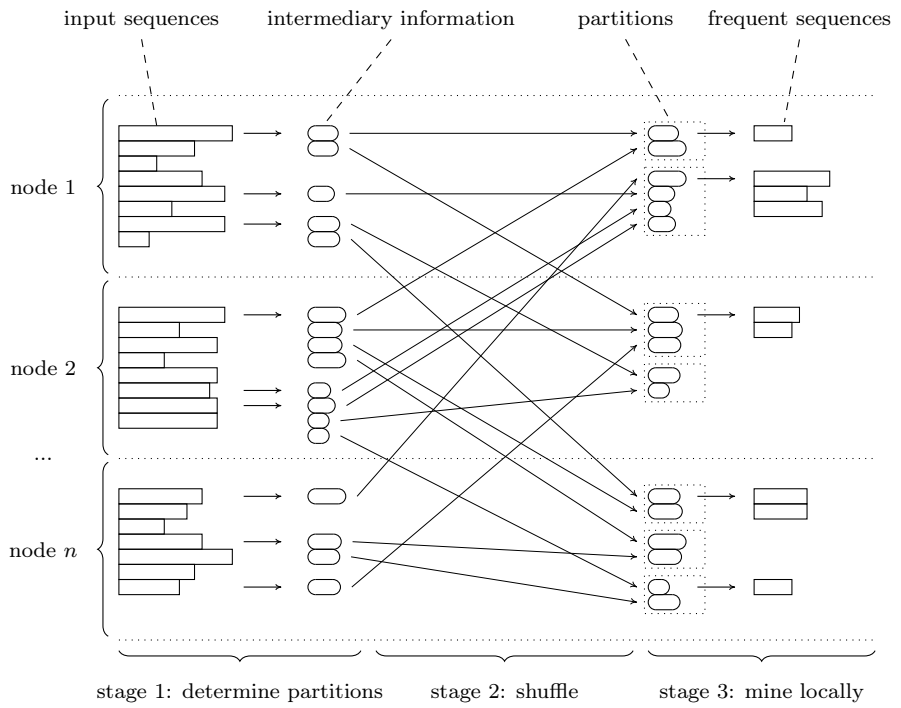


Figure 3.1: Overview of the three stages of the proposed algorithm

2. Assign each partition to a node and send information about each candidate sequence to the node responsible for its partition.

3. In parallel at each node, mine for frequent sequences in the partitions.

Figure 3.1 portrays this approach. One horizontal lane represents one node in the cluster. Each of these nodes holds a part of the input data at the start of stage 1, depicted as small rectangles with sharp corners. In step 1, each node processes each of the input sequences it holds. The goal is to decide for which partitions the input sequence generates candidate sequences and to extract information about these candidate sequences that we can send to the partition. Packages of extracted information are depicted as rectangles with rounded corners. In stage 2, after assigning partitions to physical nodes, the extracted information is redistributed to the nodes responsible for the partitions. The redistribution of data between the nodes of a distributed system is commonly referred to as a *shuffle*. In stage 3, each node mines its assigned partitions and outputs found frequent sequences. In the figure, partitions are indicated as dotted lines around groups of intermediary information. Frequent sequences are depicted as rectangles with sharp corners. Each partition outputs only frequent sequences that belong to its partition.

This two-stage approach with one shuffle is a well-known approach in distributed data processing and is also frequently employed in distributed FSM algorithms (Miliaraki et al., 2013; Beedkar and Gemulla, 2015): data is processed in parallel, redistributed between the nodes of a cluster, and then processed in parallel again. The three stages of the algorithm can be seen as a map, a shuffle, and a reduce step of the MapReduce programming model.

We now discuss each of these steps in more detail. The following section, Section 3.3, discusses our method to partition the candidate sequences. Section 3.4 explains how we can send information about the candidate sequences to the partitions. Section 3.5, Section 3.6,

and Section 3.7 discuss how to send this information efficiently. Section 3.8 then deals with the third stage of our approach, the local mining.

3.3 Decomposing the problem using item-based partitioning

So far we assumed that we have a way to partition the space of all possible candidate sequences into a number of partitions, such that each partition can be processed independently. We now discuss this method in more detail.

We have seen the approach of DESQ-COUNT, which can be seen as generating one partition for each candidate sequence. We have also seen that for some pattern expressions, this approach is infeasible, as it produces and communicates all candidate sequences.

We adopt the item-based partitioning framework of MG-FSM (Miliaraki et al., 2013), which is also used in LASH (Beedkar and Gemulla, 2015). We further adopt MG-FSM's notion of ω-equivalency for rewriting input sequences.

3.3.1 Pivot items

Key intuition is to assign a sequence, in our case, each candidate sequence, to a partition based on one specific item in the sequence. For example, one could choose the first item in a sequence. If the first item of a sequence is a_1, it is assigned to partition a_1. We call the item that decides about the partition of a sequence the *pivot item*.

Instead of choosing the first item of a sequence, we adopt the approach chosen in MG-FSM (Miliaraki et al., 2013; Beedkar et al., 2015) and LASH (Beedkar and Gemulla, 2015). These algorithms pick as pivot item the least frequent item in a sequence. As there might be two least frequent items ω and ω' with $f(\omega) = f(\omega')$ in a sequence, we establish a total order on the items of the item vocabulary, based on item frequency in the sequence database. Formally, we define a total order on all items of the item vocabulary Σ, such that either $\omega < \omega'$ or $\omega > \omega'$ for all $\omega, \omega' \in \Sigma,\ \omega \neq \omega'$. The ordering is based on item frequency, such that for $\omega < \omega'$ it must hold $f(\omega) \geq f(\omega')$. In cases of equal frequency $f(\omega) = f(\omega')$, ties are broken arbitrarily. The order in which the items appear in the f-list is such a total order.

For our running example, the ordering is as follows:

$$A < B < e < a_1 < b_1 < b_2 < d < a_2 < b_{12} < c < b_{11} \text{ with}$$

$$f(A) = f(B) = f(e) > f(a_1) > f(b_1) = f(b_2) = f(d) > f(a_2) = f(b_{12}) = f(c) > f(b_{11}).$$

This method to order items might seem counter-intuitive, as, for example, item e is 'smaller' than item a_1, although e is more frequent. We adopt this notation from MG-FSM and LASH for consistency, but acknowledge that it can take some getting used to for readers at first. It can help to think of the order in terms of the f-list: the order is equivalent to the order the items appear in the f-list: A is the first item in the f-list, B is the second item in the f-list, and so on. With this order, we can define the pivot item of a sequence.

Definition 6 (Pivot item of a sequence) *We denote as the pivot item of a sequence S the largest item in the sequence. We write*

$$p(S) = \{\omega \mid \omega \geq \omega' \text{ for all } \omega' \in S\}$$

Informally, the pivot item of a sequence S is that item ω in the sequence that is the furthest down in the f-list. The total order ensures that for every given sequence, there is an unambiguous pivot item such that each candidate sequence can unambiguously be assigned to exactly one partition.

In our approach, we assume that the f-list and therefore this order is known at every node before we start the partitioning. In our implementation, we store a copy of the f-list with the sequence database and send it to all nodes at the start of the algorithm. If the f-list is unknown, it can be computed efficiently in one map and reduce job: the map counts the generalized frequencies of items in all sequences in parallel, the reduce sums them up. We then sort items by decreasing frequency at one of the nodes.

3.3.2 Partitions

We generate one partition $\mathcal{P}_p$ for every frequent item p. Using the intermediary information sent to the partition, each partition mines for frequent sequences independently of the other partitions.

At partition $\mathcal{P}_p$, frequent sequences with pivot item p are mined and output. Consequently, frequent patterns found at partition $\mathcal{P}_p$ contain pivot item p at least once and otherwise, only items smaller p. If a sequences contains an item $\omega > p$, it is output at partition $\mathcal{P}_\omega$ if $p(S) = \omega$ or $\mathcal{P}_{p(S)}$ if the sequence contains items larger ω.

There is no need to construct partitions for infrequent items, that is, items ω with $f(\omega) < \sigma$. As we established in Section 2.3.2, a frequent sequence cannot contain an infrequent item. All sequences that $\mathcal{P}_\omega$ would output contain ω at least once and are therefore infrequent.

To produce correct results, that is, to output all frequent sequences with the correct frequency, it is necessary that each partition $\mathcal{P}_p$ is provided with information about all input sequences that produce a frequent candidate sequence with pivot item p.

3.3.3 Partitions for an input sequence

In the first step of our algorithm, for each input sequence T, it is necessary to determine which partitions require information about T. We define the set of pivot items of an input sequence as follows.

Definition 7 (Pivot items of an input sequence) *Given a pattern expression π and a minimum support value σ, an input sequence T produces a set of frequent candidate sequences $C(T, \pi)$. We call the set of pivot items of these candidate sequences the pivot items of the input sequence T:*

$$P(T, \pi) = \{p(S) \mid S \in C(T, \pi)\}.$$

We say that an input sequence T is relevant for all partitions $\mathcal{P}_p$ for $p \in P(T, \pi)$. Note that $P(T, \pi)$ contains only frequent items, as the candidate sequences in $C(T, \pi)$ contain only frequent items.

Figure 3.2 presents an example for item-based partitioning for our running example with pattern expression $\pi_1 = A([c|d][A^\uparrow|B^\uparrow]^+e)$. We set the minimum support value to $\sigma = 2$. For each input sequence, we have zero or more frequent candidate sequences $C(T, \pi)$. Each

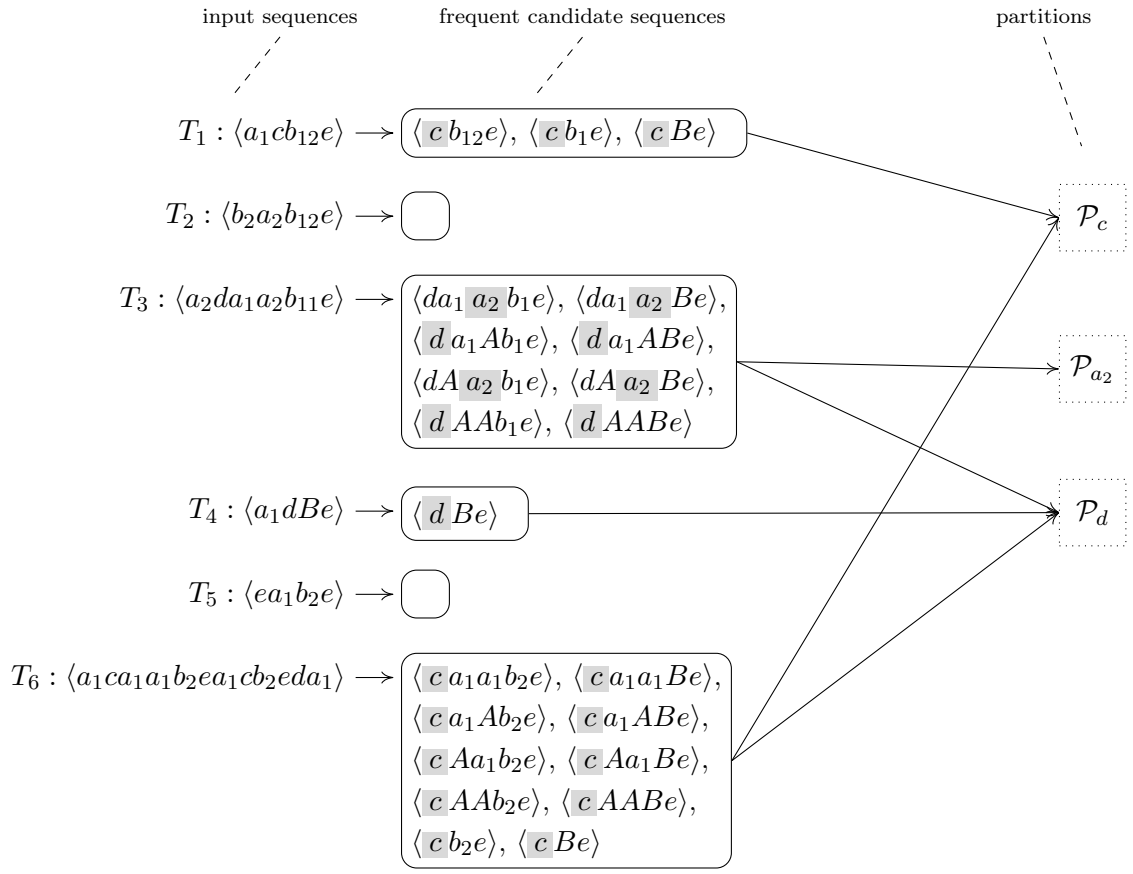


Figure 3.2: Item-based partitioning for pattern expression $\pi_1 = A([c|d][A^{\uparrow}|B^{\uparrow}]^{+}e)$ and minimum support $\sigma = 2$ on example sequence database $\mathcal{D}_{ex}$. Pivot items are marked by gray shading.

of the candidate sequences has one pivot item. In the figure, the pivot items are marked by gray shading. An input sequence is relevant for the partitions corresponding to the pivot items of all its candidate sequences. For example, sequence T_1 produces three candidate sequences, which all have item c as pivot item. Therefore, input sequence T_1 is relevant for partition $\mathcal{P}_c$. T_3 on the other hand produces frequent candidate sequences with pivot items a_2 and d, so it is relevant for the partitions $\mathcal{P}_{a_2}$ and $\mathcal{P}_d$.

For this example, there are three non-empty partitions $\mathcal{P}_c$, $\mathcal{P}_{a_2}$, and $\mathcal{P}_d$. The other partitions are empty, that is, they do not receive any intermediary information and consequently do not output any frequent sequences.

3.3.4 Discussion

Beedkar et al. (2015) show that choosing the least frequent item as pivot item can lead to more evenly distributed partition sizes. Partitions of frequent pivot items tend to receive information from many input sequences, but the information tends to be short, as longer sequences are more likely to contain other, less frequent items. These sequences are sent to other partitions. Partitions of less frequent pivot items tend to receive information from

fewer input sequences, but the size of the sent information is larger, as the sequences are longer. On average, this seems to lead to more homogeneous partition sizes.

As LASH, we use a hierarchy-aware variant of item-based partitioning. Assume input sequence T contains item b_{11}, which generalizes to b_1 and B. If all of them are pivot items, input sequence T is relevant for partitions $\mathcal{P}_{b_{11}}$, $\mathcal{P}_{b_1}$, and $\mathcal{P}_B$. This is crucial for the correctness of our framework, but requires many sequences to be shuffled, as the partitions $\mathcal{P}_{b_{11}}$, $\mathcal{P}_{b_1}$, and $\mathcal{P}_B$ might end up at different nodes. In future work, it could be interesting to examine whether one can decrease shuffle size by grouping partitions.

If all candidate sequences have one identical pivot item, item-based partitioning creates only one partition and effectively runs sequentially in the third stage. More specifically, there need to be more distinct pivot items than the desired degree of parallelism in the third stage. Ideally, there are many more distinct pivot items than the desired degree of parallelism to accommodate longer-running partitions. For typical pattern expressions, this should not be an issue. In all our experiments, the number of partitions is significantly higher than the desired degree of parallelism.

3.4 Sending information to the partitions

With item-based partitioning, we need to decide how to communicate the candidate sequences of an input sequence to the relevant partitions. There are two main options to do this, which we discuss in the following: sending the input sequence and sending candidate sequences.

The main requirement for the information we send from an input sequence T to a partition $\mathcal{P}_p$ is that we can reproduce all frequent candidate sequences of T with pivot item p from this information. It is not necessary to reproduce candidate sequences that contain infrequent items, as they cannot be a frequent sequence. We define the set of frequent candidate sequences with pivot item p as follows:

Definition 8 (Frequent candidate sequences for one partition) *The frequent candidate sequences of one input sequence T that are relevant for the partition $\mathcal{P}_p$ of pivot item p are the frequent candidate sequences of T that have pivot item p. We write:*

$$C_p(T, \pi) = \{S \in C(T, \pi) \mid p(S) = p\}$$

From each input sequence T, we aim to send information to partition $\mathcal{P}_p$ such that we can reconstruct $C_p(T, \pi)$ at the partition.

Typically, this involves significant network communication between the nodes of the cluster, which is typically a bottleneck in distributed algorithms. So we further want the information we send to the partitions to be compact in size. Of course, both constructing and reading the representation should be reasonably fast.

3.4.1 Sending the input sequence

To send input sequences as information, one proceeds as follows. For each input sequence T, we check whether it matches the given pattern expression by running the FST. If it matches, we determine the relevant partitions $P(T, \pi)$ of the input sequence and send the input sequence to these partitions.

At the partition, we can generate all frequent candidate sequences relevant for the partition by running the FST on the input sequence. The FST produces all frequent candidate sequences $C(T, \pi)$, including candidate sequences with pivot items other than p, which are irrelevant for this partition. The local mining at the partition discards these candidate sequences. For example, at partition d, input sequence T_3 produces candidate sequence $\langle da_1a_2b_1e\rangle$, which has pivot item a_2 and is therefore irrelevant for partition d.

Sending the input sequence is a comparably concise option for pattern expressions that produce a large number of candidate sequences for one pivot item. In our example, T_3 produces four candidate sequences for partition d. In this case, it is more concise to send input sequence T than to send the four candidate sequences.

Sending the input sequence means that the FST needs to be run twice on this input sequence. Once to determine the partitions for the input sequence and once at the partition, to produce the set of candidate sequences.

3.4.2 Sending candidate sequences

For some pattern expressions, it can be more concise to send the set of frequent candidate sequences. This is usually the case when pattern expressions generate a small number of candidate sequences per pivot item or if these candidate sequences are shorter than the input sequence.

Say, for example, that an input sequence generates ten candidate sequences that are much shorter than the input sequence, and each candidate sequence has a distinct pivot item. Then one could either send the long input sequence to the ten partitions or each short generated candidate sequence to its partition.

To send the candidate sequences, one would proceed as follows. For each input sequence T, we check whether it matches the given pattern expression by running the FST. If it matches, we produce the frequent candidate sequences and send the candidate sequences for a pivot item $C_p(T, \pi)$ to the corresponding partition $p \in P(T, \pi)$.

Sending candidate sequences has one crucial advantage: at the partitions, it is not necessary to run the FST again. As FST simulation is the most time-consuming part for most pattern expressions, this is a considerable advantage, which has proven crucial in our experiments.

3.4.3 Discussion

If candidate sequences are shorter than the input sequences, one could consider sending only the relevant parts of an input sequence. One could drop parts of the input sequence that are not relevant for the pattern expression. For example, in input sequence T_6, one could drop the items after the last e, producing $T_6' = \langle a_1ca_1a_1b_2ea_1cb_2e\rangle$. This trimmed input sequence T_6' produces equivalent candidate sequences and can thus be sent to the partitions instead of T_6. Beedkar et al. (2015) propose similar rewrites for MG-FSM. For the setting of traditional subsequence constraints, they propose to send only relevant parts of the input sequences. To adapt these rewrites to the more general case of pattern expressions, additional work is required.

There might be additional potential to send even more concise input sequences. Uncaptured items, such as the first A in pattern expression $A([c|d][A^{\uparrow}|B^{\uparrow}]^{+}e)$, are sent to the

partition as part of the input sequence, but do not contribute items to candidate sequences. However, these items are relevant for the pattern expression to match. If we drop these items from the input sequence, the pattern expression does not match the input sequences. One could, for example, find ways to replace these items by placeholders, potentially reducing the size of necessary communication. This idea is used by Beedkar et al. (2015) for traditional subsequence constraints. They propose to replace items, which are irrelevant for the produced subsequences, but crucial to conform to maximum-gap constraints, by placeholders, which can be encoded more concisely than the original items. In the setting of pattern expressions, such placeholders might be an option, but one could also consider modifying the FST.

Despite these considerations, one crucial disadvantage of sending the input sequence remains: the FST has to be run twice on each relevant input sequence. Once in the map stage to find the partitions relevant for the input sequence and once in the mine stage to produce the set of candidate sequences. As the FST simulation is the most computationally expensive component of our algorithm, this is an important factor.

For this reason, we focus on sending candidates sequences in this work. Nevertheless, we hypothesize that sending the input sequence with appropriate rewrites of the input sequence and adapted local mining can work well for pattern expressions that are similar to traditional FSM, that is, produce many subsequences per input sequence.

For pattern expressions that produce a small number of candidate sequences, one could send a list of the candidate sequences to the partition. However, for other pattern expressions that produce a larger number of candidate sequences, this approach can lead to an exponential increase in the size of the necessary communication, as the number of candidate sequences can be exponential in the worst case.

Therefore, this approach would not improve over the distributed version of DESQ-COUNT. In the following, we propose a more concise representation for the set of candidate sequences for a partition.

3.5 Compactly representing candidate sequences

In this section, we propose NFAs as a method to compactly represent a set of candidate sequences that is sent from one input sequence T to one partition $\mathcal{P}_p$. The requirement for this representation is that we can produce the set $C_p(T, \pi)$ at the partition. Our goal is to find a representation that is compact and can be constructed efficiently.

As discussed in the previous section, the advantage of sending candidate sequences is that the FST is run only once. However, for some pattern expressions, the set $C_p(T, \pi)$ can have size exponential in the length of input sequence T. A naive approach to sending a concatenated list of the candidate sequences allows producing all candidate sequences but is inefficient or infeasible when there are many candidate sequences. Therefore, we aim to find a representation that is more compact than that. We make use of fact that often, candidate sequences share a similar structure. Ideally, we do not have to iterate all candidate sequences to construct this representation.

3.5.1 Using NFAs to represent candidate sequences

Instead of sending a list of sequences, one can use an NFA to represent a set of sequences. Each accepting path through the NFA corresponds to one sequence of the set. An accepting path through an NFA starts in an initial state and ends in an accepting state.

For our representation, we construct one NFA to produce the set of frequent candidate sequences $C_p(T, \pi)$, which is sent from one input sequence T to a partition $\mathcal{P}_p$. Instead of sending a list of candidate sequences, we send this NFA and generate the paths through the NFA at the partition.

Consider the following example. As depicted in Figure 3.2 before, the set of all frequent candidate sequences generated by input sequence T_6 for pattern π_1 is

$$\begin{aligned} C(T_6, \pi_1) = \{&\langle ca_1a_1b_2e\rangle, \langle ca_1a_1Be\rangle, \langle ca_1Ab_2e\rangle, \langle ca_1ABe\rangle, \\ &\langle cAa_1b_2e\rangle, \langle cAa_1Be\rangle, \langle cAAb_2e\rangle, \langle cAABe\rangle, \\ &\langle cb_2e\rangle, \langle cBe\rangle\}. \end{aligned}$$

All candidate sequences in the set have pivot item c, so $C_c(T_6, \pi_1) = C(T_6, \pi_1)$. An NFA that produces the set of candidate sequences $C_c(T_6, \pi_1)$ is depicted in Figure 3.3. Each accepting path through this NFA, starting from the initial state and ending in a accepting state, produces one candidate sequence of the set $C_c(T_6, \pi_1)$. We mark an accepting state by outlining it twice.

Clearly, this is a naive way to represent the candidate sequences. This NFA is not more compact than a list of the candidate sequences. Sending this NFA to the partition would actually increase shuffle size, as we would need to encode the structure of the NFA in addition to the items of the sequences. In the following, we show how we can decrease the size of the NFA and therefore decrease the amount of necessary communication.

3.5.2 Reducing NFA size

Using an NFA to represent the candidate sequences allows us to merge parts that more than one candidate sequences contains. For example, all candidate sequences in our sample NFA start with item c and end with item e. We can put each of these in one shared transition to reduce the complexity of the NFA. The NFA in Figure 3.4 shows an NFA with one shared transition for c and e. In the depicted NFA, we also merged all final states into one final state.

To make the representation as compact as possible, we aim to minimize the number of states in the NFA. Minimizing the states of an automaton is a common problem, with a couple of existing algorithms (Moore, 1956; Hopcroft and Ullman, 1979; Brzozowski, 1962). Brzozowski's algorithm (Brzozowski, 1962) is applicable for both DFAs and NFAs. Despite its exponential worst-case run time, it is frequently used due to its simplicity, straightforward implementation, and good performance in practice (Berstel et al., 2010). Applying Brzozowski's algorithm to the example NFA in Figure 3.3 results in the NFA depicted in Figure 3.5. Compared to the naive NFA, the number of states is reduced from 47 to 6.

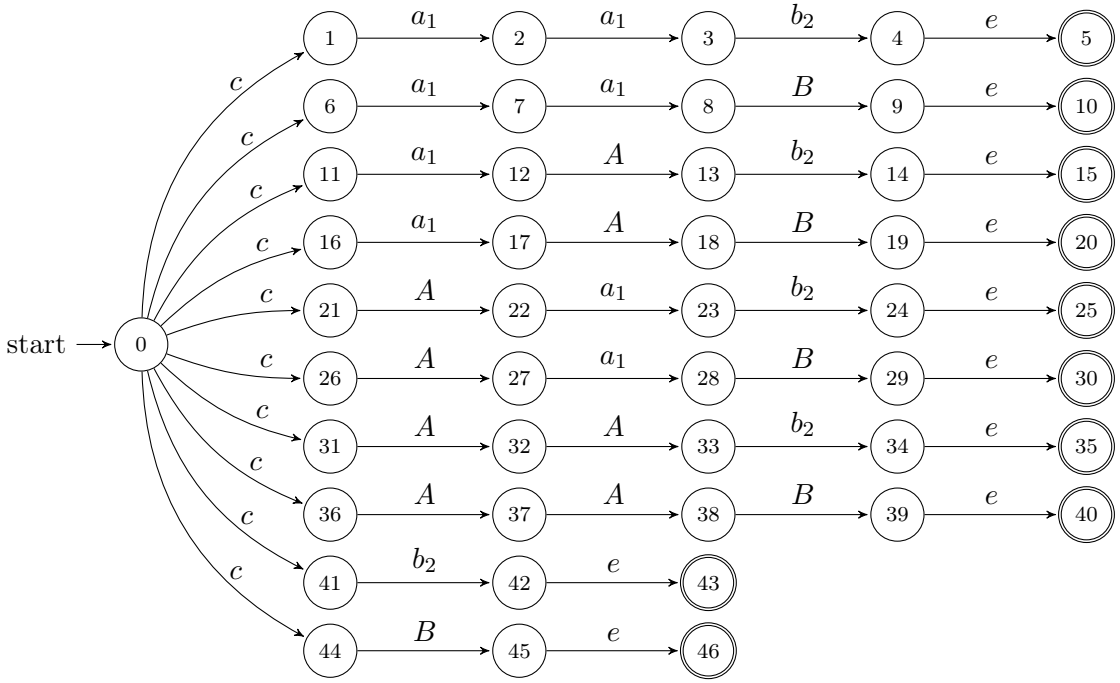


Figure 3.3: An NFA that produces the set of candidate sequences $C_c(T_6, \pi_1)$. Each accepting path through the NFA produces one candidate sequence.

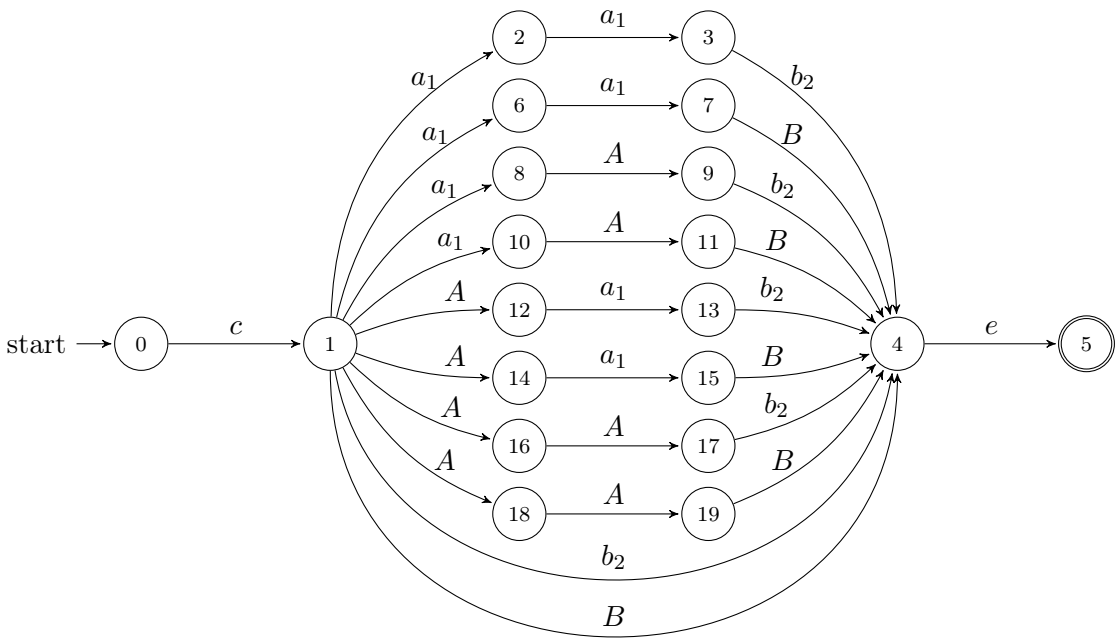


Figure 3.4: An NFA that produces the set of candidate sequences $C_c(T_6, \pi_1)$, first and last transitions merged.

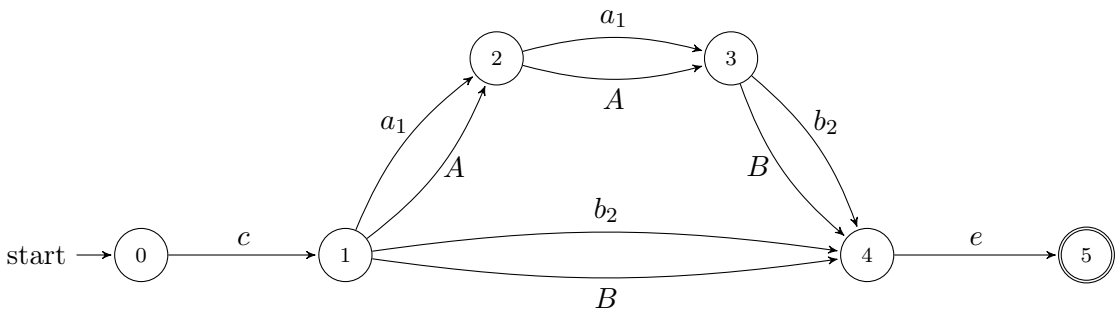


Figure 3.5: An NFA that produces the set of candidate sequences $C_c(T_6, \pi_1)$, minimized version of the NFA from Figure 3.3.

3.5.3 Discussion

An NFA with fewer states can usually be encoded more compactly for communication. However, one input sequence can potentially produce exponentially many candidate sequences. When there are many candidate sequences, generating all candidate sequences and then minimizing the NFA is inefficient. In the next section, we propose a method to build compact NFAs more efficiently.

3.6 Constructing compact NFAs efficiently

We now discuss how, given an input sequence and an FST for the pattern expression of interest, we can efficiently generate one NFA for each partition the input sequence is relevant for. Intuitively, we construct the NFA directly from the FST transitions that generate the candidate sequences.

To do this, we simulate the FST to find accepting paths through the FST. During simulation, we maintain a set of NFAs, one for each pivot item $p \in P(T, \pi)$. An accepting path can produce multiple candidate sequences, each possibly with a distinct pivot item. We add each accepting path to the NFAs for the pivot items of the path's candidate sequences.

When the simulation of the FST for T is finished, we minimize the NFAs and start sending them to the corresponding partitions.

3.6.1 Paths through an FST

In our setup, given an input sequence T, we produce the candidate sequences by simulating the FST. As proposed by Beedkar and Gemulla (2016a), we use compressed FSTs to do this. A compressed FST can produce multiple output items at each transition. We now revisit the FST for pattern expression $\pi_1 = A([c|d][A^{\uparrow}|B^{\uparrow}]^{+}e)$. We depict the FST again in Figure 3.6.

The set of candidate sequences is produced by the accepting paths through the FST. Each accepting path can produce multiple candidate sequences. Each produced candidate sequence can have a distinct pivot item. For input sequence $T_6 = \langle a_1 c a_1 a_1 b_2 e a_1 c b_2 e d a_1 \rangle$ of our running example, there are two accepting paths through the FST. Table 3.1 depicts these two paths.

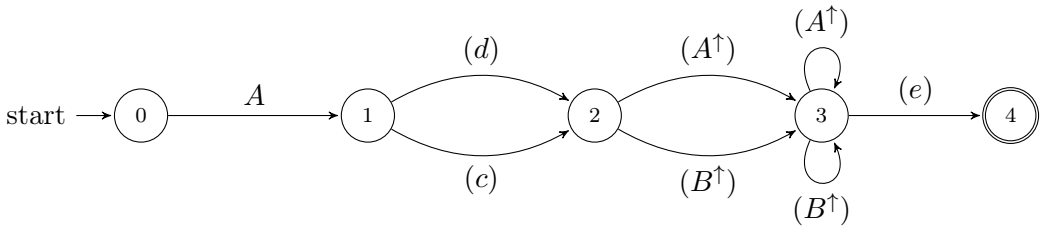


Figure 3.6: FST for pattern expression $\pi_1 = A([c|d][A^{\uparrow}|B^{\uparrow}]^{+}e)$, as seen in Figure 2.2.

Table 3.1: Accepting paths for input sequence T_6 through the FST for pattern expression π_1

	From → to states	Transition	Input item (pos.)	Output item(s)
Path 1	$0 \to 1$	A	a_1 (1)	–
	$1 \to 2$	(c)	c (2)	c
	$2 \to 3$	$(A^{\uparrow})$	a_1 (3)	a_1, A
	$3 \to 3$	$(A^{\uparrow})$	a_1 (4)	a_1, A
	$3 \to 3$	$(B^{\uparrow})$	b_2 (5)	b_2, B
	$3 \to 4$	(e)	e (6)	e
Path 2	$0 \to 1$	A	a_1 (7)	–
	$1 \to 2$	(c)	c (8)	c
	$2 \to 3$	$(B^{\uparrow})$	b_2 (9)	b_2, B
	$3 \to 4$	(e)	e (10)	e

Captured transitions as (c) or $(A^{\uparrow})$ produce output items. We define a succession of such sets of output items as an *output path*.

Definition 9 (Output path) *An accepting path through an FST produces a succession of sets of output items. If the path contains one or more non-empty sets of output items, we call the succession of non-empty sets of output items an output path. An output path R of length n is defined as a succession of n non-empty sets of output items. We write*

$$R\text{: } O_1 \text{ — } O_2 \text{ — } ... \text{ — } O_n.$$

For example, for path 1 of input sequence T_6, we write:

$$\text{Path } R_1\text{: } \{c\} \text{ — } \{a_1, A\} \text{ — } \{a_1, A\} \text{ — } \{b_2, B\} \text{ — } \{e\}$$

We omit empty sets of output items as they are not relevant for the generation of candidate sequences.

3.6.2 Determining the pivot items of a path

One path through the FST can produce multiple candidate sequences. Each candidate sequence can potentially have a distinct pivot item. The path is relevant for the partitions corresponding to the pivot items of the candidate sequences. We define the set of pivot items of a path as follows.

Definition 10 (Pivot items of a path) *Let a path R be given as a sequence of n non-empty sets of output items O_1 — O_2 — ... — O_n. Let R^+ be the set of candidate sequences that is produced by picking one item out of each O_i for $i = 1, ..., n$ and concatenating them in order as a sequence:*

$$R^+ = \{\langle \omega_1 \omega_2 ... \omega_n \rangle \mid \omega_i \in O_i \text{ for } i = 1, ..., n\}.$$

We define the set of pivot items of the path R as the set of distinct pivot items of the candidate sequences it produces:

$$P(R) = \{p(S) \mid S \in R^+\}.$$

To ensure correctness, we need to add the path R to the NFA for each pivot item $p \in P(R)$, as it produces candidate sequences that are relevant for these partitions. To do this, we need to determine the pivot items of the path. Ideally, we do this without enumerating all candidate sequences produced by the path. For our algorithm, we use a method that integrates well with the fact that we add the path to the corresponding NFA for each found pivot item. The method works with the following steps:

1. Determine the maximum item p in the path. This item is the next pivot item.
2. Add the path to the NFA for partition $\mathcal{P}_p$.
3. Remove all occurrences of item p from the path.
4. Check whether any set of output items in the path is empty now. If yes, stop. Otherwise, continue from step 1.

We discuss how to add a path to an NFA in the following subsection, Section 3.6.3.

Before that, we consider an example to see that this method finds all pivots of the path. For $T_3 = \langle a_2 d a_1 a_2 b_{11} e \rangle$ and pattern expression π_1, there is one output path through the FST:

$$R_1 : \{d\} — \{a_1, A\} — \{a_2, A\} — \{b_1, B\} — \{e\}.$$

This path creates eight frequent candidate sequences, which can be seen in Figure 3.2. The pivot items of these candidate sequences are a_2 and d. We now run the algorithm as described above on this output path. According to the f-list, the maximum item in this path is a_2 (step 1). So we add the path to the NFA for partition $\mathcal{P}_{a_2}$ (step 2). We then remove all occurrences of a_2 from the path (step 3) and obtain a modified path:

$$R_1' : \{d\} — \{a_1, A\} — \{A\} — \{b_1, B\} — \{e\}.$$

None of the sets of output items in the path are empty (step 4), so we continue in step 1. According to the f-list, the maximum item in R_1' is d (step 1). So we add R_1' to the NFAs for partition $\mathcal{P}_d$ (step 2) and remove d (step 3):

$$R_1'' : \{\} — \{a_1, A\} — \{A\} — \{b_1, B\} — \{e\}.$$

Now the first set of output items is empty, so we stop in step 4. We correctly added the path to the NFAs for pivot items a_2 and d.

This method integrates nicely with adding the path to the corresponding pivot NFA. It is unnecessary to include any items in the NFA that are larger than the pivot item: when we add a path to the NFA for partition $\mathcal{P}_p$, we do not want any items $\omega > p$ in the path, as any candidate sequence that contains ω is not a candidate sequence for partition $\mathcal{P}_p$. In our example, when we add the path to the NFA for $\mathcal{P}_d$, we already removed a_2 from the path.

We now give some intuition to see why this method finds exactly the set of pivot items for every given path. In a path R, an item ω from output item set O_i is a pivot item if there is at least one item smaller or equal ω in every other set. When we pick exactly these items from the other sets and ω from O_i, ω is the maximum item in the sequence and therefore the pivot item. If there are such smaller or equal items in every other set, the proposed method picks ω as pivot item before any other set runs empty. In the opposite case, if an item ω is not a pivot item, there is at least one other set that contains items that are all larger than ω. This set runs empty and causes the method to stop before ω is picked.

We investigated one other method for determining the pivot items of a path. It merges the sets of output items of a path according to a specific rule. We briefly describe this method in Appendix A. We picked the method described above because it integrates well with adding the path to the NFAs. However, we estimate that the method described in Appendix A can be more efficient if one does not construct NFAs. For example, that is the case when we send input sequences to the partitions instead of candidate sequences.

3.6.3 Constructing one NFA for each partition

While simulating the FST on input sequence T to find accepting paths, we maintain a set of NFAs: one NFA for each pivot item $p \in P(T, \pi)$. We add a path R to the NFAs for all the pivot items $P(R)$ of the path. If an NFA for a pivot item $p \in P(R)$ does not exist yet, we create it.

To add a path R to an NFA, we use a straightforward process for building a tree from a set of strings, or, in our case, sequences. The sets of output items of the path become the labels of edges in the tree:

1. Initialize q to be the initial state of the NFA and o to be the first set of output items of path R.
2. Check whether q has an outgoing transition with label o.
   - If it does, follow this transition and set q to the target state of this transition.
   - If it does not, add an outgoing transition to q with the label o, pointing to a newly created target state q'. Then set $q \leftarrow q'$.
3. Check whether there is a next set of output items in the path R.
   - If there is one, set o to this next set of output items and continue with step 2.
   - Otherwise, mark the current state q in the NFA as final and stop.

With this method, we build a tree for each partition. If we treat a set of output items as a distinct label for the automata, this is a DFA, as we make sure that no state has an outgoing transition twice.

If we do this for T_6 of our running example, we end up with one NFA for partition $\mathcal{P}_c$. No other NFA is created. We add two paths to the NFA and arrive at the tree depicted in Figure 3.7.

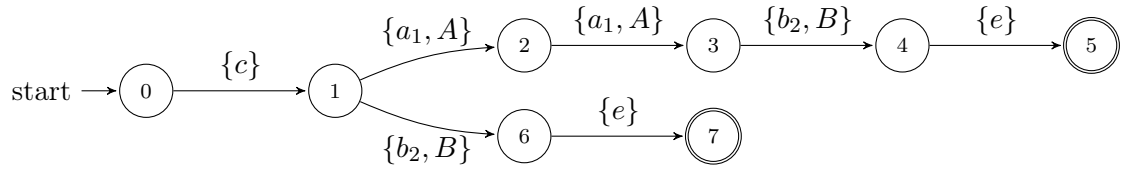


Figure 3.7: An NFA that produces the set of candidate sequences $C_c(T_6, \pi_1)$, constructed from output paths through the FST.

This NFA is not minimal. We now run an adaptation of Brzozowski's algorithm starting from the final states of the automaton backward to join paths in the NFA with a shared suffix. In this example, the two paths share the suffix $\{b_2, B\} — \{e\}$. In this case, we merge the $\{e\}$ part of the shared suffix. Figure 3.8 presents the reduced NFA, which is also a DFA.

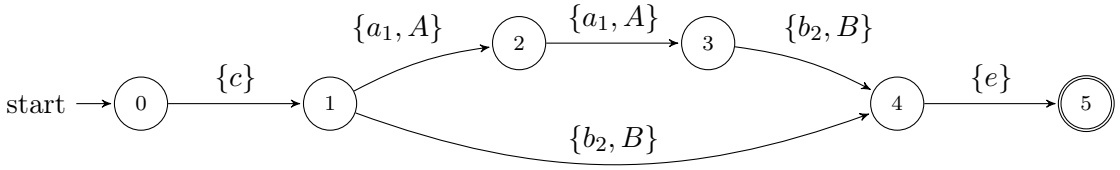


Figure 3.8: An NFA that produces the set of candidate sequences $C_c(T_6, \pi_1)$, constructed from output paths through the FST and minimized.

3.6.4 Discussion

Building an NFA directly from FST transitions is more efficient than generating all candidate sequences if the number of accepting paths through the FST is smaller than the number of candidate sequences. This is the case if the paths generate multiple candidate sequences – which they do if transitions in the paths generate multiple output items. A transition can produce multiple output items if an item can generalize to other items. So this method is especially efficient in the presence of hierarchies.

However, the method can be inefficient if there are exponentially many paths through the FST, which is the case for some pattern expressions. For these cases, our method adds many paths to the NFAs and minimizes the resulting NFA. For such pattern expressions, other approaches might be more applicable – for example, sending the original input sequence to the partition.

We explored an alternative approach to building the NFAs. In this approach, while simulating the FST, we construct only one NFA for all partitions. We add each path only

to this NFA. After the simulation, for each pivot item, we determine the parts of this NFA that are relevant for that pivot item and prepare that part for communication. In our implementation, the presented method was more efficient.

3.7 Encoding NFAs for communication

Once we finish constructing an NFA, we need to send it to the correct partition. In most of the cases, an input sequence and its relevant partitions are located at different nodes of the cluster. Therefore, we need to communicate the NFA over the network. To do this, it is necessary to encode the NFA in a way we can send over the network. This process is typically called *serialization*.

In distributed systems, sending data between the nodes via network is often slow compared to local computation. Our goal is to find a compact serialization format. It should also be efficient to write the NFA to this format and to reconstruct it later at the partition.

In this section, we first discuss how to represent an NFA for communication. Then we present and discuss two different strategies to enumerate the states and transitions of the NFA.

3.7.1 Encoding an NFA as integer array

When encoding an NFA, we encode two things: the set of output items for each transition and the structure of the NFA: from which to which state a transition goes and which states are accepting states.

In our implementation, we represent the items of the vocabulary as integers, because integers are much more compact to store than strings, for example. Therefore, it is natural to represent one NFA as an array of integers.

To achieve a more compact representation, we make use of *variable-length encoding* for these integers. In fixed-length integer encoding, each integer is typically encoded using four bytes. Variable-length encoding represents smaller integers with fewer bytes. We make sure that more frequent items are identified by smaller integers, which are then written more compactly. In our implementation, we use Apache Hadoop's variable-length integer encoding[1]. This variant of variable-length encoding represents integers $-112 <= i <= 127$ with one byte. Previous algorithms for distributed FSM have also used this approach (Beedkar et al., 2015; Beedkar and Gemulla, 2015).

We now consider the two strategies to enumerate the transitions of an NFA.

3.7.2 Serialization by state

First, we consider serialization by state. This works as follows. We run through each state of the NFA. For each state, we write information about each outgoing transition of the state. In particular, for each outgoing transition, we first write the set of output items of the transition. We then write the target state, to which the transition leads. After writing information for all outgoing transitions, we mark the end of the information about this

[1] `https://hadoop.apache.org/docs/r2.7.3/api/org/apache/hadoop/io/WritableUtils.html#writeVInt(java.io.DataOutput,int)`

state. In this marker, we include whether the state is accepting or not: the final end marker $\mathcal{E}_F$ signals that the state is accepting, the regular end marker $\mathcal{E}$ signals that it is not.

Consider the compact NFA for $C_c(T_6, \pi)$, which we depicted in Figure 3.8. Figure 3.9 presents the array that holds the serialization by state for this NFA. For easier readability, we depict output items by their letter instead of an integer identifier. State 0 is serialized first. It has one outgoing transition, which leads to state 1 with output item c. We write the output item at index 0 of the integer array and the target state at index 1. As state 0 is not a final state, we use a regular end marker $\mathcal{E}$, which we write at index 2.

The next state, state 1, has two outgoing transitions, both with two output items. The first outgoing transition leads to state 2 with the output items a_1 and A. This transition is serialized at indexes 3 to 5. The other outgoing transition leads to state 4 with output items b_2 and B. The transition is written at indexes 6 to 8. The final marker is written at index 9. We proceed accordingly for states 2, 3, 4, and 5. State 5 has no outgoing transitions, so we only note the end marker for the state, which is a final end marker $\mathcal{E}_F$, as state 5 is an accepting state.

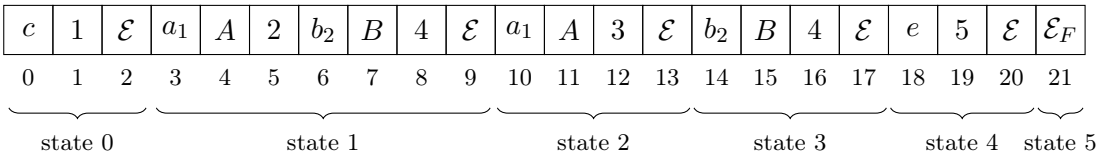


Figure 3.9: An array that holds the serialization by state for the NFA depicted in Figure 3.8. Items are represented by letters, not their integer identifiers.

To distinguish between the integers for output items, target states, and end markers, we can proceed as follows: output items are encoded as positive integers, the regular end marker is 0, the final end marker is -1, and target states are encoded as a negative integer starting from -2.

Serializing an NFA by state is a good strategy when the states of the NFA have many outgoing transitions. It is not optimal if there are many NFAs which have states with few outgoing transitions, such as our example NFA. For these simpler NFAs, serialization by state encodes much overhead for the structure of the NFA.

Imagine for example an NFA that encodes only one candidate sequence $\langle dBe \rangle$, such as the NFA that encodes $C_d(T_4, \pi_1)$. This NFA has only one outgoing transition per state and only one output item per transition. Such an NFA is depicted in Figure 3.10. Sending just the candidate sequence would require three integers, one for each of the three items in the candidate sequence. In contrast, sending the NFA includes additional information: in addition to the output items, the proposed serialization writes the target state for each transition and an end marker for each state, resulting in a total of ten integers. Compared to sending the candidate sequence alone, this is a considerable overhead.

In our experiments, we observed that, for many pattern expressions, the states in the sent NFAs tend to have few outgoing transitions, especially for the partitions of frequent items. So we implemented another way to serialize an NFA.

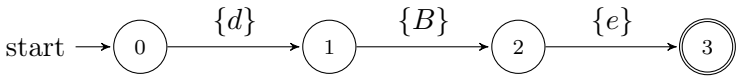


Figure 3.10: An NFA that produces the candidate sequence $\langle dBe \rangle$.

3.7.3 Serialization by path

Serialization by path allows having a compact representation of simple NFAs with few outgoing transitions per state. We can also use it to encode more complex NFAs.

In serialization by path, instead of enumerating state by state, we serialize path by path. Within a path, we do not explicitly encode the target states. We assume that a transition starts in the state the previous transition leads to. For all following paths, we only serialize new parts and refer to the states of previously written paths. For an NFA with a limited number of paths, this scheme leads to a more compact serialization.

For example, there are two paths through the NFA depicted in Figure 3.8:

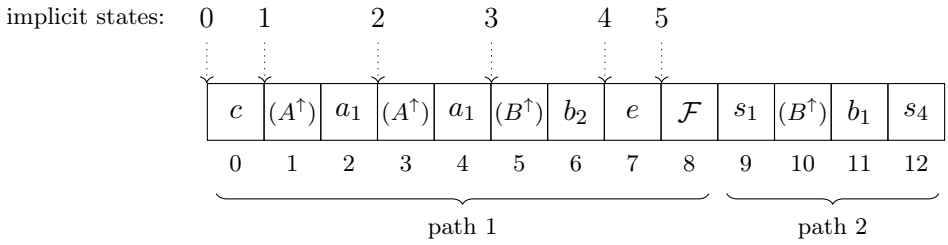
$$R_1 : \{c\} \text{ — } \{a_1, A\} \text{ — } \{a_1, A\} \text{ — } \{b_2, B\} \text{ — } \{e\}$$
$$R_2 : \{c\} \text{ — } \{b_2, B\} \text{ — } \{e\}$$

In the NFA, the two paths share transition {c} at the beginning and transition {e} at the end. We now first look at how we can write paths without explicitly noting target states. We then consider how we can serialize the output items of a transition.

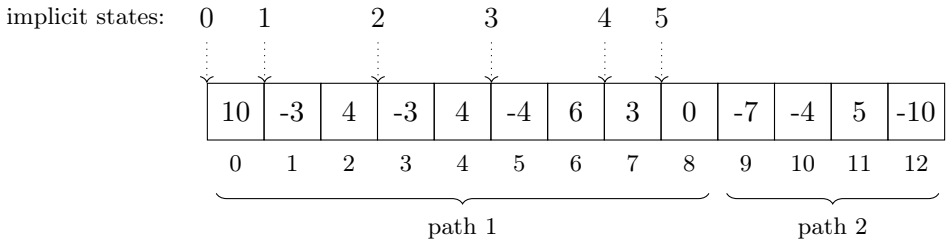
First, we serialize path R_1. We do this by serializing information about all its transitions in order. We keep an identifier for each state that we pass but do not serialize explicitly. We call these states *implicit states*. For example, we start with transition $\{c\}$ in an implicit start state s_0 and reach an implicit state s_1 with this transition. With transition $\{a_1, A\}$, we start in this implicit state s_1 and reach a new implicit state s_2. With the next $\{a_1, A\}$, we reach implicit state s_3, with $\{b_2, B\}$ we reach implicit state s_4, and with $\{e\}$ we reach implicit state s_5. If a state, which we pass implicitly, is an accepting state in the NFA, such as state s_5, so we note a final marker $\mathcal{F}$ after we serialize the transition that leads to this state, in this case after $\{e\}$. In this example, the implicit state numbers are identical to the state numbers in Figure 3.8, but this is not necessarily the case.

Next, we serialize path R_2. To do this, we serialize only the transitions of R_2 that we have not serialized with R_1 yet. That means, we only serialize transition $\{b_2, B\}$. In the NFA, the transition leads from state 1 to state 4. These states correspond to implicit states s_1 and s_4. So we write that we start in implicit state s_1, write information about this transition and then write that it leads to implicit state s_4. If there was more than one new transition in path R_2, we would write these transitions after each other as we did for the transitions in path R_1.

We now consider how to encode the output items of a transition. One transition can contain one or more output items. Serializing a list of many output items is not compact. To remedy this, if there is more than one output item, we make use of the item expression that produced this set of output items. Instead of writing a list of items in the set, we write an integer identifier for this item expression and the input item that matched this item expression. When deserializing the NFA, we can reproduce the set of output items from this information. For sets of output items that contain more than two items, this



(a) Array of symbols



(b) Array of integers

Figure 3.11: Arrays that hold the serialization by path for the NFA depicted in Figure 3.8.

encoding requires fewer integers than serializing all items. If there is only one output item in the set, we write this output item.

For example, the set of output items $\{a_1, A\}$ is produced by a transition with item expression $(A^{\uparrow})$, reading input item a_1. Instead of writing the two output items, we write an integer identifier for item expression $(A^{\uparrow})$ and input item a_1.

We now put all of this together in an example. We depict the serialization by path of the NFA for $C_c(T_6, \pi_1)$ from Figure 3.8 in Figure 3.11(a). The first path R_1 is serialized at indexes 0 to 8. The implicit state reached with each transition is indicated above the array. For transitions with one output item, as $\{c\}$ for example, we write the output item directly. For sets with more than one output item, such as $\{a_1, A\}$, we write item expression identifier and input item. We add a final marker at index 8 for the state reached with transition $\{e\}$. Path R_2 is serialized at indexes 9 to 12. We write the implicit start state s_1 at index 9, the item expression and input item at indexes 10 and 11, and the implicit target state s_4 at index 12.

To encode this representation as an integer array, we carefully partition the range of integers. We use positive integers to encode output items and input items. We use the 0 to encode the final marker $\mathcal{F}$. We use negative integers for both item expression identifiers and states. We give each distinct item expression of an FST an integer identifier. Immediately after writing an item expression identifier, we write the input item that matched the item expression.

To distinguish between identifiers for states and identifiers for item expressions, we additionally partition the range of negative integers. Let n_{ex} be the number of distinct item expressions and n_{st} the number of states in the NFA. Then the n_{ex} largest negative integers $-n_{ex} \leq i \leq -1$ are used for item expression identifiers, the next n_{st} integers $(-n_{ex} - n_{st}) \leq i \leq (-n_{ex} - 1)$ are used for start and target states. For example, the first implicit state s_0 is represented by integer $-n_{ex} - 1$. We partition the range this way

because the number of distinct item expressions in the FST is known when deserializing the NFA, whereas the number of states in the NFA is unclear, unless we encode it explicitly in the serialization. In our experiments, $n_{ex} + n_{st}$ was usually clearly lower than 112, such that both state identifiers and item expression identifiers are usually encoded using only one byte.

When deserializing an NFA, we can distinguish between output items and input items, which we both encode as positive integers, by context. If we read an integer $-n_{ex} \leq i \leq -1$ that represents an item expression identifier, we know the next item is the input item for this item expression. Otherwise, positive integers are output items.

Figure 3.12 depicts the encoding scheme for our running example, the NFA for $C_c(T_6, \pi_1)$, with $n_{ex} = 5$ distinct item expressions used in the FST and $n_{st} = 5$ implicitly serialized states. To encode output and input items, we use their position in the f-list. That is, the most frequent item A is represented by a 1, the second most frequent item B by a 2, and so on. Figure 3.11(b) gives the serialization by path for the example NFA encoded using this scheme.

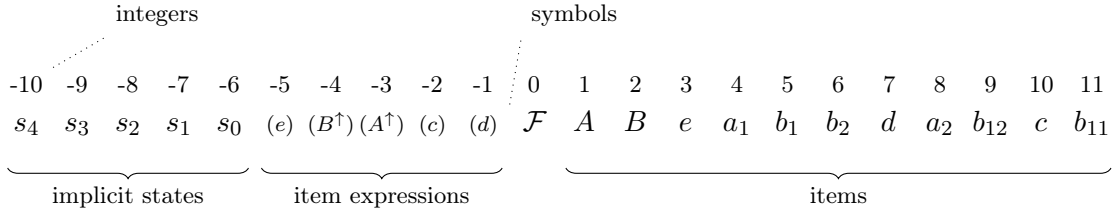


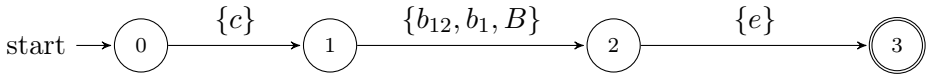
Figure 3.12: Encoding scheme for the running example with $n_{ex} = 5$ and $n_{st} = 5$

3.7.4 Discussion

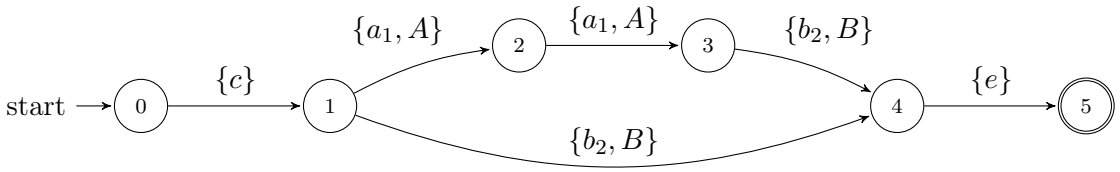
We ensure that the representation by path is canonical. That is, two identical NFAs are guaranteed to be represented by the same integer array. We achieve this by iterating paths in a fixed order. This way, depth-first search traverses the NFA in the same order for two identical NFAs. A canonical representation is important for aggregating multiple identical NFA by count.

Serialization by path is more compact than serialization by state for an NFA with a limited number of paths through the NFA. If the NFA is more complex, with many paths and therefore, many outgoing transitions per state, serialization by path can be larger than serializing by state. In our experiments, serialization by path has consistently led to lower communication between the nodes. The reason for this is that even when there are some more complex NFAs for partitions of larger pivot items, there are usually many relatively simple NFAs for partitions of smaller pivot items. For these NFAs, serialization by path can be more compact.

For $C_c(T_6, \pi_1)$, serialization by states requires 22 integers, serialization by path requires 13. For our example of an NFA with only one path, as depicted in Figure 3.10, the difference is larger. In serialization by state, we need ten integers, by path, we need only four, one for each output item and one for the final marker at the end. In our implementation, we leave out final markers at the end of the first path if there is only one path, bringing the number of necessary integers down to three.



(a) NFA N_1 for partition $\mathcal{P}_c$, generating $C_c(T_1, \pi_1)$



(b) NFA N_2 for partition $\mathcal{P}_c$, generating $C_c(T_6, \pi_1)$

Figure 3.13: Partition $\mathcal{P}_c$ in item-based partitioning for pattern expression π_1, consisting of two input NFAs, one each from input sequences T_1 and T_6.

3.8 Local mining

We now turn to local mining at the partitions. At each partition, we are given a list of NFAs $\mathcal{P}_p = \{N_1, N_2, ..., N_{|\mathcal{P}_p|}\}$ that contribute candidate sequences with pivot item p. Our goal is to find all frequent sequences with pivot item p and output each frequent sequence with its frequency.

To do this, we adapt the pattern-growth approach of PrefixSpan (Pei et al., 2001). Our approach builds on DESQ-DFS (Beedkar and Gemulla, 2016a), which we discuss in Section 2.3.3, but differs in two major ways. First, we work on NFAs that contain candidate sequences. Therefore, there is no need to simulate the FST as we run local mining. Second, we output only sequences for a specific pivot item p.

3.8.1 Pattern-growth with candidate sequence NFAs

Each of the NFAs for partition $\mathcal{P}_p$ stems from one input sequence T. It encodes the set of frequent candidate sequences $C_p(T, \pi)$ of that input sequence. A candidate sequence S at partition $\mathcal{P}_p$ is frequent if it occurs in σ or more of these sets, or: if σ or more NFAs generate S.

In this section, we consider example partition $\mathcal{P}_c$. It consists of two NFAs, one from input sequence T_1 and one from T_6. The NFAs are depicted in Figure 3.13. We refer to the NFAs as N_1 and N_2. We further set minimum support $\sigma = 2$.

An NFA N generates a sequence $S = \langle \omega_1 \omega_2 ... \omega_n \rangle$ if there is a path through the NFA that produces S. Formally, there is such a path if there is a succession of transitions $u_1 u_2 ... u_n$ in N such that: transition u_i outputs item ω_i for $1 \leq i \leq n$, transition u_1 originates in the initial state of N, and u_i originates in the target state of transition u_{i-1} for $2 \leq i \leq n$. If u_n ends in an accepting state, we call the path an *accepting path*. To decide whether a sequence is a frequent sequence or not, we define the *final support* of a sequence at a partition.

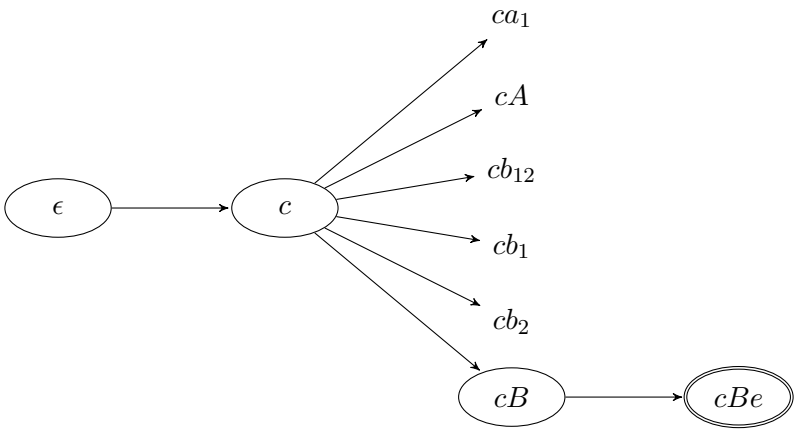


Figure 3.14: Prefix tree for partition $\mathcal{P}_c$. Prefixes with support $\geq \sigma$ are marked with an outline. Prefixes with final support $\geq \sigma$ are marked with a double drawn outline.

Definition 11 (Final support at a partition) *We define the final support of a sequence S at partition $\mathcal{P}_p$ as the set of NFAs from $\mathcal{P}_p$ that have an accepting path for S:*

$$FinalSup_{\mathcal{P}_p}(S) = \{N \in \mathcal{P}_p \mid \textit{there is an accepting path in } N \textit{ that produces } S\}.$$

A candidate sequence S is a frequent sequence at partition $\mathcal{P}_p$ if $|FinalSup_{\mathcal{P}_p}(S)| \geq \sigma$. In pattern-growth, we grow a prefix by appending items and aim to stop expanding a prefix when we are sure that it cannot lead to a frequent sequence. To do this, we further define *prefix support* as the set of NFAs that support a prefix from any path, not just accepting paths.

Definition 12 (Prefix support at a partition) *We define the support of a prefix S at partition $\mathcal{P}_p$ as the set of NFAs from $\mathcal{P}_p$ that have a path that produces S:*

$$Sup_{\mathcal{P}_p}(S) = \{N \in \mathcal{P}_p \mid \textit{there is a path in } N \textit{ that produces } S\}.$$

We say that a prefix S is *frequent* if $|Sup_{\mathcal{P}_p}(S)| \geq \sigma$ and *infrequent* otherwise. In our example partition $\mathcal{P}_c$, prefix $\langle c \rangle$, for example, has prefix support from both NFAs: $Sup_{\mathcal{P}_c}(\langle c \rangle) = \{N_1, N_2\}$. So for $\sigma = 2$, it is a frequent prefix. For $\sigma = 2$, examples for infrequent prefixes are $\langle ca_1 \rangle$, $\langle db_2 \rangle$, and $\langle ca_1 B \rangle$.

We now discuss how the local mining process works. The algorithm grows a *prefix* S item by item. It starts with an empty prefix $S = \epsilon$ and recursively appends single items ω to the prefix. When it finds a prefix to be infrequent, prefix expansion is stopped for this prefix, as any prefix $S\,\omega$ produced from an infrequent prefix S will be infrequent.

For each prefix, we maintain an inverted index *S.db*. The inverted index stores which NFA in $\mathcal{P}_p$ can produce the prefix. To do this, it stores tuples (N, q). One tuple for each NFA N that produces the prefix and the state q, in which the NFA is after producing prefix S. We call *S.db* an inverted index because we store a list of supporting NFAs for each prefix and not a list of supported prefixes for each NFA.

The prefixes that are generated this way can be arranged in a *prefix tree*, where each prefix S$\,\omega$ is a child of the prefix S. Figure 3.14 shows the prefix tree for our example. Frequent prefixes, that is, prefixes S with $|Sup_{\mathcal{P}_c}(S)| \geq 2$, are outlined. Frequent sequences, that is, prefixes S that have $|FinalSup_{\mathcal{P}_c}(S)| \geq 2$ are depicted with a double outline. At

Table 3.2: Inverted indexes for the prefixes in the prefix tree for partition c

Prefix	Inverted index	Prefix support count	Final support count
$\emptyset$	$(N_1, 0)$, $(N_2, 0)$	2	0
$\langle c \rangle$	$(N_1, 1)$, $(N_2, 1)$	2	0
$\langle ca_1 \rangle$	$(N_2, 2)$	1	0
$\langle cA \rangle$	$(N_2, 2)$	1	0
$\langle cb_{12} \rangle$	$(N_1, 2)$	1	0
$\langle cb_1 \rangle$	$(N_1, 2)$	1	0
$\langle cb_2 \rangle$	$(N_1, 4)$	1	0
$\langle cB \rangle$	$(N_1, 2)$, $(N_2, 4)$	2	0
$\langle cBe \rangle$	$(N_1, 3)$, $(N_2, 5)$	2	2

partition $\mathcal{P}_c$, there is one frequent sequence: $\langle cBe \rangle$, with support $|FinalSup_{\mathcal{P}_c}(\langle cBe \rangle)| = 2$. Prefix $\langle c \rangle$ is expanded to many prefixes, for example, to $\langle ca_1 \rangle$. However, most of these children prefixes are infrequent, so they are not expanded further. Only $\langle cB \rangle$ is frequent, so we expand it.

Table 3.2 depicts the inverted indexes for the prefixes in the prefix tree of our example. Along with the inverted index, we also depict the prefix support count $|Sup_{\mathcal{P}_p}(S)|$ and the final support count $|FinalSup_{\mathcal{P}_p}(S)|$ of each prefix S.

Algorithm 1 depicts the algorithm more formally. The empty prefix has support from the initial states 0 of all NFAs in $\mathcal{P}_p$. This is set in line 2. Expansion of a prefix S is done by the function *expand(S)*. To expand a prefix S, we iterate through the NFAs stored in the inverted index *S.db* (line 11). For each NFA in the list, we go to stored state q and follow the outgoing transitions of the state. Each outgoing transition produces a set of output items ω and leads to a state q'. For each tuple (ω, q') (line 12), we form a new child prefix $S\,\omega$ and update the data structures for the child prefix (lines 13 to 17).

First, we add tuple (N, q') to the inverted index of prefix $(\mathrm{S}\,\omega)$ (line 14), because N can produce prefix $(\mathrm{S}\,\omega)$ in state q'. For each prefix, we maintain two additional structures (lines 15 and 17): *support* and *finalSupport*, corresponding to prefix support and final support. We use the support to decide whether to expand a prefix (line 19) and final support to decide whether to output a prefix as frequent sequence (line 8).

As mentioned, we ensure to output only sequences with pivot item p at partition $\mathcal{P}_p$. As we send only pivot-relevant candidate sequences to the partition, most of the generated sequences have pivot p. In particular, the NFA does not produce any items $\omega > p$ at partition $\mathcal{P}_p$ as the pivot item of that sequence would be ω and not p. There are some cases though in which it is possible that we generate sequences S with $p(S) < p$. Consider for example an NFA with a path $\{b_1, B\}$ — $\{A\}$ — $\{b_1, B\}$ for pivot item b_1. In pattern-growth, one of the prefixes we will expand (if frequent) is $\langle BAB \rangle$, but $p(\langle BAB \rangle) = B$. We must not output this sequence at partition b_1, even if we find it to be frequent. Therefore, the filter step in line 8 is included.

Algorithm 1: Mine frequent sequences for partition $\mathcal{P}_p$

```
Data: List of NFAs P_p, pivot item p, minimum support σ
Result: Frequent sequences with pivot item p
S ← ϵ
S.db ← {(N_1, 0), (N_2, 0), ..., (N_|P_p|, 0)}
S.support ← {N | N ∈ P_p}
S.finalSupport ← {}
expand(S)

Function expand(S)
    if |S.finalSupport| ≥ σ and p(S) = p then
        output (S, |S.finalSupport|) as a frequent sequence
    S.children ← {}
    foreach (N,q) ∈ S.db do
        foreach (ω, q') ∈ outgoing(N,q) do
            S.children ← S.children ∪ ω
            (S ω).db ← (S ω).db ∪ (N, q')
            (S ω).support ← (S ω).support ∪ N                    // initially empty
            if q' is a final state in N then
                (S ω).finalSupport ← (S ω).finalSupport ∪ N       // initially empty
    foreach ω ∈ S.children do
        if |(S ω).support| ≥ σ then
            expand(S ω)

Function outgoing(N, q)
    // returns a list of (output item, target state) pairs for all outgoing items leaving state q of
       NFA N
```

3.8.2 NFA representation for local mining

Before the local mining process, we read the variable-length encoded NFAs, convert the path-wise representation to a state-wise representation, and keep the state-wise representation in memory as integer arrays during the mining process. This has two main advantages. First, the serialized format has a low memory footprint compared to an approach in which we read in the NFA and create an object structure. Second, we can efficiently implement the state pointers in the inverted index as the array index of the starting position of the state in the integer array.

Although serialization by path was the more concise representation for communication of the NFAs in our experiments, it is not a suitable format for the local mining process. Recall that we store tuples (N, q), pointing to a specific state q of NFA N. Later, in a subsequent expand step, we visit this state q and aim to quickly generate the output items of all outgoing transitions along with their target states q'. If the state q has more than one outgoing transition, serializing by path has the consequence that the outgoing transitions of the state are kept in multiple different locations of the array. Consider the serialization in Figure 3.11(a). The NFA can be seen in Figure 3.8. The two outgoing transitions of state 1 can be found at indices 1 and 10. Moreover, without reading the entire serialization, there is no straightforward way to determine the other index when knowing one of them.

To remedy this, one could either store multiple indices for each state or read in the NFA completely before the mining process and hold either an object-representation or additional data structures for the integer array in memory during the mining. However, as there might be many NFAs at one partition, this can be memory consuming. We found that it is more efficient to convert the NFA to a state-wise representation when deserializing the NFA.

The representation by state allows implementing state-pointers efficiently. Consider the representation by state depicted in Figure 3.9 again. The information about all outgoing transitions, their target states, and the finality of the state are stored in one common place in the array. We can easily retrieve all this information by storing one pointer to the start index for a state q. For state 0 of the example NFA N_2, we store a pointer to index 0, for state 1 to index 3, for state 2 to index 10, for state 3 to index 14, for state 4 to index 18, and for state 5 to index 21.

It might seem counterintuitive to serialize NFAs by path for communication and then convert them to a representation by state. However, one has to consider that writing, reading, and sending data between nodes often is a bottleneck in distributed algorithms. In our experiments, sending the path-wise representation and converting it to a state-wise representation proved to be faster than sending the state-wise representation in the first place.

3.8.3 Discussion

There is further work possible to improve the local mining process. As mentioned, for some paths through the NFA, sequences are produced that are not relevant for the partition. For some pattern expressions, this can result in considerable additional computation. For these pattern expressions, it is desirable to add methods to prevent the generation of these prefixes.

We use an adapted version of pattern-growth in this work, but other methods for local mining could be employed. In essence, our distributed setup reduces the problem of local mining to finding all σ-frequent common paths in a given set of NFAs, which is a relatively general problem.

In our implementation, we aggregate identical NFAs by count both before communication to the partitions and locally at the partition before the mining process starts. That means, if an NFA N_1 arrives twice at a partition, we put it once in the list $\mathcal{P}_p$ with a weight of two. We adapt the algorithm for local mining accordingly by taking the weight of the NFAs into account when calculating the size of sets S.support and S.finalSupport.

Chapter 4

Experimental evaluation

In our evaluation, we proceed as follows. After describing our experimental setup, we compare our algorithm, which we call *DDIN*[1], to baseline algorithms for FSM with declarative subsequence constraints and assess the effect of minimizing NFAs and of aggregating NFAs by count. We then examine performance against a state-of-the-art algorithm for FSM with traditional subsequence constraints and further investigate the scalability of the proposed algorithm. Our main findings can be summarized as follows.

- For pattern expressions that generate many candidate sequences, DDIN is up to 50 times faster than the distributed version of DESQ-COUNT.
- For pattern expressions that generate only few candidate sequences, DDIN is not slower than the distributed version of DESQ-COUNT.
- For FSM with hierarchies and traditional subsequence constraints, the proposed algorithm performs competitively to LASH, a state-of-the-art algorithm specialized for this problem. For some parameter settings, it outperforms LASH.
- DDIN is not efficient for some pattern expressions that produce an exponential number of candidate sequences.
- DDIN exhibits linear scalability and can mine large datasets, which cannot be mined efficiently using sequential algorithms.

4.1 Experimental setup

4.1.1 Implementation

We implemented DDIN based on the source code of the sequential DESQ-COUNT and DESQ-DFS implementation by Beedkar and Gemulla (2016a), which is available online[2]. Since the publication in 2016, the authors have developed the code further, both in terms of performance and usability. Our implementation is in Java, using Java Development Kit 1.8. High-level code is written in Scala, using Scala version 2.11.8. The source code is available online[3].

[1] Distributed Desq with Item-based partitioning and Nfa shuffle

[2] `http://dws.informatik.uni-mannheim.de/en/resources/software/desq/`

[3] `https://github.com/alexrenz/ddin`

We represent items using integer identifiers based on the order established by the generalized f-list, such that more frequent items are represented by smaller integers, which take less space in variable-length encoding. We store the mapping between frequency-based identifiers and original identifiers in a *dictionary*. This dictionary also holds the parent items for each item, as defined by the item hierarchy. We send this dictionary to every node of the cluster at the start of the algorithm.

In DDIN, for both communication between partitions and local mining, we aggregate the NFAs that encode the candidate sequences by count. This means that, if two input sequences in a group of input sequences produce the same NFA for one pivot, we transfer this NFA once with a count of 2. At the mine partitions, the two NFAs are aggregated again. For example, if one NFA arrives with counts 2 and 3 from two different map partitions, we aggregate it so we have this NFA once with a count of 5. To do this, we use Apache Spark's *combineByKey* function.

4.1.2 Cluster setup

We run our experiments on a local cluster. The cluster consists of nine Dell PowerEdge R720 computers. One of the machines is configured as a master node for Hadoop, the other eight are worker nodes. Each worker node is equipped with two Intel Xeon E5-2640 v2 8-core CPUs and 128 GB of main memory. For data storage, each worker node is equipped with four 2 TB NL-SAS 7200 RPM hard disks. The driver node is equipped with one of the CPUs instead of two, 64 GB of main memory and is not used as data storage for Hadoop. The nodes are connected with 10 GBit Ethernet.

The cluster nodes run CentOS Linux, release 7.3.1611 with Linux Kernel 3.10.0. We are running version 2.5.0 of Apache Hadoop and version 2.0.1 of Apache Spark.

For Hadoop, we run one node as a master node and the other eight machines as nodes for data storage and computation. For Spark, no master node is required. We run Spark on the eight worker nodes.

For our experiments, we store data in the distributed file system HDFS on the cluster nodes. We submit applications to the Hadoop resource manager YARN in cluster mode. The algorithm reads the input sequences from HDFS, runs all computations, and stores found frequent sequences to HDFS again. We set the HDFS replication factor to three, meaning, each part of the data is stored on three nodes. Whenever possible, YARN schedules tasks at the nodes that hold the input data for the task.

As described, each worker node has 16 CPU cores. Each of these CPU cores supports two threads. Therefore, YARN counts 32 virtual CPU cores per worker node. When running an application in Spark, one can specify the number of *executors* for a job and the resources for one executor. Spark creates the specified number of executors, and each executor creates a Java Virtual Machine with the specified resources and runs as many tasks in parallel as it was assigned CPU cores. Unless otherwise specified, we run our experiments using 8 executors, each equipped with 8 virtual CPU cores and 64 GB of memory, that is, 8 GB per task. YARN adds 10% memory overhead, so the YARN containers effectively run with 70.4 GB memory.

Unless otherwise noted, we partition the input data into 256 input splits, which creates 256 execution partitions in Apache Spark. We do not change the number of execution

Table 4.1: Characteristics of datasets and hierarchies

	NYT	AMZN-S	AMZN-L	CW50
Average sequence length	22.8	4.5	3.9	19.0
Maximum sequence length	21 000	25 630	44 557	20 993
Total sequences	49 593 066	6 643 666	21 176 522	567 032 222
Total items	1 129 536 263	29 667 966	82 677 131	10 774 007 413
Unique items	8 148 932	2 374 096	9 874 211	22 642 566
Hierarchy items	9 874 089	2 385 775	10 557 785	22 642 566
Leaf items	8 148 932	2 374 096	10 528 545	22 642 566
Root items	31	2 623	1 173 798	22 642 566
Average number of ancestors	2.8	3.6	4.6	1
Maximum number of ancestors	3	8	104	1

partitions throughout the job. So in both the map and the mine stage, we work with 256 Apache Spark partitions. Note that these Spark partitions are not to be confused with the partitions $\mathcal{P}_p$ we introduced in Section 3.3. In the map stage, one Spark partition processes many input sequences. In the mine stage, one Spark partition typically processes many partitions $\mathcal{P}_p$.

Throughout our experiments, we use the *Garbage First Garbage Collector* on all nodes (Java option -XX:+UseG1GC). We also use an increased stack size of 8 MB (Java option -Xss8m), as we use recursion to simulate FSTs and for some pattern expressions, regular stack size does not suffice to simulate some FSTs on some input sequences.

4.1.3 Measures

We report *run times* as measured by Apache Spark. Apache Spark reports both run time for the map stage and the mine stage. We either report the total run time as the sum of these two stages or both of them separately. The time required for the shuffle is included in these two stages.

We report the number of bytes transferred between the map stage and the mine stage as *shuffle size*. To measure this, we use Apache Spark's *shuffleWriteBytes* metric for the map stage.

All measurements we report are the mean of three independent runs that were run with no other applications running on the cluster.

4.1.4 Data

We use four real-world datasets for our experiments. Table 4.1 depicts statistics about these datasets and the hierarchies we use with the datasets. In the following, we give a brief description of the these.

New York Times corpus dataset. The *New York Times Annotated Corpus*, released in 2008 by the Linguistic Data Consortium[4], consists of more than 1.8 million articles published by the New York Times between 1987 and 2007.

[4] `https://catalog.ldc.upenn.edu/ldc2008t19`

We interpret sentences in the corpus as sequences for our sequence database. Each word is an item. This results in a sequence database with 49.6 million sequences. We refer to this dataset as *NYT*.

We use a hierarchy of annotations from the Stanford CoreNLP tools[5]. Named entities generalize to their type (PERSON, ORGANIZATION, LOCATION, MISC) and then to ENTITY. For example, *Clinton* ⇒ PERSON ⇒ ENTITY.

Other words generalize to their lemma and then their part-of-speech tag. For example, *winning* ⇒ *win* ⇒ VB. This hierarchy is taken from Beedkar and Gemulla (2016a).

An example sequence from the NYT corpus is:

$$T_{\text{NYT}} = \langle \text{ The increase was attributed to a recommendation by Goldman,Sachs\&Company . } \rangle.$$

Amazon reviews dataset. We use two datasets that contain reviews of Amazon users. The original *Amazon review dataset* (McAuley and Leskovec, 2013) consists of 24.7 million product reviews by 6.6 million Amazon users between 1995 and 2013. We refer to this dataset as *AMZN-S*. We also use a larger version of the dataset (McAuley et al., 2015), to which we refer as *AMZN-L*. It contains 82.7 million reviews of 21.2 million Amazon users between 1996 and 2014.

Both datasets have an inherent hierarchy: the Amazon product hierarchy. An item generalizes to a category of items. This category can generalize to a broader category, which might then generalize to a department. For example, a specific book can generalize to the category *Fiction books*, which in turn generalizes to the department *Books*. Between the publication of the two datasets, both the Amazon product catalog and product hierarchy became more fine-grained, which can be seen in Table 4.1: both the number of unique items and the number of items in the hierarchy grew.

For this dataset, we interpret the succession of reviews by one user as a sequence in our sequence database. An example sequence from the Amazon dataset is:

$$T_{\text{AMZN-S}} = \langle 0307264556 \text{ B000NY133G B0002Q1CFO B000FIS5U4 B00004R6JY B00004S5SK B00004ROBR B00004VY19 B000002OLA} \rangle.$$

Each of the items in this sequence is an Amazon product identifier. An Amazon client reviewed these products over a period of time.

Using the Amazon product catalog one can map each product identifier to its corresponding product. First, this person reviewed two books, *I Feel Bad about My Neck* by Nora Ephron and *The Secret* by Rhonda Byrne. The books stem from the categories *Humor* and *Psychology & Counseling*, respectively. The person later went on to review a knee strap and a polishing cloth. The last five reviews are all for products related to the movie *Far and Away* with Nicole Kidman and Tom Cruise. The person reviewed two VHS tapes, two DVDs, and an audio cassette containing the soundtrack of the movie.

ClueWeb. The fourth dataset is a 50% sample of the ClueWeb09-T09B subset of ClueWeb[6]. Our subset contains the text of 25 million English web pages. Beedkar et al. (2015) split the text into sentences and removed boilerplate parts. Each sentence in the

[5] `http://stanfordnlp.github.io/CoreNLP/`

[6] `http://lemurproject.org/clueweb09/`

text is then treated as one input sequence. Unfortunately, we were able to obtain only a 50% sample of this dataset from the authors. We use no hierarchy for this dataset. We refer to this dataset as *CW50*.

4.1.5 Pattern expressions

We use a set of pattern expressions for our experiments. We include both constraints that cannot be expressed using traditional constraints and pattern expressions that model traditional subsequence constraints. Table 4.3 depicts the pattern expressions we use along with some example frequent sequences from corresponding datasets. The pattern expressions are adapted from Beedkar and Gemulla (2016a). Some pattern expressions make use of quantification, such as ?, $\{3\}$, and $\{1, 4\}$. These quantifications work as they do in regular expressions.

Table 4.2 reports statistics about the matched sequences and produced frequent candidate sequences for some pattern expressions. As it depends on σ whether a candidate sequence is considered frequent or not, we report these statistics for specific values of σ. The table depicts the absolute number of matched sequences for each pattern expression, as well as the share of matched sequences compared to the total number of sequences. For example, $N_2(100)$ matches 1 883 581 sequences of the NYT dataset, which is 3.8% of the sequences in NYT. Here, we count a sequence as a matched sequence if it produces at least one frequent candidate sequence. It can happen that a sequence matches the pattern but produces no frequent candidate sequence.

The table further depicts the total number of produced frequent candidate sequences. It also reports both the mean and median number of frequent candidate sequences produced by one matched sequence. For example, $N_2(100)$ generates 16 011 402 frequent candidate sequences, which is 8.5 frequent candidate sequences per matched sequences on average. The median number of produced candidate sequences for $N_2(100)$ is 9.

New York Times corpus. The pattern expressions N_1–N_5 are aimed at the NYT dataset. Abbreviations used in the table are DET for determiner such as *the*, PREP for preposition such as *in* or *at*, ADV for adverb, and ADJ for adjective.

Pattern expressions N_1, N_2, and N_3 are selective pattern expressions: they produce a manageable amount of candidate sequences. They further match only few of the input sequences. $N_1(10)$ and $N_2(100)$ match 3.8% of the sequences in NYT, $N_3(10)$ only 0.9%. $N_1(10)$, $N_2(100)$, and $N_3(10)$ produce on average 1, 8.5, and 2.9 frequent candidate sequences per matched input sequence. The total number of frequent candidates sequences stays manageable for these pattern expressions: below 2 million for $N_1(10)$ and $N_3(10)$ and at around 16 million for $N_2(100)$.

N_1 extracts verb-based relations used between two entities, for example ⟨lives in⟩. Multiple nouns and a preposition can be part of the relation, but are optional. Typically, N_1 extracts exactly one candidate sequence from a sentence. N_2 works similarly to N_1, but captures the surrounding entities as well. It allows for generalization of both these entities, so it produces more candidate sequences per matched sequence than N_1, as can be seen from the mean number of frequent candidate sequences per matched sequence in Table 4.2.

Pattern expressions N_4 and N_5 are less selective: they produce many candidate sequences. They extract 3-grams, a common concept in language modeling: contiguous

Table 4.2: Statistics on select pattern expressions: number of input sequences matched, total number of frequent candidate sequences produced, share of matched sequences compared to the total number of sequences, and the mean and median number of frequent candidate sequences produced by one matched sequence.

		Absolute number of		% matched	Candidates / matched	
	Dataset	matched s.	freq. candidate s.	of total s.	mean	median
$N_1(10)$	NYT	1 888 731	1 966 265	3.8	1.0	1
$N_2(100)$	NYT	1 883 581	16 011 402	3.8	8.5	9
$N_3(10)$	NYT	434 017	1 273 918	0.9	2.9	3
$N_4(1K)$	NYT	43 871 765	5 051 558 542	88.5	115.1	99
$N_5(1K)$	NYT	48 657 981	6 334 670 196	98.1	130.2	119
$A_1(500)$	AMZN-L	1 149 166	5 049 862 729	5.4	4 394.4	30
$A_2(100)$	AMZN-L	1 081 873	41 690 987	5.1	38.5	1
$A_3(100)$	AMZN-L	125 060	3 216 153 183	0.6	25 716.9	989
$A_4(100)$	AMZN-L	54 110	204 949 085	0.3	3 787.6	25
L(10,1,5)	AMZN-S	3 029 105	51 972 718 770	45.6	17 157.8	103
L(100,1,5)	AMZN-S	3 028 765	30 340 605 519	45.6	10 017.5	60
L(100,1,4)	AMZN-S	3 028 765	5 676 618 616	45.6	1 874.2	54
L(100,3,5)	AMZN-S	3 028 765	139 828 522 247	45.6	46 166.8	60
L(100,0,5)	AMZN-S	3 028 594	6 803 880 901	45.6	2 246.5	55

sequences of three items. Consequently, they match almost every input sequence. Further, they can produce many candidate sequences from one input sequence.

N_4 extracts generalized 3-grams occurring directly before a noun. For $\sigma = 1\,000$, it produces an average of 115.1 frequent candidate sequences for 88.5% of the input sequences in NYT. N_5 extracts 3-grams from any position in the sequence, and generalizes one of the three words in the 3-gram at a time. It does not produce a frequent candidates sequence only if the input sequence consists of less than three items or if it contains infrequent items such that no candidate sequence is frequent. Consecutively, the number of frequent candidate sequences is much higher for $N_4(1k)$ and $N_5(1k)$: around 5 billion and 6 billion, respectively.

Amazon reviews. The hierarchy for the AMZN-L dataset is deeper than the one for NYT. That is, an item on average has more ancestors and therefore can generalize to more items. As can be seen in Table 4.1, in AMZN-L an item has 4.6 ancestors (including itself) on average, in NYT it is 2.8.

A_1, A_3, and A_4 are cases that match only few input sequences, but generate many candidate sequences from these few input sequences (see Table 4.2). One reason for this is the deeper hierarchy: a matched item produces more items if it has more ancestors. Another reason is that — due to the possible gaps between matched items — these pattern expressions can match many combinations of items. For example, in A_1, each matched *Electronics* item can generalize to its ancestors. Moreover, multiple combinations of *Electronics* items can be matched.

A_1 and A_4 are similar pattern expressions with one difference: Amazon customers review products of category *Music Instrument* less often than the ones of the category *Electronics*. $A_1(500)$ matches 5.4% of the AMZN-L sequences, $A_4(100)$ matches 0.4%.

Table 4.3: Pattern expressions and example frequent sequences. Adapted from Beedkar and Gemulla (2016a).

Notation	Pattern expression	Example sequences from NYT (freq.)
$N_1(\sigma)$	Relational phrases between entities ENTITY (VERB$^+$ NOUN$^+$? PREP?) ENTITY	lives in (4 322), is survived by (1 749)
$N_2(\sigma)$	Typed relational phrases (ENTITY$^\uparrow$ VERB$^+$ NOUN$^+$? PREP? ENTITY$^\uparrow$)	ORG is offering ENTITY (2 239), PER was born in LOC (11 581)
$N_3(\sigma)$	Copular relations for an entity (ENTITY$^\uparrow$ $be^\uparrow_=$) DET? (ADV? ADJ? NOUN)	PER be professor (1 582), LOC be great place (99)
$N_4(\sigma)$	Generalized 3-grams before a noun (.$^\uparrow$){3} NOUN	NOUN PREP DET (8 163 372), DET ADV ADJ (760 714)
$N_5(\sigma)$	3-grams, one item generalized ([.$^\uparrow$. .]|[. .$^\uparrow$.]|[. . .$^\uparrow$])	human rights NOUN (21 883), who VERB also (22 223)
		Example sequences from AMZN-L
$A_1(\sigma)$	Generalized sequences of (up to 5) electronic items, which are at most 2 items apart in the input sequences (*Electronics*$^\uparrow$)[.{0, 2}(*Electronics*$^\uparrow$)]{1, 4}	'Mice' 'Keyboards' 'Computers & Accessories' (875), 'MP3 Players' 'Headphones' (11 761)
$A_2(\sigma)$	Sequences of books (*Book*)[.{0, 2}(*Book*)]{1, 4}	'Storm of Swords' 'A Feast for Crows' (153)
$A_3(\sigma)$	Type of products reviewed after a digital camera *DigitalCamera*[.{0, 3}(.$^\uparrow$)]{1, 4}	'Lenses' 'Tripods' (158), 'Batteries' 'SD&SDHC Cards' (149)
$A_4(\sigma)$	Generalized sequences of musical instruments (*MusicInstr*$^\uparrow$)[.{0, 2}(*MusicInstr*$^\uparrow$)]{1, 4}	'Musical Instrument' 'Bags & Cases' (2 158)
		Example sequences from AMZN-S
$L(\sigma, \gamma, \lambda)$	FSM with hierarchy, maximum gap γ, and maximum length λ (as LASH) (.$^\uparrow$)$\Big[$.{0, γ}(.$^\uparrow$)$\Big]${1, λ^-} with $\lambda^- = \lambda - 1$.	$L(10, 1, 5)$: 'Pop Music' 'Pop Music' 'Pop Music' 'Pop Music' (288 021), 'Books' 'Movies & TV' 'Toys & Games' (4 052)
		Example sequences from NYT
$M(\sigma, \gamma, \lambda)$	FSM with maximum gap γ and maximum length λ, no hierarchy (as MG-FSM) (.)$\Big[$.{0, γ}(.)$\Big]${1, λ^-} with $\lambda^- = \lambda - 1$.	$M(100, 1, 5)$: in what (46 905), most of the (115 243), spoke on condition anonymity (9 995)

A_3 is also using generalization, but is different from A_1 and A_4. It mines any combination of up to four items reviewed after a digital camera, with up to three other items between the matched items. This creates an average of 25 716.9 frequent candidate sequences per matched input sequence for $\sigma = 100$, as can be seen in Table 4.2.

A_2 does not make use of generalization and typically produces only few candidate sequences. It mines books reviewed in sequence. Typically, it produces only one frequent candidate sequence per input sequence. For users who reviewed many books, it can generate many candidate sequences per input sequence.

Traditional constraints. We can use the pattern expression language to express traditional subsequence constraints. These pattern expressions are applicable to all of our datasets. For example, pattern expression $L(\sigma, \gamma, \lambda)$ presented in Table 4.3 models the setting of LASH (Beedkar and Gemulla, 2015): a *maximum length* constraint, a *maximum gap* constraint, and support for hierarchies. The brackets [...] in the pattern expression are used to depict a succession of items, not a set of matching items.

The maximum length constraint λ allows limiting the maximum number of items in a subsequence. For example, for $\lambda = 4$, only subsequences with four items or less are considered. The maximum gap constraint γ allows to limit the gap between two items in the subsequence. With $\gamma = \infty$, all subsequences are considered. For $\gamma = 0$, only contiguous subsequences are considered. In general, for $0 \leq \lambda < \infty$, subsequences are considered if, for two consecutive items of the subsequence, only a maximum of λ items lie between the originating items in the input sequence. For example, for an input sequence $\langle abcd \rangle$, sequence $\langle ad \rangle$ is considered as a subsequence for $\lambda \geq 2$ and $\langle ac \rangle$ for $\lambda \geq 1$.

Pattern expressions for such traditional constraints produce a vast number of candidate sequences, as can be seen from Table 4.2. The AMZN-S dataset is smaller than the other datasets, with only 6 643 666 input sequences in total. However, these unselective pattern expressions generate many frequent candidate sequences.

Pattern expression $M(\sigma, \gamma, \lambda)$ models the setting of MG-FSM: the same constraints as $L(\sigma, \gamma, \lambda)$, but without support for hierarchies. One can see that, in the pattern expression, we do not generalize any items: we use item expression (.) instead of (.$^\uparrow$).

4.2 Performance for declarative subsequence constraints

We now turn to evaluating our algorithm. First, we examine its performance for FSM with subsequence constraints that cannot be expressed using traditional constraints. As we know of no other distributed algorithms for this settings, we evaluate our approach against two baseline algorithms.

4.2.1 Baseline algorithms

We implemented two baseline algorithms: *DDCount* and *DDIS*. DDCount, the distributed version of DESQ-COUNT, is described in Section 3.1. For each input sequence, it produces all frequent candidate sequences and aggregates them by count. We implement this using Spark's *reduceByKey* function.

DDIS is a variant of DDIN that sends the input sequence to the partitions instead of candidate sequences, as discussed in Section 3.4.1. As DDIN, it is based on item-based

partitioning. For each input sequence T, DDIS determines the pivot items $P(T, \pi)$ and sends the entire input sequence to the partitions corresponding to the pivot items. At each partition $\mathcal{P}_p$, we run DESQ-DFS on the input sequences to produce all frequent sequences with pivot item p.

DDIS is a rather naive approach because there are at least two clear opportunities for improvement. First, it sends the full input sequences. However, often, only a part of the input sequence is relevant. Therefore, one could send only the relevant part of the input sequence. In some cases, it might be possible to just drop irrelevant parts of the input sequence. In other cases, this might be more involved, and one would need to modify the FST or rewrite the input sequence. In any case, our naive implementation sends full input sequences.

Second, as the algorithm sends full input sequences, many non-pivot sequences are produced at the partitions. We discard these input sequences when they are produced. However, DESQ-DFS can spend much time in parts of the prefix tree where no pivot item is produced or will be produced. A more efficient approach would prevent these sequences in the first place. For LASH, Beedkar and Gemulla (2015) propose a specialized pivot sequence miner, which improves this behavior in the more general case of traditional subsequence constraints. Essentially, their pattern-growth approach starts with the pivot items (instead of the empty prefix at the beginning of the sequence) and grows the pattern to the left and to the right from there. It might be possible to adapt this approach for DDIS. We hypothesize that this could improve mining performance significantly.

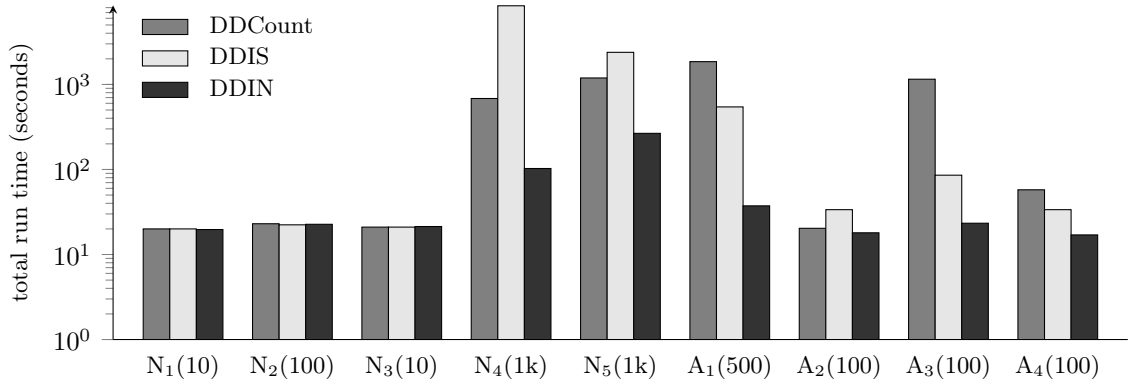
4.2.2 Results

Run time. Figure 4.1(a) depicts the times the three algorithms took for mining pattern expressions that cannot be expressed using traditional subsequence constraints: A_1 to A_4 and N_1 to N_5. It also gives the minimum support thresholds σ we used for these pattern expressions, which we adopt from Beedkar and Gemulla (2016a). The run times are shown using a logarithmic scale.

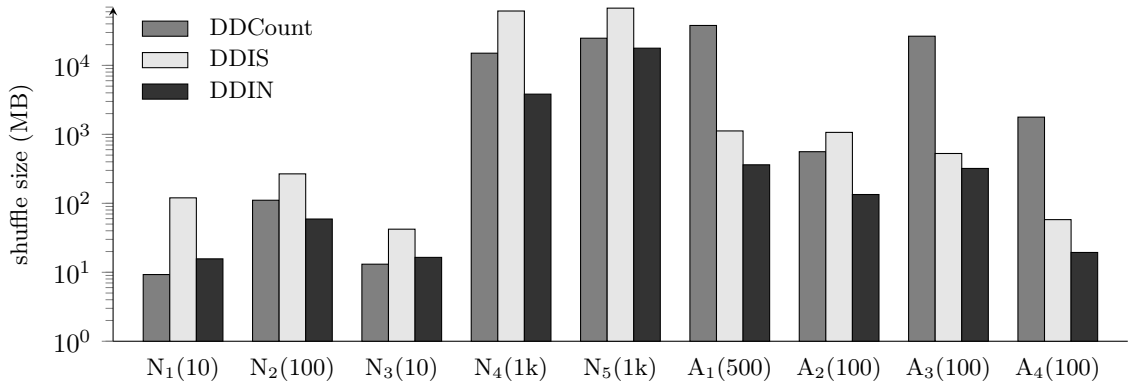
We first consider selective pattern expressions, such as N_1, N_2, N_3, and A_2. These do not generate many candidate sequences. For these pattern expressions, the run times of the three algorithms are similar. This is noteworthy, as DDIN and DDIS execute additional computation. They determine pivot items and run DESQ-DFS or an adapted version of DESQ-DFS for local mining, respectively. DESQ-COUNT, on the other hand, just produces candidate sequences and aggregates them by count. We conclude that this additional computation does not result in a significant run time overhead for selective pattern expressions.

We now consider unselective pattern expressions, such as N_4, N_5, A_1, A_3, and A_4. These pattern expressions generate many candidate sequences. DDIN consistently outperforms the baseline approaches for these pattern expressions. For A_1, A_3 and A_4, respectively, it is 47, 49, and 3.3 times faster than DDCount. For N_4, it is 6.7 times faster, for N_5 it is 4.5 times faster. As stated in Section 1, building an algorithm that is more efficient than DDCount for unselective pattern expressions is the goal of this thesis.

We further observe that DDIS outperforms DDCount for some of the unselective pattern expressions that generate many candidate sequences for one input sequence: DDIS is 3.5



(a) Total run time (seconds)



(b) Shuffle size (MB)

Figure 4.1: Performance of DDCount, DDIS, and DDIN for subsequence constraints that cannot be expressed using traditional constraints

times faster than DDCount for A_1, 14 times faster for A_3, and 1.7 times faster for A_4. For these pattern expressions, it is apparently more efficient to send input sequences, even though we send the full input sequences.

For N_4 and N_5, DDIS is significantly slower than DDCount, mostly because the mining phase takes long. We estimate that this can be improved significantly by developing a local mining algorithm that is specialized for mining only pivot sequences.

In contrast to DDIS, both DDIN and DDCount run most of the work in the map stage. We do not report map and mine times separately in Figure 4.1(a), because the separation could be misleading due to the logarithmic scale. In general, DDCount has short mining phases, as its mining consists only of aggregating the produced candidate sequences by count and filtering by the minimum support. DDIN has slightly longer mining phases, as it needs to find frequent paths through the NFAs. Both these algorithms shift the main work of simulating the FST to the map phase. In contrast, DDIS often uses most of the total run time in the mining stage.

Shuffle size. Using a logarithmic scale, Figure 4.1(b) depicts the amount of data shuffled between the map and the mine stage of our distributed algorithms.

We observe that encoding candidate sequences as NFAs is effective. In contrast to DDCount, DDIN can represent candidate sequences much more concisely when the pattern expression is unselective. For example, for A_1, DDCount shuffles 37.04 GB of data, whereas DDIN shuffles only 0.35 GB. For N_4, DDCount and DDIN shuffle 25.8 GB and 0.32 GB, respectively. For selective pattern expressions, the shuffle sizes of DDIN are smaller or on par with DDCount. For N_1 and N_2, the shuffle sizes of DDIN are slightly larger than the ones of DDCount because the candidate sequences are very short and DDIN additionally shuffles the pivot item. These results indicate that our techniques to encode simple NFAs concisely are effective.

Shuffling full input sequences leads to encouraging results for pattern expressions that make extensive use of generalization. For A_1, A_3, and A_4, sending the full input sequence to the partitions, as DDIS does, is considerably more concise than sending all generalized candidate sequences, as DDCount does. For N_4 and N_5, which allow less generalization, DDCount is more concise, as it can send only the relevant parts of the sequence and DDIS sends the full sequence. For these pattern expressions, further work on the approach of DDIS is necessary.

We can summarize that DDIN performs at least as well as the baseline approaches for all pattern expressions and clearly outperforms them for unselective ones. Further, encoding candidate sequences as NFAs is effective.

4.3 Effect of proposed enhancements

We now examine the effect of some of the enhancements we proposed in Chapter 3. In particular, we assess the effect of minimizing the NFAs and the effect of aggregating the NFAs by count.

4.3.1 Algorithm variants

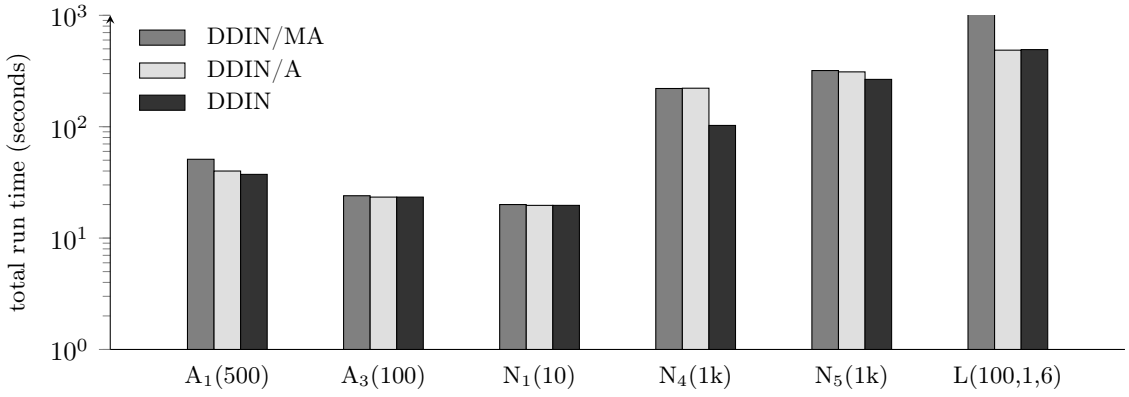
For this, we implemented two algorithm variants: *DDIN/MA*[7] and *DDIN/A*[8]. DDIN/MA does not minimize the NFAs before sending them to the partitions and does not aggregate NFAs by count. When processing an input sequence, for each partition $\mathcal{P}_p$, it builds a tree as described in Section 3.6.2 and then serializes this tree without merging common suffixes. DDIN/A minimizes the NFAs as DDIN, but does not aggregate the NFAs by count.

4.3.2 Results

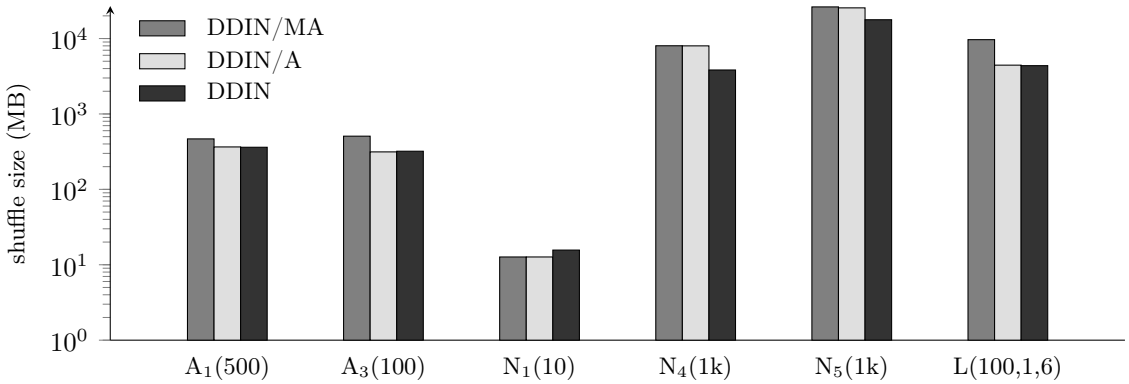
Figure 4.2 depicts both run time and shuffle size of these algorithm variants and DDIN for a select set of pattern expressions. Both graphs use a logarithmic scale for the y-axis.

For some pattern expressions, minimization and aggregation have only a negligible effect on both shuffle size and run time: they do not improve performance, but also do not make it much worse. Such pattern expressions are N_1 and A_3 or other ones A_2, A_4, N_2, and N_3, which we omitted in the graphs. These pattern expressions are the ones that already run fast in comparison to the other cases.

[7]DDIN without Minimization and Aggregation
[8]DDIN without Aggregation



(a) Total run time (seconds)



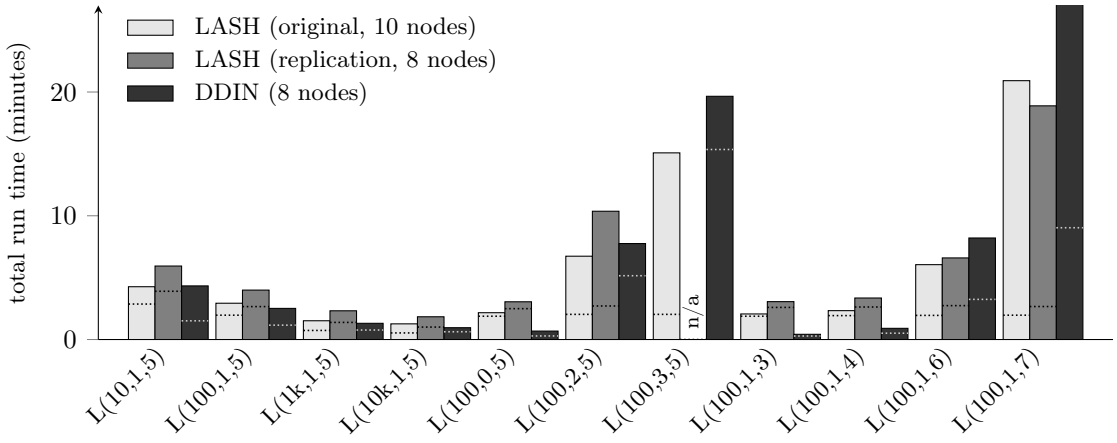
(b) Shuffle size (MB)

Figure 4.2: Performance of DDIN/MA, DDIN/A, and DDIN for select non-traditional and traditional subsequence constraints

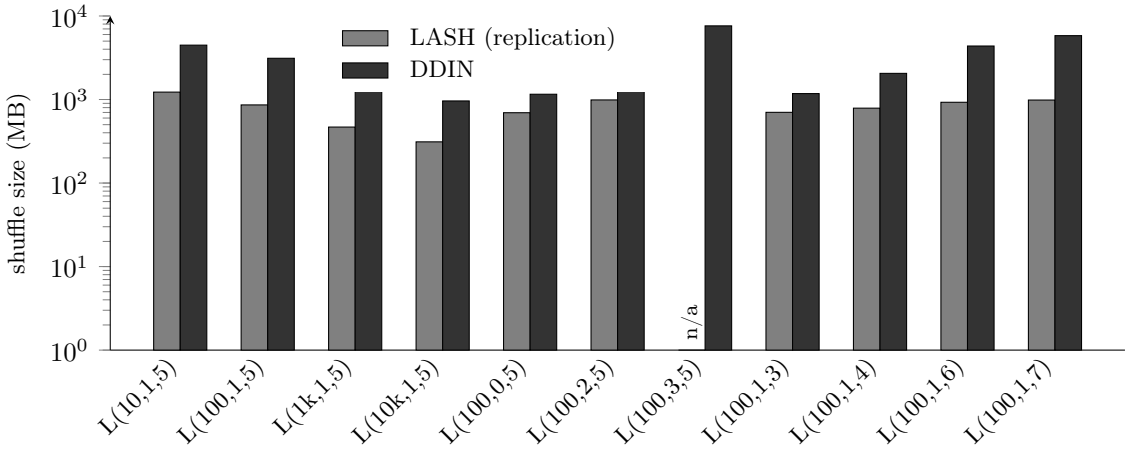
For more involved mining tasks, the enhancements can have a noticeable effect. Merging the suffixes of the NFAs has an effect for pattern expressions with many shared suffixes. For A_1, minimization reduces run time by 24%, from 51 to 39 seconds. For some cases of traditional FSM, which we discuss more in the upcoming section, the effect is larger. For $L(100, 1, 6)$, it leads to a speed-up of 50%, from 16 to 8 minutes.

Aggregation has a significant effect whenever identical NFAs are produced multiple times. In these cases, aggregation can lead to both lower shuffle and faster mining, as there are less NFAs to be processed and held in memory. For N_4, the effect is most noticeable: aggregation decreases shuffle size and run time by roughly half, from around 7.8 GB to 3.7 GB and from 224 to 103 seconds, respectively. For N_5, the effect on both shuffle size and run time is a reduction by roughly 30%.

In summary, one can say that the proposed enhancements do not slow down mining in general and can decrease run times for some pattern expressions.



(a) Total run time (minutes), the times for map and mine stages are separated by a dotted line.



(b) Shuffle size (MB)

Figure 4.3: Performance of LASH and DDIN for various parameter settings of traditional subsequence constraints using hierarchies

4.4 Performance for traditional constraints

So far, we compared DDIN to baseline approaches for FSM with declarative subsequence constraints. In this section, we examine how the algorithm performs for traditional constraints. In a first step, we evaluate its performance for some traditional constraints with the support for hierarchies. To do this, we compare it to LASH, a state-of-the-art distributed FSM algorithm with support for traditional subsequence constraints and hierarchies (Beedkar and Gemulla, 2015). In a second step, we examine the performance of DDIN without hierarchies.

4.4.1 Reference algorithm

LASH supports only a subset of the subsequence constraints DDIN offers. It allows to constrain the maximum gap between two items of a subsequence in the input sequence and the maximum length of the subsequences. LASH was specifically created and tuned

for this scenario, with specialized subsequence extraction and specific rewrites of the input sequences for the shuffle. We built DDIN for the more general case of declarative subsequence constraints. Therefore, it cannot take advantage of optimizations specific to this particular set of constraints.

We replicate an experiment series run by Beedkar and Gemulla (2015), which was run on the smaller Amazon dataset AMZN-S. We obtained this dataset from the authors. The dataset is smaller than the AMZN-L dataset we used in our other experiments. Dataset characteristics can be seen in Table 4.1.

LASH is developed for the MapReduce framework. We obtained the LASH code online[9]. Beedkar and Gemulla (2015) ran their experiments on a different cluster of 10 worker nodes and 1 master node, with 8 map and 8 reduce tasks per worker node. Each task was given 4 GB of main memory, resulting in a maximum of 64 GB used per worker node. We run LASH on our cluster also with 8 map and 8 reduce tasks per worker node, but only 8 worker nodes. We start each task with 8 GB of main memory. For DDIN, we run 8 executors that each have 8 CPU cores and 64 GB of memory.

In Hadoop MapReduce, reduce tasks can start before all map tasks are finished. During that time, more than 8 tasks can run in parallel on a worker node. Therefore, more than 8 cores per worker and more than 64 GB of memory can be used per worker. In our implementation of DDIN in Apache Spark, the mine stage begins once all map tasks have finished. We decided for the above settings because, in this setting, during the majority of execution time, LASH uses 8 tasks per worker, which occupy 64 GB of memory and use 8 cores per worker.

4.4.2 Results

With hierarchy. Figure 4.3(a) reports run times for various settings of the parameters minimum support, maximum gap, and maximum length using a linear scale. The parameter settings are adapted from the original experiment series. We include the run times originally reported by Beedkar and Gemulla (2015), of our runs of LASH, and the ones of DDIN.

In the figure, we depict time spent in the map stage and time spent in the mine stage separately. A dotted line in the bars indicates the end of the last map task. For LASH, we include the shuffle time in the time of the mine stage. Apache Spark does not report separate times for the shuffle. Shuffle time is included in the other two stages.

The results of our replication mostly confirm the run times reported in the original experiment. In the replication, we use two workers less and therefore 16 map and 16 reduce tasks less. The work of these tasks is distributed to the other 8 workers, so given the linear scalability demonstrated in Beedkar and Gemulla (2015), on an identical cluster, 25% longer run times are expected. We work with different hardware for the nodes of the cluster and different Hadoop and Java versions. Additionally, we took the LASH code and ran it, without extensive tweaking of settings. In three parameter settings, our runs deviate unexpectedly. For $L(100, 2, 5)$, the replication runs are slower than expected. For $L(100, 3, 5)$, our replication runs all failed with a null pointer exception. Moreover, for

[9] `https://github.com/uma-pi1/lash`

$L(100, 1, 7)$, the replication runs are faster than the original run times, despite the fact we use less map and reduce tasks.

Even though we test on cases that LASH is optimized for, our experiments show that DDIN offers run times that are competitive to LASH. It is faster than the LASH replication runs for eight parameter settings and slower for two.

Before we discuss this in more detail, we briefly consider shuffle size. Figure 4.3(b) depicts the shuffle sizes for both our LASH runs and DDIN using a logarithmic scale. Shuffle size is consistently higher for DDIN. As discussed previously, traditional subsequence constraints generate a large number of candidate sequences. In contrast to DDIN, LASH shuffles input sequences, so it can encode these candidate sequences efficiently. DDIN requires between 1.7 and 5.9 times more bytes to shuffle candidate sequences than LASH requires to shuffle input sequences.

DDIN is particularly fast when the number of candidate sequences is small, and candidate sequences are short. For settings $L(100, 0, 5)$, $L(100, 1, 3)$, and $L(100, 1, 4)$, DDIN outperforms LASH noticeably. One can see in Figure 4.3(b) that for these cases, encoding candidate sequences as NFAs leads to shuffle sizes close to the ones of LASH. These parameter settings allow either no gaps between items or only short sequences. Both reduces the number of candidate sequences. For these cases, respectively, DDIN sends only 1.7, 1.7, and 2.6 times more bytes.

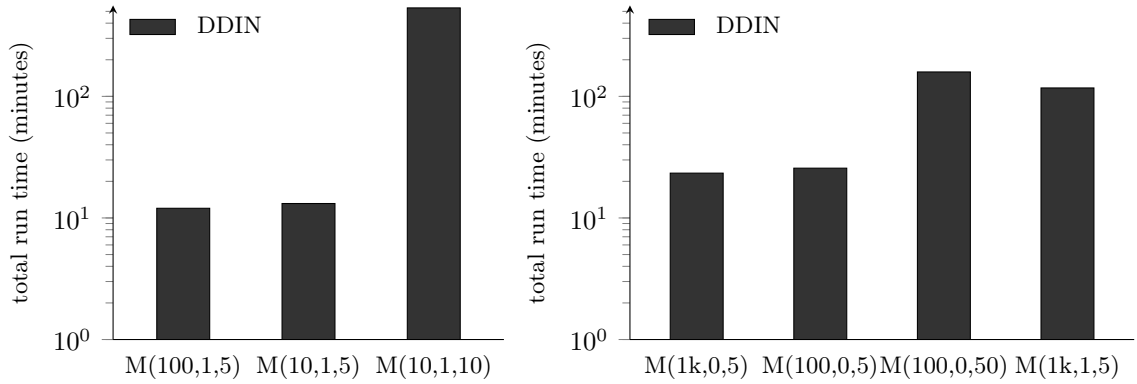
The opposite effect can be observed when the number of candidate sequences is large. $L(100, 3, 5)$ allows gaps of 3 items. This gap constraint allows many matches on the input sequence. Each match creates an accepting path in the FST, which DDIN adds to an NFA. Consequently, DDIN requires a long time to construct the NFAs. $L(100, 1, 7)$ allows long subsequences, which also causes many possible matches and many candidate sequences, which also results in long NFA construction times.

One would expect DDIS to perform well for these cases, as its general architecture is similar to LASH: it also sends input sequences to the partitions and then mines them locally. In our experiments, local mining took long with DDIS, presumably because we have no pivot-specific mining algorithm. LASH makes use of such a pivot-optimized local mining algorithm.

DDCount also performs poorly due to the large number of candidate sequences. For most cases, when running with 64 GB of memory per executor, the jobs fail because they run out of memory to store the candidate sequences. For cases with fewer candidate sequences, as $L(10\text{k}, 1, 5)$, $L(100, 0, 5)$, $L(100, 1, 3)$, and $L(100, 1, 4)$, DDCount finishes, but is slower than DDIN by factors 18, 40, 7, and 40, respectively.

Without hierarchy. We further run DDIN on two datasets without considering hierarchies. We run it on the NYT dataset and discard the NYT hierarchy by using the pattern expressions $M(\sigma, \gamma, \lambda)$, which does not make use of the hierarchy. We further run it on CW50.

The run times for NYT are depicted in Figure 4.4(b). The experiments are inspired by an experiment run by Beedkar and Gemulla (2015, Figure 4(e)), where LASH is compared to MG-FSM for mining without hierarchies. Unfortunately, we were not able to obtain the NYT dataset used in these experiments. Our NYT dataset was preprocessed differently such that it contains many more distinct items. Unfortunately, in the time available, we were unable to get LASH to run on our NYT dataset. For both 8 GB and 16 GB of memory



(a) NYT dataset

(b) CW50 dataset

Figure 4.4: Performance of DDIN for various parameter settings of traditional constraints without considering hierarchies

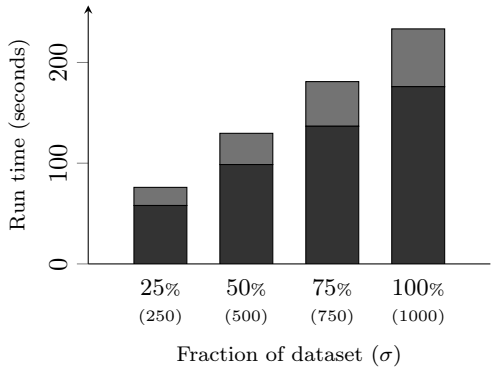
for all types of MapReduce tasks, LASH consistently ran out of memory when reading the f-list for the dataset.

The run times in the figure show that the run time of DDIN increases as we increase the maximum length constraint λ from 5 to 10: from 13 minutes for $M(10, 1, 5)$ to 8.9 hours for $M(10, 1, 10)$. As the number of candidate sequences increases exponentially in the length of the input sequence, this produces exponentially more candidate sequences. DDIN adds all these candidate sequences to the NFAs, which is inefficient in this case. The original experiment indicates that LASH is more robust in handling this increase of λ (Beedkar and Gemulla, 2015, Figure 4(e)).

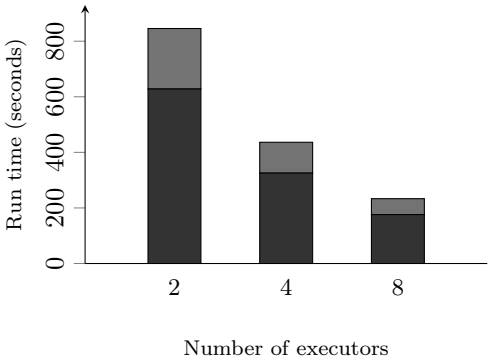
We further ran DDIN on the larger CW50 dataset. We divided CW50 into 2 048 input splits instead of 256, such that the number of input sequences per split is similar to the same number for NYT with 256 input splits. The run times reported in Figure 4.4(b) indicate that DDIN is able to mine larger datasets efficiently. The first three pattern expressions $M(1\text{k}, 0, 5)$, $M(1\text{k}, 0, 5)$, and $M(100, 0, 50)$ mine n-grams from the ClueWeb sentences and are inspired by an experiment of Beedkar et al. (2015, Figure 4(b)). Unfortunately, we were able to obtain only a 50% sample of the ClueWeb dataset used in that experiment. Additionally, our count of distinct items in the CW50 dataset does not match what we would expect from a 50% sample of the dataset used in the original MG-FSM experiment. Beedkar et al. report that their dataset contains 7 361 754 distinct items. The dataset we were able to obtain contains 22 642 566 distinct items. As we were unable to clear up the cause of this difference, we are hesitant to directly compare the results.

An increase in the maximum length for n-grams does lead to an increase in run time, but it is not as drastic as seen for maximum gap $\gamma = 1$ on the NYT dataset. Constraining the mining to consecutive subsequences lowers the number of candidate sequences. Again, the original experiment shows that MG-FSM is more robust at handling the increase than DDIN.

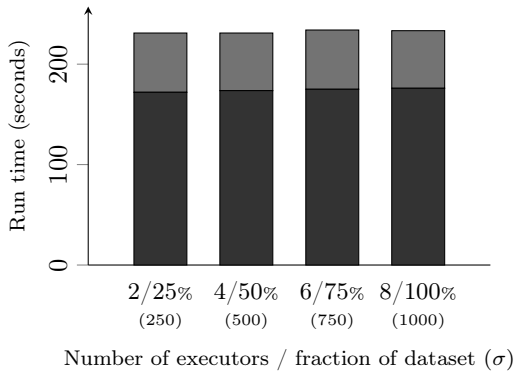
We can summarize that despite the fact that DDIN is developed for a more general case, it offers run times that are competitive to a state-of-the-art distributed algorithm for FSM with traditional subsequence constraints when hierarchies are used. The fewer candi-



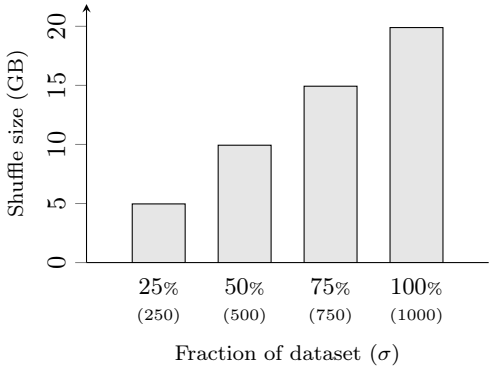
(a) Increasing dataset size



(b) Strong scalability ($\sigma = 1000$)



(c) Weak scalability



(d) Shuffle size

Figure 4.5: Scalability of DDIN for pattern expression N_5.

date sequences the traditional constraints allow, the better DDIN performs in comparison. DDIN can mine large datasets, but does not handle an exponential increase in candidate sequences well.

4.5 Scalability

In this section, we examine the scalability of the proposed algorithm DDIN. First, we investigate performance as we vary dataset size. We then look at strong and weak scalability. In a second step, we examine the speed-up we can achieve with our distributed algorithm compared to sequential mining.

4.5.1 Data subsets

We test the scalability of DDIN by varying dataset size, the number of executors, and both at the same time. To vary dataset size, we created random subsamples of the NYT dataset with 25%, 50%, and 75% of the original sequences.

When running a pattern expression on two different sizes of a dataset, one has two options for setting the minimum support threshold σ. One can either adapt σ proportionally to the dataset size. This leads to a shuffle size that is roughly proportional to the smaller dataset size, but results in roughly the same number of frequent sequences. Alternatively,

Table 4.4: Sequential and distributed run times for select pattern expressions

		Run time (minutes)		
	Dataset	DESQ-DFS (1 core)	DDIN (65 cores)	Speed-up
N_4(1k)	NYT	99.37	1.71	58×
N_5(1k)	NYT	67.46	4.43	15×
L(10,1,5)	AMZN-S	96.07	4.33	22×
L(10k,1,5)	AMZN-S	7.60	0.96	8×
L(100,3,5)	AMZN-S	157.59	19.66	8×
M(1k,0,5)	CW50	n/a (out of memory)	23.37	n/a
M(1k,1,5)	CW50	n/a (out of memory)	117.28	n/a

one can leave σ at the value for the full dataset. That leads to a shuffle size that is much smaller than proportional, but to a proportional number of frequent sequences. For our experiments, we opted for the first choice: we set σ proportional to the dataset size. Figure 4.5(d) shows that setting σ proportional to the size of the dataset leads to proportional shuffle sizes for N_5 with minimum support σ set to 250, 500, and 750 for the 25%, 50%, and 75% samples, respectively. As before, we set $\sigma = 1000$ for the full dataset.

4.5.2 Results

We tested scalability for a couple of pattern expressions and all presented linear scalability. Figure 4.5 depicts the results for N_5. In the figure, we report time taken by the map and mine stage separately. The time spent in the map stage is indicated at the top of the bars by a lighter shading.

We first examine the performance of DDIN as we vary dataset size. For this, we let it run on 25%, 50%, 75%, and 100% of the data with 8 executors. Results are shown in Figure 4.5(a). We observe that DDIN handles this well with both mine and map time increasing almost linearly as we increase dataset size. There is a constant factor at the start of each job, as there is some time required to set up the executors and especially to send the dictionary to each executor and read it into memory, which takes around 16 seconds for the NYT dataset.

We evaluate strong scalability by running DDIN on the full NYT dataset with 2, 4, and 8 executors. In Figure 4.5(b), one can see that DDIN demonstrates good strong scalability with both map and mine times decreasing linearly as we double the number of executors.

We evaluate weak scalability by increasing the number of executors as we increase the size of the dataset. Results are depicted in Figure 4.5(c). The results show that DDIN exhibits good weak scalability.

4.5.3 Speed-up compared to sequential mining

To examine how our distributed algorithm compares against sequential execution, we obtained the latest code revision from the authors of DESQ-DFS and ran it for a group of pattern expressions. We ran the sequential experiments on one machine of our cluster. For the experiments, we disconnected this machine from the cluster. We ran DESQ-DFS with 124 GB of maximum Java heap memory. We limited the number of threads garbage collection can use in parallel to 4.

As a comparison, we ran DDIN with our standard settings, that is 8 executors with 8 CPU cores and 64 GB of memory each. In total, with one core used by the driver process, we ran DDIN on 65 CPU cores in parallel.

In our experiments, DDIN was between 8 and 58 times faster than the sequential DESQ-DFS. Table 4.4 shows the run times for the sequential and distributed algorithm. For N_4, DDIN was 58 times faster than DESQ-DFS. We estimate that this is mainly due to the large number of NFAs that can be aggregated for N_4, as discussed previously. For the LASH pattern expressions, speed-up is lower, presumably because many candidate sequences have to be added to NFAs and the NFAs have to be shuffled.

Distributed execution makes it possible to mine larger datasets. For both CW50 cases depicted in Table 4.4, sequential execution consistently failed with *out of memory* exceptions in our experiments with 124 GB of maximum heap space. The experiments also failed when we increased the maximum heap space to 204 GB, making use of the swap space of the machines.

In summary, DDIN exhibits linear scalability. Distributed execution can make mining possible for datasets that cannot be mined efficiently sequentially and can speed up execution compared to sequential algorithms.

Chapter 5

Conclusions

5.1 Summary

We proposed the distributed algorithm DDIN for FSM with declarative subsequence constraints. The algorithm divides up the computation using item-based partitioning. For each input sequence, it produces candidate sequences and sends these to the corresponding partitions. The candidate sequences are encoded as one NFA for each partition. To reduce the size of the NFA, we use information about the structure of these candidate sequences and include shared parts of candidate sequences only once.

Our experiments show that DDIN is faster than baseline algorithms for distributed FSM with declarative subsequence constraints by a factor of up to 50 for pattern expressions that produce many candidate sequences. It is not slower than baseline algorithms for cases that produce few candidate sequences.

Our experiments further show that DDIN can be competitive to LASH, a state-of-the-art distributed algorithm for FSM with traditional subsequence constraints. For some parameter settings, DDIN outperforms LASH. We further found that DDIN scales linearly and makes it possible to mine large datasets that cannot be mined efficiently using sequential algorithms.

5.2 Limitations and future work

DDIN can be inefficient for pattern expressions that produce a vast number of candidate sequences. We hypothesize that for these cases, it can be more efficient to send input sequences to the partitions and run the FST again to generate the candidate sequences. This has the disadvantage that one has to run the FST twice, but has the advantage that it does not generate and shuffle all frequent candidate sequences. We implemented a naive version of this approach in our baseline algorithm DDIS. Rewrites similar to the work in MG-FSM and LASH could reduce the size of the shuffle by sending only the relevant parts of the input sequence. A local mining algorithm specialized for mining pivot sequences could improve mining time.

The pattern expression language we use in this work has limitations. Although we can express many of the traditional constraints and entirely new ones, there exist constraints that cannot be expressed using this language. For example, the language offers no support for aggregate constraints, which operate on multiple items of a subsequence (Pei et al.,

2002). Beedkar (2016) extends the pattern expression language with more fine-grained controls over which items are matched and which items are produced by an item expression. Further work could integrate more types of constraints into the language or make it more convenient to express some constraints, which can be specified with custom hierarchies at the moment.

In this thesis, we focus on the special case where sequences consist of single items. Other literature often studies sequences that consist of itemsets, which is relevant for many applications. Extending DDIN to support sequences of itemsets would make it practical for these applications.

Bibliography

Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, ICDE '95, pages 3–14. IEEE, 1995.

Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 207–216. ACM, 1993.

Hunor Albert-Lorincz and Jean-François Boulicaut. Mining frequent sequential patterns under regular expressions: a highly adaptative strategy for pushing constraints. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, SDM '03, pages 316–320. SIAM, 2003.

Kaustubh Beedkar. *Methods for Frequent Sequence Mining with Subsequence Constraints*. Doctoral thesis, University of Mannheim, Mannheim, 2016.

Kaustubh Beedkar and Rainer Gemulla. Lash: Large-scale sequence mining with hierarchies. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 491–503. ACM, 2015.

Kaustubh Beedkar and Rainer Gemulla. Desq: Frequent sequence mining with subsequence constraints. In *Proceedings of the 16th IEEE International Conference on Data Mining*, ICDM '16, pages 793–798. IEEE, 2016a.

Kaustubh Beedkar and Rainer Gemulla. Desq: Frequent sequence mining with subsequence constraints. Technical report, Data and Web Science Group, University of Mannheim, Germany, Mannheim, Germany, 2016b. URL `https://arxiv.org/abs/1609.08431v2`.

Kaustubh Beedkar, Klaus Berberich, Rainer Gemulla, and Iris Miliaraki. Closing the gap: Sequence mining at scale. *ACM Transactions on Database Systems*, 40(2):8:1–8:44, 2015.

Klaus Berberich and Srikanta Bedathur. Computing n-gram statistics in mapreduce. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 101–112. ACM, 2013.

Jean Berstel, Luc Boasson, Olivier Carton, and Isabelle Fagnot. Minimization of automata. Technical report, Université Paris-Est Marne-la-Vallée, Paris, France, 2010. URL `https://arxiv.org/abs/1010.5318`.

Janusz Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical theory of Automata*, Volume 12 of MRI Symposia Series, pages 529–561. Polytechnic Press, Polytechnic Institute of Brooklyn, New York, 1962.

Gregory Buehrer, Srinivasan Parthasarathy, Shirish Tatikonda, Tahsin Kurc, and Joel Saltz. Toward terabyte pattern mining: An architecture-conscious solution. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 2–12. ACM, 2007.

Shengnan Cong, Jiawei Han, and David Padua. Parallel mining of closed sequential patterns. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '05, pages 562–567. ACM, 2005.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, pages 137–150. USENIX, 2004.

Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Spirit: Sequential pattern mining with regular expression constraints. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 223–234. Morgan Kaufmann Publishers Inc., 1999.

Valerie Guralnik and George Karypis. Parallel tree-projection-based sequence mining algorithms. *Parallel Computing*, 30(4):443–472, 2004.

Valerie Guralnik, Nivea Garg, and George Karypis. Parallel tree projection algorithm for sequence mining. In *Proceedings of the 7th International Euro-Par Conference on Parallel Processing*, Euro-Par '96, pages 310–320. Springer, 2001.

Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 1–12. ACM, 2000.

John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Co., Reading, MA, 1979.

Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Chang. Pfp : Parallel fp-growth for query recommendation. In *Proceedings of the 2nd ACM Conference on Recommender Systems*, RecSys '08, pages 107–114. ACM, 2008.

Nizar Mabroukeh and Christie Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43(1):3:1–3:41, 2010.

Julian McAuley and Jure Leskovec. Hidden factors and hidden topics: Understanding rating dimensions with review text. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pages 165–172. ACM, 2013.

Julian McAuley, Rahul Pandey, and Jure Leskovec. Inferring networks of substitutable and complementary products. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 785–794. ACM, 2015.

Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

Iris Miliaraki, Klaus Berberich, Rainer Gemulla, and Spyros Zoupanos. Mind the gap: Large-scale frequent sequence mining. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 797–808. ACM, 2013.

Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

Carl Mooney and John Roddick. Sequential pattern mining – approaches and algorithms. *ACM Computing Surveys*, 45(2):19:1–19:39, 2013.

Edward Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th International Conference on Data Engineering*, ICDE '01, pages 215–224. IEEE, 2001.

Jian Pei, Jiawei Han, and Wei Wang. Mining sequential patterns with constraints in large databases. In *Proceedings of the 11th International Conference on Information and Knowledge Management*, CIKM '02, pages 18–25. ACM, 2002.

Hongjian Qiu, Rong Gu, Chunfeng Yuan, and Yihua Huang. Yafim: A parallel frequent itemset mining algorithm with spark. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium*, IPDPSW '14, pages 1664–1671. IEEE, 2014.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, pages 1–10. IEEE, 2010.

Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '96, pages 3–17. Springer, 1996.

Ramakrishnan Srikant, Quoc Vu, and Rakesh Agrawal. Mining association rules with item constraints. In *Proceedings of the 3rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '97, pages 67–73. AAAI, 1997.

Roberto Trasarti, Francesco Bonchi, and Bart Goethals. Sequence mining automata: A new technique for mining frequent sequences under regular expressions. In *Proceedings of the 8th IEEE International Conference on Data Mining*, ICDM '08, pages 1061–1066. IEEE, 2008.

Vinod Vavilapalli, Arun Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16. ACM, 2013.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '10, pages 1–7. USENIX, 2010.

Mohammed J Zaki. Parallel sequence mining on shared-memory machines. *Journal of Parallel and Distributed Computing*, 61(3):401–426, 2001.

Morteza Zihayat, Zane Hu, Aijun An, and Yonggang Hut. Distributed and parallel high utility sequential pattern mining. In *Proceedings of the 2016 IEEE International Conference on Big Data*, BIG DATA '16, pages 853–862. IEEE, 2016.

Appendix A

Finding pivot items of a path by merging sets

We aim to find the pivots $P(R)$ of a given path R by merging the sets of output items of the path using the following merge rule:

$$U \oplus Q = \{\omega \in U \mid \omega \geq \min(Q)\} \cup \{\omega \in Q \mid \omega \geq \min(U)\}$$

The update rule is associative and commutative. We omit a proof here. To obtain the set of pivot items of a path R, one merges the sets of output items of the path like this:

$$P(R) = O_1 \oplus O_2 \oplus ... \oplus O_n.$$

We do not give a proof for the correctness of this method here. To get an intuition how the method works, we apply the method to an example, path R_1:

$$R_1 : \{c\} — \{a_1, A\} — \{a_1, A\} — \{b_2, B\} — \{e\}.$$

Let us start with a part of the path, which we call R_1': $\{b_2, B\} — \{e\}$. This path generates sequences $\langle b_2 e\rangle$ and $\langle Be\rangle$. From our total ordering of the items, we have $B < e < b_2$. Therefore, the pivots of the candidate sequences are $p(\langle b_2 e\rangle) = b_2$ and $p(\langle Be\rangle) = e$. So the set of pivots for this path is $P(R_1') = \{b_2, e\}$.

Using the update rule, the minimal items of $U = \{b_2, B\}$ and $Q = \{e\}$, are $\min(U) = B$ and $\min(Q) = e$. Further, $\{\omega \in U \mid \omega \geq e\} = \{b_2\}$ and $\{\omega \in Q \mid \omega \geq B\} = \{e\}$. Therefore, $\{b_2, B\} \oplus \{e\} = \{b_2, e\}$.

For path R_1, we have

$$P(R_1) = \{c\} \oplus \{a_1, A\} \oplus \{a_1, A\} \oplus \{b_2, B\} \oplus \{e\} = \{c\}.$$

This is easy to see, as $\min(\{c\}) = c$ and for all other items ω in the path $\omega < c$.

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass diese Arbeit von mir persönlich verfasst wurde und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann.

Mannheim, den 13. April 2017