MapReduce Algorithms in Sequential Pattern Mining of Precise Data

Alexander Khovansky
The Open University of Israel
Department of Mathematics and Computer Science
Seminar in Database Management Systems, 22953
Spring 2017

Abstract

Sequential Pattern Mining (SPM) is an important technique to extract knowledge from data, aiming to find interesting subsequences from a set of sequences. The interestingness of a subsequence can be measured in terms of various criteria such as its frequency or profit. As SPM is computationally difficult, there is on-going research to design more and more efficient algorithms for this problem. [1] performs a comprehensive survey of the SPM problem, including its variations and extensions, with corresponding serial SPM algorithms. However, it does not provide a detailed survey of parallel SPM algorithms, which are becoming increasingly important as the dataset sizes continue to grow. As the Map/Reduce frameworks such as Hadoop and Apache Spark have turned into the main parallel execution platforms, most recent parallel algorithms are employing the Map/Reduce paradigm. This paper surveys Map/Reduce algorithms for both the basic SPM problem and its variations and extensions. After introducing the main concepts, it will provide an overview of the algorithms, elaborating on two of them, which gives an opportunity to explore the new techniques in more detail. This paper excludes SPM algorithms in uncertain databases and SPM extensions related to time series as these problems present a set of very different challenges which deserve their own survey.

Index Terms: sequential pattern mining, mapreduce

1. Introduction

1.1. Key Concepts

Itemsets and sequences

Pattern mining consists of discovering interesting patterns in databases. Initially the task was to find ***frequent itemsets***, i.e. groups of items frequently appearing together in a database, such as a group of products that are frequently bought together. The discovered frequent itemsets can be used to generate ***association rules***, e.g. rules such as 'if a customer bought items X and Y then s/he is likely to buy item Z', where the likelihood is above a user-defined minimal confidence level. However, these tasks don't take into account the sequential ordering of events, which is important in many domains. The ***sequential pattern mining*** problem addresses this issue and aims to find interesting subsequences in a set of sequences, where the interestingness of a subsequence is measured in terms of various criteria such as its frequency or profit.

Let I = $\{i_1, i_2, . . . i_m\}$ be a set of all items. An ***itemset*** is a subset of items in I and is normally shown in curly brackets, such as *{a, b}* or *{ab}*. An itemset of size k is called ***k-itemset***. A ***sequence*** is an ordered list of itemsets, is normally shown in angular brackets such as $s=<I_1,...,I_n>$, where each I_k is an itemset, also called an ***element*** of a sequence. The total number of items in all itemsets of a sequence is called the ***length of the sequence*** and the sequence of length k a ***k-sequence***. A sequence database SDB is a list of sequences, $SDB=<s_1,s_2,...,s_p>$, having sequence identifiers (***SIDs***) 1, 2,...,p.

Horizontal and vertical representation

The representation of a sequence database as a collection of (SID, sequence) pairs (or just as a list of sequences) is called ***horizontal*** representation.

Alternatively, a database can be represented ***vertically*** as followed. Each itemset of a sequence is given its own identifier, TID, unique in the given sequence. The *vertical* database representation is a map (item → *IDList*), where *IDList* is a list of (SID, TID) pairs (might be grouped by SID). *IDList*s can be concisely represented as bit vectors. Continuing the product set example, each itemset can represents a set of products that a certain customer has bought at a specific day. Then a sequence might represent the products that a customer had bought over a month. The SID will then represent a customer ID, and TID will be a day (or time) of the customer transaction. Note that *IDList* can be used to represent any sequential pattern s and the support of s is simply its *IDList*'s size.

It is assumed some **order between the items**, such as e.g. lexicographical order and the itemsets are represented in lexicographic order of their items.

Goals of sequential patterns mining

A sequence $s_a = <A_1, A_2, ..., A_n>$ is said to be a '***subsequence of***' or '***contained in***' another sequence $s_b = <B_1, B_2, ..., B_m>$ if there exist integers $1 \leq i_1 < i_2 < ... < i_n \leq m$ such that $A_1 \subseteq B_{i1}, A_2 \subseteq B_{i2}, ..., A_n \subseteq B_{in}$. The goal of sequential pattern mining is to discover interesting subsequences in a sequence database. S_b is a ***super-sequence*** of s_a if s_a is a subsequence of s_b.

Given a sequence database SDB, the ***absolute support*** of a sequence s, ***sup(s)***, is the number of sequences that contain s. The ***relative support*** is the number of sequences containing s divided by the total number of sequences in the database: ***relSup(s_a)*** = $sup(s_a)/|SDB|$. A sequence s is said to be a ***frequent sequence*** or a ***sequential pattern*** if relSup(s) ≥ some minimal support threshold. Originally, the measure of interest was a subsequence support. Later, other measures of interestingness, such as subsequence total profit, have been introduced.

Search space exploration

All sequential pattern mining algorithms explore the search space of sequential patterns by performing two basic operations called s-extensions and i-extensions. ***S-extension*** adds a new itemset with a single item to the end of the given sequence. ***I-extension*** adds a new item to the last itemset of the given sequence.

The basic mechanism for pruning the search space is based on the property called ***Apriori property, downward closure property, or anti-monotonicity property***. This property states that for any sequence s_a and s_b, if s_a is a subsequence of s_b, then s_b must have a support that is lower or equal to the support of s_a. This property holds in the basic SPM problem and in many, but not all of its extensions and variations.

Note that in the vertical database representation, an *IDList* of a new sequential pattern can be generated from the *IDLists* of its subsequences. (Specifically, any pattern s_a obtained by performing an i-extension or s-extension of a pattern s_b with an item *i* can be created by joining the *IDList* of s_b with the *IDList* of *i*).

Many papers refer to a real or imagined ***Lexicographic Tree (LST)***. The tree enumerates all the sequences and it is unique once the order between items is established. The LST is defined as followed. The nodes are sequences, the root is an empty sequence. Two sequences u and v are connected by edge $u \rightarrow v$ if v is either s-extension of i-extension of u, that is, the edges might be of two types. Instead of enumerating all the sequences, LST can enumerate only the actual sequences of the input database.

More details and examples on the SPM concepts can be found in [1] which also explains how a database of time-series could be represented as a sequence database.

1.2. SPM Problem: Variations and Extensions

The basic SPM problem has a number of variations and extensions. In line with [1], the term '***variation***' is used when modifications to the original problem do not require any additional information in the input database, and '***extension***' when the input sequences are annotated with some additional data.

This paper will not list all the existing variations and extensions, but only those relevant to the new Map/Reduce algorithms surveyed in this paper. [1] strives to provide the full list.

Variation: concise representation of results

SPM might produce a huge number of results, especially with low minimal support threshold, which not only requires additional resources such as time, memory, and disk space, but also make it much more difficult for the user to analyse the results. One type of SPM variation is to look for some concise representation of the results. There are three main concise representations, two of which are relevant for this paper.

The first is ***closed sequential patterns***, defined as a set of sequential patterns that are not included in other sequential patterns having the same support. In other words: a closed sequence pattern can't be extended without decreasing its support. It's a '***lossless***' representation of results, as the original results, i.e. all frequent sequences and their support, can be recovered from the closed ones. To do this, sort the closed sequences by their length and iterate over their subsequences; if a subsequence is not covered previously, its support equals to the support of its closed sequence pattern.

Another representation is ***maximal sequential patterns***, defined as set of sequential patterns that are not included in other sequential patterns, i.e. those that can't be extended without ceasing to be frequent. Clearly, any maximal pattern is closed, but the opposite is not true: a super-sequence of a closed pattern might have lower support, but still be frequent. The advantage of maximal patterns is that their number is usually several orders of magnitude less than that of closed patterns. However, they are not lossless: although the maximal patterns may be used to obtain all sequential patterns without scanning the database, their support can only be recovered by performing an additional database scan.

The last representation is ***generator sequential patterns***, defined as set of sequential patterns that have no subsequence having the same support. They are not relevant to this paper. [1] describes them in more detail and provides further references.

Variation: constraints

A constraint is an additional set of criteria that the user provides to indicate more precisely the types of patterns to be found. There is a number of different constraints. The ones relevant to this survey are:

- ***Length constraints***: minimum and/or maximum length of a sequential pattern.
- ***Duration constraints***: if each itemset of a sequence is annotated with a time, these constraints restrict the maximal time duration of a sequence pattern.
- ***Gap constraints***: in its simplest form, a gap is defined as the number of sequence elements between the chosen ones. For instance, in the subsequence <*{a}, {d}*> of a sequence <*{a}, {b}, {c}, {d}*>, the gap between *{a}* and *{d}* is 2. Gap constraints restrict the minimal and/or maximal number of elements between the chosen ones. Continuing the example, if the gap constraint is 1, then <*{a}, {c}*> might be a valid sequential pattern, while <*{a}, {d}*> is not. If the itemsets are annotated with a time, the gap constraints can restrict the minimal and/or maximal amount of time between two consecutive itemsets in a pattern.

All the constraints above are anti-monotone, i.e. they can be used to prune the search space by applying the downward closure property.

Variation: incremental sequential pattern mining

The basic SPM problem seeks to find sequential patterns using batch algorithms that run once and obtain the set of results. However, in many applications, databases are updated incrementally. For example, customer shopping transaction database is growing
daily due to the appending of newly purchased items for existing customers. ***Incremental sequential pattern mining*** algorithms strive to be efficient in response to a small change to the existing, already processed, database.

Progressive SPM problem generalizes the incremental SPM by also discarding changes that are too old. Formally, the progressive SPM problem is defined as followed: given an interesting time period called period of interest (POI) and a minimum support threshold, find the frequent subsequences whose frequencies ≥ (minimal support threshold) x (the number of sequences whose elements are in the current POI in a progressive sequence database).

Extension: high-utility sequential pattern mining

A utility sequential database is an extension of regular sequential database where

1. Each item is assigned a weight to indicate its relative importance.
2. Each occurrence of an item in an itemset is annotated with its quantity.

An example of an utility sequential database is a database of all purchases performed by all customers over some time. Items' weights are their prices. A specific itemset in a specific sequence might contain all items purchased by a certain customer in a certain day, and a sequence records all the transactions for a specific customer. **Q-item** is a pair (item, quantity), **q-itemset** is a set of q-items, a **q-sequence** is a sequence of q-itemsets, and a **q-sequence database** is a set of pairs (SID, q-sequence).

Defining a problem of ***high-utility sequential pattern mining*** (***HUSPM***) requires first to properly define the concept of q-containment and a utility of a subsequence.

- **Q-itemset containment**: given two q-itemsets $l_a = \{(i_{a1}, q_{a1})...(i_{an}, q_{an})\}$ and $l_b = \{(i_{b1}, q_{b1})...(i_{bm}, q_{bm})\}$, **q-itemset l_b contains l_a, $l_a \subseteq l_b$**, iff there exist integers $1 \le j_1 \le ... \le j_n \le m$ such that $i_{ak} = i_{bjk} \wedge q_{ak} = q_{bjk}$ for each k in [1, n]; in other words, if l_a is a subset of l_b's items and the items' quantities in l_a and l_b are the same.
- **Q-sequence containment**: given two q-sequences $s=<l_1..l_n>$ and $s'=<l'_1...l'_{n'}>$, **q-sequence s' contains s, $s \subseteq s'$** iff there exist integers $1 \le j_1 \le ... \le j_n \le n$ such that $l_k \subseteq l'_{jk}$ for each k in [1, n].
- Given a q-sequence $s = <s_1...s_n>$ and a sequence $t= <t_1..t_m>$ **s matches t, $t \sim s$** iff n = m and s_k contains the same items as t_k for each k in [1, n].
- A **utility of a q-sequence** s, **u(s)** is defined as $\sum_{i \in e,\, e \in s} weight(i) * quanity(i)$. The total utility of a q-sequence database is the sum of utilities of its q-sequences.
- Utility of a sequence in a q-sequence:
  In general, the same sequence can mean multiple different q-subsequences of the same q-sequence. Consider a sequence t=<{e}, {a}> and a q-sequence s= $<\{(b,2)(e,2)\},\{(a,7)(d,3)\},\{(a,4)(b,1)(e,2)\}>$ and suppose the weight of each item is 1. A sequence t might mean both $<\{(e,2)\},\{(a,7)\}>$ yielding a utility 9 and $<\{(e,2)\},\{(a,4)\}>$, yielding a utility 6. Normally, the ambiguity is resolved by taking the maximum, i.e. 9 in the above example. So a ***utility of a sequence t in a q-sequence s, v(t, s)*** is defined as $\max_{s' \subseteq s \wedge s' \sim t} u(s')$. A ***utility of a sequence t in a q-sequence database, v(t)***, is the sum of utilities of t in each q-sequence of the database.

- In a more general case, the operators such as 'sum' and 'multiplication' can be replaced by any user-defined functions.

The goal of HUSPM is to find sequential patterns t that have their utility v(t) above a user-defined minimum utility threshold. Continuing the example of customers and purchased items, the goal might be to find sequences of purchased items that generated the maximum profit. The difficulty of the HUSPM problem is that the Apriori property does not hold, i.e. extending a low-utility sequence can still generate a high-utility one, so most SPM algorithms will not work.

Extension: item hierarchies

In Sequential Pattern Mining With Hierarchies the items are arranged in ***hierarchy***, that is each item has zero or more children and at most one parent. Defining the problem requires the following concepts:

- For two items u and v, u ***directly generalizes to item*** v, $u \rightarrow v$, if v is a child of u in the hierarchy. Denote by $\rightarrow\ast$ the symmetric transitive closure of $\rightarrow$.
- Sequence $T = t_1...t_n$ ***directly generalizes to sequence*** $S = s1...s_{n'}$, $T \rightarrow S$, if $n = n'$ and there exists an index j such that $t_j \rightarrow s_j$ and for each $i \neq j$ $t_i = s_i$.
- Sequence $S = s_1...s_n$ is a ***generalized subsequence*** of $T = t_1...t_m$ if there exists integers $1 \leq i_1 \leq ... \leq i_n \leq m$ such that $t_{ij} \rightarrow\ast s_j$. The (absolute) ***support*** of S in the given sequence database is the number of database sequences for which S is generalized subsequence.

The Sequence Pattern Mining With Hierarchies problem is to find all generalized subsequences whose support ≥ some minimal support threshold.

Taking again products as an example, “Canon EOS 70D” may generalize to “digital camera”, which generalizes to “photography”, which in turn generalizes to “electronics”. The result pattern might be <{“digital camera”}, {“photography book”}, {“flash”}>, although the actual sequences might never contain the item “digital camera” and will only contain specific products such as “Canon EOS 70D”.

SPM on Multiple Databases

Normally, an SPM problem is explored in the context of a single database. However, some works start to explore the patterns mining in the context of multiple databases, trying to address queries such as “find all patterns with frequency > α in database A whose frequency in database B is < β”. This kind of problem requires collaboration between the involved databases. This survey only addresses parallel algorithms in the context of a single database. [26] provides a starting point for the patterns mining problem on multiple ones.

The rest of this paper will provide an overview of parallel algorithms on SPM, concentrating on the most recent ones that employ the MapReduce paradigm. Two of the algorithms will be shown with an additional level of detail which might help to provide a better clarity to the description of the new techniques.

1.3. Parallel Algorithms Landscape

Many papers differentiate between the data-parallel and task-parallel execution schemes.

- In *data-parallel scheme*, each machine runs part of the algorithm on its local block of data, then shares the results with other machines (the process can be repeated).
- In *task-parallel scheme*, completely-independent tasks are created and distributed to the machines which run each task to the end, producing a subset of the end results. Each task can require an access to the entire database or its subset, but it is not limited to the machine's own block of data.

Many algorithms that use the task-parallel scheme also use the data-parallel scheme as a preparation step.

Using the above distinction, all the recent parallel algorithms surveyed in this paper could be assigned to one of the following classes:

1. Algorithms that use task-parallel scheme at its main step and utilize the underlying serial algorithm's ability to partition the work into independent tasks. As the tasks are not equal in size, this approach requires steps to load balance the tasks between the machines.
2. Algorithms that use task-parallel scheme at its main step, but strive to split the work into balanced tasks using its own way of work partitioning.
3. Data-parallel scheme without iterations, when at the first step each machine runs a serial algorithm to the end on its own block of data to generate candidate subsequences, and in the second step the candidates are filtered out to produce the globally-frequent results.

The next sections will provide details on these approaches and survey the relevant algorithms.

2. MapReduce SPM algorithms

2.1. Early Parallel Algorithms on SPM

NPSPM and HPSPM

One of the first papers on parallel sequential pattern mining was [3] published in 1998. It explored parallel execution of the GSP algorithm ([2]) and assumed **shared-nothing** architecture. It assumes that the input sequence database is split into blocks and the blocks are distributed among the machines and are stored in their local disks. The GSP algorithm finds frequent sequences similarly to how the Apriori algorithm ([8]) finds frequent itemsets: in the first database scan it finds frequent 1-sequences, in the second database scan it generates 2-item candidates and counts them to generate frequent 2-sequences and so on. The algorithm will need n database scans where n is the length of the longest frequent sequence. The paper explores 3 strategies for distributed execution of GSP:

- NPSPM ("Non-Partitioned SPM"): the strategy is similar to parallel Apriori's "Count Distribution" strategy used for frequent itemset mining. It partitions the input sequences, but not the candidates. On each iteration k, every node scans its own input sequences, generates k-candidate sequences and sends them with their local counts to the coordinator node that sums the local counts to determine the frequent k-sequences.
- SPSPM ("Simply-Partitioned SPM"): this strategy is similar to parallel Apriori's "Data Distribution" approach. It partitions the candidates, but not the input sequences. On each iteration k, every node receives some k-candidate sequences in a round-robin fashion and counts their support by reading the entire input sequence database to generate frequent k-sequences.
- HPSPM ("Hash-Partitioned SPM"): the strategy is similar to parallel Apriori's "Candidate Distribution/Hybrid" approach. It strives to use the benefits of both previous approaches. On each iteration k, every node scans its own input sequences and generates k-length candidates with their local counts. But now it sends each candidate directly to a specific node, according to the candidate's hash value. So each node will compute global counts for its own share of candidates and will generate its own set of k-sequences.

The paper reports the HPSPM as the best with NPSPM coming second, but becoming much slower with lower minimal support values due to a large number of candidate sequences being sent to the coordinator node. The implementation was using 'manual' message passing between the nodes, but HPSPM strategy is easily achievable in today's MapReduce frameworks. **Note also that the algorithm requires synchronization of the nodes on each iteration which slows down the execution.**

pSPADE

Another approach for parallel SPM was the pSPADE algorithm ([4], 2000). Unlike most other algorithms, it assumed a **shared-memory architecture** for all the nodes. pSPADE is based on the SPADE SPM algorithm. SPADE ([32]) uses vertical database representation which generates *IDLists* of new sequences from *IDLists* of shorter ones, which doesn't require additional database scans.

SPADE algorithm outline

The algorithm first creates a vertical representation of the database, where each item is associated with its *IDList* (a list of (SID, TID) pairs). *IDLists* of 2-sequences are obtained by joining *IDLists* of two items. *IDLists* of (k+1)-sequences are obtained by joining *IDLists* of two k-sequences with common (k-1)-prefix. The support of a sequence is simply the number of distinct SIDs in its IDList.

Example (minimal support = 2):

Let the input database be:

1:<{cd}:10, {abc}:15, {abf}:20, {acdf}:25>, 2:<{abf}:15>, 3:<{abf}:10>, 4:<d:10, {bf}:20, a:25>

The vertical representation of this database is:

a: <(1, 15), (1, 20), (1, 25), (2, 15), (3, 10), (4, 25)>

b: <(1, 15), (1, 20), (2, 15), (3, 10), (4, 20)>

d: <(1, 10), (1, 25), (4, 10)>

f: <(1, 20), (1, 25), (2, 15), (3, 10), (4, 20)>

From this representation, 2-sequences can be obtained, e.g.:

<db> (S-extension): *<(1, d:10, b:15), (1, d:10, b:20), (4, d:10, b:20)>*

- Because of the common (k-1)-prefix, it's enough to represent it as *<(1, 15), (1, 20), (4, 20)>*, i.e. only storing the largest TID in each occurrence.

<df>: <(1, d:10, f:20), (1, d:10, f:25), (4, d:10, f:20)> or just *<(1, 20), (1, 25), (4, 20)>*

Next-size sequences:

<d{bf}> (I-extension)*: <(1, d:10, {bf}:20), (4, d:10, {bf}:20)>* or just *<(1, 20), (4, 20)>*

<dbf>: <(1, d:10, b:15, f:20), (1, d:10, b:15, f:25), (1, d:10, b:20, f:25)> or just *<(1, 20), (1, 25)>* – infrequent

Note: **all k-sequences with the same (k-1)-prefix can be processed completely independently.** The algorithm imposes no restriction on the search space exploration order – it could be either BFS, DFS, or any hybrid one.

pSPADE approaches to parallelization

The paper ([4]) explored several approaches to parallelize the execution and the best-performing approach was: partition *IDLists* of k-sequences by the common (k-1)-prefix, then process each partition (called '***equivalence class***') independently in a separate processor. The paper explored three different approaches to distribute the IDList 'classes' between the processors:

- SLB ("Static Load Balancing"): assume the weight of each 'class' is proportional to its number of elements (IDLists), sort them by decreasing weights and then distribute them between the processors in a round-robin fashion.
- ICDLB ("Inter-Class Dynamic Load Balancing"): store the unprocessed classes in a global queue, each processor will pick the next available class from the queue and will process it completely.
- RDLB ("Recursive Dynamic Load Balancing"): each processor picks the next tasks from a global queue, where the task is to only expand a specific class to the next level (i.e. to generate IDLists for (k+1)-sequences from IDLists of k-sequences). It then pushes the new classes to the global queue for further processing.

As expected, RDLB was the fastest with ICDLB coming second. Note that the fact that the memory is shared between the processors was only utilized by pSPADE in its use of the global queue, meaning might be also applicable to the shared-nothing architectures. Indeed, both SLB and ICDLB approaches are achievable in today's MapReduce frameworks. ICDLB is achievable if the number of 'equivalence classes' is much higher than the number of nodes, in which case the framework will dynamically distribute them between the nodes. In contrast, the completely dynamic load-balancing scheme used in RDLB is not easily achievable in today's MapReduce frameworks.

Tree-Projection SPM

Serial database projection algorithms

An important class of SPM algorithms is ***pattern-growth algorithms*** based on ***database projections***. The algorithms strive to reduce the number of candidates by creating them from the actual input sequences and reduce the cost of database scans by working with ***projected databases***. The idea of projected databases is simple: partition the sequences of the original database by some common property such as a prefix subsequence and process each partition independently. These partitions are called '*projected databases*'. The idea comes originally from FP-Growth ([5]) algorithm for frequent itemset mining, and one of the best known original algorithms for SPM utilizing this idea was PrefixSpan ([6]).

PrefixSpan algorithm outline:

The algorithm uses a *PrefixSpan(P, D|P)* subroutine, where P is the prefix, D is the original database, and D|P is the projected database created from D with prefix P. The **projected database** is created by dropping the prefix from each sequence of D.

The algorithm: just call *PrefixSpan((), D).*

Simplified outline of PrefixSpan(P, D|P):

1. Find and output L_1, a list of sequences of the form Pv, where v is an item that is either assembled into the last itemset of P (I-extension) or appended to P as a singleton itemset {v} (S-extension). It is normally done by simply counting the frequencies of v in D|P, but there are some subtleties (will see in the example below).
2. For each v in L_1, call *PrefixSpan(Pv, D|Pv)*

For example (minimal support = 2):

Let D be *<a{abc}{ac}d{cf}>, <{ad}c{bc}{ae}>, <{ef}{ab}{df}cb}>, <eg{af}cbc>.*

Initially, P=<> and L_1=*(<a>:4, <b>:4, <c>:4, <d>:3, <e>:3, <f>:3)* – just count the frequency of each item.

The projected databases:

D|<a>: <{abc}{ac}d{cf}>, <{_d}c{bc}{ae}>, <{_b}{df}cb>, <{_f}cbc>

- **Note the use of '_'**: {_v} means that the last itemset of the prefix, together with v, forms an itemset in this sequence

D|<b>: <{_c}{ac}d{cf}>, <{_c}{ae}>, <{df}cb>, <c>

…

So we call *PrefixSpan(<a>, D|<a>), PrefixSpan(<b>, D|<b>), ...*

Let's see how we proceed with *PrefixSpan((a), D|<a>)*:

1. In D|(a), output L_1=*(<aa>:2, <ab>:4, <{ab}>:2, <ac>:4, <ad>:2, <af>:2)* – essentially, we are just counting frequencies of items *a...f* in D|<a>, but also count entire itemsets such as *{ab}* if we see them.
2. Projected databases:

   D|<aa>: *<{_bc}{ac}d{cf}> , <{_e}>* – no chance to proceed, can stop here

   D|<ab>: *<{_c}{ac}d{cf}>, <{_c}{ae}>, <c>* – will have to recursively proceed further

   D|<{ab}>: *<{_c}{ac}d{cf}>, <{df}cb>* – will have to recursively proceed further

   ...

Data-parallel and task-parallel schemes in the Tree-Projection SPM:

The paper "Parallel tree-projection-based sequence mining algorithms" ([7], 2004), explores how this class of algorithms can be adopted to a **shared-nothing architecture**, exploring different approaches to task distribution and load balancing between the nodes. Like most shared-nothing algorithms, it assumes that the input sequence database is split into blocks and the blocks are distributed among the machines. In the following discussion a PrefixSpan is assumed to be the underlying serial SPM algorithm.

First, the paper differentiates between the data-parallel and task-parallel parallel execution schemes:

- The ***data-parallel*** execution scheme is similar to the NPSPM strategy described earlier. The solution space is explored in BFS order and the execution is performed in iterations. On each iteration k, every node scans its own input sequences, generates k-candidate sequences and sends them with their local counts to the coordinator node that sums the local counts to determine the frequent k-sequences. The only difference with NPSPM is the underlying sequential algorithm such as PrefixSpan. The solution requires synchronization between the nodes on every iteration.
- In the ***task-parallel*** scheme each projected database is processed independently on a separate node.

The task-parallel scheme is clearly more efficient as it does not require any synchronization and data movement between the nodes once they receive their databases. The paper shows that the following approach is the most efficient:

1. First, use the *data-parallel* scheme for initial k iterations to create sufficiently small projected database. For instance, in the PrefixSpan it will generate projected databases based on common k-item prefix. A typical value of k is from 2 to 4. Smaller k value causes the tasks to be too large, meaning a high risk that all machines will stay idle waiting for the node that received the heaviest task. Larger value of k means potentially good load distribution between the nodes, but might incur significant overhead as the number of the projected databases grow.
2. Then proceed with the *task-parallel* scheme: distribute the tasks (i.e. prefixes in the context of PrefixSpan) between the nodes for independent execution.

Static and dynamic load-balancing in Tree-Projection SPM

The paper also explores a number of algorithms to distribute the tasks between the nodes in a task-parallel scheme. Similarly to pSPADE, it distinguishes between the static distribution of tasks and dynamic load-balancing scheme.

- Static task distribution: the paper suggests to estimate the weight of a k-prefix by summing the frequencies of frequent (k+1)-sequences having the same prefix. Then employ some algorithm to statically distribute the tasks between the nodes based on their weights. The paper explores two algorithms, bin-packing and bipartite graph partitioning. The latter not only strives to partition the tasks equally, but also to minimize the data movement by sending tasks with similar items to the same node.
- Dynamic load-balancing: if a processor becomes idle it asks another random machine for additional work. The response could either be an actual task or refusal, in which case it will ask another machine.

The dynamic load-balancing scheme had performed the best, but the static one had also shown very good performed, especially for k>2 (i.e. with finer-grained projected databases).
The paper used direct message communication between the machines, however it is fully applicable for execution in a MapReduce framework. The only exception is the dynamic load-balancing scheme described above which can't be easily achieved in these frameworks.

2.2. MapReduce Paradigm and Frameworks

As the MapReduce frameworks such as Hadoop ([28]) and Apache Spark ([29]) had become the main parallel execution platforms, most algorithms on parallel SPM started to employ the MapReduce paradigm and use an existing MapReduce framework in their experiments. Originally Hadoop was the platform of choice, but the most recent papers use Apache Spark as it offers simpler API and is usually much more efficient than Hadoop mostly due to its reduced work with the disk.

Let's briefly recap how MapReduce frameworks work. The MapReduce framework assumes **shared-nothing** architecture. Each MapReduce operation consists of 3 steps: 'map', 'shuffle', and 'reduce'. The 'map' step processes the input to produce a list of (key, value) pairs; if the input data is split into shards located on multiple machines, each machine would normally execute the map operation on its own shard. The 'shuffle' operation is performed by the framework (no programmer code is required); it takes the output of the 'map' operation as input and produces (key, list of values) pairs as output. Finally, the 'reduce' operation uses the output of the 'shuffle' operation as the input and produces the final output, normally as (key, values list) pairs.

2.3. Distribution of Tasks and Load-Balancing

Algorithms based on independent tasks

As seen previously, there are at least two separate families of serial algorithms capable of splitting their work to completely independent tasks. The first family works with vertical database representation and includes the SPADE algorithm and its newer and faster variations such as SPAM ([9]) and its derivatives. All *IDLists* of k-length candidates with the same (k-1)-prefix form an '*equivalent class*' that can be processed independently. The second family is comprised of the *database-projection* algorithms that work with horizontal database representation. It includes PrefixSpan ([6]) and its derivatives and adaptations for different SPM variations and extensions. Each projected database can be processed independently. [1] provides a review of all serial SPM algorithms and describes many algorithms from both families. Note that not all SPM algorithms split their work into independent tasks, e.g. GSP does not do it.

Data-parallel and task-parallel execution

Let's recap the two parallel execution options (introduced in the 'Tree-Projection SPM) section. In the distribution execution environment the input database is split into blocks which are distributed among the machines ('***nodes***') and stored in their local disks. When a serial algorithm capable of splitting its work into independent task is adapted to the distributed environment, two options are available: data-parallel and task-parallel execution.

- In the data-parallel execution each node runs the algorithm on its local data partition, benefiting from the data locality. The drawback is that each node might have to frequently stop and synchronize intermediate results such as candidate sequences' local counts or local IDLists with other nodes. It means that the nodes might frequently stay idle waiting for other nodes to complete the iteration. It also means that large amount of data will have be frequently moved between the nodes.
- In the task-parallel execution scheme the tasks are distributed between the nodes and each node completes its own task independently, without any synchronization with other nodes and without any data movement between the nodes. Note that the tasks mean independent group of sequences to be processed independently, for instance, all sequences with the same k-length prefix.

The task-parallel scheme is promising as once the task are distributed, no communication and synchronization is required between the machines, but its performance heavily depends on the distribution of tasks between the nodes. In the worst case, all the nodes will stay idle most of the time waiting for a single node to complete its work.

The tasks distribution problem

Probably the easiest way to split the tasks (i.e. sequences) is by the sequence first item. However, multiple papers on SPM, including the very early papers such as [7] and [4], report that in SPM this approach produces too coarse-grained tasks which leads to major imbalance of load between the nodes.

The ideal solution would probably be to have relatively small tasks that are distributed to nodes in a completely dynamic fashion, while preferring nodes that have the maximum amount of data for the task on their local disk ([7]). Unfortunately, the existing MapReduce frameworks do not provide an easy and natural way to achieve this (for the same partition key). Each individual 'map' task is assumed to run on a single node and there is no easy way for this node to delegate this task further into additional nodes. On other hand, customized static partitioning of the tasks is possible in some MapReduce framework. For instance, Apache Spark allows the programmer to implement its own custom Partitioner that can be used to assign individual 'map' and 'reduce' tasks to 'partitions' that will be then automatically assigned to nodes by the framework.

Simple static task partitioning scheme that splits the sequences to finer-grained groups has been shown in multiple papers to produce reasonably good results. For instance, the already-reviewed 'Tree-Projection SPM' paper ([7]) has shown that splitting the sequences by a common frequent 3-length prefix produces good balancing, although not as good as the fully-dynamic load balancing. The data-parallel scheme could be used to produce the frequent prefixes for the partitioning. Other works explore different approaches surveyed in the rest of this section.

Static sample-based tasks distribution

In “Parallel Mining Of Closed Sequential Patterns”, ([12], 2005), the authors describe their ***Par_CSP*** algorithm that adapts BIDE algorithm ([11]) for mining closed frequent sequence patterns in a distributed environment. BIDE is based on PrefixSpan and traverses the search space in DFS order. It adopts a closure checking scheme, called BI-Directional Extension, which mines closed sequential patterns without maintaining the set of previously-discovered closed patterns.

Par_CSP uses a simple static task distribution scheme which groups the sequences into tasks by a common prefix. Once the tasks efforts ("weights") are estimated, the tasks are distributed between the nodes using greedy bin-packing algorithm, i.e. by sorting the tasks by decreasing effort and distributing them to nodes in a round-robin fashion, quite similarly to the static task distribution scheme described in [7]. However, the *Par_CSP* algorithm uses a sophisticated routine to estimate the expected effort for the task. It first creates a **small sample** of the input database by removing from it all items except the N most frequent ones (where N is a user parameter) and running BIDE on this sample. If N is 75% of the total frequent items, BIDE is reported to take only about 1% of the time if being run on the original database, i.e. the time required for the estimation can be relatively small and yet the routine is reported to produce very good task effort estimation. Moreover, if a task's expected effort is greater than a user parameter (such as e.g. 3% of total), it is further split into smaller tasks with a longer common prefix. Only after the routine produces sufficiently small tasks based on the estimation, the algorithm distributes the tasks between the nodes and starts the actual execution, i.e. runs BIDE independently on each group of sequences. The algorithms run on the MPI framework ([30]), but as mentioned earlier, it can be adapted to run on the MapReduce framework as it only uses a static task partitioning scheme.

Tasks distribution with multiple MapReduce iterations

Although completely-dynamic load-balancing and tasks distribution is difficult to achieve in today's MapReduce frameworks, it is possible to approximate it by using multiple MapReduce iterations. Both BIDE-MR ([13], 2012) and MR-BIDE ([14], 2017) implement BIDE in Hadoop distributed environment in a similar fashion. Initially, the search space is explored in BFS order. Iteration k of MapReduce produces frequent k-sequences and updates the result with the new closed patterns. The frequent k-sequences are then used as the common prefixes to group the sequences for the next iteration. Once a certain depth is reached (a user parameter), each node runs BIDE on the group of sequences till the end. The papers do not provide performance comparison data with the older Par_CSP ([12]) algorithm.

- MR-PrefixSpan ([15], 2012) implements a similar approach, but use PrefixSpan as the sequential partition-mining algorithm

The previous algorithms explore the search space (the Lexicographic Tree) in a BFS order until a certain depth is reached and then switch to exploring the entire branch depth-first on each node. SPAMC ([16], 2013) uses a different task distribution strategy. SPAMC is an adaptation of serial SPAM algorithm ([9]) for a basic SPM problem to run in a distributed environment. The SPAM algorithm works similarly to SPADE, but uses bit vector representation for the IDLists. On each iteration, SPAMC constructs a partial sub-tree with a pre-defined limited tree depth. In other words, if the task's common prefix is of length *p*, it will construct sequential patterns of lengths *p+1,... ,p+d*, where *d* is a user parameter. IDLists with the same (*p+d*)-length prefix will form a task for the next iteration. Unfortunately, the paper does not provide performance comparison with other task distribution approaches. In particular, the obvious alternative of two iterations, one to construct all patterns until length *d* and another to execute the task till the end, was not explored.

Dynamic load-balancing with Streaming MapReduce

The problem of the iterative MapReduce solution is the need to synchronize between the nodes after each iteration: in each iteration, all the machines wait for the slowest one, quite similar to the data-parallel scheme. SPAMC-UDLT ([17], 2017) strives to achieve better task distribution and load-balancing by using a ***streaming MapReduce framework*** and a ***distributed queue***.

Similarly to SPAMC, the SPAMC-UDLT algorithm is based on the SPAM algorithm and its goal is to solve the basic SPM problem. Again, the IDLists are grouped by the common prefix and can be processed completely independently. Each **task** receives a group of IDLists having a common prefix of length k and returns frequent (k+1)-sequences and new tasks associated with the common (k+1)-length prefix, i.e. going just one step down the Lexicographic Tree. Each individual mapper executes just one task (as defined above), working very similarly to mappers in the basic iterative MapReduce approach described earlier. However, the use of a streaming MapReduce framework and a distributed queue allows to avoid the synchronization problem of the iterative MapReduce solution. In general, the goal of a streaming framework is to work with live data streams, continuously processing the incoming changes. In case of SPAMC-UDLT, the machines receive the tasks from the distributed queue and place the new tasks back to the queue in completely **asynchronous** way, potentially achieving a **completely-dynamic load balancing**. The onus of discovering the arrival of the new tasks and efficiently dispatching them to the machines is delegated to the streaming framework. SPAMC-UDLT uses Spark Streaming ([31]) as a streaming MapReduce framework and reports to outperform the SPAMC algorithm due to much better load balance between the machines. Unfortunately, the SPAMC-UDLT work does not report performance comparison with other task distribution approaches.

2.4. Partitioning into Independent Tasks

The previous section surveyed approaches to efficiently **distribute** independent tasks between the machines. The problem of **partitioning** the computation into independent tasks was easy, as the underlying serial algorithms were providing a natural way to split the work. However, for some variations and extensions to the SPM problem, splitting the work into independent tasks might be challenging. This section provides a summary of two interesting algorithms, MG-FSM and LASH that overcome these challenges. Both reports results that outperform the previous algorithms to their respective problems. The algorithms will be described with higher level of detail than in the other sections which will hopefully contribute to a better clarity of the description.

MG-FSM: Distributed SPM with Gap Constraints

MG-FSM ([18], 2015) addresses the problem of mining sequential patterns with ***gap constraints*** in distributed environment. In addition, it can be easily extended to produce either maximal or closed patterns. The work generalizes the concept of *projected database* by introducing a notion of ***w-equivalency*** and offers a number of novel optimization techniques for efficient database partitioning that reduce computation and communication costs.

Problem statement

[18] describes MG-FSM assuming the input sequences of the form <$item_1$, ..., $item_n$>, i.e. assuming sequences' itemsets to be of length 1. However, the assumption was done for simplicity of description only and the algorithm can be easily extended to a general case.

Defining a problem requires a number of basic definitions. Given two sequences S and T and a maximal gap constraint γ≥0, S is a ***γ-subsequence*** of T, $S \subseteq_\gamma T$, if S is a subsequence of T and there is a gap of at most γ between consecutive items selected from T. For example, if S=*acd*, T=*abcd*, then $S \subseteq_1 T$ and $S \subseteq_2 T$, but $S \not\subseteq_0 T$ as the gap between *a* and *c* in T is 1 element, *b*. The ***γ-frequency*** of a sequence S in a given sequential database is the number of database sequences T for which $S \subseteq_\gamma T$. MG-FSM seeks a solution to the following problem:

> Given a minimal support threshold σ, a maximal gap γ≥0 and maximal pattern length λ≥2, find sequential patterns of length at most λ whose γ-frequency is at least σ and output the found sequential patterns with their γ-frequencies.

W-equivalency

Following the *PrefixSpan*, the majority of database-projection algorithms split the original database by a common prefix. No two projected databases can yield the same sequential pattern as each projected database is created from its own prefix. The sequences selected for a particular projected database are normally reduced by removing the common prefix, but no additional effort is usually exercised to further minimize the projected databases on each machine. This approach has two drawbacks:

1. The work is not split equally as mining a projected database created from a more frequent prefix usually requires more work than mining a projected database created from a less frequent prefix.
2. The projected databases might still be very large, leading to additional memory requirements from a single machine.

The previous algorithms tried to address these problems by either further splitting large projected databases into smaller ones or by creating a large number of projected databases from the start, which imposes additional computation and communication costs. MG-FSM takes a different approach. It seeks to create balanced projected databases from the start by using an approach that naturally reduces larger projected databases more aggressively than smaller ones. The approach is inspired by the PFP-Growth algorithm ([10]) used for frequent sequence mining, but MG-FSM significantly extends it, utilizing the gap and the maximal length constraints for more aggressive pruning.

Explaining its approach requires a number of additional definitions. Let ***Σ*** be all the frequent items sorted by decreasing frequency. Σ defines the **order between the items**: **w<w'** if it is located earlier in Σ, i.e. if w is more frequent than w'. In the following examples let's assume that *Σ = {a, b, c, d, e, f}*, i.e. $a < b < \ldots < f$. The least frequent (the 'largest') item of a sequence S is called a ***pivot item*** and is denoted as ***p(S)***, e.g. ***p***(*acdb*)=*d*. Given a database sequence T and a frequent item w, let $\boldsymbol{G_{w,\gamma,\lambda}(T)}$ = {S | $S \subseteq_\gamma T$, 2 ≤ |S| ≤ λ, and <u>p(S)=w</u>}. For instance, $G_{c,1,2}(acbfdeacfc)$ = {*ac, ab, cb, cc*}. Given a database D, let $G_{w,\gamma,\lambda}(D)=\bigcup_{T\in D}(T)$. Now it is possible to define the w-equivalency: given a frequent item w, databases P and D are ***w-eqivalent*** or ***(w, γ, λ)-equivalent*** if $G_{w,\gamma,\lambda}(P)=G_{w,\gamma,\lambda}(D)$. In other words, two sequences or sequence databases are w-eqivalent if they produce the same <u>relevant</u> set of γ-subsequences with the same γ-frequences. Given a database P and a frequent item w, the sequences of the set $G_{w,\gamma,\lambda}(P)$ are called ***pivot sequences***.

MG-FSM algorithm creates a projected database P_w for each frequent item w. **The only requirement from P_w is that it should be w-equivalent to the original database** (in other words, **contain the same set of pivot sequences as the original database**). Note that unlike the previous algorithms, MG-FSM permits two projected databases to yield the same sequential pattern. For instance, P_w might just be comprised of all the sequences containing w. **This flexibility allows MG-**

FSM to more aggressively reduce the projected databases, especially the ones corresponding to the more frequent items. In particular, it can remove from the sequences of P_w all items less frequent than w (we'll see how it is done without compromising the w-eqivalency later). The projected database sizes will become much more balanced, as partitions corresponding to highly frequent pivot items will contain many short aggressively-reduced sequences, while partitions corresponding to less frequent pivot items will contain fewer sequences.

The MG-FSM algorithm

1. Preprocessing: use a regular Map/Reduce phase to compute the list frequent items, Σ, and sort them by decreasing frequency.
2. Computation:

   Map:

   - Input: original database sequence T, list of frequent items Σ
   - Action: for each distinct $w \in T \cap \Sigma$

     Construct a sequence database $P_w(T)$ that is *(w, γ, λ)-equivalent* to {T}

     Output (w, S) for each $S \in P_w(T)$

   Reduce:

   - Input: (w, P_w) where P_w is the union of all $P_w(T)$ produced by the 'map' operation
   - Action:

     F ← use any sequential algorithm to mine γ-frequent patterns with their γ-frequencies from P_w

     Result ← pick up only the *pivot sequences* from F (i.e. only the patterns whose least frequent item is w)

     Output Result

The preprocessing phase computes frequent items, similarly to most other algorithms. The easiest implementation under the MapReduce paradigm is just to output *(w, 1)* pairs for each $w \in T$ on the 'map' phase and sum the counts in the 'reduce' one. More efficient implementations will compute item frequencies in each partition and then union this information across the partitions.

The 'map' operation of the computation phase effectively partitions the original database into a set of w-eqivalent databases P_w for each frequent item *w*, where each P_w could be mined independently. The algorithm does not dictate how to map the original sequences T into the output sequences $P_w(T)$, it only requires their w-eqivalency. The next subsection discusses the desirable properties and techniques to create (w, γ, λ)-equivalent sequences that are much smaller than the original ones.

The paper calls the process of mapping a sequence T into $P_w(T)$ a ***sequence rewriting***. Note that $P_w(T)$ can be comprised from more than one sequence.

The 'reduce' operation of the computation phase just runs an arbitrary sequential algorithm to mine the γ-frequent patterns, similarly to many other MapReduce algorithms. However, due to an open nature of w-equivalency, this time multiple partitions might produce the same patterns and might even output frequent patterns that are not frequent in the original database. Hence the output of the serial algorithm should be post-processed to pick only patterns relevant to the given partition P_w, i.e. the patterns whose least frequent item is w. Due to the definition of (w, γ, λ)-equivalency, the reported patterns and their frequencies are guaranteed to be the same as if run on the original database (the paper provides a formal prove of this).

Construction of w-equivalent sequences

Given an input sequence T and a frequent item $w \in T$, the 'map' operation of MG-FSM algorithm needs to produce (w, γ, λ)-equivalent sequences $P_w(T)$. While the easiest implementation can just return $P_w(T)=\{T\}$, the efficiency of the algorithm depends on its ability to create sequences that are significantly smaller than the original sequence T, especially for frequent items w. This subsection describes some useful properties of $P_w(T)$ that help to achieve this and also steps to create such $P_w(T)$.

- **<u>Minimality</u>**:
  - Definition: $P_w(T)$ is *(w, γ, λ)-minimal* if $\forall S \in P_w(T)\, G_{w,\gamma,\lambda}(S) \neq \emptyset$, in other words if $P_w(T)$ contains no irrelevant subsequences.
  - Example: Let T=*addcd*, w=*c*, γ=1 and λ≥2. Similar to all other examples, we are assuming *Σ = {a, b, c, d, e, f}* (sorted by decreasing frequency). There is no 1-subsequences of T with length 2 or above that contain no items less frequent than *c*, i.e. $G_{c,1,2}(T)=\emptyset$. Hence, T can be simply dropped.
  - Steps: sequences can be pruned if they don't contain the pivot item w or if w is surrounded by only *irrelevant items* (i.e. items less frequent than w) within the distance of γ+1. For instance, in T=*addcd*, item '*a*' is surrounded only by *d* within the distance of 2, so T can be pruned for pivot item *c* and $\gamma \leq 1$.
- **<u>Irreducibility</u>**:
  - Definition: a sequence S is *(w, γ, λ)-irreducible* if $\neg \exists R \mid |R|<|S| \wedge G_{w,\gamma,\lambda}(R)=G_{w,\gamma,\lambda}(S)$, in other words if there is no shorter way to

write it without dropping some relevant subsequence. If such sequence R exists, then S is said to *reduce* to R.

- Example: Let S=*acdeb*, R=*acb* and pivot *c*. (R is obtained by simply removing all irrelevant items from S). R is (*c*, 2, 2)-reduction of S, but it is not its (*c*, 1, 2)-reduction since $cb \in G_{c,1,2}(R)$, but $cb \notin G_{c,1,2}(S)$. That is, dropping irrelevant items does not always produce a correct result. In general, creating a fully irreducible sequence is computationally challenging, so MG-FSM uses a weaker form of irreducibility.
- Weaker form: let's define a right distance of index i in T, $r_{w,\gamma,\lambda}(i,T)$, as the minimal number of relevant items we need to 'step onto' when moving to the right from i to the nearest pivot item w. Assume the indexes start with 1. The left distance is defined similarly. For instance, for T=**ca**d**bab**d**a**d**c**ef**a**d, $r_{c,1,2}(1,T)=1$ (just step onto *c*), $r_{c,2,2}(2,T)=2$ (step onto *c*, *a*), $r_{c,1,2}(4,T)=3$ (step onto *c, a, b*), $r_{c,1,2}(6,T)=4$ (step onto *c, a, b, b*). Index i is said to be *(w, γ, λ)-reachable* if $min\{l_{w,\gamma,\lambda}(i,T), r_{w,\gamma,\lambda}(i,T)\} \leq \lambda$. We can remove the unreachable indexes as any γ-subsequence of T that contains such an index would either violate the maximum length constraint or will not contain w.
- Steps (example for pivot c, γ=1, λ=2)
  1. Replace all irrelevant items with blanks, e.g.: *cadbabdadcefad → ca␣bab␣a␣c␣␣a␣*
  2. Remove unreachable indexes, e.g. *ca␣bab␣a␣c␣␣a␣ → ca␣a␣c␣␣*
  3. Remove prefixes and suffixes consisting of '␣' only, e.g. *ca␣a␣c␣␣ → ca␣a␣c*
  4. Replace any consequent blanks chain longer than γ+1 by blanks chain of the length γ+1, e.g. *c␣␣␣c → c␣␣c.*

- **Inseparatibility**
  - Definition: T is *(w, γ, λ)-separable* if $\exists T_1, T_2 \mid G_{w,\gamma,\lambda}(T) = G_{w,\gamma,\lambda}(\{T_1, T_2\}) \wedge |T_1| + |T_2| \leq |T|$ i.e. if T could be split into smaller subsequences. Covering all cases is computationally expensive, but in a weaker form, parts that are separated by γ+1 blanks can be extracted if they don't contain a common γ-subsequence of length ≥2 (otherwise we'll change the frequency of this subsequence by splitting). The only exception are parts contained in other parts, in which case they can just be dropped.
  - Example: *acb␣␣bca* can be (c, 1, 2)-split into {*acb, bca*}, but *bcb␣␣bca* can't as both *bcb* and *bca* contain a common 1-subsequence *bc*.

- Steps (example for pivot *c*, γ=1, λ=3):
  Split sequences with parts separated by γ+1 blanks that don't contain a common γ-subsequence with length ≥2 or are contained within others (in which case ignore them), e.g. *bcb␣␣bca␣␣ac␣␣bca_c* → {*bcb␣␣bca, ac*}. Then drop irrelevant subsequences (see the 'minimality' subsection above).
- **Aggregation**: aggregate repeated sequences that comprise $P_w(T)$, e.g. {*abc, abc*} → {(*abc, 2*)}.

The paper presents a relatively efficient algorithm ($O(|T^2|)$ due to $O(|T|)$ number of backward and forward scans over T) that aggregates the techniques described above. The $O(|T^2|)$ performance can be further reduced to $O(|T|)$ with some additional data structures, which might be useful for longer sequences.

Summary

MG-FSM presents a sophisticated algorithm to create minimal and balanced partitions of the original sequence database which can be processed independently. The algorithm can use any serial SPM algorithm for independent mining of each partition. The flexibility of the w-eqivalency concept allows to adapt it to other SPM problem extensions and variations. The paper also shows how to extend the algorithm to mine maximal or closed patterns with gap constraints (not covered here). The next subsection will describe an algorithm that adapts MG-FSM to the problem of mining sequential patterns with hierarchies.

LASH: Distributed SPM with Gap Constraints and Hierarchies

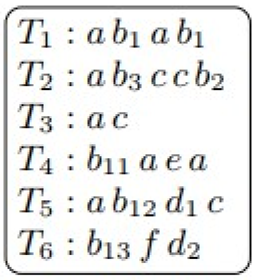
The LASH algorithm ([19], 2015) extends the MG-FSM algorithm to deal with hierarchies. Items are arranged in hierarchy and the reported sequential patterns might use higher hierarchy level than the input sequences. LASH extends the MG-FSM's notion of *w-equivalency* to take care of items' hierarchy. It also suggests a **new sequential partition-mining algorithm** that **only produces the patterns 'relevant' to the given partition**, which allows to eliminate the filtering step of the 'reduce' operation of the MG-FSM algorithm, thus reducing the total run time. (Using it is optional as any correct sequential algorithm that mines patterns with gap constraints and hierarchies will work, but it might require the filtering step as in the MG-FSM algorithm).

Problem statement

The paper explores a generalized form of frequent sequence mining, referred as ***generalized sequence mining (GSM)***, in which the item hierarchies are specifically taken into account. In particular, the patterns might use higher hierarchy level than the input sequences. For instance, consider a semantic hierarchy: a specific person (e.g. “Emmanuel Macron”) generalizes to something like “politician” that generalizes to PERSON, a specific city such as “Paris” generalizes to CITY etc. GSM is able to find patterns that do not actually occur in the input sequences, e.g. (PERSON lives in CITY).

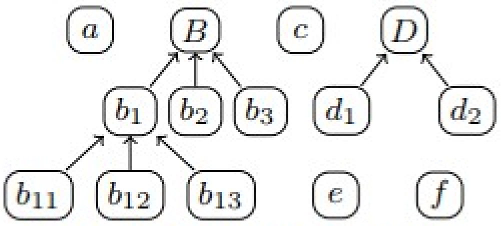
- Similarly to MG-FSM, LASH assumes the input sequences of the form $<item_1, ..., item_n>$, i.e. assuming sequences' itemsets to be of length 1 for simplicity.

In this subsection we'll use the following sequential database example:



$T_1 : a\, b_1\, a\, b_1$
$T_2 : a\, b_3\, c\, c\, b_2$
$T_3 : a\, c$
$T_4 : b_{11}\, a\, e\, a$
$T_5 : a\, b_{12}\, d_1\, c$
$T_6 : b_{13}\, f\, d_2$

(a) Example database


$$\begin{tikzcd} a & & B & & c & D & \\ & b_1 \arrow[ur] & b_2 \arrow[u] & b_3 \arrow[ul] & d_1 \arrow[ur] & & d_2 \arrow[ul] \\ b_{11} \arrow[ur] & b_{12} \arrow[u] & b_{13} \arrow[ul] & & e & & f \end{tikzcd}$$

(b) Vocabulary and hierarchy

GSM introduces a number of additional concepts. The item hierarchy tree can contain *leaf items, root items* and *intermediate items*. If item *u* is a child of *v*, then *u* is said to ***directly generalize to v***, ***u → v***. Given two sequences of the same length T=$t_1...t_n$ and S=$s_1...s_n$, ***T → S*** if

$\exists j \mid t_j \rightarrow s_j \wedge t_i = s_i \forall i \neq j$. The transitive closure of '→' is denoted as '$\rightarrow^*$'. The concept of *γ-subsequence* defined in the subsection on MG-FSM is naturally extended to hierarchies: S=$s_1...s_n$ is ***generalized γ-subsequence*** of T=$t_1...t_m$, $a \subseteq_\gamma^g b$, if

$\exists i_1..i_n, 1 \leq i_1 \leq ... \leq i_n \leq m \mid t_{i_j} \rightarrow^* s_j \wedge 0 \leq i_{j+1} - i_j - 1 \leq \gamma$. For instance, $ad_1 \subseteq_1^g ab_{12}d_1c$ and

$aD \subseteq_1^g ab_{12}d_1c$ (even though $ab_{12}d_1c$ does not contain item D). The ***generalized γ-frequency*** of a sequence S, ***$f_\gamma(S)$***, in a given sequential database (might also be referred simply as 'frequency' in this subsection) is the number of database sequences T for which $S \subseteq_\gamma^g T$. LASH seeks a solution to the following problem:

Given a minimal support threshold σ, a maximal gap γ≥0 and maximal pattern length λ≥2, find sequential patterns S, 2 ≤ |S| ≤ λ whose **generalized** γ-frequency is at least σ and output the found sequential patterns with their **generalized** γ-frequencies.

For instance, for σ=2, γ=1 and λ=3, the example database has the following patterns and frequencies: *(aa, 2), (ab$_1$, 2), (b$_1$a, 2), (aB, 3), (Ba, 2), (aBc, 2), (Bc, 2), (ac, 2), (b$_1$D, 2), (BD, 2).*

Notes:

- The problem statement excludes the frequent items (but they are found by the algorithm anyway)
- The problem as stated above can generate many redundant patterns: for instance, it reports both *(b_1D, 2)* and *(BD, 2)*, also the fact that b_1D is frequent implies its generalized supersequences such as *BD* are also frequent (but might have a larger frequency in general case).

The LASH algorithm

The structure of the LASH MapReduce algorithm is the same as MG-FSM. The differences are in the parts:

(1) Extend the order '<' between the items to take into account the items hierarchy
(2) Modified construction of $P_w(T)$ that is based on the generalized w-equivalency and is modified to take care of item hierarchy
(3) Optionally, use a dedicated serial algorithm, **PSM**, to mine patterns on each partition.

For completeness, the entire algorithm is copied below and the changed parts are highlighted:

1. Preprocessing: use a regular Map/Reduce phase to compute the list **generalized** frequent items, Σ, and **sort them by the extended '<' relation**.
2. Computation:

   Map:

   - Input: original database sequence T, list of frequent items Σ
   - Action: for each distinct $w \in T \cap \Sigma$

     Construct a sequence database $P_w(T)$ that is *(w, γ, λ)-equivalent* to {T}

     //The w-equivalency and the $P_w(T)$ construction are redefined

     Output (w, S) for each $S \in P_w(T)$

   Reduce:

   - Input: (w, P_w) where P_w is the union of all $P_w(T)$ produced by the 'map' operation
   - Action:

     F ← use any sequential algorithm to mine **generalized** γ-frequent patterns with their γ-frequencies from P_w //**A dedicated algorithm, *PSM*, can optionally be used**

     Result ← pick up only the *pivot sequences* from F //**No need in this step if the PSM algorithm is used in each partition, in which case Result ← F**

     Output Result

Generalized order '<' and w-equivalency

The the order '<' between items and the concept of w-eqivalency introduced by MG-FSM can be naturally extended to include hierarchies.

Given two items w_1 and w_2 let $\underline{\boldsymbol{w_1<w_2}}$ if

1. $f_\gamma(w_1) > f_\gamma(w_2)$, i.e. larger frequency, or
2. $f_\gamma(w_1) = f_\gamma(w_2)$ and w_1 occurs at a higher level of the item hierarchy than w_2
3. The remaining ties are broken arbitrary, e.g. alphabetically

In particular, it implies that if $w_2 \rightarrow w_1$ then $w_1<w_2$. $\boldsymbol{\Sigma}$ is again the list of all frequent items sorted by '<' in increasing order, e.g. in our example $\Sigma=\{a{:}5, B{:}5, b1{:}4, c{:}3, D{:}2\}$. Similarly, given a sequence T, ***p(T)*** is the 'largest' item in T.

$\boldsymbol{G_{w,\gamma,\lambda}(T)} = \{S \mid \boldsymbol{S \subseteq^g_\gamma T}, 2 \le |S| \le \lambda, \text{ and } \underline{p(S)=w}\}$, i.e. a set of **generalized** γ-subsequences of T whose *pivot item* is *w*. Same as in MG-FSM, for a database D, $G_{w,\gamma,\lambda}(D)=\bigcup_{T\in D}(T)$. Given a frequent item w, databases P and D are ***w-eqivalent*** or ***(w, γ, λ)-equivalent*** if $G_{w,\gamma,\lambda}(P)=G_{w,\gamma,\lambda}(D)$. I.e. the only difference with the MG-FSM algorithm is generalization of the '<' relation and of the set $G_{w,\gamma,\lambda}(T)$.

Construction of w-equivalent sequences

Given an input sequence T and a frequent item $w\in T$, the 'map' operation of the LASH algorithm needs to produce (w, γ, λ)-equivalent sequences $P_w(T)$. The goal is again to produce $P_w(T)$ that minimizes the size of its sequences especially for frequent pivot items.

The LASH paper adapts the same the techniques to construct $P_w(T)$ as in MG-FSM, but first prepares T with a technique that is called 'w-generalization' in the paper.

- **<u>W-generalization</u>**: while MG-FSM replaces the irrelevant items (the ones larger than the pivot item w) with blanks, LASH can't afford this because of the hierarchies: generalizations of irrelevant items might still be relevant. For instance, consider the sequence $T_2=ab_3ccb_2$ from our example database and let B be the pivot item. Replacing B-irrelevant items with blanks yields $T'_2=a\text{␣␣␣␣}$.Clearly, $aB \not\subseteq^g_1 T'_2$, but $aB \subseteq^g_1 T_2$. So the new rule for irrelevant items replacement is:

    Replace an irrelevant item with its largest ancestor that is still relevant or with a blank if such an ancestor does not exist.

  In the above example the proper replacement is $T'_2=aB\text{␣␣}B$. Although this technique does not reduce the size of the original sequence, it opens the door for other techniques.

Once the w-generalization is applied, the algorithm can reuse the rest of the techniques described in MG-FSM.

PSM: optimized mining on a single partition

Incentive: Mining of partition P_w is supposed to generate only patterns S with the *pivot item* w (p(S)=w). However, partitioning by *w-eqivalency* allows P_w to contain other patterns as well: w-equivalency only requires P_w to contain the right set of the *pivot sequences* and says nothing about other patterns. It means that any partition-mining sequential algorithm that is not aware of the partition's pivot item will generate these irrelevant patterns, so MG-FSM has an additional step to filter them out. This is wasteful as both generating the irrelevant patterns and filtering them afterwards takes additional time. LASH proposes a sequential partition-mining algorithm that is **aware of the partition's pivot item and does not generate irrelevant patterns**. The paper calls it '***Pivot Sequence Miner (PSM)***'. This subsection will describe the algorithm. The paper produces both experimental results and theoretical analysis that shows that PSM only explores a fraction of the solution space that would be explored by a pivot item-unaware algorithm. It also tried the algorithm on regular databases without hierarchies and showed that it can also significantly improve the overall run time.

- Note that the 'w-generalization' technique used in construction of P_w guarantees that for each $T \in P_w$ p(T)=w, i.e. T is a pivot sequence. However, the set of generalized γ-subsequences of T will normally contain non-pivot sequences. For instance, consider the item hierarchy of our example database, T=cab_1D, and pivot item *D*. Clearly, p(T)=*D*, but generalized 1-subsequences of T include *ca*, ab_1, and *aB*, non of which is a pivot sequence relative to *D*. PSM avoids generation of these subsequences.

PSM algorithm: Similar to most other algorithms, PSM explores the solution space by growing patterns. However, to avoid generation of irrelevant patterns **it starts from the pivot item and then grows it both to the right and to the left**. Since PSM starts with the pivot item w and P_w only contains w and items 'smaller' than w, every intermediate sequence will be a pivot sequence. Except the left and right expansion direction, PSM is similar to the PrefixSpan. The core of the algorithm is the '***Expand****(S, D_S, direction)*' subroutine that expands the given pattern in the given direction. It takes the following input parameters:

- S: the sequence to expand
- D_S: the subset of P_w that contains S
- Direction (either 'left' or 'right'): the direction of expansion

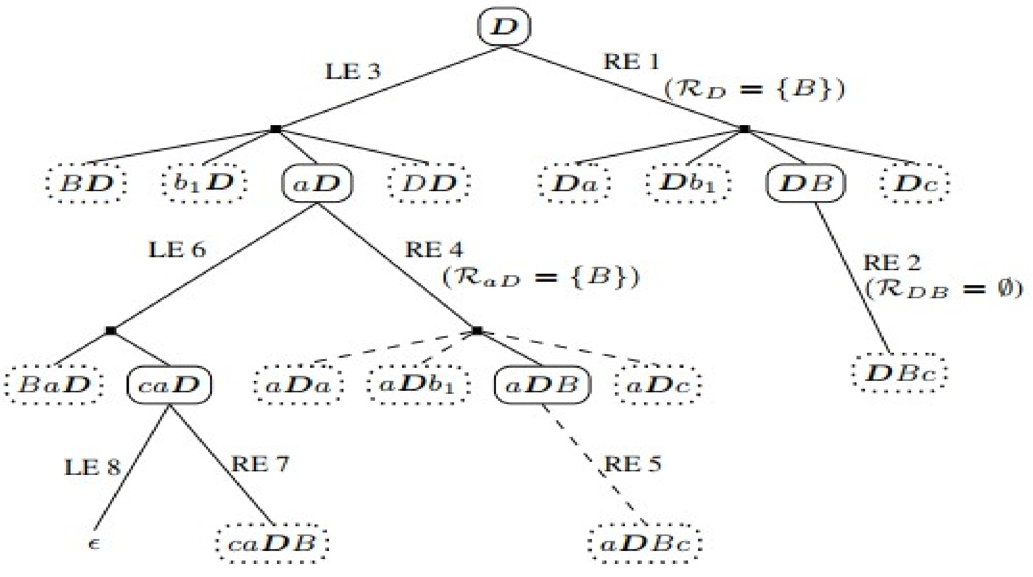
Given the '*Expand*' subroutine, the PSM algorithm is simple:

```
PSM(w, $P_w$) //σ, γ, λ omitted for brevity
    Expand(w, $P_w$, 'right')
    Expand(w, $P_w$, 'left')
```

An outline of ***Expand(S, D$_S$, dir)* subroutine**:

```
    if (|S| = λ) then return
    if (dir = 'right'):
        for each item u, u≤w, for which Su is generalized γ-frequent:
            Output Su with its frequency
            Expand(Su, $D_{Su}$, 'right')
    else:
        for each item u, u≤w, for which uS is generalized γ-frequent:
            Output uS with its frequency
            Expand(uS, $D_{uS}$, 'right')
            Expand(uS, $D_{uS}$, 'left')
```

Example: The figure below shows an example of the PSM run on partition P_D = {*aDDa, cab$_1$D, caDB, BaaDb$_1$c*} for σ=2, γ=1, λ=4, and an item hierarchy as in the rest of the examples. Solid nodes represent frequent sequences.



The algorithm first starts with the pivot item *D*. It then performs a right expansion and finds that only *DB* is frequent. Expansion of *DB* produces no frequent patterns so it backtracks. It then performs left expansion of *D* which yields only *aD* as a frequent pattern. *aD* is first right-expanded which yields *aDB* and then backtracks after not finding any frequent children, and then left expanded that only yields *caD* as a frequent pattern. No non-pivot sequences are explored.

Summary

LASH extends the MG-FSM algorithm to mine patterns in a sequential database with hierarchies, creating minimal and balanced partitions of the original sequence database which can be processed independently, based on the concept of *w-eqivalency*. In addition, it suggest to modify the sequential partition-mining algorithm to only explore subsequences that are relevant to the given partition, thus exploring only a fraction of the solution space. It achieves so by starting with a pivot item and expanding it to the right and to the left, thus guaranteeing that no irrelevant subsequences are explored. This idea is not directly related to the item hierarchies and could be used in any sequential database. Although the sequential algorithm is essentially based on PrefixSpan, other, more modern algorithms could potentially be adapted to efficiently mine only the pivot sequences. For this, they should implement the same approach to the search space exploration (i.e. start with a partition's pivot item and then expand it to the right and to the left).

2.5. Data-Parallel Execution Without Iterations

Alternative approach to data-parallel execution

All the algorithms surveyed so far were using a *task-parallel execution* approach: the initial database partitions might be used in the first step, but then the computation is split into independent tasks that are executed on different machines. Once the computation is split into the tasks, no communication and synchronization is required between the machines. However, as seen previously, the task-parallel execution depends on the quality of the tasks distribution between the machines: in the worst case all the machines will stay idle waiting for the one with the heaviest load. In addition, unlike the data-parallel execution approach, it does not benefit from the data locality. The input database is usually split into blocks distributed between the machines. Each task usually requires an access to all the blocks. On other hand, the original *data-parallel scheme* exemplified in e.g. NPSPM algorithm (surveyed in the 'Early Parallel Algorithms on SPM' subsection) suffers from communication and synchronization costs. For instance, to mine patterns of length N, NPSPM performs N iterations and the nodes have to synchronize twice in each iteration: the first one to share the locally-frequent candidates and their local counts (which also involves heavy communication cost), and the second one after summing up the local counts. So a number of papers came up with refined approach to the data-parallel execution scheme: **run a**

sequential partition-mining algorithm end to end on the initial data partition and then use the results afterwards. In general case, these algorithms involve three phases:

1. Run a preparation step (one or two MapReduce phases) that will help to reduce unnecessary work in the next phases. For instance, find all frequent and infrequent items.
2. Find locally-frequent patterns on each initial database partition by running some sequential partition-mining algorithm and benefiting from the data locality.
3. Distribute the candidates between the machines, count the frequencies of all the candidates on each machine and then sum them up to produce the globally-frequent patterns.

Although this general scheme might slightly vary between the algorithms, it shows the general idea of this approach. Unlike the data-parallel execution used in NPSPM, the number of times the nodes have to synchronize is limited. This scheme will be referred to as '***data-parallel execution without iterations***' scheme in the rest of this section. The simplest example of this scheme can be found in ([20], 2015) that implements Distributed GSP (DGSP) algorithm in two MapReduce jobs (the second phase in the above scheme is implemented in the second 'map' job of DGSP, and the third phase is implemented in its second 'reduce'). However, it's unclear whether DGSP is faster than any of the algorithms explored so far - most probably not as GSP is much slower than other more recent sequential algorithms. The next subsections explore three other algorithms that focus on a specific variant or extension of the SPM problem and offer competitive performance for their kind of problem.

PMSPX: Initial Database Partitioning to Minimize Skewness

In a distributed environment the input database is normally partitioned between the machines before running any SPM algorithm. Most algorithms containing *data-parallel execution* phases benefit from a good balance between these partitions: the more balanced this partitioning is, the less the machines will stay idle waiting for the most unfortunate ones. However, most such algorithms just assume that simple partitioning of the input database by size is good enough. This is not the case for the PMSPX algorithm ([22]) that highly depends on how good each partition represents the entire database. Hence the paper proposes the following approach, describing both steps:

1. Run once an algorithm to partition the input database between the machines in a way that minimizes skewness between the partitions. The goal is to partition the input database so that each partition could represent the entire database as good as possible.
2. Run PMSPX algorithm any number of times, with different parameters, and benefit from the quality of the initial database partition.

PMSPX algorithm: Let's first take a quick look on the PMSPX algorithm to understand why it depends so much on the quality of initial database partition. PMSPX is partially based on the sequential MSPX algorithm ([21]) and its goal is to mine maximal sequence patterns. The PMSPX algorithm is as followed:

1. Perform two iterations to find frequent items and frequent 2-sequences using NPSPM strategy (i.e. count locally and then exchange the counts on each iteration).
2. On each partition, run any sequential algorithm that mines maximal sequential patterns and determine the set of local maximal frequent sequences. (The paper uses MSPX, but any algorithm will do). Then unite all these sets to produce a set MFS* of all potential global maximal frequent sequences and distribute MFS* between all the nodes.
3. Iterate until MFS* is empty. In each iteration: let k be the current maximal sequence length in MFS*, then count all the k-sequences from MFS* and distribute this information between all the nodes.
   Each node updates its local copy of MFS* in the same way:
   If a k-sequence S turns out to be frequent, report it as the maximal frequent sequence and remove all its subsequences (if they exist) from the MFS*.
   Otherwise, add all (k-1)-subsequences of S to the MFS*.

If all the partitions are very similar, then on the phase 2 all nodes will report the same set of local maximal frequent sequences. The phase 3 will be very quick, as |MFS*| ≈ the size of global maximal frequent sequences and there will be no need to generate new candidates. However, the more dissimilar the partitions, the larger the MFS* will be and the more (k-1)-candidates will have to be generated from infrequent k-sequences.

Initial database partition strategies: The paper reports three strategies to achieve very similar partitions, sorted by increasing sophistication:

- *Simple Random Distribution (SRD)*: each node randomly chooses its next sequence
  - Performs reasonably well most of the time, but not always
- *k-SRD*: perform the SRD k times and choose the best partitioning, based on the quality function defined in the paper.
  - The quality function is computed as followed: take t (a user parameter) of the most locally frequent items, F_1^i , on a given partition, generate the set F_2^i of all candidate 2-sequences from F_1^i and compute their local counts. Compute $F_1 = \bigcup_i (F_1^i)$,

create F_2 of all candidate 2-sequences from F_1 and compute their global counts. Then measure the sum distance between vectors F_2^i and F_2.

- *cluSRD*: use a well-known clustering algorithm based on edit-distances between the sequences to create k clusters of similar sequences, then perform the SRD partitioning on each cluster, so each partition will have the same number of representatives from each cluster.

Note that this database partitioning is performed only once, before running the actual SPM algorithms. While the performance of most algorithms is less dependent on the quality of the initial database partition than PMSPX, all the algorithms that contain data-parallel execution steps might benefit from it. This might be especially true for the algorithms that utilize the '*data-parallel execution without iterations*' scheme that characterize all the algorithms in this section.

BigHUSP: Absence of Downward Closure Property

All the algorithms explored so far could rely on the downward closure property. This is not the case for High-Utility SPM (HUSPM): for instance, a total utility of an item *i* might be smaller than a total utility of a pair of items containing *i*. The BigHUSP algorithm ([24], 2015) utilizes the sequential USpan algorithm ([23]) for HUSPM in distributed environment. As USpan does not offer a natural way to partition the execution into independent tasks, BigHUSP employs the '*data-parallel execution without iterations*' scheme. It is able to run a sequential partition-mining algorithm (USpan) till the end on each data partition and use its results later.

The HUSPM problem has been defined previously. Given items with *weights* and a *q-sequence database* (each item occurrence has a quantity), the goal of HUSPM is to find sequential patterns whose utility in the database is above a user-defined minimum utility threshold δ. A *utility of a q-sequence s, u(s)* is $\sum_{i \in e,\, e \in s} weight(i) * quanity(i)$, q-sequence s= <s_1...s_m> *matches* a sequence t= <t_1..t_m> , $t \sim s$ if each s_k contains the same items as t_k, a *utility of a sequence t in a q-sequence s, v(t, s)* is $\max_{s' \subseteq s \wedge s' \sim t} u(s')$, and a *utility of a sequence t in a q-sequence database, v(t, D)*, is the sum of utilities of t in each q-sequence of the database D.

In addition to these terms, BigHUSP also defines the concept of ***Sequence-Weighted Utility (SWU)*** of a sequence: given a sequence t and a database D, *SWU(t, D)* is the sum of utilities of all sequences in D that contain t. *SWU(t, D)* has two interesting properties (both formally proved in the paper): (1) *SWU(t, D)* ≥ *v(t, D)* (2) SWU observes the downward closure property. It means that if for an item *i SWU(i, D)* < δ, the utility of any sequence that contains *i* will also be less than δ, so such items can safely be pruned. BigHUSP uses this to reduce the search space. It also uses a data structure that it calls ***Utility Matrix (UM)*** to represent each sequence. *UM* maps each occurrence of an item in a sequence to a pair (utility of the item occurrence i.e. item-quantity x item-weight, remaining utility of the tail of the sequence after this occurrence).

The outline of the BigHUSP algorithm:

1. Compute a *UM* per sequence and then perform a MapReduce iteration to filter out items whose *SWU* < δ in the standard 'word counting' fashion ('map' computes the local SWUs of the items and 'reduce' sums them and filter out the unpromising items).
2. Compute local high-utility sequential patterns (***L-HUSPs***) with their local utilities on each partition using any sequential algorithm. BigHUSP uses USpan for this purpose.
3. Perform a single MapReduce iteration to compute global utilities for the union of all L-HUSPs discovered in each partition and pick up the global high-utility sequential patterns.
    - Map: accepts the union of all L-HUSPs from all the partitions and returns the same L-HUSPs with their local utilities. Naturally, if some L-HUSP has been already discovered on the given partition in phase 2, there is no need to do anything as its utility is already known. But there might be sequences that are not high-utility in this partition and come from other partitions. The local utilities of these sequences are unknown and have to be computed. The algorithm computes them by gradually expanding a local lexicographic tree holding the intermediate sequences and their local utilities until the utilities of all L-HUSPs are computed.
    - Reduce: just sum the local utilities per L-HUSP pattern and pick up the sequences whose global utility ≥ δ.

To summarize, the BigHUSP algorithm employs the '*data-parallel execution without iterations*' scheme, using 4 MapReduce iterations. The bulk of the work is done on the local database partitions. Similarly to PMSPX, it might benefit from a balanced partitioning of the input database as suggested by the PMSPX paper ([22]).

DPSP: Distributed Algorithm on Progressive SPM

All the previous sections surveyed the batch parallel algorithms. DPSP ([25], 2010) suggests an algorithm to solve the problem of Progressive SPM, introduced previously. As a reminder, given the minimal support threshold σ and the *Period Of Interest (POI)* with duration ρ, the goal of a Progressive SPM is to find the frequent subsequences whose frequencies ≥ σ x (the number of sequences whose elements are in the current POI), where each sequence element is annotated with its timestamp. The system implementing the Progressive SPM should continuously read updates every t seconds and output all frequent patterns relevant to the given POI.

Similarly to the two previously explored algorithms, DPSP employs a '*data-parallel execution without iterations*' approach. It uses two MapReduce jobs (being run every t seconds):

1. Given the portion of updates, each machine computes the locally-frequent POI-relevant candidates (the candidates are computed in the 'map' phase and united in the 'reduce' one).
2. Produce POI-relevant frequent patterns from the set of candidates in a standard 'word counting' fashion: produce local count for each candidate in the 'map' phase and sum them to produce global counts in the 'reduce' phase, then pick up the globally frequent ones.

Similar to the previous algorithms in this section, DPSP relies on the sequential partition-mining algorithms to do the bulk of the computation on each partition.

2.6. Summary

While some of the past works explored parallel algorithms in the context of shared memory, most recent papers assume the *shared-nothing* architecture. In this context, the focus has shifted from message-passing frameworks (either completely manual or MPI) to the *MapReduce* ones which automate many tasks that previously had to be done by the developer. In the context of MapReduce, the focus is shifting from Hadoop to Apache Spark as it is faster and offers simpler API.

The parallel SPM algorithms can employ either *data-parallel* or *task-parallel* execution scheme; the task-parallel algorithms often employ a data-parallel approach at the preparatory phase. The works surveyed in this paper can be roughly categorized into the following three types:

- Parallel algorithms based on a sequential algorithm that offers a natural way to split the work into independent tasks. These algorithms employ a task-parallel execution scheme, but need to deal with load-balancing between the executing nodes as the tasks might be very different in size.
- Parallel algorithms based on inherently more balanced task partitioning, originally used in PFP-Growth algorithm for frequent patterns mining and then generalized for SPM problems. These works also employ task-parallel execution scheme, but focus on how to adapt this partitioning approach to a specific variant of the SPM problem and how to avoid unnecessary work on each machine.

- Parallel algorithms that are based on a sequential algorithm that does not offer a natural way of splitting the work into independent tasks, often to solve a variant of the SPM problem that is very different from the basic one in some aspect. The recent algorithms that employ the data-parallel execution approach strive to avoid distributed iterations to decrease communication and synchronization costs. Instead, a sequential algorithm is run on the original database partitions till the end to produce intermediate results which are used in the next phase to pick up the desired patterns. As the performance of such algorithms might depend on the quality of the initial database partitioning, one of the works offers ways to achieve it.

The scope of the survey excludes the following three classes of the SPM problem that might be significantly different from the rest:

- Parallel algorithms for SPM in uncertain databases
- Parallel algorithms for SPM extensions related to time series
- Patterns mining in multiple distributed databases

Observation: most works surveyed in this paper do not use the most recent sequential algorithms such as e.g. CM-Spade or CM-ClaSP ([27]) that offer faster execution than their predecessors in their respective SPM problems. In addition, many of the surveyed works seem to be unaware of some of the other works on parallel SPM. This shows the importance of proper categorization of the works on sequential patterns mining. While [1] provides a summary of serial SPM algorithms, to the best of the author's knowledge no similar work has been done on the parallel ones.

3. Conclusion

This paper surveys the works on parallel algorithms on the sequential patterns mining problem and its extensions and variations, excluding uncertain databases, multiple databases, and time series. The majority of the recent works use the MapReduce approach and could be broadly categorized into three classes described in this paper. Another observation is that many papers on parallel sequential patterns mining are only aware of a subset of the works in the field due to the sheer number of them. While there are papers that summarize serial sequential patterns mining algorithms, no similar work has been done on the parallel ones. This paper strives to help in bridging this gap.

References

1. Fournier-Viger P., Lin J. C., Kiran R. U., Koh Y. S., Thomas R. (2017). A survey of sequential pattern mining. Data Science and Pattern Recognition. 1(1), pp. 54-77.
2. Srikant R., Agrawal R. (1996). Mining sequential patterns: Generalizations and performance improvements. The International Conference on Extending Database Technology. 1(1), pp. 1–17.
3. Shintani T., Kitsuregawa M. (1998). Mining algorithms for sequential patterns in parallel: Hash based approach. In: Wu X., Kotagiri R., Korb K.B. (eds) Research and Development in Knowledge Discovery and Data Mining. PAKDD 1998. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence), vol. 1394. Springer, Berlin, Heidelberg.
4. Zaki M.J. (2000). Parallel Sequence Mining on Shared-Memory Machines. In: Zaki M.J., Ho CT. (eds) Large-Scale Parallel Data Mining. Lecture Notes in Computer Science, vol. 1759. Springer, Berlin, Heidelberg.
5. Han J., Pei J., Yin Y. et al. (2004). Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Mining and Knowledge Discovery. 8(1), pp. 53–87.
6. Pei J., Han J., Mortazavi-Asl B., Wang J., Pinto H., Chen Q., Dayal U., Hsu M. C. (2004). Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. IEEE Transactions on knowledge and data engineering. 16(11), pp. 1424–1440.
7. Guralnik V., Karypis G. (April 2004). Parallel Tree-Projection-Based Sequence Mining Algorithms. Journal of Parallel Computing. 30(4), pp. 443 – 472.
8. Agrawal R., Srikant R. (September 1994). Fast algorithms for mining association rules in large databases. In Bocca Jorge B., Jarke Matthias, and Zaniolo Carlo (Eds.). In Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487-499. Santiago, Chile.
9. Ayres J., Flannick J., Gehrke J., Yiu T. (2002). Sequential pattern mining using a bitmap representation. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 429–435.
10. Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, Edward Y. Chang (2008). PFP: Parallel FP-Growth for query recommendation. In Proceedings of the ACM Conference on Recommender Systems (RecSys'08). pp. 107–114.
11. Wang J., Han J., Li C. (2007). Frequent closed sequence mining without candidate maintenance. IEEE Transactions on Knowledge Data Engineering. 19(8), pp. 1042–1056.

12. Shengnan Cong, Jiawei Han, David Padua (August 2005). Parallel mining of closed sequential patterns. KDD '05 Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. pp. 562-567.
13. Yu D., Wu W., Zheng S., Zhu Z. (2012). BIDE-Based Parallel Mining of Frequent Closed Sequences with MapReduce. In: Xiang Y., Stojmenovic I., Apduhan B.O., Wang G., Nakano K., Zomaya A. (eds) Algorithms and Architectures for Parallel Processing. ICA3PP 2012. Lecture Notes in Computer Science, vol. 7440. Springer, Berlin, Heidelberg.
14. Wang Lifeng, Liu Dong, Wang Zhiyong (January 2017). Closed Sequential Pattern Mining Algorithm on Hadoop Platform. Revista de la Facultad de Ingenieria. 32(7), pp. 369-374.
15. Wei Yong-qing, Liu Dong, Duan Lin-shan (August 2012). Distributed PrefixSpan algorithm based on MapReduce. Information Technology in Medicine and Education (ITME), 2012 International Symposium on. 2, pp. 901-904.
16. Chen C. C., Tseng C. Y., Chen M. S. (2013). Highly scalable sequential pattern mining based on MapReduce model on the cloud. IEEE international congress on big data (BigData Congress' 13). pp 310–317.
17. Chun-Chieh Chen, Hong-Han Shuai, Ming-Syan Chen (2017). Distributed and scalable sequential pattern mining through stream processing. Knowledge and Information Systems, November 2017. 53(2), pp 365–390.
18. Beedkar K., Berberich K., Gemulla R., Miliaraki I. (2015). Closing the Gap: Sequence Mining at Scale. Journal of ACM Transactions on Database Systems (TODS), June 2015. 40(2), Article No. 8.
19. Beedkar K., Gemulla R. (2015). LASH: Large-Scale Sequence Mining with Hierarchies. Proceeding of SIGMOD '15 Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 491-503.
20. Yu X., Liu J., Liu X., Ma C., Li B. (2015). A MapReduce Reinforced Distributed Sequential Pattern Mining Algorithm. In: Wang G., Zomaya A., Martinez G., Li K. (eds) Algorithms and Architectures for Parallel Processing. Lecture Notes in Computer Science. vol. 9529. Springer, Cham.
21. Luo C., Chung S.M. (2005). Efficient mining of maximal sequential patterns using multiple samples. Proc of SIAM int'l conf on data mining.
22. Luo C., Chung S.M. (2012). Parallel mining of maximal sequential patterns using multiple samples. The Journal of Supercomputing, February 2012. 59(2), pp. 852–881.
23. Yin U., Zheng Z., Cao L. (2012). Uspan: An efficient algorithm for mining high utility sequential patterns. Proc. of ACM SIGKDD, 2012. pp. 660–668.

24. Zihayat Morteza, Zhenhua Hut Zane, An Aijun, Hut Yonggang (2016). Distributed and parallel high utility sequential pattern mining. 2016 IEEE International Conference on Big Data. pp. 853-862.
25. Huang J. W., Lin S. C., Chen M. S. (2010). DPSP: Distributed Progressive Sequential Pattern Mining on the Cloud. Advances in Knowledge Discovery and Data Mining. 6119, pp. 27-34.
26. Xingquan Zhu, Bin Li, Xindong Wu, Dan He, Chengqi Zhang (2011). CLAP: Collaborative pattern mining for distributed information systems. Decision Support Systems, December 2011. 52(1), pp. 40-51.
27. Fournier-Viger P., Gomariz A., Campos M., Thomas R. (2014). Fast Vertical Mining of Sequential Patterns Using Co-occurrence Information. In: Tseng V.S., Ho T.B., Zhou ZH., Chen A.L.P., Kao HY. (eds) Advances in Knowledge Discovery and Data Mining. PAKDD 2014. Lecture Notes in Computer Science. Vol. 8443. Springer, Cham.
28. http://hadoop.apache.org/
29. http://spark.apache.org/
30. http://mpi-forum.org/docs/
31. https://spark.apache.org/streaming/
32. M. J. Zaki (2001). SPADE: An efficient algorithm for mining frequent sequences. Machine learning, 2001. 42(1-2), pp. 31–60.