École polytechnique de Louvain (EPL)

CP based Sequence Mining on the cloud using spark

Dissertation presented by
Cyril DE VOGELAERE

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Pierre SCHAUS

Reader(s)
John AOGA, Guillaume DERVAL , Nijssen SIEGFRIED , Roberto D'AMBROSIO

Academic year 2016-2017

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Sequential pattern mining (SPM) is a challenging problem concerned with discovering frequent pattern in a given sequence dataset. Due to it's broad range of applications and it's relative complexity, it has been widely studied in the last two decades and a wide variety of efficient approaches have been developed, notably CP-based approaches which combine both great efficiency and flexibility through the addition of constraints.

With the advent of Big-data allowing large-scale data processing through parallel computation, most specialised approaches evolved further and displayed a qualitative step forward in efficiency. However, CP-based approaches have yet to be adapted to support parallel computations as parallel CP frameworks have yet to be developed.

In this paper we thus propose a novel parallel sequential pattern mining approach based on constraint programming modules, and designed to efficiently tackle large-scale data mining problems through parallel computations in a scalable environment.

In this endeavour, we use the recently designed CP-based algorithm PPIC, which outperform both other CP-based and specialised approaches by using ideas from both data mining and CP on a generic constraint solver, to locally solve sub-problems generated through a generic, map-reduce based, scalable PrefixSpan algorithm implemented using Spark.

We then show through detailed experimentation on popular datasets that, through this novel implementation, great performances can be obtained in a scalable architecture without losing much flexibility on the supported constraints.

1 Introduction

Sequential pattern mining (SPM) is a widely studied problem concerned with discovering frequent sub-sequences in a sequence database. The problem of SPM is to find all patterns in a sequence database with respect to the frequency constraint. The frequency is the number of number of occurrences of a pattern in the database. This problem has broad applications, including but not limited to: Analysis of customer purchase patterns, web-log mining, medical research, DNA sequencing, text-mining, human mobility mining, ... [1].

First introduced by Agrawal and Srikant in 1995 [2], sequential pattern mining problems have since been widely studied for more efficient way to find their combinatorially explosive number of possible subsequence patterns. Many techniques such as Apriori [2], GSP [3], cSPADE [4], and PrefixSpan [5] have thus been proposed to efficiently solve this problem.

Recently, constraint programming (CP) has been proposed as a framework for sequential pattern mining problems to solve the lack of flexibility and modularity of the previously cited approaches. Thus, various efficient CP-based algorithm such as CPSM [6], or Gap-Seq [7] where developed. The main benefit of those algorithms is their modularity, since their implementations allow the addition of multiple constraints to restrict the search space of the problem, and more efficiently work toward specific solutions. Those constraints encompassing a wide range of possibilities such as imposing frequency restrictions, imposing restrictions on the pattern length, ensuring the respect of regular expressions, ...

However, this increased flexibility can come with a performance cost. Many of the developed algorithm staying uncompetitive in comparison to state-of-the-art specialised methods. However, many recently made improvements, notably by Kemmar et al [8, 7], which had further extended this work by introducing one constraint (module) for both the pseudo-projection and the frequency pruning, and allowed CP based sequence mining algorithms to become competitive with state-of-the-art techniques such as Zaki's cSPADE [9].

A recent paper [10] went even further by designing a CP based implementation that greatly out-performs state-of-the-art implementations. This PrefixSpan based algorithm called PPIC, designed to find patterns on sequences of individual symbols and implemented on the open source CP-solver OscaR [11], achieved those improved performances by combining ideas from pattern mining and constraint programming.

More specifically, this new algorithm improves the efficiency of computing prefix-projected databases (the usual bottleneck of PrefixSpan based techniques) by using ideas from the LAPIN-SPAM algorithm [12]. This algorithm also improves the time needed to restore projected database when using a depth-first search (DFS) approach through the use of trailing which avoid unnecessary copies of the data.

However, the previously mentioned implementations were not ready for the recent advent of Big Data, as they were not adapted for large-scale data processing. Those implementation thus needed to evolve and support parallel computations, so that they could be executed in scalable 'big-data' environments and display further improved performances.

While non-CP algorithms were quickly adapted to those new environments with the introduction of a scalable implementation based on PrefixSpan [13] and cSPADE [14], CP-based approaches have yet to be adapted to support parallel computations.

Our objective in this paper is thus to adapt the recently designed PPIC algorithm to efficiently support parallel computations in a scalable architecture. Additionally, we took extra care to try keeping the inherent flexibility of CP-based implementation without sacrificing performances.

The reminder of this paper is thus organized as follows: In Section 2 we present the necessary background knowledge allowing one to understand SPM problems. Section 3 then explains and motive the different implementation created during this master thesis, including the final implementation which combine their strength. Finally Section 4 present the performance measured on each implementations, as well as the scalable performances of both the original implementation and our final implementation.

2 Sequential Pattern Mining

Sequence pattern mining (SPM) is a widely studied problem focusing on discovering sub-sequences in a dataset of given sequences. Each (sub) sequence being an ordered list of symbols, or sets of symbols. SPM has applications ranging from web log mining and text mining, to biological sequence analysis.

2.1 Sequential Pattern Mining Background

2.1.1 Definitions and Concepts

Definition 1. Symbol, Item and Item-sets: A symbol (s_i) is an element of a sequence, symbols can be repeated in sequences but, in a given database, a symbol will keep the same meaning. Symbols can be grouped into sets of symbols.
An item (i) is the integer representation of a symbol. While symbols could technically be anything from an Object to a primitive type, items are integer representation of those symbols allowing shorter representation of the database. Similarly, items can be grouped in sets, we refer to sets of items as Item-sets. Thus given S = $\{s_1, ... , s_M\}$ a set of M symbols where there is no ambiguity, the set of items representing these symbols is given by I $\in$ {1, ..., N}.
In the remainder of this paper, we refer to elements of sequence as symbols unless a representation as Item is necessary for the algorithm/concept concerned.

Definition 2. Sequence: A sequence seq = $\{S_1, S_2, ..., S_M\}$ is an ordered list of sets of symbols S_j, $j \in [1, M]$. The sequence of length N (N = $\sum_{j=1}^{M} |Sj|$) is called an N-length sequence. For example: ⟨a(ab)⟩ and ⟨abc⟩ are both 3-length sequences.

In the remained of this paper, we distinguish between two types of sequences. If a sequence of length N specifically contains M=N sets of symbols, we refer to it as a sequence of symbols (SoS). Otherwise, if a sequence contains a non-specified number of item M ($1 \leq M \leq N$), we refer to it as a sequence of sets of symbols (SoSS).

Example 1. ***Type of Sequences***

- *⟨(A B)D E(C A)⟩ is a sequence of sets of symbols (SoSS), where parenthesis act as delimiters between symbol sets. We are have two symbol sets (A B) and (C A) containing two symbols each for a total sequence length of 4.*
- *⟨A B C A⟩ is a sequence of symbols (SoS) of length 4 containing three different symbols. Symbol A being repeated multiple times in the sequence. Similarly ⟨(A)(B)(C)(A)⟩ is also a SoS despite the presence of separators, since each symbol set contains at most one element.*

Definition 3. Sub-Sequence ($\preceq$) / Super-Sequence: A sequence $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_m\}$ is a subsequence of seq = $\{S_1, S_2, ..., S_n\}$ and seq is a super-sequence of α if and only if: m $\leq$ n and there exist a one-to-one order-preserving function f that maps sets in α to sets in seq such that $\forall i \in [1, M]$, $\alpha_i \subset f(\alpha_i)$ and $\forall i, j$ with j > i, $f(\alpha_i)$ occurs before $f(\alpha_j)$ in seq.

Definition 4. Sequence database: A sequence database is a non-ordered collection of set of tuples (sid, seq). Where 'sid' is a sequence identifier and 'seq' a sequence. If a sequence database can only contains sequences of symbols, we shall denote it as SDB in the remained of this paper. In any other case we shall refer to the database by using the acronym SSDB.
Any algorithm mining sequential pattern for an SSDB should find the same solutions for SDB datasets as an SDB only algorithm. SDB specialised algorithms may never be used on SSDB.

Example 2. ***Type of Sequences Database***

- *(⟨A B C A⟩, ⟨D E C B⟩) is an SDB, since it contains only sequences of symbols.*

- *($\langle$(A)(B)$\rangle$, $\langle$(D)(E)(C)(B)$\rangle$) is also an SDB, since it similarly only contains sequences of symbols, the maximal Symbol Set size being one.*
- *($\langle$(A)(B)(C A)$\rangle$, $\langle$(D)(E)(C)(B)$\rangle$) is an SSDB, since it contains at least one sequence of sets of symbols, the first sequence of the database which contains (C A), a symbol set with multiple symbols.*

Definition 5. Cover, Support, Pattern, Frequent Pattern: The **cover** of a sequence seq in SSDB, denoted by $cover_{SSDB}(seq)$, is the subset of sequences in SSDB that are a super-sequence of seq. i.e. $cover_{SSDB}(seq) = \{(sid, s) \in SSDB | seq \preceq s\}$
The **support** of a sequence seq in SSDB, denoted $nbSupport_{SDB}(seq)$, is the number of sequence in the cover. i.e. $nbSupport_{SDB}(seq) = |cover_{SSDB}(seq)|$
Any sequence seq can be a **pattern** as long as it appears in the database, but we call **frequent patterns**, patterns where nbSupportSSDB(seq) $\geq \theta$, where θ is a given minimum support threshold.

Definition 6. Sequential Pattern Mining (SPM): Given a minimum support threshold θ and a sequence database SSDB, the SPM problem is to find all frequent patterns of the SSDB. i.e. all patterns p such that nbSupportSSDB(p) $\geq \theta$

Definition 7. Prefix, Suffix: Let $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_m\}$ be a pattern. If a sequence $\beta = \{\beta_1, \beta_2, ..., \beta_n\}$ is a super-sequence of α ($\alpha \preceq \beta$) then the prefix of α in β is the smallest sequence of symbols in β that is still a super-sequence of α (i.e. $\{\beta_1, \beta_2, ..., \beta_j\}$ such that $\alpha \preceq \{\beta_1, \beta_2, ..., \beta_j\}$ and $\nexists k < j$ such that $\alpha \preceq \{\beta_1, ..., \beta_k\}$).
The remains of β ($\{\beta_{j+1}, ..., \beta_n\}$) that are not part of the prefix are called suffix, and can be obtained by projecting the prefix away.

Definition 8. Prefix Projected database: A prefix-projected database of a prefix α, denoted by SDB_α, is the set of prefix-projections of all sequences in SDB that are a super-sequence of α.

Example 3. ***Type of Sequences***
In the database of Figure 1.a, we can observe that:

- *The database is an SSDB.*
- *$\langle$(ab)$\rangle$ is a frequent pattern for $\theta = 2$.*
- *$\langle$(abc)$\rangle$ and $\langle$(ab)c$\rangle$ are both super-sequences of $\langle$(ab)$\rangle$, but only $\langle$(ab)c$\rangle$ is a frequent pattern given $\theta = 2$ since $nbSupport_{SDB}(\langle(abc)\rangle) = 1$.*
- *The cover of $\langle$(ab)$\rangle$ in the given database is the subset of sequences ($\langle$a(abc)(ac)d(cf)$\rangle$, $\langle$(ef)(ab)(df)cb$\rangle$). Thus $nbSupport_{SDB}(\langle(ab)\rangle) = 2$.*
- *The prefix of $\langle$(ab)c$\rangle$ in sequence 10 is $\langle$a(abc)(ac)$\rangle$, the suffix of the same pattern in the sequence 10 is $\langle$d(cf)$\rangle$*

2.1.2 Existing specialised approaches

2.1.2.1 Apriori

The Apriori algorithm, created in 1995 [2], was designed for frequent item-sets mining in a transactional database. This breath first search (BFS) algorithm, whose performances are now surpassed by more modern techniques, finds all frequent item-sets by iteratively growing its sequential patterns.

First, at the start of each iteration, all N-length candidate will be generated (N being the number of the iteration) using the result of the previous iteration and the 1-length candidates as basis. The support of those candidates will then be checked with the databases and un-frequent pattern will be deleted. Finally, frequent N-length patterns will be sent onto the next iteration, until none can be grown.

While historically relevant, this algorithm was inefficiently generating all possible N-length candidates using the N-1-length candidates as basis. Among those a large amount of candidate wouldn't be frequent, and would thus waste huge amount of memory and CPU time being stocked and uselessly verified across the database.

One of its good points however, lies in its ability to be used to detect association rules (Example: When A is present, B has 75% chance of being also present), which can give indications about the general trends in the database and allow human operators to get a general understanding of the inputted dataset.

2.1.2.2 GSP

The Generalised Sequential Pattern (GSP) algorithm [3] was based on the Apriori algorithm but redesigned for sequential pattern mining instead of frequent item-set mining. One of the good points of this new algorithm lied in the possibility to add time constraints that specified a minimum and/or maximum time period between adjacent elements in a pattern.

As Apriori, the algorithm start by detecting frequent 1-length pattern, it then proceeds with generating all 2-length patterns from there. However, since order now matters as items that should be grouped together can come from multiple transactions, there is now a lot more candidates to create.

More specifically, given two items A and B, the generated candidate would be AB, BA and (AB). (AB) being the representation of those two items occurring in the same transactional time frame. Those candidates will then be projected on the database, and their support will be counted. All candidates having a lesser amount of support than the threshold θ will then be cleaned.

The algorithm should then generate further candidates, but unlike Apriori, the method has been complexified to become far more efficient. Instead of growing sequential patterns by adding all 1-length patterns to each of their end and creating a tremendous amount of unsupported patterns, we create those candidates more efficiently.

First, for each sequential patterns of N-length found in the previous iterations, we detect its first and last N-1 item, thus creating two sub-sequential patterns by omitting an element either at the end or start. We will then create N+1-length candidates by composing sequential patterns with similar end and start sub-sequential pattern.

For example, given the sequential patterns AA, (AB), AB and BA, we would be able to generate the following candidates: A(AB), (AB)A, B(AB), (AB)B, ABA, BAB, AAB, BAA.
Those candidates should then be pruned to remove impossible combinations found in previous iterations. In the case of our example, since BB was not retained as a 2-length sequential pattern due to being un-frequent, we can delete all candidate sequential patterns containing BB, as they similarly cannot be present.
We are thus left with: A(AB), (AB)A, ABA, AAB, BAA

Finally, we count the number of support for each candidates having passed the pruning phase, and remove unsupported sequential patterns. Another iteration of generating candidates, pruning and count support will then start, until no supported sequential patterns is found at the end of an iteration, or no candidates can be generated.

If after counting the support of each candidate we detect that only ABA and BAA are supported, the next iteration would only create the candidate ABAA. Since it would be our only 4-length pattern, no 5-length pattern will be generated and the execution will stop having determined that all solutions were found.

As you may expect, this new method to generate candidates allowed GSP to surpass Apriori's efficiency by a wide margin. Since the algorithm additionally supported a wide range of new constraints allowing reduction of the search space, and was so efficient, it became a reference algorithm in frequent pattern mining.

2.1.2.3 PrefixSpan

The PrefixSpan approach [5] relies on pattern growth. As you may see in our simple example displayed in Figure 1, the idea of this algorithm is to find the complete set of patterns through iteratively growing prefixes by projecting them on the database, and finding extensions.

The main selling point of the algorithm lies in its ability to quickly generate candidate extensions, projecting prefixes on the database significantly reducing the time needed to find prefixes extensions, as only the suffixes need to be searched for those extensions to be found. Another selling point lies in its scalability, as this method is easily implementable under a Map-Combine-Reduce programming scheme.

However, this method also has disadvantages, mainly in the fact that projecting databases is a time expensive process. Making building those projected databases the efficiency bottle-neck of PrefixSpan.

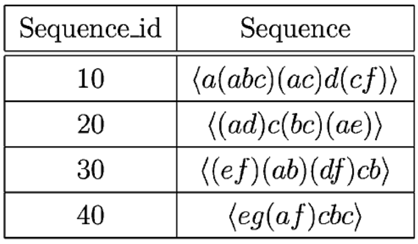
Fortunately, techniques exists to break past this bottle-neck and mitigate its effects. Techniques among which we should mention the 'Bi-level projection' technique, which consists in creating a triangular matrix (S-matrix [24]) registering the number of supports for all 2-length sequences that can be assembled from 1-length prefixes. A quick scan of the S-matrix then allows the detection of all supported 2-length prefixes, prefixes which can then be extended again through applying the same technique on their projected database. Thus allowing a reduction of the number of necessary prefix projections during the execution of the algorithm, since only half of the prefixes will need to be projected.

A second technique we should mention is the 'Pseudo-Projection' technique which is based on keeping and maintaining a start index for each sequence. This start-index registering the lowest position at which the last projected prefix was supported, thus allowing quicker projection of extending prefixes by avoiding to re-project previous prefix elements.

2.1.2.4 cSPADE

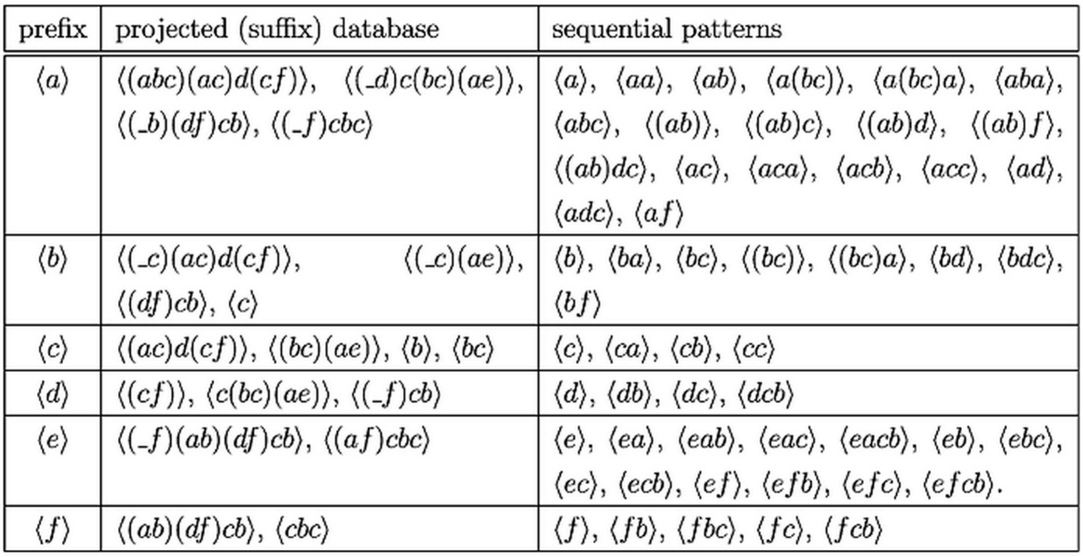
The cSPADE algorithm [9] uses combinatorial properties to divide the original sequence pattern mining problem in smaller sub-problems that can be solved independently using simple id-list join operations. This algorithm is thus parallelisable, furthermore, the parallelisation has a linear scalability with respect to the size of the inputted database.

cSPADE's first step is to compute all 1-length sequential patterns using a simple database scan. Then, we generate all 2-length sequential patterns and count the number of supporting



Sequence_id	Sequence
10	$\langle a(abc)(ac)d(cf)\rangle$
20	$\langle (ad)c(bc)(ae)\rangle$
30	$\langle (ef)(ab)(df)cb\rangle$
40	$\langle eg(af)cbc\rangle$

(a) Original database of our SPM example



prefix	projected (suffix) database	sequential patterns
$\langle a\rangle$	$\langle (abc)(ac)d(cf)\rangle$, $\langle (\_d)c(bc)(ae)\rangle$, $\langle (\_b)(df)cb\rangle$, $\langle (\_f)cbc\rangle$	$\langle a\rangle$, $\langle aa\rangle$, $\langle ab\rangle$, $\langle a(bc)\rangle$, $\langle a(bc)a\rangle$, $\langle aba\rangle$, $\langle abc\rangle$, $\langle (ab)\rangle$, $\langle (ab)c\rangle$, $\langle (ab)d\rangle$, $\langle (ab)f\rangle$, $\langle (ab)dc\rangle$, $\langle ac\rangle$, $\langle aca\rangle$, $\langle acb\rangle$, $\langle acc\rangle$, $\langle ad\rangle$, $\langle adc\rangle$, $\langle af\rangle$
$\langle b\rangle$	$\langle (\_c)(ac)d(cf)\rangle$, $\langle (\_c)(ae)\rangle$, $\langle (df)cb\rangle$, $\langle c\rangle$	$\langle b\rangle$, $\langle ba\rangle$, $\langle bc\rangle$, $\langle (bc)\rangle$, $\langle (bc)a\rangle$, $\langle bd\rangle$, $\langle bdc\rangle$, $\langle bf\rangle$
$\langle c\rangle$	$\langle (ac)d(cf)\rangle$, $\langle (bc)(ae)\rangle$, $\langle b\rangle$, $\langle bc\rangle$	$\langle c\rangle$, $\langle ca\rangle$, $\langle cb\rangle$, $\langle cc\rangle$
$\langle d\rangle$	$\langle (cf)\rangle$, $\langle c(bc)(ae)\rangle$, $\langle (\_f)cb\rangle$	$\langle d\rangle$, $\langle db\rangle$, $\langle dc\rangle$, $\langle dcb\rangle$
$\langle e\rangle$	$\langle (\_f)(ab)(df)cb\rangle$, $\langle (af)cbc\rangle$	$\langle e\rangle$, $\langle ea\rangle$, $\langle eab\rangle$, $\langle eac\rangle$, $\langle eacb\rangle$, $\langle eb\rangle$, $\langle ebc\rangle$, $\langle ec\rangle$, $\langle ecb\rangle$, $\langle ef\rangle$, $\langle efb\rangle$, $\langle efc\rangle$, $\langle efcb\rangle$.
$\langle f\rangle$	$\langle (ab)(df)cb\rangle$, $\langle cbc\rangle$	$\langle f\rangle$, $\langle fb\rangle$, $\langle fbc\rangle$, $\langle fc\rangle$, $\langle fcb\rangle$

(b) Prefix projected databases relative to 1-length sequential patterns &
final list of sequential patterns eventually extended from those 1-length prefixes

Please take note that underscores in the projected databases represent itemSet's items that have been 'consumed' by the projection. Since we consider that itemSets are sorted in PrefixSpan, smaller item are 'consumed' when projecting larger item on itemSets. For example, projecting pattern ⟨b⟩ on sequence 10 will also 'consume' item a in the first itemSet (abc).

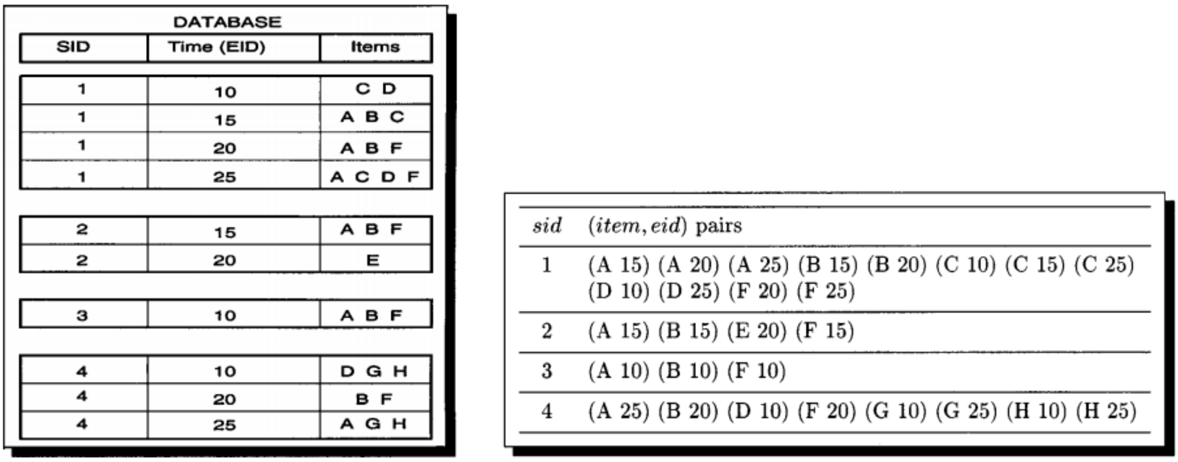
Figure 1: PrefixSpan example

sequences for each pair of items in a bi-dimensional matrix. Counting the number of supporting sequence being realised through another database scan by first transforming the original vertical representation of the database into an horizontal representation (see Figure 2).

Subsequent N-length sequence patterns can then be formed by joining (N+1)-length patterns using their id-lists (list of positions in sequences for each item, see Figure 3). The support of each item can also be easily calculated from ID-list, as we just need to count the number of different sequences in which it appears.

It is also important to note that this method of joining ID-lists is only efficient from 3-length patterns onward, as ID-lists for length 1 and 2 patterns can be extremely large and potentially wouldn't fit in memory.

Of course, at the end of each round, un-frequent sequential pattern should be cleaned as to guarantee only frequent patterns will be extended. This algorithm can be executed using either breadth first or depth first search, and ends its execution once no patterns can be further extended.



DATABASE

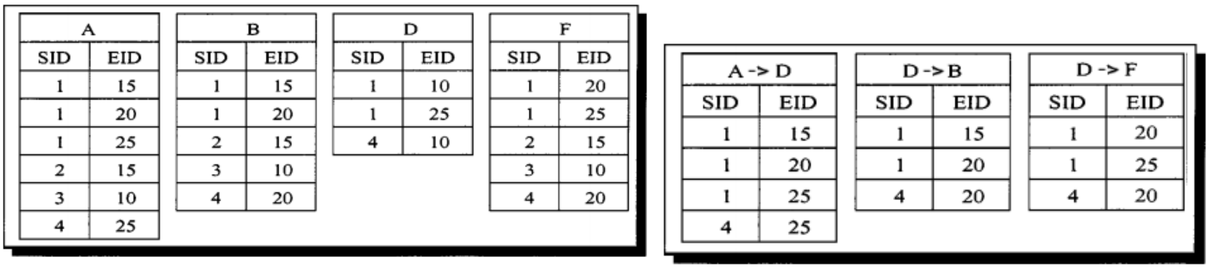
SID	Time (EID)	Items
1	10	C D
1	15	A B C
1	20	A B F
1	25	A C D F
2	15	A B F
2	20	E
3	10	A B F
4	10	D G H
4	20	B F
4	25	A G H

sid	$(item, eid)$ pairs
1	(A 15) (A 20) (A 25) (B 15) (B 20) (C 10) (C 15) (C 25) (D 10) (D 25) (F 20) (F 25)
2	(A 15) (B 15) (E 20) (F 15)
3	(A 10) (B 10) (F 10)
4	(A 25) (B 20) (D 10) (F 20) (G 10) (G 25) (H 10) (H 25)

(a) Vertical representation of the database

(b) Horizontal representation of the database

Figure 2: cSPADE database representation



A		B		D		F	
SID	EID	SID	EID	SID	EID	SID	EID
1	15	1	15	1	10	1	20
1	20	1	20	1	25	1	25
1	25	2	15	4	10	2	15
2	15	3	10			3	10
3	10	4	20			4	20
4	25						

A -> D		D -> B		D -> F	
SID	EID	SID	EID	SID	EID
1	15	1	15	1	20
1	20	1	20	1	25
1	25	4	20	4	20
4	25				

(a) ID-list of 1-length sequence patterns

(b) ID-list of 2-length sequence patterns

Figure 3: cSPADE's ID-lists

2.1.3 Existing CP Based approaches

2.1.3.1 CPSM

The CPSM approach [6] was implemented to solve sequential pattern mining problems involving only sequences of symbols (no symbols set) using generic Constraint Programming (CP) solvers. It creates a global constraint[1] allowing a search for frequent patterns over the database. Thanks to its integration in CP solvers, it also easily support additional constraints such as size constraints or regular-expression constraints.

The global constraint, which forms the basis of this approach, is called the global exists-embedding constraint. Based on an incremental propagator (DFS approach), the algorithm will incrementally extend a prefix, until no extensions can be successfully projected.

At each step, the solver will take care of assigning the next extensions in our search space. The constraint will then search said extensions to verify its validity. Of course, the whole sequence won't be searched, but only positions larger than the smallest position supporting the previous valid prefix in each sequence.

If the extension is valid and its support exceeds the threshold θ, the prefix will be considered 'embedded' in the database. The algorithm thus works by creating all prefixes through various incremental extensions and numerous backtracking, then verifying their embedding in

[1] The paper also present an alternative to this global constraint allowing a decomposition with per-sequence constraint. It is whoever far less efficient, we thus decided to emit the presentation of this algorithm here.

the database.

Since trying all possible extensions at each incremental step would be inefficient, the algorithm was also designed to prune the next possible extensions that should be considered by the solver, and only keep extensions that are sufficiently supported in the database. Thus, symbols that are considered unfrequent won't be branched over during the execution. This pruning is done by counting the number of support for each extension during the embedding existence verification.

Although impressive for its modularity, the efficiency of this algorithm didn't quite caught up to specialised non-CP algorithms such as PrefixSpan or cSPADE. It, however, surpassed their efficiency at searching for specific solutions thanks to its modularity.

2.1.3.2 PP

This second approach based on CPSM proposed a slightly different global constraint by taking ideas from prefix-projection [8]. Similarly to its predecessor, this algorithm was designed to solve sequential pattern mining problems involving sequence of symbols (no sets of symbols).

The main idea behind this improved constraint being that we have no need to check sequences for extensions support when they did not support sub-sequences of the current prefix.

This new algorithm thus takes notes of the ID of sequences which support the current pattern so that its extensions may be projected only on those sequences. The key element of this new feature being that we keep ID of sequence and not copy of sequences, making it extremely memory and time efficient.

During the extensions pruning phase, only supportive sequences will need to be checked in a similar fashion, as they are the only relevant sequences to find extensions with.

While the rest of this global constraint implementation was very similar to CPSM's global constraint, the prefix projection improvement allowed this new implementation to largely overtake its predecessors in efficiency. Making this new algorithm competitive with state-of-the-art non-CP method while keeping the improved modularity inherent to CP algorithms.

2.1.3.3 Gap-Seq

This new algorithm introduced a global constraint adding support for time gap constraint [7]. This allows, for example, to analyses purchase behaviours and find products usually bought by customers at regular time intervals.

Similarly to its predecessors, the algorithm was designed to solve sequential patterns mining problems involving sequences of symbols, doing so by incrementally extending and verifying prefixes over multiple pass of a prefix projected database.

The new global constraint, however, allow tighter search spaces where one can specify the minimal/maximal distance allowed between two symbols for them to remain solution.

For this constraint to work efficiently, additional pruning was added when pruning for extending items, so that extensions which would violate the time gap constraints couldn't be taken into account.

2.1.3.4 PPIC

Largely based on its predecessor, CPSM and PP, PPIC[2] is an algorithm designed for solving sequence pattern mining problems involving sequences of symbols. Unlike Gap-seq, constraint over time gaps are not supported.

This new algorithm speciality lying in its record-breaking efficiency, surpassing even state-of-the-art non-CP solver, generally by a wide margin (see Figure 25).

PPIC's implementation is based on the PrefixSpan approach, it's execution can be separated in two stages:

1. **Pre-processing:** In this first stage, we first clean the received sequences from unfrequent symbols, renaming them into unique items. Three matrices are then build from the sequence database:

   (a) The 'first-position' matrix: A #SDB*N sized matrix allowing O(1) jumps to the first occurrence of a given item.

   (b) The 'last-position' matrix: A #SDB*N sized matrix allowing O(1) check for the presence of a given item in the remains of a sequence.

   (c) The 'interesting-position' matrix: A matrix with the same size as the original sequence database, but whose content is changed from the items forming those sequences, to the positions of the next 'interesting' item. That is the next position where an item last appears in a sequence.
   Although, at first glance, this matrix may seem redundant with the last-position matrix, its purpose appears when one realises that to achieve the same goal a whole column of the last-position matrix would have to be checked. Similarly, keeping only this matrix would also be less efficient, since there would be no way to efficiently check if an item is present in the remains of a sequence. Both matrices are thus needed to achieve the greatest efficiency gain.

   The pre-processing stage will also take care of adding multiple constraints, depending on the wishes of the user. Thus restricting the search space to fit more tightly the desired solutions and improve the algorithm's performances at reaching those specific solutions.

2. **Execution:** Once the pre-processing is finished, the algorithm will truly start to run. Using the three matrices, prefixes will be extended efficiently using an incremental propagator (DFS approach). An approach possible thanks to trailing, that is more efficient than BFS approach were multiple copies of the database would need to be kept, or where the prefixes would need to be re-projected. Thanks to this incremental propagation, memory consumption will thus be minimal during the execution of the algorithm.

   Through each step of the DFS execution, the last item of the current prefix will be projected efficiently as, similarly to CPSM, the algorithm keeps track of the minimal index after which the previous valid pattern was considered supported. Thus, only the remains of the sequence will need to be searched for confirming the validity of the new projection. Once projected, the algorithm will then prune possible solutions for the next pattern efficiently, thanks to the lastPosition list, and continue the execution.

   Additionally, to keep the increased performances of the PP algorithm, the algorithm keep notes of which sequences supports the current prefix while projecting the current extension. For further extensions of the prefix, only those sequences will thus have to be considered during any projection or pruning. Since the algorithm also keeps track of the number of

[2] Open-source code available at `http://sites.uclouvain.be/cp4dm/spm/`

support for each extension, an improvement was also made to stop searching for projections once that number of supporting sequences have been found during the prefix's projection. Since the remaining sequences definitely won't support the item and will remain irrelevant further down this branch of the search tree.

Each time a valid solution is found. That is, a solution that satisfy all constraints injected in the solver. The execution will be momentarily interrupted. The solution will then be translated back to the symbols corresponding to the recorded items and saved in a result list.

Once the solver determines that no further solution can be found under the specified constraints. The execution will terminate, and all resulting pattern will be returned.

An example of a complete execution of PPIC can be found in Figure 4. The pseudo-code can be found in Algorithm [2, 3].

Also, there exist a version of PPIC (called PPICt [15])supporting time constraint (GAP, SPAM) and that provides better performances than Gap-Seq and cSPADE.

2.2 Parallelisation

As said previously in the introduction, our goal is to achieve a scalable implementation based on PPIC, a CP algorithm based on the PrefixSpan approach.

2.2.1 The Benefits of Parallelisation

The benefits of achieving such a scalable algorithm are many. First, local algorithms are limited to the power of the computer they run on, achieving parallelism allows to break that limit and increase the overall efficiency by running on multiple computers at once.

Another advantage lies in costs as, with the arrival of cloud technologies, running algorithms on clusters of small machines is gradually becoming cheaper than buying and maintaining supercomputers.

For both those reasons, and the fact that a large quantity of SPM problems cannot be run on local architectures, designing implementations that are both scalable and efficient has gradually become extremely important for the industry.

2.2.2 Tool Selection

Since, fortunately, SPM problems are embarrassingly parallel problem, we had the opportunity to choose from a selection of widely used open-source libraries. We, however, restricted ourselves to Scala compatible framework, as the CP library supporting PPIC was implemented in this language. Rapidly, we were left to choose from two major options:

1. Hadoop mapreduce
2. Spark

2.2.2.1 Hadoop

Hadoop is an open-source framework that allows large-scale data processing across clusters of machines. Based on a Map-reduce programming model, this framework allows larger scale iterative computations on humongous quantities of data. At each iteration, the data is read from

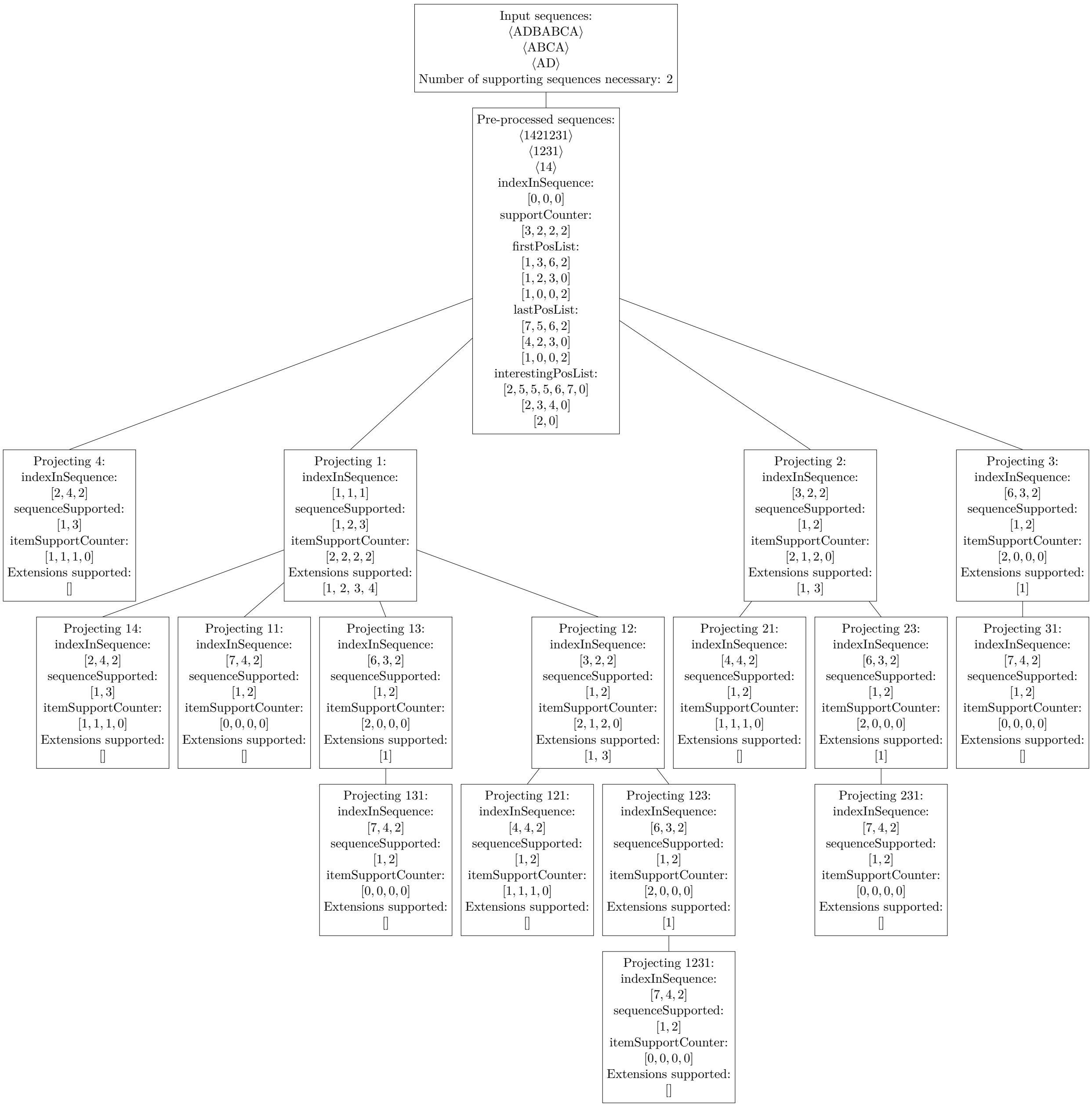

Input sequences: ⟨ADBABCA⟩ ⟨ABCA⟩ ⟨AD⟩ Number of supporting sequences necessary: 2
Pre-processed sequences: ⟨1421231⟩ ⟨1231⟩ ⟨14⟩ indexInSequence: [0, 0, 0] supportCounter: [3, 2, 2, 2] firstPosList: [1, 3, 6, 2] [1, 2, 3, 0] [1, 0, 0, 2] lastPosList: [7, 5, 6, 2] [4, 2, 3, 0] [1, 0, 0, 2] interestingPosList: [2, 5, 5, 5, 6, 7, 0] [2, 3, 4, 0] [2, 0]
Projecting 4: indexInSequence: [2, 4, 2] sequenceSupported: [1, 3] itemSupportCounter: [1, 1, 1, 0] Extensions supported: []
Projecting 1: indexInSequence: [1, 1, 1] sequenceSupported: [1, 2, 3] itemSupportCounter: [2, 2, 2, 2] Extensions supported: [1, 2, 3, 4]
Projecting 2: indexInSequence: [3, 2, 2] sequenceSupported: [1, 2] itemSupportCounter: [2, 1, 2, 0] Extensions supported: [1, 3]
Projecting 3: indexInSequence: [6, 3, 2] sequenceSupported: [1, 2] itemSupportCounter: [2, 0, 0, 0] Extensions supported: [1]
Projecting 14: indexInSequence: [2, 4, 2] sequenceSupported: [1, 3] itemSupportCounter: [1, 1, 1, 0] Extensions supported: []
Projecting 11: indexInSequence: [7, 4, 2] sequenceSupported: [1, 2] itemSupportCounter: [0, 0, 0, 0] Extensions supported: []
Projecting 13: indexInSequence: [6, 3, 2] sequenceSupported: [1, 2] itemSupportCounter: [2, 0, 0, 0] Extensions supported: [1]
Projecting 12: indexInSequence: [3, 2, 2] sequenceSupported: [1, 2] itemSupportCounter: [2, 1, 2, 0] Extensions supported: [1, 3]
Projecting 21: indexInSequence: [4, 4, 2] sequenceSupported: [1, 2] itemSupportCounter: [1, 1, 1, 0] Extensions supported: []
Projecting 23: indexInSequence: [6, 3, 2] sequenceSupported: [1, 2] itemSupportCounter: [2, 0, 0, 0] Extensions supported: [1]
Projecting 31: indexInSequence: [7, 4, 2] sequenceSupported: [1, 2] itemSupportCounter: [0, 0, 0, 0] Extensions supported: []
Projecting 131: indexInSequence: [7, 4, 2] sequenceSupported: [1, 2] itemSupportCounter: [0, 0, 0, 0] Extensions supported: []
Projecting 121: indexInSequence: [4, 4, 2] sequenceSupported: [1, 2] itemSupportCounter: [1, 1, 1, 0] Extensions supported: []
Projecting 123: indexInSequence: [6, 3, 2] sequenceSupported: [1, 2] itemSupportCounter: [2, 0, 0, 0] Extensions supported: [1]
Projecting 231: indexInSequence: [7, 4, 2] sequenceSupported: [1, 2] itemSupportCounter: [0, 0, 0, 0] Extensions supported: []
Projecting 1231: indexInSequence: [7, 4, 2] sequenceSupported: [1, 2] itemSupportCounter: [0, 0, 0, 0] Extensions supported: []

Figure 4: A simple execution of PPIC's algorithm.
The solution patterns are the projected prefixes

the distributed file system (HDFS), modified through a MapReduce, then stored back on the file system.

The advantages of Hadoop thus lies in the simplicity of its usage. Aside from implementing the Map and Reduce process, Hadoop will take care of scheduling, data repartition and failure recovery.

Widely used since its initial release on December 10, 2011. Hadoop slowly climbed to become one of the big standards in terms of large scale computation. We thus selected it as our potential scalable framework, discovering shortly after that an efficient implementation of PrefixSpan on Hadoop was already available on the internet.

2.2.2.2 Spark

Spark is an open-source engine for large-scale data processing. Mainly reputed for its speed, ease of use, and ability to efficiently implement sophisticated problems.

Originally developed at UC Berkeley in 2009, Its entire implementation revolves around an immutable read-only data structure called the **resilient distributed dataset (RDD)**. Maintained in a fault-tolerant way, those lazily computed RDD, built through deterministic coarse-grained transformation, have been designed to be efficiently distributed over a cluster of machines, allowing resolution of complex iterative problems in scalable environments.

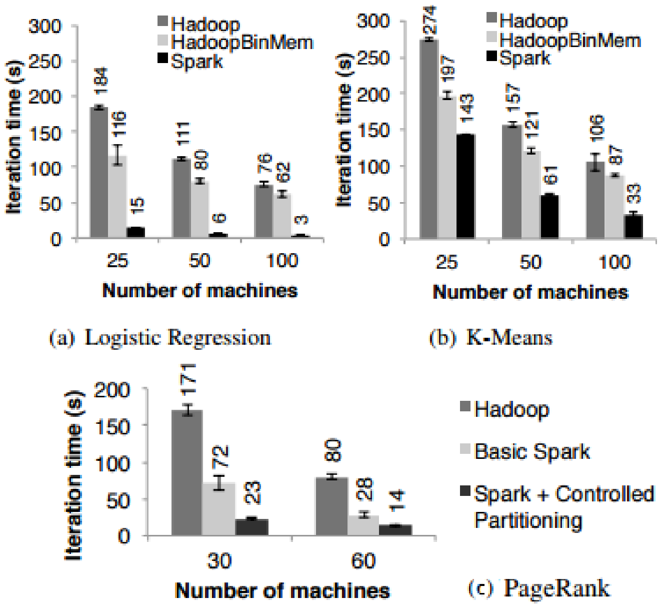
Furthermore, Spark has been designed to make use of its clusters RAM memory efficiently, allowing the engine to distance itself from slow HDD memory access. Of course, should the RAM memory be insufficient, Spark is perfectly able to run using nothing but the hard-drive.

Spark's was thus an extremely valid choice from a technical standpoint and, similarly to Hadoop, we were surprised to discover an existing implementation of PrefixSpan available in Spark's machine learning library.

2.2.2.3 Final Choice

As said earlier, both of those libraries already disposed of a scalable PrefixSpan implementation, and both were efficient and widely recognised framework to achieve parallelism. It was thus a matter of determining whose performances were better, and whether those implementations could be efficiently extended through CP technologies.

Fortunately for us, performance comparison had already been done in a widely recognised scientific paper on Spark's RDD [16]. Those performances are presented in Figure 5



(a) Logistic Regression

(b) K-Means

(c) PageRank

Figure 5: Performance comparison of Hadoop and Spark

As we can see, performance-wise, Spark vastly outperform Hadoop thanks to its ability to use both memory and disk for its computations. Allowing up to 100x speed-up under the right circumstances, as we can see in the logistic regression problem. According to the official website, Spark would also boast a 10x speed-up through on disk computation only, but no performance benchmarks were provided to back that claim.

In terms of extensions through CP technologies, we quickly realised that Hadoop would be far less practical. Although MapReduce can be used to execute the standard PrefixSpan algorithm, and could certainly be modified to introduce CP elements, Spark can support any coarse-grained transformation with its RDDs, allowing a more precise implementation where only required transformations would be made, instead of simple sequences of Map-Combine-Reduce.

3 Implementation of a Scalable CP Based Algorithm

In this section, we shall present the various implementation we created in an attempt to improve Spark's original algorithm's performances. The performance of those implementations, however, will be tested in a later section.

3.1 Spark's original implementation

Before introducing our various implementation, let us present Spark's original algorithm. An algorithm based on the PrefixSpan approach, and can be separated in four stages:

1. **Pre-processing:** The goal of this stage is to replace each symbol of the sequence database by a unique item, to separate item-sets through a zero delimiter, and to clean the database from unfrequent items.
   For example, the sequence ⟨(ABD)(ABC)A⟩ will become 0120123010, assuming only symbols A, B and C are frequent in the sequence database.

2. **Scalable execution:** The core of the algorithm. Its execution consists in extending prefixes through a three sub-stage process, starting from the empty prefix.

   (a) First, a large prefix is projected on the database, meaning that only the suffix of supporting sequence remain in the resulting database. When there is no prefix to project, the scalable execution comes to an end and the algorithm goes on to the next stage.

   (b) Then, from the set of supporting sequences, we discover symbols that can extend the current prefix. If no such extension exists, we try the next large prefix.

   (c) Finally, for each possible symbol extension, we extend the corresponding prefix. We then determine how long further expanding each extended prefix may take by calculating the projected database size. Depending on the calculated projected size and of the value of a user defined parameter, we then either further extend this prefix using another iteration of the scalable execution, or store it for use in the local execution stage.

3. **Local execution:** The local execution is completely similar in its implementation to the scalable execution. Its only use is in significantly improving the algorithm's performance by calculating all extensions from a Prefix locally, instead of doing so while shuffling information around the scalable architecture.
   This stage is only launched once all large prefixes have been extended sufficiently, making the projected databases that need to be processed to find future extensions small enough. Depending on the parameters inputted by the user, this stage may be skipped.

4. **Post-processing:** During the post-processing step, we translate back the unique items into the corresponding symbols they each represented. Then we send back the collected results to the user.

During the prefix projection phase of the scalable and local execution, the algorithm will also detect which item-sets in the sequence comply with what has already been projected and can still be extended in some way. Storing such items positions in a 'partial projection' list.
That way, if we are projecting an item-set containing multiple symbols from a prefix, until the end of that item-set, the items projection phases will know where to search possible extensions, preventing the algorithm from having to search the whole sequence.
Also, should we be computing on a database of sequence of symbols, no partial start would be created and kept in memory, since the algorithm would automatically detect that those item-sets

cannot be extended.

When the current item-set end, the full remains of the sequence, from the earliest partial position recorded, will have to be searched for extensions. When extending the first item of the new item-set, new potential partial projection will be recorded, and the old ones will be discarded.

During the prefix extension phase, the current partial projection will also be used to find extensions of the current item-set quicker. The remainder of the database will also be searched, but only for extension that starts new item-sets.

An example of a fully scalable execution from this algorithm can be found in Figure 6. A pseudo-code can also be found in Algorithm [4, 5, 6, 7].

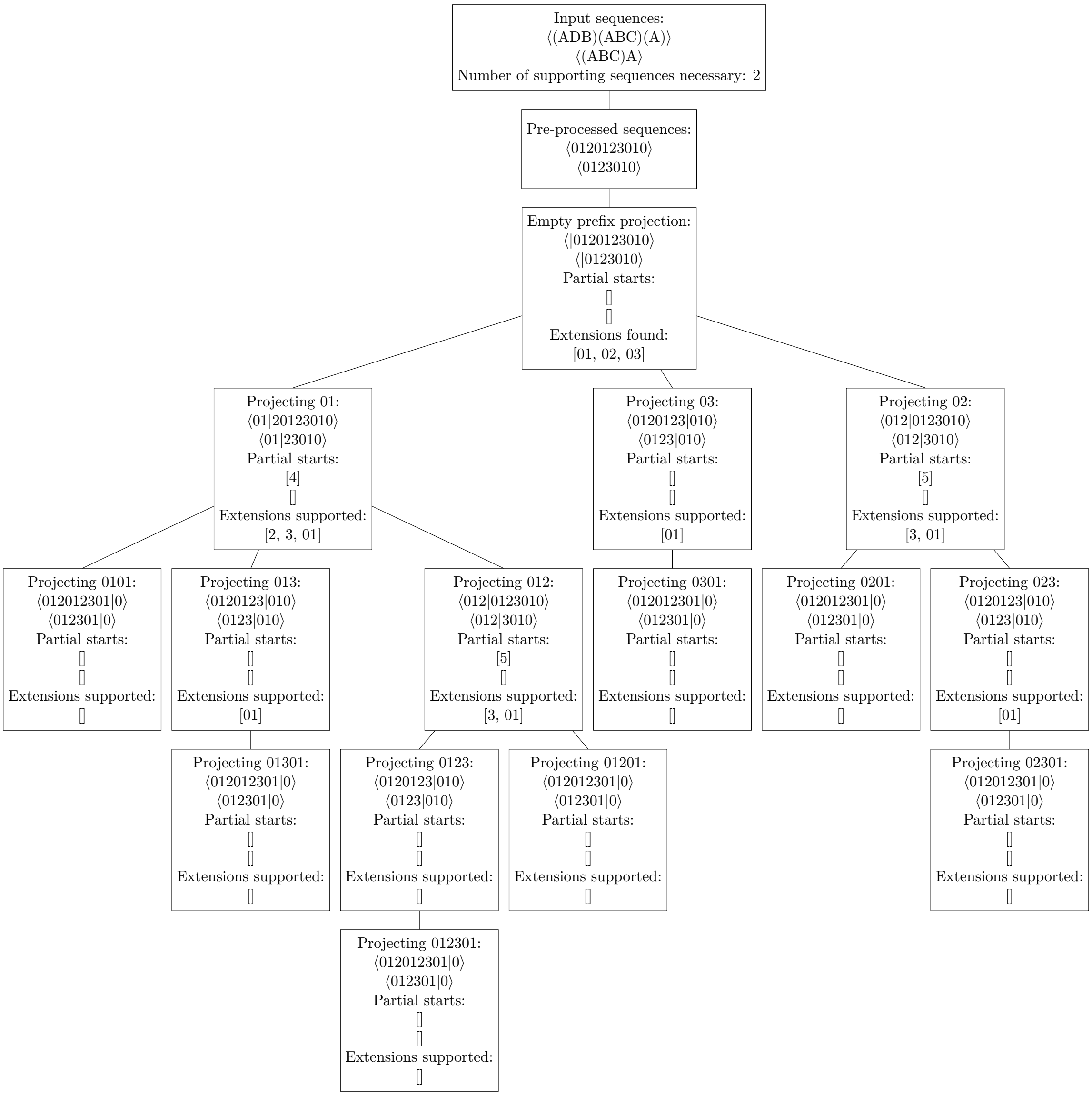


Figure 6: A simple execution of Spark's algorithm.
The solutions are the projected prefixes (except the empty prefix)

NB: During our first analysis of Spark's pre-processing stage, we noticed a small inefficiency in the cleaning of the database's sequences. When multiple item-sets were fully cleaned of their item, the algorithm had a tendency of creating sequences of zero delimiters, as the algorithm still delimited empty item-sets.

Although the internal representation was still correct and results weren't modified, those trailing zeroes substantially slowed the algorithm down. The performance improvement of this small correction is reflected in the annexes, Figure 26.

Later, this small correction was proposed to Spark's community and quickly accepted into the default implementation. In the remains of this paper, we thus consider this corrected version which has been approved by the community as the original algorithm for Spark, and compare our performance improvement with this corrected version as the basis.

3.2 A First Scalable CP Based Implementation

As mentioned earlier in this paper, Spark's original algorithm is composed of two different execution stages. We thus analysed each stage independently to understand how CP technique could improve their implementation. Although we rapidly discovered that the scalable stage could hardly be modified to efficiently incorporate CP techniques, at least not without completely incorporating Spark into the solver, we realised that the local execution stage could be significantly improved.

In fact, the entire local execution could be easily replaced by a CP based algorithm. For SoS problems where PPIC is applicable, we could even use a nearly identical implementation where only the pre-processing stage would need to be modified, as to fit Spark's middle-put.

To remain able to solve SoSS problems through a local execution. A simple boolean was also added, so that users could specify whether PPIC could be used on the input dataset. In case it couldn't be used, the original local execution of Spark would be used.

This first implementation's pseudo-code can be found below, in Algorithm 8.

3.2.1 Improvements Pathways

To improve this first scalable CP based implementation, we identified three promising options that needed to be studied:

1. Improve the link between Spark's middle-put and PPIC's input. The easiest option, but also the most promising, since both algorithms have been separately optimised in terms of performances.
2. Improve Spark's performances by incorporating more ideas from pattern mining. Which would improve both the scalable execution and the original local execution component of Spark. Thus improving performances on both single and multi-item pattern problems.
3. Developing a new version of PPIC which can efficiently be applied to multi-item pattern. Making CP usable for every local execution opportunity and, hopefully, improving performances.

Additionally to those performance improving options, we also decided to prove that PPIC's modularity can be conserved in a scalable environment through the addition of multiple functionalities in our future implementations.

3.3 Adding new functionalities

To first task we undertook was to add four new functionalities to our first CP based Implementation. The implemented functionalities were as follows:

1. Unbounded max pattern length: Although Spark's implementation already disposed of a way to control the maximum length of a pattern, no special value existed to allow unlimited max pattern length. We thus added this minor functionality, specifying 0 as a special value that would allow searching for all solutions pattern of any length. Additionally, we changed the default value of 10 to this new special value, so that all solutions could be found by default.

2. Min pattern length: Although Spark's original implementation allowed to control the maximal length of a pattern. No such functionalities existed to control their minimal length. We thus added a functionality to specify the minimal length a pattern should have before being considered solution. As all patterns containing fewer non-zero items than the specified input wouldn't be outputted, we decided to set the default value of this parameter at 1, so that the returned solutions wouldn't be restricted.

3. Limit on the maximal number of items per item-set: This functionality was added so that users could, once again, better control their outputted results. Supposing an hypothetical business would like to find all sequences of items bought in pairs in a dataset, it would have no need for solutions where item-sets are larger than two. We can thus stop searching for further item-set extensions once the limit has been reached and, de facto, improve our algorithm performances for returning those specific solutions. Additionally, we created a special value (0) so that all item-set of any length could be outputted, and set that special value as our default for this parameter.

4. Soft limit on the number of sub-problems created: By default inactivated, this parameter is enabling far better performance on protein like datasets where projected databases tend to have rather similar sub-problems. Its implementation is a simple check at the beginning of each iteration of the scalable algorithm. If the number of sub-problems created is larger than the user inputted value, the implementation will forcefully put an end to the scalable stage, and switch to a local execution on each worker. For the default value of this parameter, a special value (0) was created so that the number of created sub-problems wouldn't be limited.

   As we can see in Figure 27, performance improvement are only observed on the protein, Kosarak, slen2 and slen3 datasets, but the increase in performance is significant. The problem being that the loss of performance on other datasets is generally even more significant. This loss in performance comes from large differences in sub-problem sizes appearing due to the forced local execution. Since the largest problems tend to be created and processed last, only a few executors have problems to work on toward the end of the execution. The remaining executor staying idle for the reminder of the local execution stage. Moreover, those remaining problems take a very long time to compute, greatly slowing down the measured performance.
   As is, this parameter should be used with extreme care. However, we consecrate an implementation design to improve this improvement's performances later in this paper.

While these additions did not have any measurable impact on performances at their default value, they still allowed better control of the search space. Proving that, although using a CP solver in Spark seriously affected modularity, a certain level could be easily kept from the original CP solver.

A pseudo-code demonstrating the changes brought to the code can be found in Algorithm 9.

We then looked into improving our performances, leading to the discovery of two potential inefficiency.

3.3.1 Quicker - Start

During the pre-processing of Spark's original implementation, unfrequent items are cleaned from the database. Frequent items are thus found in the process, only to be discarded and searched for once more when projecting an empty prefix at the beginning of the scalable execution.

We thus modified our first CP implementation to remove this 'inefficiency', deciding to pass frequent items directly to the scalable stage, instead of discarding them before searching for them once more through another complete iteration over the database.

The code was modified accordingly, as shown in Algorithm 10.

3.3.2 Cleaning Sequence before the Local Execution

The next inefficiency we found was that, during PPIC's local execution, three matrices were built whose size depended on the number of unique symbols in the input databases. Yet, in our first implementation, the various projected Databases that reached the local execution stage often had unfrequent symbols that could potentially be cleaned.

Cleaning them would not only reduce the input database's size, it would also reduce the size of those matrices. Making it a potentially worthwhile deal to pre-process PPIC's input for each projected database of the local execution.

We thus modified our code accordingly, producing the code found in Algorithm 11

Since, as we see later in the performance testing section, large increase in performances can be observed through these two improvements, and since so many other improvements needed to be implemented, **we decided to use this implementation as reference for the remainder of this paper**. All further improvements were thus added separately to this version, since it would later allow us to better compare the performance gains brought by each implementation.

We will thus end this implementation section with a final algorithm regrouping all implementation which showed performance improvements.

3.4 Improving the Switch to a CP Local Execution

In this section, we discuss the improvement we tried to bring to the translation from Spark's middle-put to our local executions input, and the results we obtained. For each improvement, we explain its nature and the trade-off's its implementation may encompass.

3.4.1 Automatic Choice of the Local Execution Algorithm

Our next attempted improvement was to automate the choice between PPIC and Spark's local execution. Using the former only for sequence of symbols problems, and the latter for sequence of sets of symbols problems.

This would allow to be more efficient by default, without needing involvement from the user to decide whether PPIC or Spark's original local execution should be used during the local-execution stage.

In that endeavour, we decided to recalculate the 'maxItemPetItemSet' argument dynamically during the pre-processing stage. Since it allows us to determine whether we were dealing with a SoS or SoSS database, and thus, whether PPIC could be used during the local execution.

Additionally, should 'maxItemPerItemSet' be left to its default value or should it have been put to much too high a value, we could theoretically slightly improve performance by refraining from searching extensions to solution patterns having already reached the max recalculated length. Of course, this theoretical performance gain would only apply for databases where the number of Items per item-set are mostly constant.

But this additional feature would be worth a small loss of performance on SoSS, as long as the usage of PPIC could be guaranteed on SoS problems.

Of course, should the non-recalculated parameters be used with anything but it's default value, the algorithm would make sure only the requested solutions would be computed.

A pseudo-code demonstrating the implementation of this automatic choice can be found in Algorithm 12

3.4.2 Generalised Pre-Processing Before the Local Execution

Extending our previous idea of cleaning the sequence database before PPIC's execution, we decided to create an implementation where the database would be cleaned before any local execution, including SoSS problems. Wondering, if cleaning the sequence database before Spark's original local execution could bring similar improvements.

We thus designed a new cleaning process suited both for Spark and PPIC's local execution input. During this process, we also realised we could also easily check whether the cleaned sequences could be solved using PPIC or Spark. We thus decided to include this feature too, and to compare its performance gain to our previous 'maxItemPetItemSet' based implementation.

While creating this new implementation, we also discovered more than a few inefficiency on the old one. Such as the use of ArrayBuffer structure instead of the much more efficient ArrayBuilder, or un-needed extra iteration during the matrices creation.

We thus expected this new implementation to be more efficient on both types of sequential pattern mining problems and, hopefully, for every dataset.

A pseudo-code representing this implementation can be found in Algorithm 13.

3.5 Improving the Scalable Execution

In this section, we discuss the improvement we tried to bring to Spark's execution stages, and the results we obtained. For each improvement, we first explain its nature and the trade-off's its implementation may encompass.

3.5.1 Position Lists

The first idea we had to improve Spark's performances was to use LAPIN-SPAM's position list to our advantage.

In Spark's original implementation, the scalable and local stages of the algorithm perform their duty in three phases. First they receive a solution prefix which needs to be extended, and project it on the whole database, allowing them to know which sequence support that prefix. Then, in the supporting sequences only, they search for symbols which may extend the prefix. Finally, if some symbols are found and they respect the constraint applied to the solutions, Spark's will save them as solutions and try to extend them further.

While this implementation is very efficient in a scalable environment, we thought it could be improved through the addition of position list. More specifically, during the prefix projection phase, we determined it would improve performance if the algorithm knew earlier when a sequence couldn't possibly hold the currently projected pattern, or if the algorithm didn't have to analyse half of the sequence before starting to project this aforementioned pattern.

Of course, the trade-off would be a more important use of memory, as the positions list would need to be kept on RDD, along-side their corresponding sequences. To adopt this solution, the measured performance improvement would thus need to be important enough to motivate the benefits of the trade-off.

From this idea, we created three new implementations. The first using only a last position list, the second using only a first position list, and the last using both position-lists together.

In Algorithm 14, you will find a pseudo-code of the implementation including both positions lists together. To obtain the two other implementation, simply remove all pieces of code concerned with either the firstPositionList or lastPositionList.

3.5.2 Specialising the Scalable Execution

To improve the scalable execution's performances further, we then had the idea to separate the scalable execution stage of SoS and SoSS problems.

This idea stemming that, for SoS problems, the database's internal representation could be seriously shortened by removing unnecessary separators, effectively reducing the size of their internal representation by two. Furthermore, this more compact representation would allow us to switch twice sooner to the local execution step, as the projected database is now smaller.

We thus implemented a new scalable stage specialised for sequence of symbols problems. Its main features being the absence of spacial start and the much more compact database representation without delimiters.

A trade-off was however made, as we now needed to detect in which type of pattern mining problem we were before starting the execution. Since we now needed to create a different internal representation depending on the database's type, and since simply removing the delimiters after a database type check wouldn't be efficient.

The most efficient way we found to detect the type of problem was during the frequent symbol detection part of the pre-processing step, where through a simple modification we could efficiently detect the type of each sequence, and thus whether we could use the shortened SoS representation on the database.

We thus implemented the code found in Algorithm 15 to successfully put in practise this idea.

3.5.3 Priority Scheduling for Sub-Problems

The final idea we explored to improve Spark's performances, was to modify the order in which sub-problems are computed during the local execution stage.

In the original code, problems are decomposed until they become smaller than a size specified by the 'maxLocalProjDBSize' parameter. As mentioned before, in our reference algorithm, an extension of that idea, the 'subProblemLimit' parameter, was also implemented to allow better control of the number of sub-problems created.

However, we have seen that a consequence of this new functionality is that sub-problems can largely vary in size, making some problem far harder to solve than others. Something which would rarely appear in the original version, unless the maxLocalProjDBSize parameter was put way past its default value. Coincidentally, we also realised that major drops in performance were experienced if those large problems were solved last, since some executor would be left with nothing to do while others would be stuck with catastrophically large workload.

The solution was clear, large problems needed to be solved first in the various executor, so that smaller workload could be shuffled between executor in the later stage of the execution.

3.5.3.1 Analysing sub-problem creation

The piece of code which created the various sub-problems from the various projected prefixes collected during the scalable stage was fundamentally a mapReduce process. The sequences from the original sequence database being projected one by one with different prefixes, then mapped to some reducer, depending on the prefix's ID (see Figure 7).

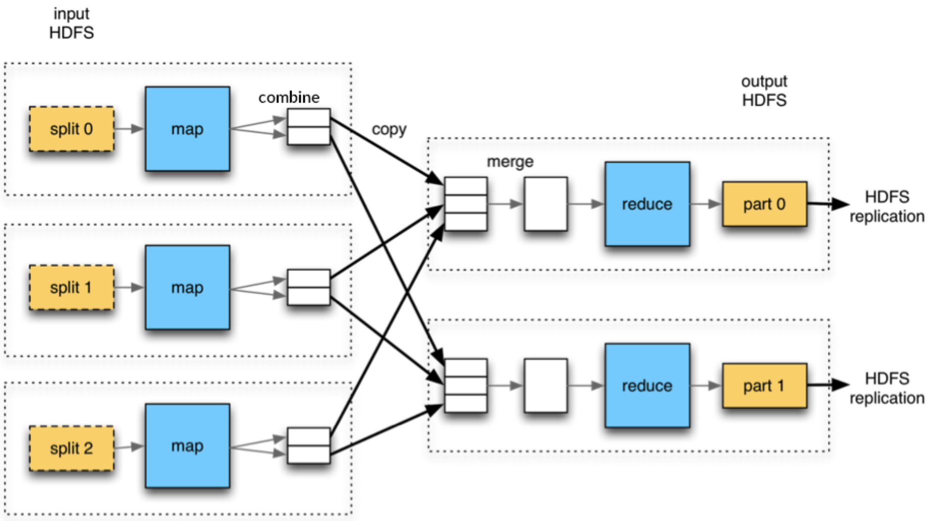


Figure 7: Spark's mapReduce

3.5.3.2 Sort sub-problem on the reducer

We thus concluded that a simple solution could be implemented. According to the specification of Spark's sortBy function, the sorting stage will be exacted locally on each reducer. We thus implemented this quick change by adding a simple sort between our map and reduce stage, obtaining the implementation found in Algorithm 16.

But, as you will see during our performance tests, although this algorithm produced better performance. A memory issue now appeared, to the point of crashing our 10G memory executor.

As it turns out, to sort the various sub-problems depending on their size, Spark's obviously needed to evaluate and simultaneously hold in memory all the sub-problems assigned to an executor. Since those sub-problems were transiently created data, they were not stocked on an RDD, and thus couldn't be stored on disk.

To remain scalable, we thus had to come up with another solution to sort our sub-problems. One that did not involve computing multiple sub-problems while keeping them in memory..

3.5.3.3 Sort sub-problem on the mapper

We thus realised we needed to sort our sub-problems during the mapping phase of the mapReduce process. After a few trial and error, we realised that the mapping function of Spark created sub-problems one by one, following the mapping code and sent them directly to the reducer through the groupBy function, which delivered them in the very same order they were sent.

We thus modified our implementation to change the order in which we created our sub-problem, instead of sorting them afterwards. The created sub-problems would now be mapped to the reducer and computed in the same order that they were mapped.

Meaning that, by mapping the hardest problems first, they would also be executed first.

We then modified our code to sort our prefixes in descending order through their projected database size, a size which was already computed during the scalable stage of our algorithm, and could simply be stored in the prefix until used to sort. This made sorting sub-problems far more efficient, at a small cost in memory.

As expected, sorting a few prefixes used an insignificant amount of memory in comparison to sorting huge RDDs containing complete projected database, while producing equivalent, if not better, performances. As you will see in the dedicated performance testing section.

A pseudo-code based on the implemented changes can be found in Algorithm 17

3.6 CP Based Local Execution for Sequence of Sets of Symbols

Our final implementation attempts were to create a CP-based implementation to solve pattern mining problems involving sets of symbols. Of course, to replace the original local execution, this implementation would need to be more performant than its predecessor.

3.6.1 Pushing PPIC's Ideas Further

Our first attempt at creating such an implementation was to try pushing PPIC's ideas further.

We decided to forgo all three pre-computed matrices, and to change the structure of our sequence database to fill those matrices purposes more efficiently. Instead of an array, we changed each sequence into a map containing unique symbols as key and the various positions of each the respective symbol as value.

Additionally, we represented the sequence's symbol positions list (including delimiter position) by a ReversibleArrayStack structure, thus allowing efficient backtracking of a symbol's remaining position through trailing.

This new structure's purpose was to allow us to make distant jumps and checks more efficiently, at the cost of a slightly higher memory consumption. Theoretically, finding the next position of a given symbol would be O(1), be this next, last or first position of the symbol. Should a symbol's position list be empty, or should the last recorded position be smaller than our current position in the sequence, we would also immediately know that no more occurrences of said symbol were contained in the remains of the sequence.

The trade-off lied in the number of reversible points that needed to be maintained. With one ReversibleArrayStack per symbol for each sequence. This number could grow quite quickly, the results may thus greatly vary between datasets but we had good faith it the performance tests would yield satisfying results.

Translating our improvement to pseudo-code, we obtain Algorithm 18.

3.6.2 Adding Partial Projections to PPIC

We also decided to try another approach at developing an efficient CP based implementation for problems involving sequences of sets of symbols. This second attempt focusing on bringing the partialStart structure of Spark in a PPIC-like algorithm.

First we realised that keeping all three matrices wouldn't be efficient. In a set of symbols pattern mining context. To keep the same function, the 'interesting position' matrix needed to be modified to indicate the next last appearance of an item in an itemset, instead of the next last appearance of an item in the sequence, as it did before. This in turn, makes this matrix useless as the next such position would nearly always be the next item of the database.

We thus decided to remove this matrix, but to keep the first and last position lists, as they remained relevant in a sequence of sets of symbols context. We also modified our implementation to keep zero delimiters during cleaning and to change the partial start received from Spark during the pre-processing, so that they still referred to the same item-set after cleaning (lest the item-set completely disappear in which case that particular partial start will be scraped).

The disadvantage of partial starts, however, was that they need to be maintained and advanced at each step of our DFS execution. This meant that, when starting a new item-set, all remaining sequences of the database would need to be fully explored.

For item-set extensions, however, only the position list as partial starts would need to be checked, along the sub-sequent positions appearing before the next separator. We thus expect greater performance improvement the longer item-sets could be extended in the database.

A pseudo-code based on the implemented changes can be found in Algorithm 19.

3.7 Final implementation

Our final implementation regroups the characteristics of all performance benefiting algorithm presented above. In this section, we first present our design choices for this final implementation as well as motivate those choices. We then present a new functionality added exclusively to this implementation.

A pseudo-code representing this final implementation may be found in Algorithm 1

3.7.1 Reaching Greater Performances

The execution proceeds as follows: First, unfrequent symbols are cleaned from the original database. During the detection process, the algorithm also takes time to detect whether we are dealing with a database of sequences of symbols, or with a database of sequences of sets of symbols.

This allows us to keep the specialised scalable execution improvement which brought great performance improvements to SoS database mining problems.

After that, the scalable execution proceeds normally with the new functionalities added in the reference algorithm. We however removed the quicker-start 'improvement' made before the reference implementation, as it brought negative performance improvements (See 'Performances of Quicker-Start' section for details on its negative performances).

During the scalable execution, the algorithm will store small prefixes for the local execution stages, alongside the size of their projected database. So that, should it be necessary, sub-problems may be sorted before the beginning of the local execution stage.

As condition for triggering an automatic sort of the sub-problem, we decided to use the presence of a non-default value for the 'subProblemSoftLimit' parameter (> 1) or , depending on the type, either a 'maxLocalProjectedDBsize' value twice greater than the default of 32000000 for sequences of sets of symbols, or a 'maxLocalProjectedDBsize' greater or equal to 32000000 for sequences of symbols .

The reason we settled on those conditions is quite straightforward. As you may see in the performance test of our implementation 'Sorting sub-problems on the mapper', sorting may not always be beneficial when sorting sub-problems created with the default-value of 'maxLocal-ProjectedDBsize'. We thus decided to aim a bit higher, and settled on 64000000, where sorting always shows benefits.

Considering the condition on our 'subProblemSoftLimit' parameter, the choice was even simpler, since this parameter is meant to be used for showing the benefits of the local-execution stage, and since performance results may become hazardous when sub-problems aren't sorted properly.

Finally, before each local execution stage, the projected databases will be cleaned similarly to our 'Generalised Pre-Processing Before the Local Execution' improvement. Of course, with a few minor differences as the sequences representation now differs depending on the database type.

An algorithm will then be chosen to solve the sequential pattern mining sub-problem locally. We settled on using PPIC for problems involving sequences of symbols, and our 'Adding Partial Projections to PPIC' implementation for problems involving sequences of sets of symbols, since it gave far better performances on the slen3 dataset than any of our position list implementation and it allows greater modularity on the constraint that can be applied to our algorithm.

Of course, should a problem involving sequences of sets of symbols become a problem involving only sequences of symbols, an automatic switch to PPIC's local execution would be triggered. Ensuring the most performant algorithm is always in use.

3.7.2 A New Functionality

Additionally to our performance related design choices, we also decided to add a new functionality to this final implementation, allowing a user to set precise constraint on the occurrences of symbols in solutions pattern.

In our final implementation, a user may thus input a list of tuples of the form (symbol, relation, count) allowing precise control of the solution patterns returned.

For example, by inputting the tuple ('A', <=, 1), a user can ensure that symbol A won't occur more than once in a solution pattern. If the user inputted tuples allow to rule out the presence of a symbol in solution patterns, for example with a tuple such as ('X', <, 1), our algorithm will automatically brand those items as unfrequent and clean them during the pre-processing stage. As they have no chance of appearing in a solution anyway.

The supported relations are '==', '!=', '<', '>', '>=', and '<='. Thus allowing a wild range of control over the solution pattern returned. Of course, if the user doesn't impose any symbol constraint, the execution proceeds correctly and doesn't restrict the solution space.

As before, this new functionality was also designed to affect performance as little as possible when it is not in use. By default, no constraint are applied on the occurrences of symbols.

Algorithm 1: Final implementation

```
Function scalableExecution(database, minPatLen, maxPatLen, maxLocalSize, $\theta$,
limitItemPerItemset, softSubProblemLimit, freqItemAndCount, symbolConstraints):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.  For SoS
       database, use the shortened database representation for the
       PostFixes.                                                          */
    postfixes = $\emptyset$
    if database.type() == SDDB then
        database.map(seq => PostFixSSoS(seq))
    else
        database.map(seq => PostFixSoS(seq))
    end
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = [Prefix.empty]
    // Start scalable execution
    while largePrefixes is not empty do
        // Project and extend prefix through a map-reduce
        /* When storing prefixes to the local execution, the algorithm also
           stores their projected database size, so that they can be sorted
           accurately later.                                               */
        largePrefixes = projectAndExtend(largePrefixes)
    end
    if len(smallPrefixes) > 0 then
        // For each prefix, project the whole database and solve locally
        // Sorting sub-problems is done only when necessary.
        toExecuteLocaly = postifixes.flatMap( postfix =>
         smallPrefixes.SortByDBSizeIfNecessary().flatMap( prefix =>
            (prefix.ID, postfix.project(prefix).compress())
        )).groupByKey().flatMap(sequences =>
        // Use argument to determine which local execution to use
        sequences = cleanUnFrequentItemsAndRename(sequences)
        matrices = generatePPICMatrices()
        if sequences.type() == SSDB then
            solutionPatterns ++= PPICwithPartialProjections(sequences, maxPatLen,
              minPatLen, maxItemPerItemSet, $\theta$, matrices, prefix, symbolConstraints)
        else
            solutionPatterns ++= PPIC(sequences, maxPatLen, minPatLen,
              maxItemPerItemSet, $\theta$, matrices, prefix, symbolConstraints)
        end
        )
    end
    return solutionPatterns
end
```

4 Performances

To compare the performances of our various implementations, we first need to discuss how those performances were measured, and on which dataset they were measured.

4.1 Datasets

For our performance tests, eight datasets were chosen for the different characteristic they displayed. The goal being to prove the efficiency of the developed algorithm in a wide range of situations. The chosen datasets and their characteristic are displayed in Table 1:

	Dataset	#SDB	N	avg(#S)	avg(#Ns)	max(#S)	Sparsity	description
1.	BIBLE	36369	13905	21.64	17.85	100	1.18	text
	FIFA	20450	2990	36.24	34.74	100	1.19	web click stream
	Kosarak-70	69999	21144	7.98	7.98	796	1.0	web click stream
	LEVIATHAN	5834	9025	33.81	26.34	100	1.25	text
	PubMed	17237	19931	29.56	24.82	198	1.17	bio-medical text
	protein	103120	25	482.25	19.93	600	24.21	protein sequences
2.	slen1	50350	41911	13.24	13.24	60	1.0	generated dataset
	slen2	47555	62296	17.97	17.97	74	1.0	generated dataset
	slen3	287676	81368	17.07	17.07	85	1.0	generated dataset

Table 1: Datasets features. The datasets of category 1 contain only SoS, while the datasets of category 2 contain only SoSS
- #SDB = number of sequences;
- N = Number of different symbols in the dataset;
- #S = Length of a sequence S;
- #Ns = Number of different symbols in a sequence S;
- Sparsity = $\frac{1}{\#SDB} * \sum \frac{\#S}{\#Ns}$

As we can see, our datasets vary largely in their characteristic, be it in their sparsity or in the size of their set of symbols. Although the selected datasets focus slightly more on sequence of symbols, since it has been our focus in most of our developed improvement, we also made sure to correctly represent sequence of sets of symbols with our last three datasets.

4.2 Number of Partitions

As seen previously in Figure 5, Spark's performances may vary greatly depending on the number of partitions created from the Input dataset.

The number of created partition must thus be carefully selected as, should there be too many partitions created, the algorithm will lose considerable amount of time switching its execution between partitions. Also, should there be too little partition, scalability may be affected, as executor will need to hold large partitions in memory. In extreme cases, Spark's shuffler may even crash while shuffling partitions around.

We have thus decided to keep a constant, carefully selected, number of partitions for each dataset. So that our algorithm's performances can remain comparable while guaranteeing a complete absence of shuffler crash due to too large partitions.

The number of partitions created for each dataset are displayed in Table 2. As we can see, the number of created partition is closely related to the database's size.

Dataset	File size (Ko)	Number of partitions
BIBLE	3065	250
FIFA	2594	300
Kosarak-70	2166	250
LEVIATHAN	713	100
PubMed	1646	200
protein	126046	5000
slen1	5400	500
slen2	6896	500
slen3	39654	1000

Table 2: Number of partitions used during performance tests

4.3 Performance Testing Procedure

4.3.1 Distribution Choice & Cluster Architecture

Our performances will be tested running different custom distribution of Spark, compiled independently for each specific implementation. The goal being to test the performances of an actual distribution containing our changes, rather than simply running the program on an existing distribution.

Although this will take us longer to obtain results, the goal behind this choice, is that our code could then easily be proposed as the standard for future distributions of Spark.

Additionally to that, unless specified otherwise, our algorithm will be executed on Spark's standalone cluster, in cluster deploy mode (not locally). More specifically, during our tests, we run a simple architecture composed of a single master and of a worker with four executors, each executor running three threads. A representation of this simple architecture is available in Figure 8.

Both the driver and the executors will also dispose of a large amount of memory (10G each) during our tests. As it serves no purpose to limit their abilities when purely comparing performances.

Later in our paper, scalable performances will also be tested in a memory-restricted architecture. The architecture then used shall be specified at the beginning of the concerned section.

4.3.2 Program Parameters

At all time, unless specified otherwise, the program parameters will be kept at their default value, so that differences in parameters will never be reflected in our performance comparisons.

An exception to that rule being the 'maxPatternLength' parameter of Spark, which its default value of 10 doesn't fit our purpose as it prevents all sequences to be found on our largest datasets. We will thus use INTEGER.maxValue instead, so that every solution pattern from our datasets may be outputted.

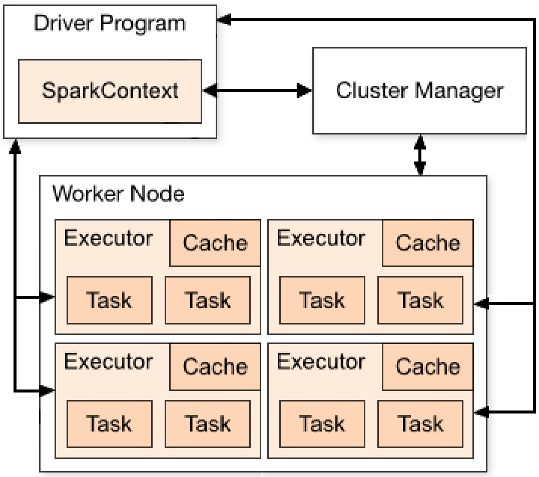


Figure 8: The simple architecture used during the majority of our tests

Also, as mentioned previously, all additional functionalities implemented have been designed to affect performance as little as possible when left to their default value. Their default value will thus similarly be used, unless specified otherwise before the performance test.

4.3.3 Measurement Span

For our performance tests, we measure not only the running time of our algorithm, but also its pre-processing and dataset loading performances. The aim being to develop a new implementation that entirely surpasses the old one.

4.4 Testing the Implementations

4.4.1 Comparing Spark and PPIC's original implementations

Since PPIC is neither scalable nor concurrent, we exceptionally used an even simpler architecture for our test on Spark, restraining our worker to a single executor so that both implementation can work under equivalent resources.

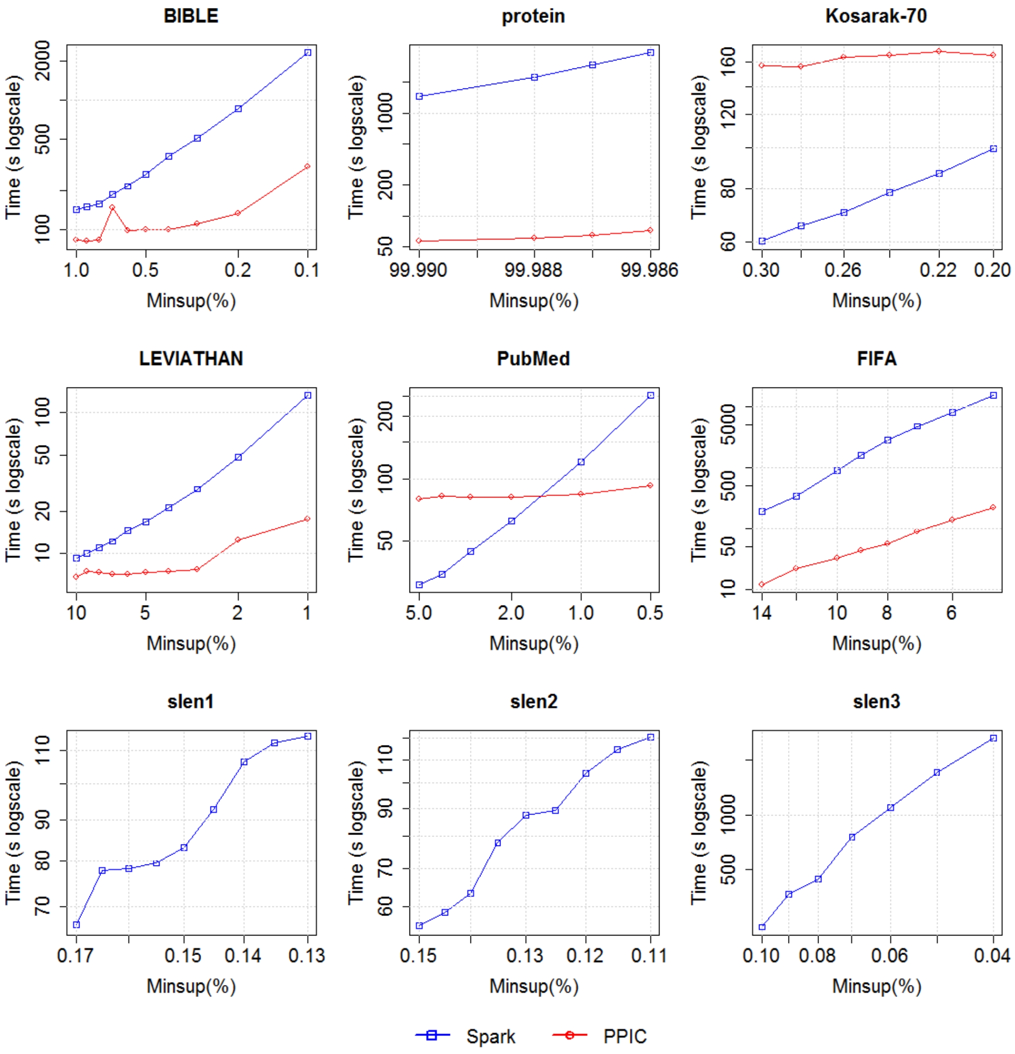
As we can see in Figure 9, PPIC greatly overcomes Spark on nearly every dataset given the same resources, the only exceptions being the Kosarak-70 and PubMed dataset.

We can however notice that, on those two datasets, PPIC's performances are extremely stable. As we can see in the performance tests involving lower amount of supporting sequence on PubMed, PPIC becomes the most efficient in the long run. We have no doubt that a similar situation would happen for Kosarak-70, should we have been PPIC the opportunity in our tests.

As said previously, the reason behind those stable performance lies in the required pre-processing of PPIC being quite long. The small difference we can see in those stable performances are thus the running time of the main algorithm itself.

4.4.2 Performances of our First CP Based Implementation

As we can see in figure 10, this first implementation, which simple merges the two algorithms by replacing Spark's local execution by PPIC, is already much more efficient on sequence of symbols problems than Spark's original algorithm. We can however notice than the single-thread execution of PPIC sometimes overcome our new scalable implementation, despite the lower



⚠ Test realised on a simplified architecture ⚠

PPIC: 1 thread with 10G memory
Spark: 1 driver + 1 worker with 1 executor. 10G memory each

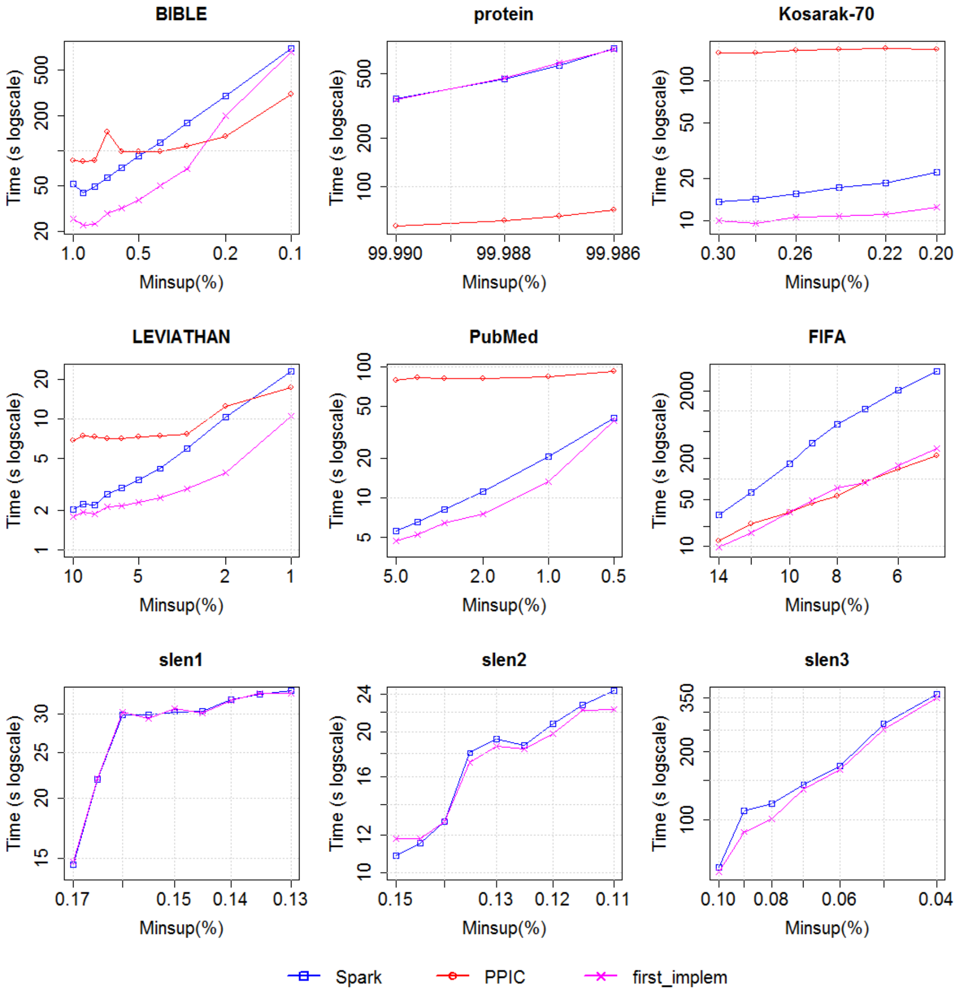
Figure 9: Original performances of PPIC and Spark

amount of resources at his disposal. The worst such case being protein, a dataset on which PPIC remain far superior despite the lower resource it disposes, which may appear disconcerting at first glance.

Another disconcerting thing to notice may be the relative flatness of PPIC's performance measurement. Making it seem as if the decrease in the amount of required support isn't really felt by the algorithm. On Kosarak and PubMed, the performances of PPIC remains worse by a 10x factor which cannot be explained by the lack of resources alone. Similarly, on protein, the performance of our new algorithm remain worse by a 10x factor, despite the additional resources given to its execution.

However, those oddities in our performance measurements are perfectly explainable, they actually result from three major factors:

1. First, the pre-processing of the original PPIC algorithm wasn't implemented efficiently,



⚠ PPIC's performances were still measured using a single thread and 10G of memory ⚠
Other performances represented in this graph were measured using the 4-executor test architecture previously described.
As we can see, when given more resources, Spark's performances slowly start to overcome PPIC's performances on a majority of our datasets.

Figure 10: A first implementation that makes PPIC scalable

probably since only the execution time had been compared on PPIC's original paper. PPIC's true efficiency was thus brought down, while our new implementation wasn't, since this pre-processing was rewritten from scratch to fit Spark's middle-put. This time, with efficiency in mind. Explaining the slow, yet stable, performances of PPIC on datasets like Kosarak and PubMed.

2. Second, as explained earlier, three matrices are built from the original sequence database before the beginning of PPIC's execution. However, in PPIC's original implementation, those matrices could be very large. As their size depends not only on the number of sequences, but also on the number of unique items present in the input database.

   However, our first-implementation surprisingly partly solved this problem. Since only the interesting parts of the sequence database are kept before launching the local execution, many items and sequences which appeared in the initial problem don't appear during the local execution. For each sub-problem treated in the local execution, the constructed matrices will thus be much smaller in size. Which explains further our new implementation's efficiency gain on datasets with large amounts of distinct items.

3. Finally, our third major factor is a major inefficiency that comes as a by-product of scalability. It appears mostly on datasets composed of a small number of repeated symbols, such as our protein dataset. In those cases, since the various sub-problems created before our local execution are actually very similar to each other, our new implementation will lose an important amount of time recreating the three input matrices before launching computations on those nearly identical sequence databases. While the original PPIC implementation would create those matrices once and use them through the reminder of the execution.

   Sadly, this problem cannot be dynamically fixed without seriously affecting scalability or efficiency, as it would require us to compare the projection of multiple prefixes. This means that, either we would have to project the prefixes multiple time to obtain the results of those comparisons, either we would project it once and keep multiple version of the database during comparison, which would be a disaster for memory consumption and scalability.

   Fortunately, although a complete dynamic fix of the problem is impractical for the implementation of a scalable and efficient solution. It is possible to give users the possibility to reduce or even negate the effects of this by-product inefficiency, through giving them control over the amount of sub-problem created. Since the less such sub-problems, the less those matrices will need to be recalculated.

   In fact, Spark's implementation already contains a way to control the number of sub-problems created, thanks to the 'maxLocalProjDBSize' parameter. Stopping further sub-problem creation from problems that are already below the inputted parameter value in size. A more direct and precise way to control the amount of generated problems was thus added among other functionalities in the Quicker-Start implementation.

On the other hand, we can notice that multi-item performance weren't impacted at all. Which is normal since the modification made only concerned single-item problems.

4.4.3 Performances of Quicker-Start

As we can see, very small gain in time are generally made. With the exception of the Kosarak-70, slen2 and slen3 datasets, which are our datasets possessing the largest number of unique items after cleaning the database. At first, we believed those small losses of performances were due to the delay needed to transfer the already calculated frequent items to the various executor being larger than the time needed to actually recalculate those items from the small cleaned database. It was only much later, when we added the slen3 dataset, that we realised this interpretation was incomplete.

In the original implementation, since the items need to be recalculated, they will be recalculated only on the sequences contained in the partitions of the executor. In this new version, we force every executor to receive those items and to store prefixes object for them. Those objects then created will then be kept on the executor for the entirety of the execution as, should an RDD need to be recalculated, they would be a necessary input for their recreation.

Meaning that, when the number of items are large, this additional feature will require more memory usage, and more computation time to receive the items and their frequencies and create

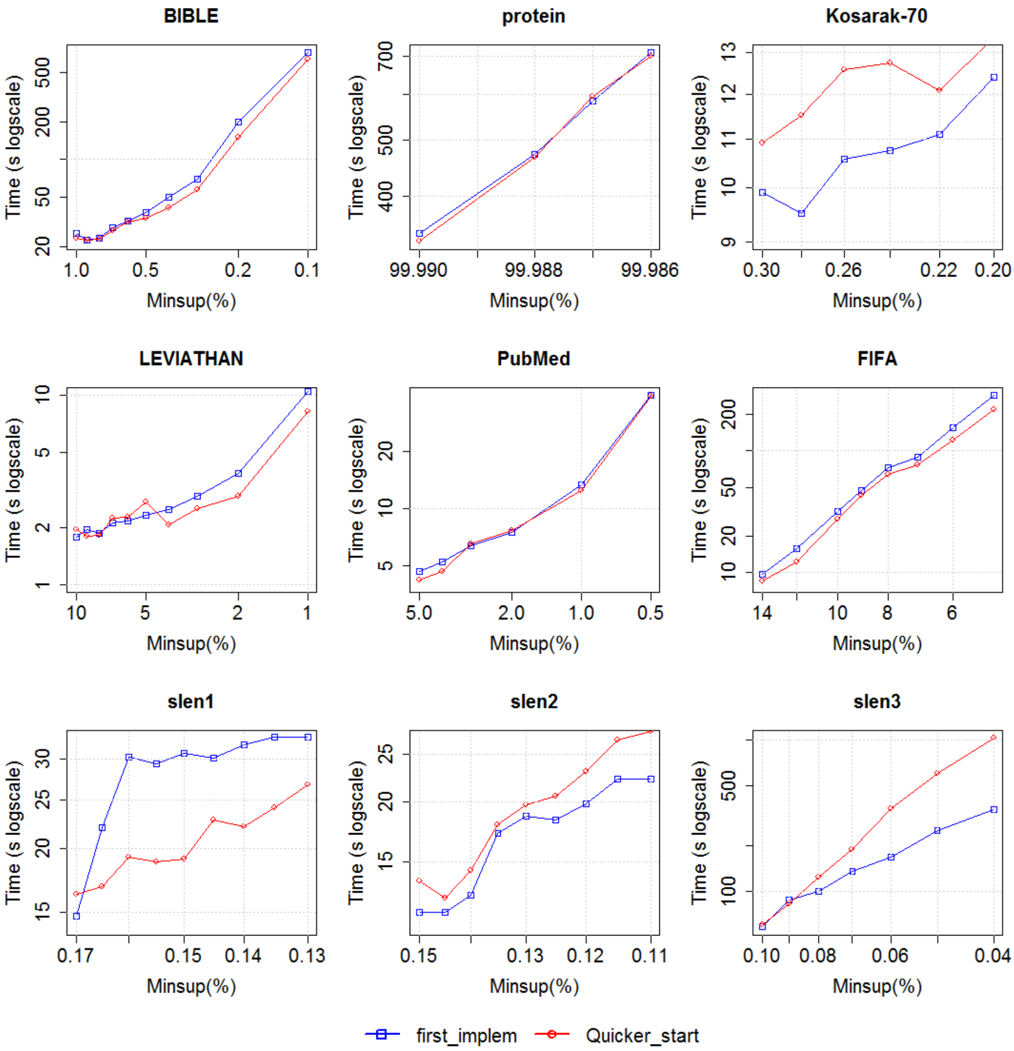


Figure 11: The performance impact of not recalculating frequent items

the prefixes. Especially when, as in our slen3 dataset which is a combination of multiple generated datasets, every partition generally concerns a whole different set of items.

Sadly, we realised this mistake much too late, as only the slen3 dataset proved it's quite disastrous implications and, at the time, we had decided to keep this improvement despite the small inefficiency on Kosarak and slen2. Since other datasets displayed rather interesting efficiency gain at the time. This inefficiency was thus introduced in our reference algorithm, and in nearly all the subsequent implementation.

We will however correct this mistake in our final version of the algorithm, and remove the quicker-start functionality, since the change is harmful to the scalability of the program.

4.4.4 Performances of Adding Pre-processing Before PPIC's Local Execution

As we can see in figure 12, this additional pre-processing step allows a 3x performance speed-up on the Bible dataset and a 2x speed-up on Fifa. Lesser performance gains were also observed on protein and the latter stages of PubMed.

However, we can observe worsening performances in smaller datasets and in executions

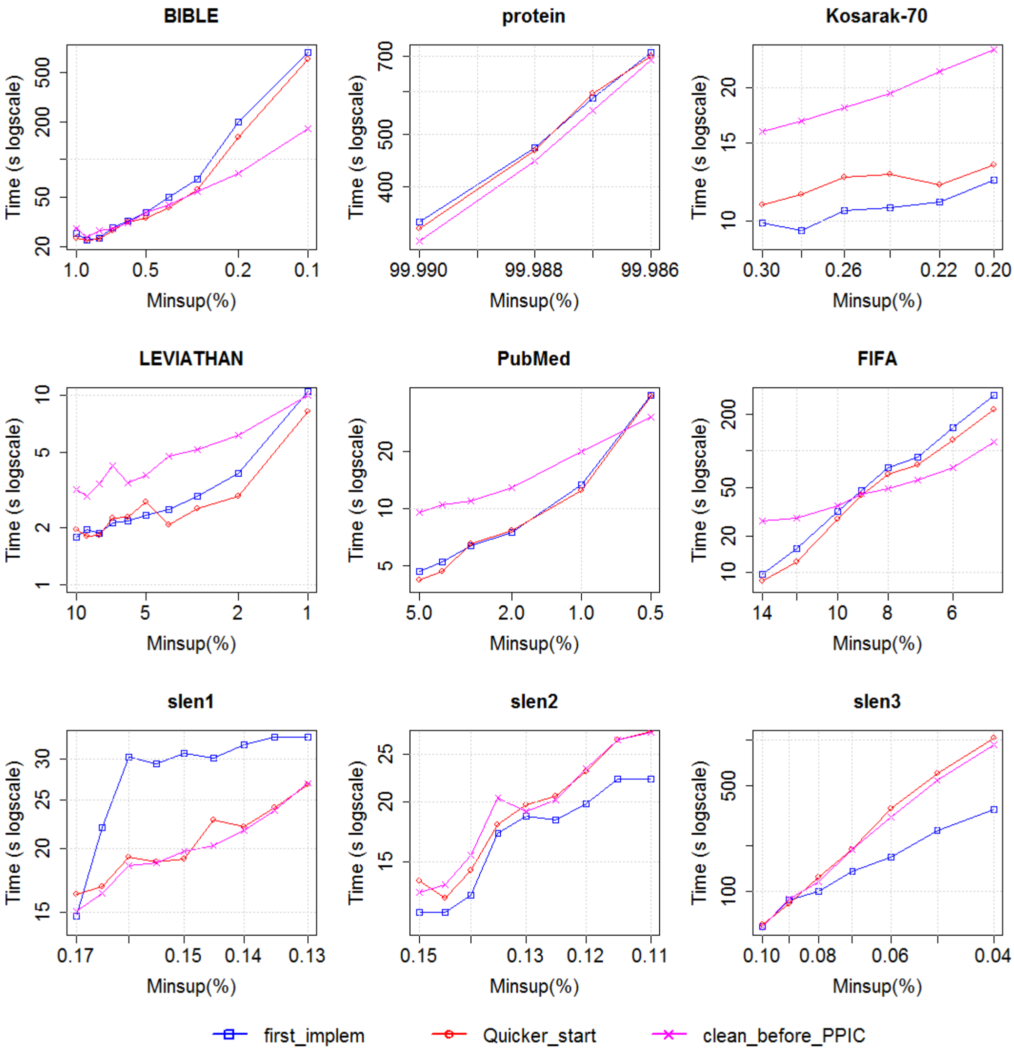


Figure 12: The performance impact of Database Pre-processing Before PPIC's Execution

with larger minsup values. Theses losses however, pales in comparison to the performance gain observed. Although at first glance they may seem large on log-scale graph, they do not even exceed ten second, while the performance gains can be counted in hundreds of seconds on Bible and Fifa.

The reason being those performance gains is that the larger the dataset and the lower the minimal amount of support, the longer prefixes will need to be before being executed locally. The larger those prefixes are before local execution, the more items can be cleaned from the sequences, the faster we can search the remains of those very sequences for patterns. Finally, if fewer items are present in our sequences during our local execution of PPIC, the three matrices needed for PPIC's execution can be created faster. Thus, hastening performance another fold.

However, cleaning sequences comes with a variable cost depending on the database size. If, much like Kosarak, the database only contains uncleanable sequences, performances will be slightly worsened. Of course, since this change is only applied before PPIC, the performance measured on our three slen datasets weren't affected.

In the end, since such large increase in performances were observed, and since so many other improvements needed to be tested, **we decided to use the performance obtained here as reference for the remainder of this paper**. To allow better comparison of performances gains/losses, all further improvements were thus added to this version, and tested separately. Finally, by the end of this paper, a final version containing all changes providing performance improvements will be compiled into a single algorithm, algorithm whose performances will be similarly tested.

4.4.5 Performances with Automatic choice of Local Execution

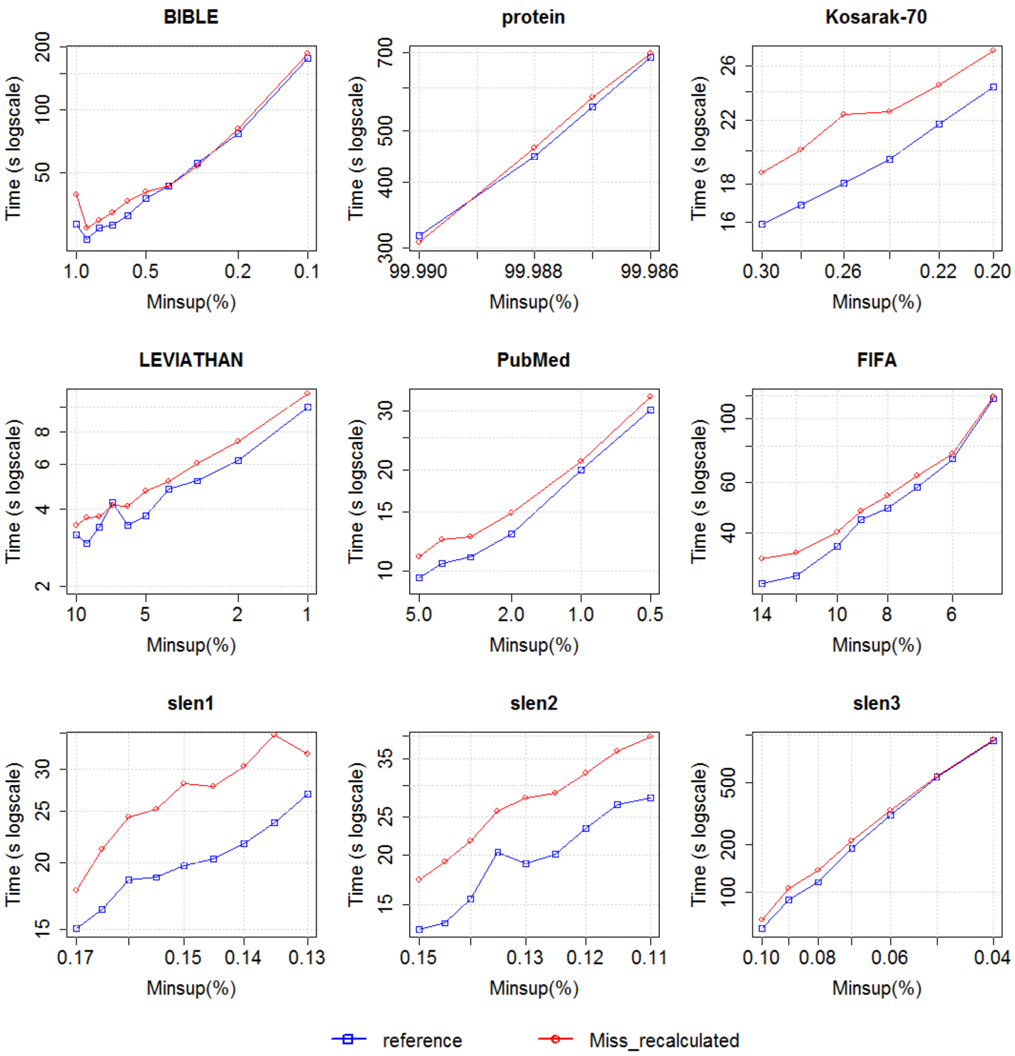


Figure 13: Performance improvement of recalculating 'maxItemPerItemset' to avoid search for multi-item extension when unnecessary

As we can see on figure 13, this improvement achieves its goal with a slight performance degradation, due to the need for calculating the 'maxItemPetItemSet' value.

While, at first, we thought those small losses were worth this additional feature, as it guarantees the use of the much more performant PPIC. We, however, found soon after that there was a more efficient way to achieve the same goal, as you will see in the next section.

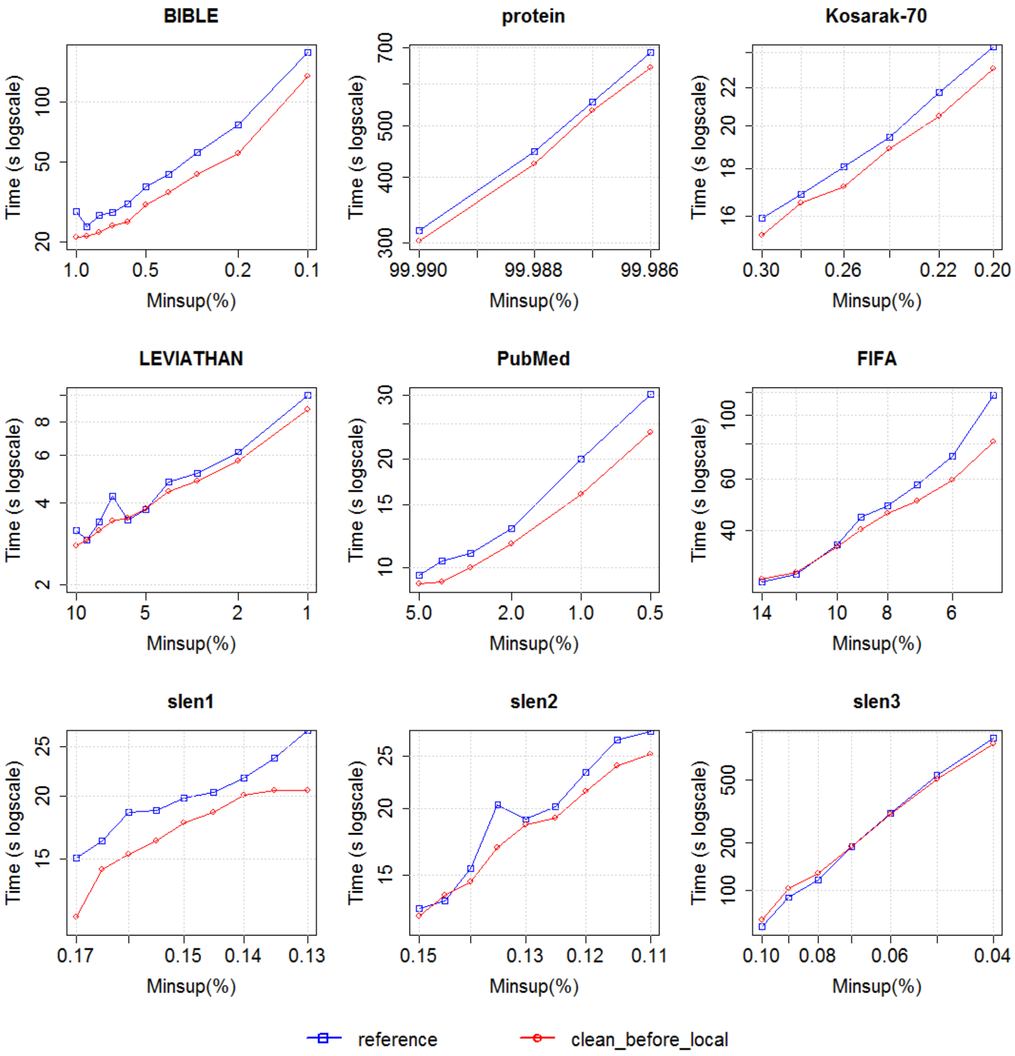


Figure 14: Efficiency gains of cleaning the sequence database before any local execution

4.4.6 Performances of Adding Pre-processing Before Any Local Execution

As we can see in figure 14, this implementation was, as expected, much more performant, for both types of datasets.

Advantages are, however, lesser when mining sequences of sets of symbols, as the local execution still relies on Spark's implementation, which doesn't take as much advantages of this additional cleaning step. The various improvement we made for cleaning sequences of symbols, on the other hand, are quite strongly reflected in these new performance measurement, surpassing our expectation on their effectiveness.

While, as said earlier, cleaning before the local execution allows us to detect witch algorithm should take charge the inputted database, another advantage also appeared. Should a sub-problem from a database of sequence of sets of symbols become a database of sequence of symbols through cleaning, its local execution would be switched through PPIC, thus improving the significantly the performance on these projected database.

Sadly those cases should only appear rarely, and depend strongly on the inputted data. Still, it remains a nice addition to have, and which give occasional small boost to performances.

4.4.7 Performances with Position lists

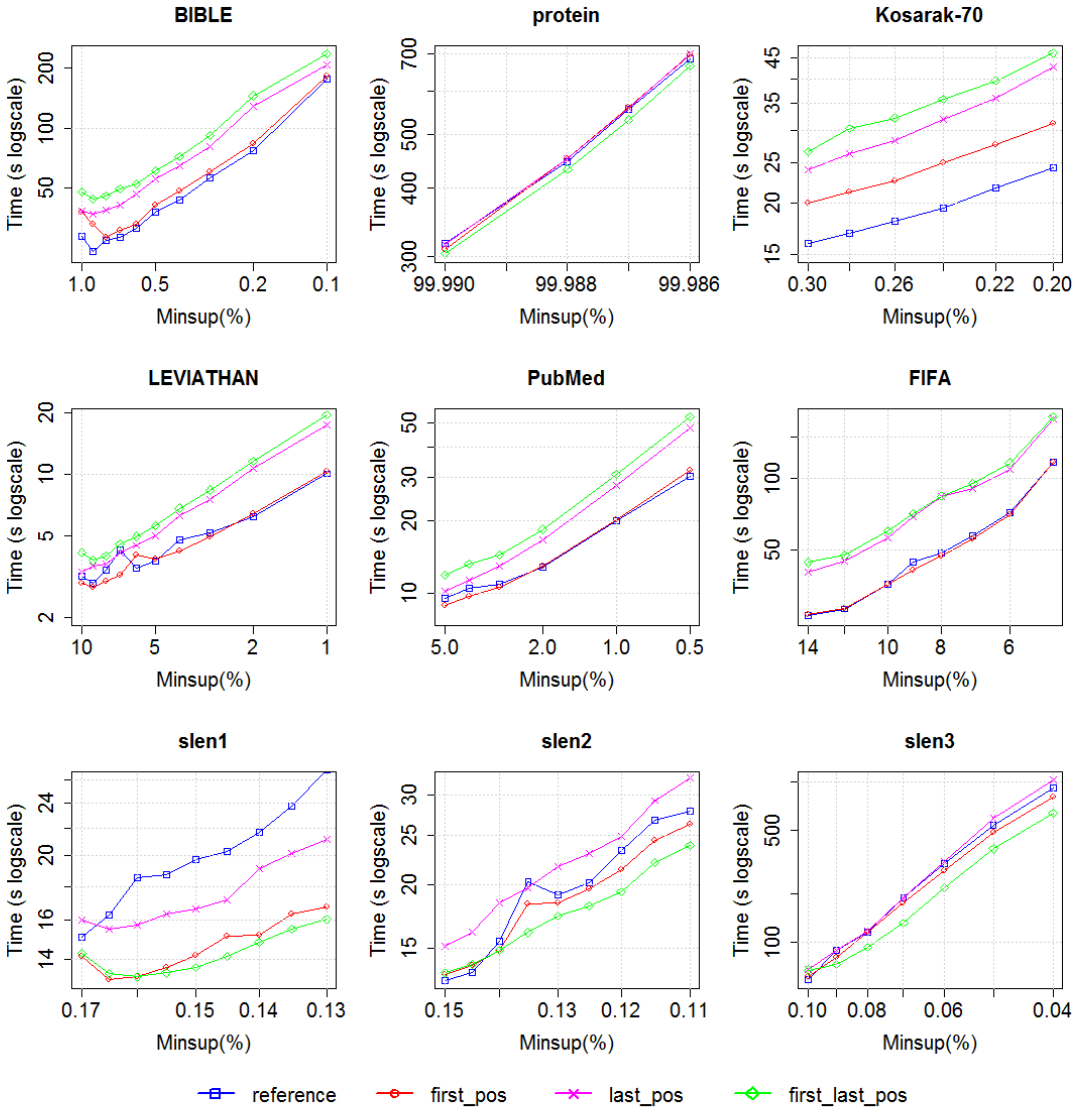


Figure 15: Performance gain of first/last position lists

As we can see in Figure 15, the resulting performances of those three implementations weren't conclusive. On every database of sequence of symbols but protein, which is a very sparse dataset on which this type of techniques excel, the modified algorithms delivered worse performances than our reference algorithm.

Those worsening performance appears because the benefit of those additional computation simply do not have enough time to appear. After being calculated during the pre-processing step, they only stay of use during the scalable execution of our algorithm. Since that, before the local execution, the sequence database will be compressed and the positions list will need to be recalculated.

Another reason for those worsening performance would be the transfer time of the calculated positions list, which can be very large depending on the number of frequent items. The delay imposed by their calculation and their transfer through the executors are thus far from negligible.

However, while suffering from the same afflictions, an improvement in performance can be observed on databases of sequences of sets of symbols, where those techniques weren't already in use during the local execution stage. Especially on the slen1 dataset, where the measured performances improve significantly.

Despite the exception of the slen1 dataset, however, the measured performances gain does not justify a nearly doubled memory consumption (nearly tripled in the case of the 'first_last_pos' improvement). Should no better implementation be found, this improvement should thus be restrained to only appear in Spark's local execution, where additional memory consumption won't affect data transfer times. Although the measured performances improvement would thus be reduced, keeping such large RDD during the scalable execution step simply wouldn't make sense for so little gains.

4.4.8 Performances with Specialised scalable execution

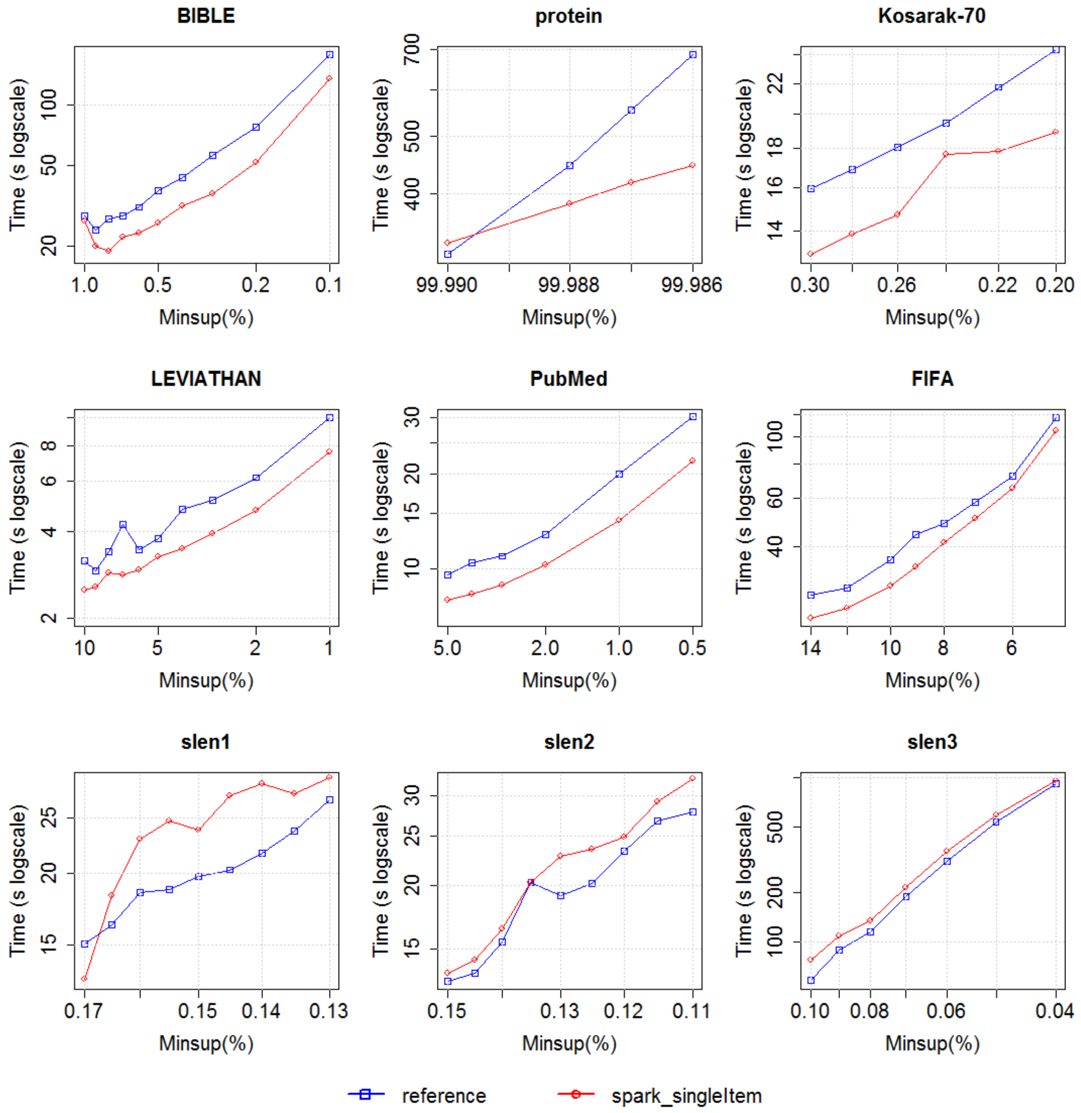


Figure 16: Performance improvement of specialising spark's scalable stage.

As we can see on the slen datasets of Figure 16, this new implementation introduced a small loss in performance for dataset of sequence of sets of symbols. As it needs to go through the added detection but doesn't profit of the advantages.

However, for sequences of symbols, performances improved greatly. Not only that, but the new internal database representation make much more sense from a memory-consumption point of view.

Thus, we judged these great advantages to be worth the slight loss of performance, and decided to include this improvement in our final implementation.

4.4.9 Performances of Using Priority Scheduling for the Local Execution

4.4.9.1 Performances of Sorting Sub-Problems on the Reducer

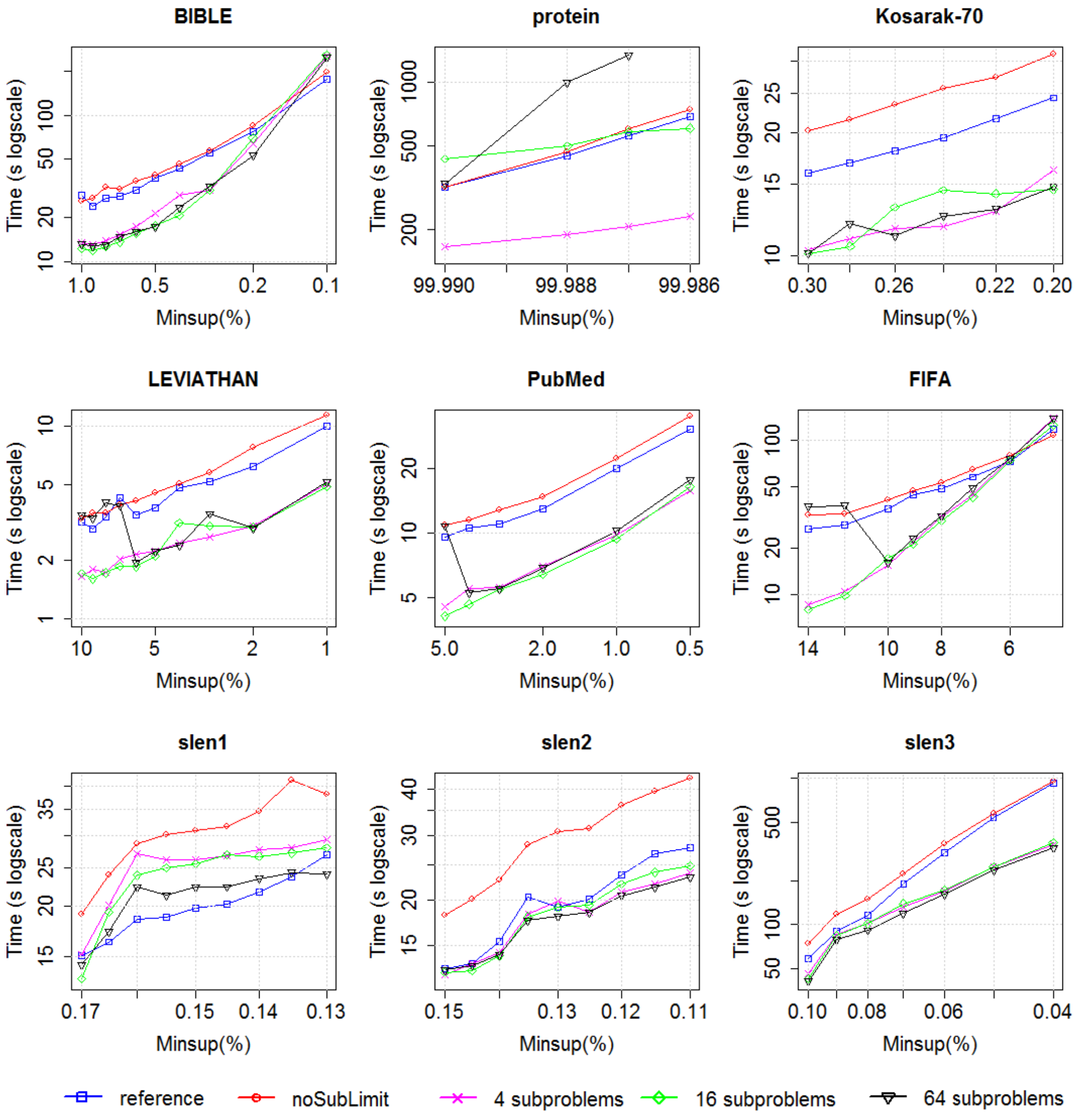


Figure 17: Performance of a 'biggest problem first' scheduling executed through the default sortBy() function of Spark.

As we can see in Figure 17, our first try at this implementation, impressive performance improvements can be observed, especially for the Kosarak, protein, slen2 and slen3 datasets. While BIBLE, FIFA and slen1 suffer from slight performance losses in their later stages, the damage is rather limited in comparison to the otherwise gain. Applying the sort function to the algorithm when unnecessary also isn't too damaging even when sub-problems aren't limited, as we can see with the red line, it should however be avoided in the final version. The sort function should only be called when relevant.

At first glance, this thus looks like a nice little improvement. However, as explained in the implementation section, a huge problem remained. As we can see on protein's measurements, the black line lacks its fourth point. The reason being a constant crash of the algorithm due to a lack of memory ! (The reasons behind this failure is explained in the corresponding implementation section)

Thus motivating our second attempt at implementing this improvement, this time by sorting the problem on the mapper rather than the reducer.

4.4.9.2 Performances of Sorting Sub-Problems on the Mapper

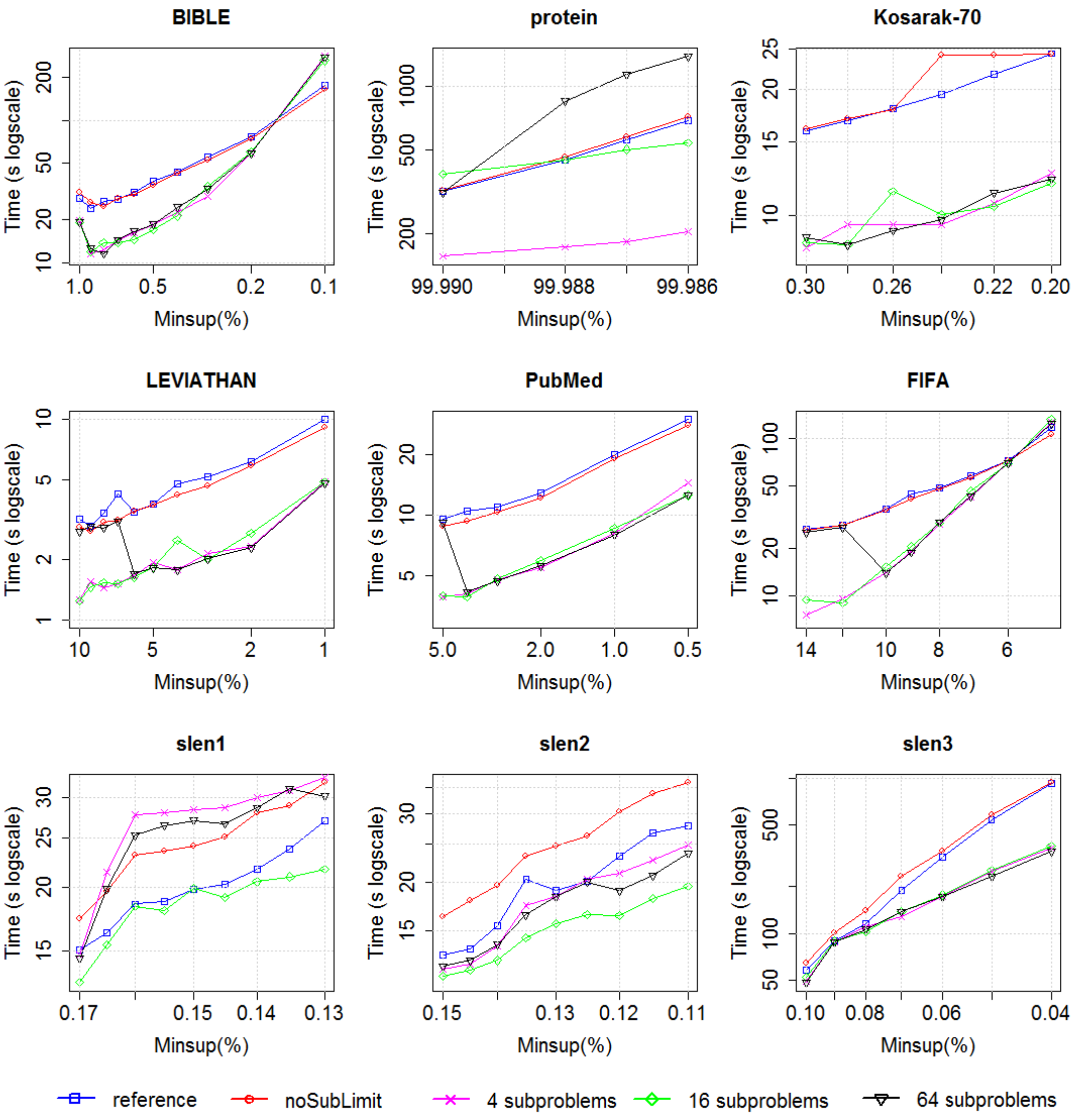


Figure 18: Performance improvement of sorting sub-problems during map stage.

This new implementation's performances were a huge success, as we can see on Figure 18.

Not only have we obtained our previous boost in performance, we also successfully avoided any memory consumption problem. This implementation advantages should thus be kept for the final version of our algorithm, since it allows far better performances when using the correct set. However, an automatic detection should be implemented, to detect when sorting sub-problems could yield increased performances, and only sort in these circumstances.

4.4.10 Performances of using a Map Based Sequence Database Structure

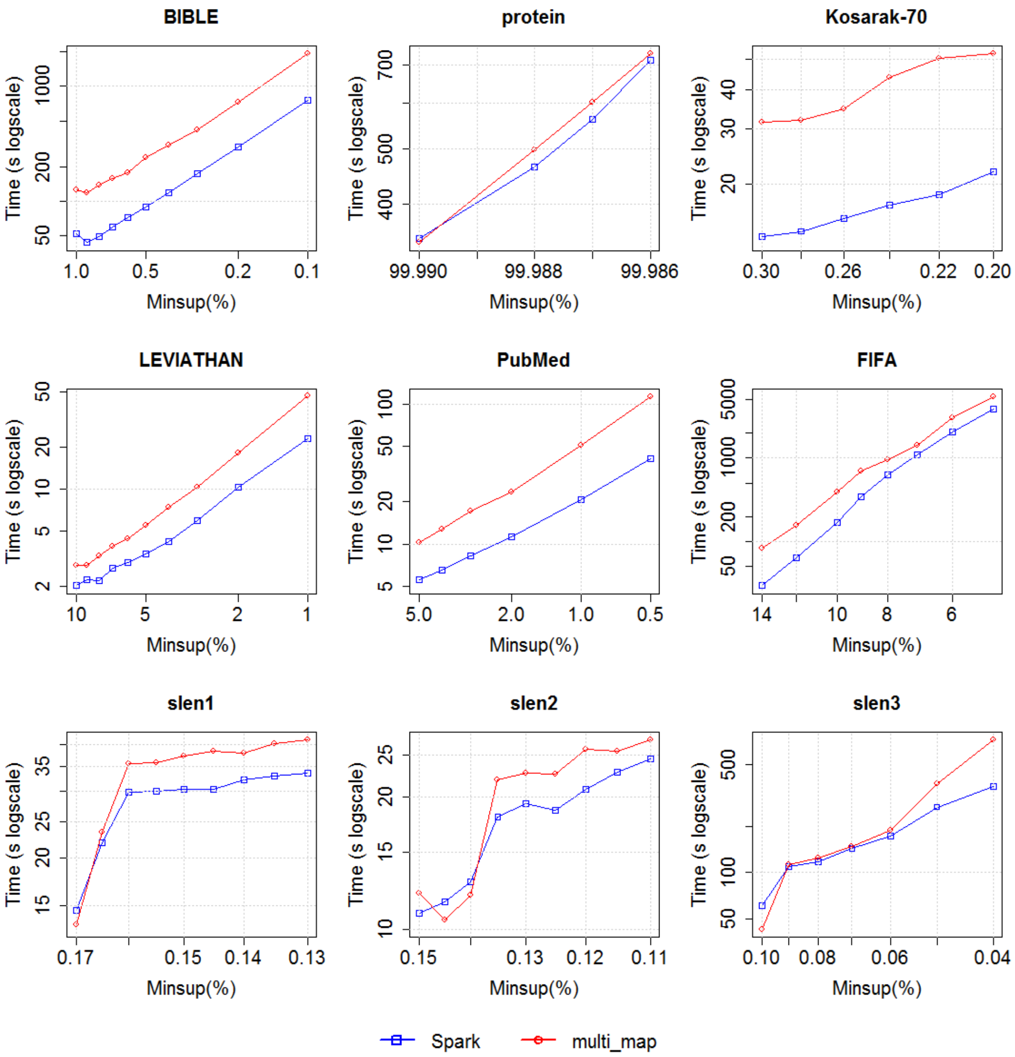


Figure 19: Performance of a CP based algorithm replacing all three pre-processed matrices from PPIC by a Map structure (Please take notice that we are here comparing performances with the original Spark implementation instead of the reference algorithm. Since the featured implementation doesn't possess the quicker-start improvement and the comparison is, in this case, much more relevant.)

As we can see in Figure 19, the performances of this implementation were terrible.

Profiling our algorithm revealed it came from two major factors. First, as feared, to attain the same purpose as the three matrices with our map structure, we needed many more trailing points which had to be backtracked at each step of the search. Regularly generating significant periods of time were our algorithm did nothing but backtrack the various sequences of its database.

The second factor was solution pruning. In PPIC, pruning our search space could be done by checking the last positions of each item, quickly asserting that if the start of a sequence was after the last position of an item in a sequence, the sequence was no longer supporting the item.

Here, however, the situation is different. To efficiently prune a multi-item pattern problem, checking the last position of items was no longer sufficient. To prune efficiently, each item-set now needed to be checked so that, should it match the current prefix, every possible extensions in that item-set could be have their support increased.

Which amount to this algorithm doing twice the work for the same results. As we found out through our tests, pruning using PPIC's original method (last position list) instead of fully, as spark did, produces better performance. A comparison of the two being available in the annexes, Figure 28.

Although the performances were better when not fully pruning our search space. Spark's original local execution was still much more efficient.

4.4.11 Performances PPIC with Partial Projection

The performance tests realised at the end of our implementation, shown in figure 20, revealed improved efficiency in comparison to Spark's original algorithm and our previously created map algorithm on sequence of sets of symbols. On slen3, this implementation was even much more efficient than any performances obtained earlier through the three position lists implementation.

These performances, however, came at the cost of an inefficiency on datasets of sequences of symbols, where performances fail to overcome Spark's standard on anything but the FIFA dataset.

This implementation should thus only be used for dealing with sets of symbols, while sequence of symbols should be found through PPIC, whose efficiency was focused on this kind of problem.

4.4.12 Performances of our final implementation

As we can see in Figure 21, our final implementation is much more performant than both our old reference implementation, and the slightly improved 'Clean before any Local Execution' implementation. Surpassing previous performances on nearly every single dataset, but particularly on protein and Kosarak-70 where the 'Specialised Scalable Execution' improvement was extremely effective.

We can also observe that the performance loss brought from the quicker-start 'improvement' have disappeared on the concerned datasets. Improving performance further on our largest datasets.

We a can, however, notice that performance worsens slightly on the slen1 dataset, a light performance loss easily explained by our choice to use our 'PPIC with Partial Projection' implementation, which showed this exact same loss of performance on slen1, but much greater performance gain on slen3.

Finally we can observe that on the FIFA and BIBLE dataset, our performance tends toward the performances of our 'Clean before any Local Execution' implementation as the number of required support gets lowered.

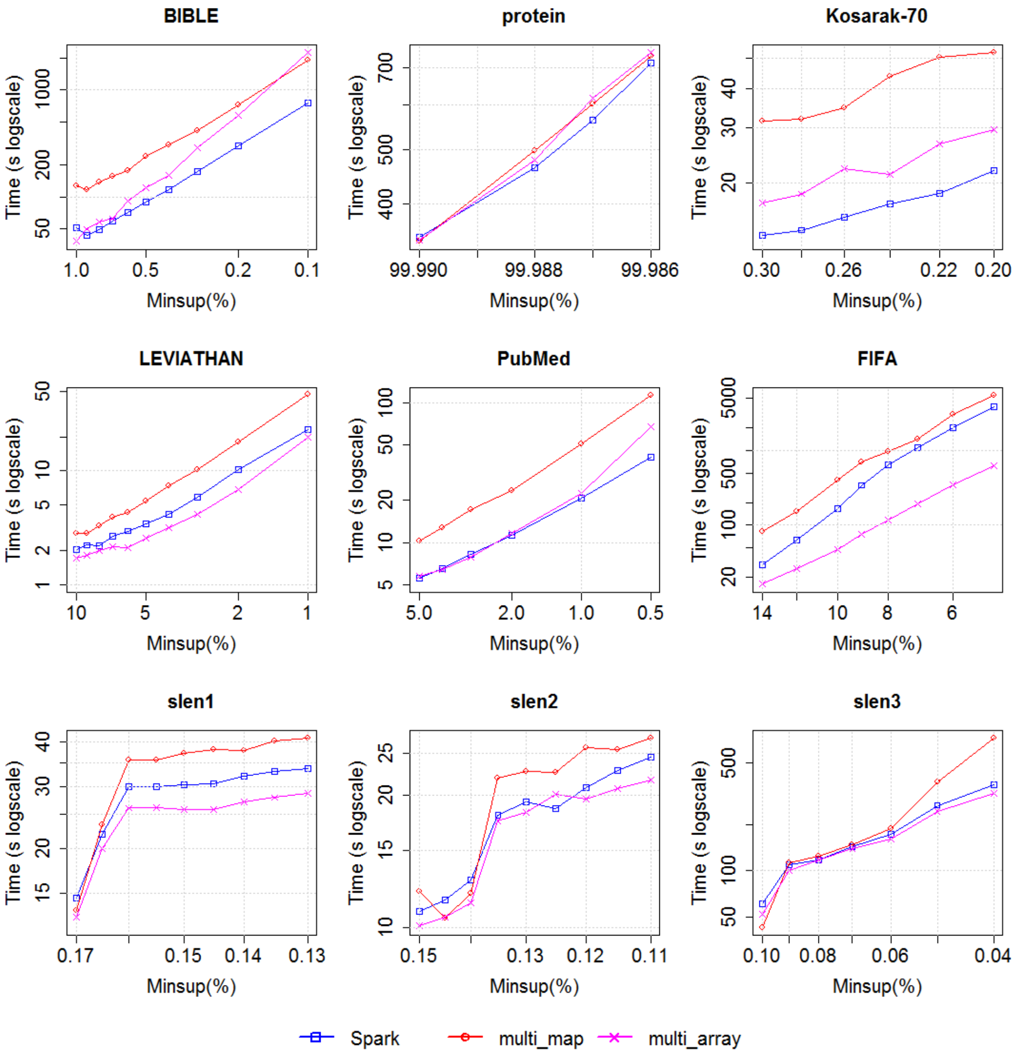


Figure 20: Performance of a CP based algorithm extending PPIC's implementation by maintaining partial starts.
(Please take notice that we are here comparing performances with the original Spark implementation instead of the reference algorithm. Since the featured implementation doesn't possess the quicker-start improvement and the comparison is, in this case, much more relevant.)

It thus seems that, on those kinds of datasets, the combined improvement fail to bring out further significant performance gains. Of course, in comparison to the original performances of Spark, this final implementation's performances remain much better.

Another interesting thing to note is that those performances were measured using the default parameters value. Greater performance improvement may thus be achieved by reducing the number of created sub-problems, since it would mean an earlier usage of our local execution algorithms, which are much more efficient than the scalable stage.

4.5 Scalability Tests

To conclude our performance testing section, let us measure the scalability of both Spark's original implementation and of our final implementation.

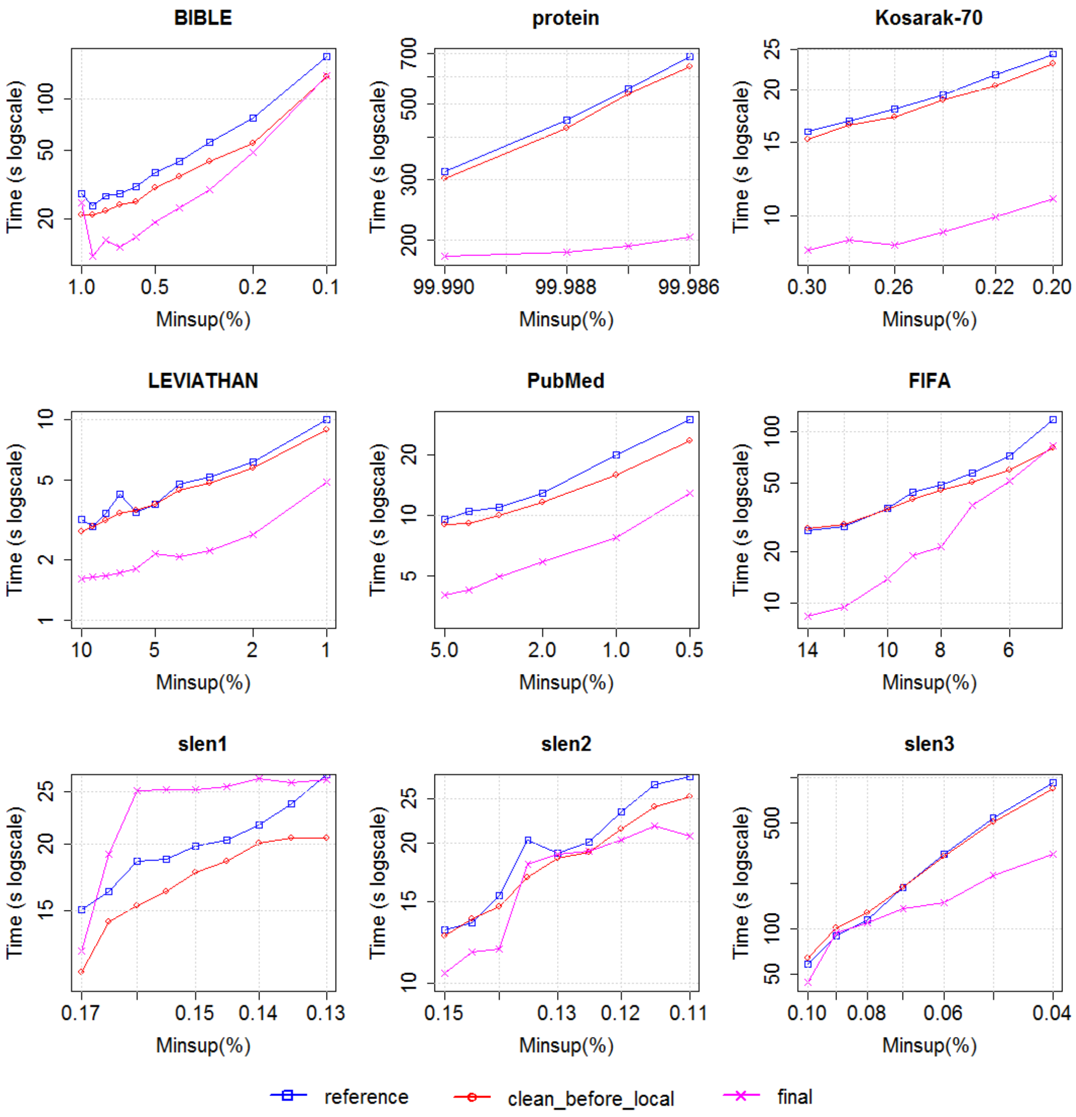


Figure 21: Performance of our final implementation

The tests will be conducted as follows: We will run an architecture containing 1 driver (given 5G memory to make sure all solution could be successfully received), and a variable number of workes. Each worker possessing one executor running a single thread, and disposing of only 1G ram memory.

Sadly, since the architecture on which we can run our tests has limited performances, running more than 10 workers will actually display slightly lowered performances, due to necessary switches between threads caused by an insufficient number of cores. We will thus limit ourselves to testing our implementation on architecture running 1,2,4, and 8 workers respectively, and compare the resulting performances under those conditions.

Additionally, we had to specify a reduced value of 64000 for the 'maxLocalProjectedDatabaseSize' parameter whose default value is set at 32000000, as otherwise it could use too much memory during the local execution step and surpass the 1G ram allocated to our workers.

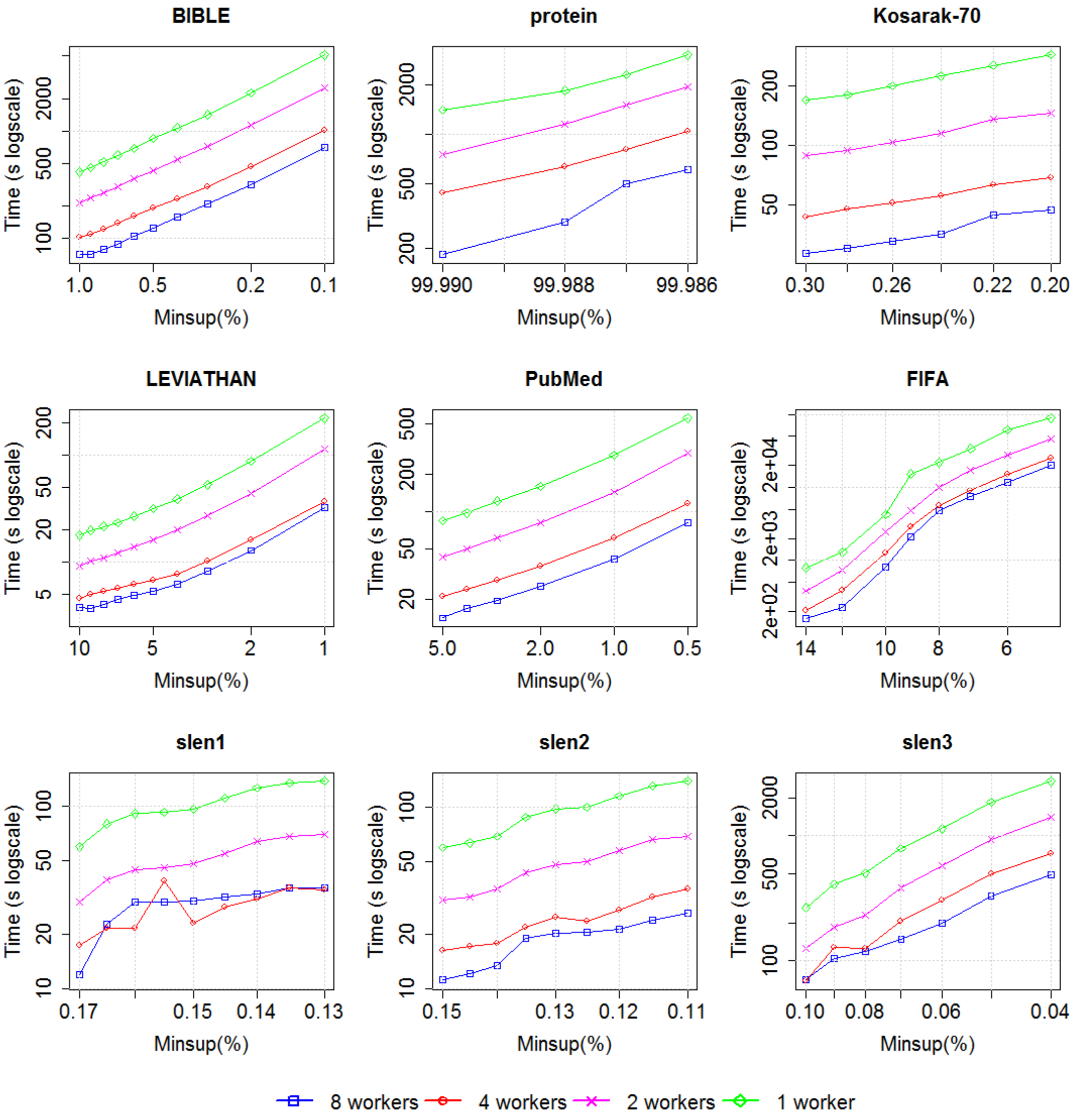


Figure 22: Scalability performances of the original implementation

4.5.1 Scalability of the Original Implementation of Spark

As you may see on Figure 22, the original implementation of Spark scales really well up to a certain point. We can even notice that between 1,2, and 4 workers, doubling the number of workers means twice as good performances. However, this apparent linear scalability disappears when increasing the number of workers to 8 in our architecture, since few changes can be seen in our measured performances.

The original implementation of Spark is thus quite scalable, but has limitations beyond what can be justified by the limited performances of our architecture. More specifically, it appears that when dealing with greater number of executors, Spark gradually use more and more of its time exchanging information instead of doing work. Thus the observed limitations.

4.5.2 Scalability of our final implementation

As you may see on Figure 23, our final implementation also shows greatly improved performances in a scalable environment, in comparison to the original algorithm.

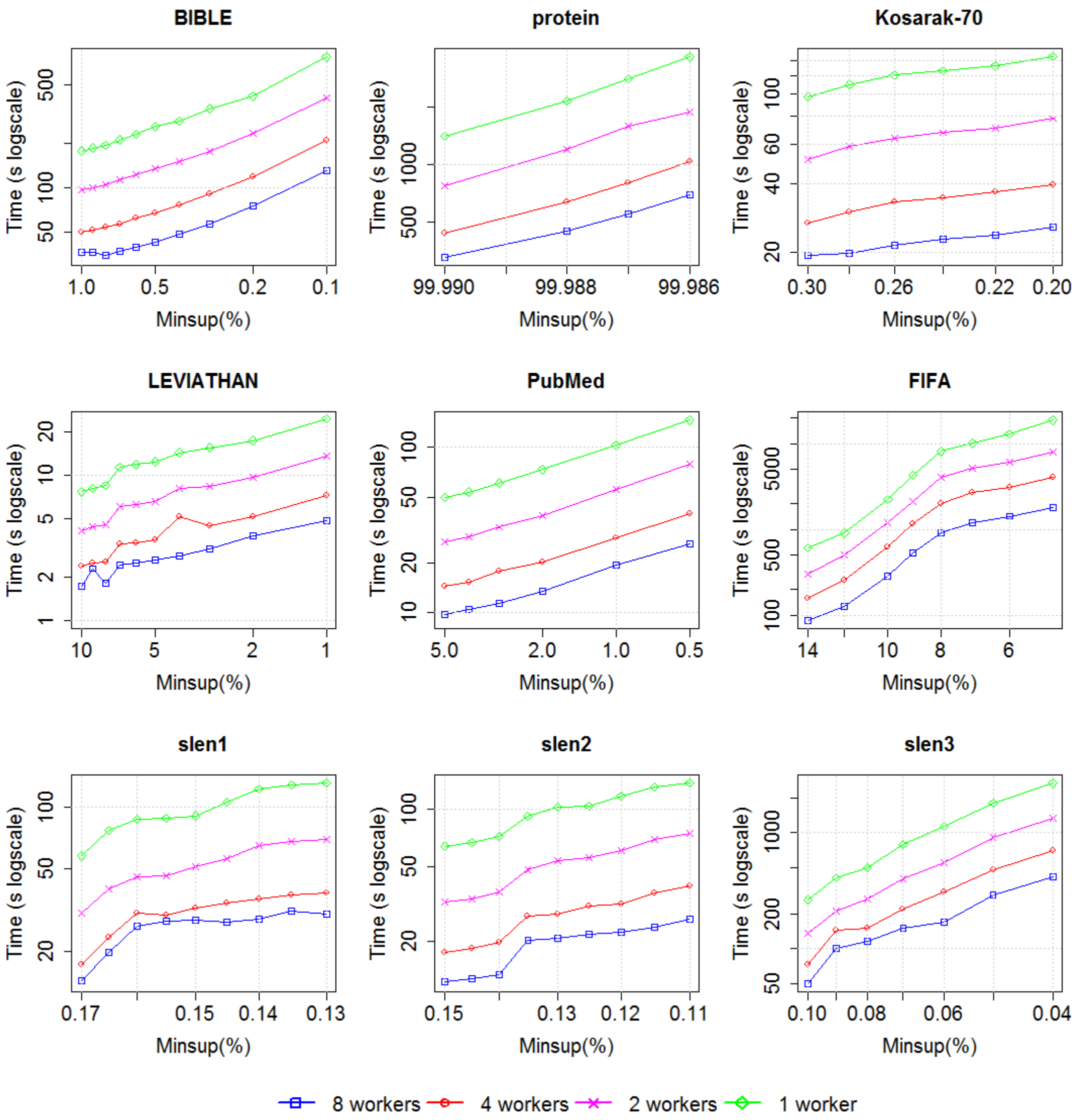


Figure 23: Scalability performances of our final implementation

Although this implementation similarly shows reduced performance gain as the number of workers increase, it has been greatly mitigated. Our implementation is thus not only more performant in general, but also more scalable. Although not quite to the point of achieving true linear scalability.

The greatest advantage of this new implementation being that we can launch the local execution step faster if given more memory. We thus have a closer relation, beyond simple access time to the hard drive, between allocated memory and performances.

A default, however, would be that there is no way to allocate large sub-problems to executors disposing of extra amount of memory instead of other executor which would have more limitation. The 'maxLocalProjectedDatabaseSize' parameter must thus be chosen while taking into account the lowest amount of memory available to an executor. Otherwise, the algorithm may crash during the local execution stage due to insufficient memory.

5 Conclusion

In this paper, we have developed a novel parallel sequential pattern mining approach based on constraint programming modules that is both scalable and efficient, greatly outperforming the original implementation available on Spark.

To do so, we used a generic map-reduce based scalable PrefixSpan implementation to divide the original problem in appropriately sized sub-problems. We then efficiently solved those sub-problems locally using a standard CP-solver running the PPIC algorithm for databases of sequences of symbols, and a newly designed PPIC-like CP algorithm for database of sequences of sets of symbols. Our various performance analysis then demonstrated that this novel approach not only outperformed Spark's original implementation in memory abundant environments, but also on scalable memory limited environment.

Additionally, we also demonstrated that greater performances could be attained by limiting the number of generated sub-problem depending on the provided number of executors and/or by sorting the generated sub-problems to compute larger sub-problems first. Finally, we demonstrated that cleaning large databases after operation large prefix projections allows noticeable efficiency gain on the long run, as it reduces the time necessary to project prefixes during the local execution stage.

However, this novel approach still has an important flaw. While we have certainly demonstrated through the addition of multiple new functionalities that a certain level of modularity inherent to CP-solver was kept in this novel parallel algorithm. It remains a fact that using a CP-solver beneath Spark's scalable framework isn't as modular and flexible as an implementation fully englobed in a CP-solver.

The next step forward in progressing this research should thus be taken by incorporating Spark's parallel execution inside a CP framework. Thus allowing efficient scalable execution on large scale databases **and** great modularity through the simple addition of constraints.

6 References

[1] N. R. Mabroukeh and C. I. Ezeife, “A taxonomy of sequential pattern mining algorithms,” *ACM Computing Surveys (CSUR)*, vol. 43, no. 1, p. 3, 2010.

[2] R. Agrawal and R. Srikant, “Mining sequential patterns,” in *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pp. 3–14, IEEE, 1995.

[3] R. Srikant and R. Agrawal, “Mining sequential patterns: Generalizations and performance improvements,” *Advances in Database Technology—EDBT’96*, pp. 1–17, 1996.

[4] M. J. Zaki, “Sequence mining in categorical domains: incorporating constraints,” in *Proceedings of the ninth international conference on Information and knowledge management*, pp. 422–429, ACM, 2000.

[5] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, “Mining sequential patterns by pattern-growth: The prefixspan approach,” *IEEE Transactions on knowledge and data engineering*, vol. 16, no. 11, pp. 1424–1440, 2004.

[6] B. Negrevergne and T. Guns, “Constraint-based sequence mining using constraint programming,” *arXiv preprint arXiv:1501.01178*, 2015.

[7] A. Kemmar, S. Loudni, Y. Lebbah, P. Boizumault, and T. Charnois, “A global constraint for mining sequential patterns with gap constraint,” in *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pp. 198–215, Springer, 2016.

[8] A. Kemmar, S. Loudni, Y. Lebbah, P. Boizumault, and T. Charnois, “Prefix-projection global constraint for sequential pattern mining,” in *International Conference on Principles and Practice of Constraint Programming*, pp. 226–243, Springer, 2015.

[9] M. J. Zaki, “Spade: An efficient algorithm for mining frequent sequences,” *Machine learning*, vol. 42, no. 1, pp. 31–60, 2001.

[10] J. O. Aoga, T. Guns, and P. Schaus, “An efficient algorithm for mining frequent sequence with constraint programming,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 315–330, Springer, 2016.

[11] OscaR Team, “OscaR: Scala in OR,” 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[12] Z. Yang, Y. Wang, and M. Kitsuregawa, “Lapin: effective sequential pattern mining algorithms by last position induction for dense databases,” *Advances in Databases: Concepts, Systems and Applications*, pp. 1020–1023, 2007.

[13] J. Deng, Z. Qu, Y. Zhu, G.-M. Muntean, and X. Wang, “Towards efficient and scalable data mining using spark,” in *Information and Communications Technologies (ICT 2014), 2014 International Conference on. IET*, pp. 1–6, 2014.

[14] M. J. Z. C.-T. Ho, “Large-scale parallel data mining,” *edited by J. Siekmann J. G. Carbonell, Lecture Notes in Artificial Intelligence: Springer*, 2000.

[15] J. O. Aoga, T. Guns, and P. Schaus, “Mining time-constrained sequential patterns with constraint programming,” *Constraint*, pp. 1–23, 2017.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[17] T. Guns, S. Nijssen, and L. De Raedt, "Itemset mining: A constraint programming perspective," *Artificial Intelligence*, vol. 175, no. 12-13, pp. 1951–1983, 2011.

[18] J. Han, J. Pei, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth," in *proceedings of the 17th international conference on data engineering*, pp. 215–224, 2001.

[19] Z. Yang and M. Kitsuregawa, "Lapin-spam: An improved algorithm for mining sequential pattern," in *Data Engineering Workshops, 2005. 21st International Conference on*, pp. 1222–1222, IEEE, 2005.

[20] F. Bonchi and C. Lucchese, "Extending the state-of-the-art of constraint-based pattern discovery," *Data & Knowledge Engineering*, vol. 60, no. 2, pp. 377–399, 2007.

[21] L. De Raedt, T. Guns, and S. Nijssen, "Constraint programming for itemset mining," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 204–212, ACM, 2008.

[22] L. De Raedt, T. Guns, and S. Nijssen, "Constraint programming for data mining and machine learning," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, pp. 1671–1675, 2010.

[23] L. D. Raedt and A. Zimmermann, "Constraint-based pattern set mining," in *proceedings of the 2007 SIAM International conference on Data Mining*, pp. 237–248, SIAM, 2007.

[24] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, "Freespan: frequent pattern-projected sequential pattern mining," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 355–359, ACM, 2000.

[25] Q. Zhao and S. S. Bhowmick, "Sequential pattern mining: A survey," *ITechnical Report CAIS Nayang Technological University Singapore*, pp. 1–26, 2003.

7 Annexes

7.1 Glossary:

BFS breath-first search.

CP constraint programming.

DFS depth-first search.

PPIC Prefix Projection Incremental Counting propagator.

RAM random access memory.

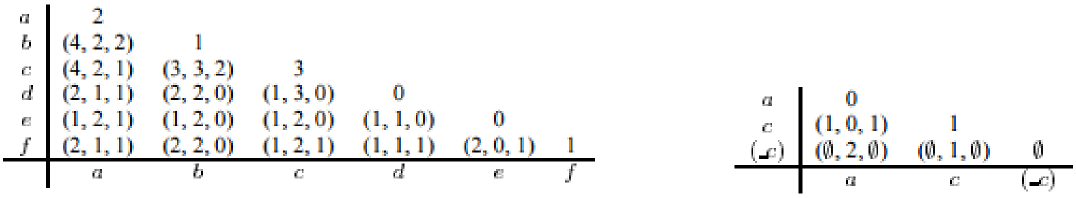
RDD resilient distributed dataset.

SoS sequence of symbols.

SoSS sequence of sets of symbols.

SPM sequential pattern mining.

7.2 Additional example images



a	2					
b	(4, 2, 2)	1				
c	(4, 2, 1)	(3, 3, 2)	3			
d	(2, 1, 1)	(2, 2, 0)	(1, 3, 0)	0		
e	(1, 2, 1)	(1, 2, 0)	(1, 2, 0)	(1, 1, 0)	0	
f	(2, 1, 1)	(2, 2, 0)	(1, 2, 1)	(1, 1, 1)	(2, 0, 1)	1
	a	*b*	*c*	*d*	*e*	*f*

(a) S-matrix in original database

a	0		
c	(1, 0, 1)	1	
(_*c*)	(∅, 2, ∅)	(∅, 1, ∅)	∅
	a	*c*	(_*c*)

(b) S-matrix in <AB>-projected database

S-M[a, b] = (4, 2, 2) => support(AB) = 4; support(BA) = 2; support((AB)) = 2;

Figure 24: Example of an S-matrix for PrefixSpan bi-level projection

7.3 Additional Performance Comparisons:

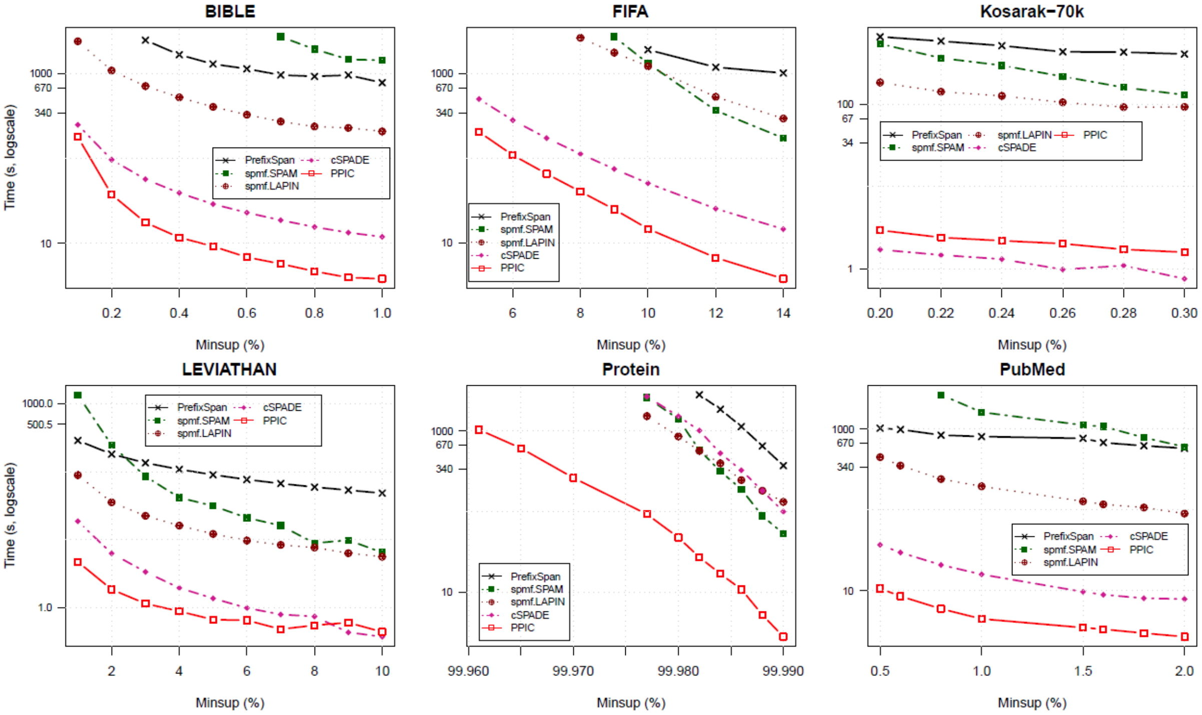


Figure 25: PPIC's performances VS other specialized algorithm

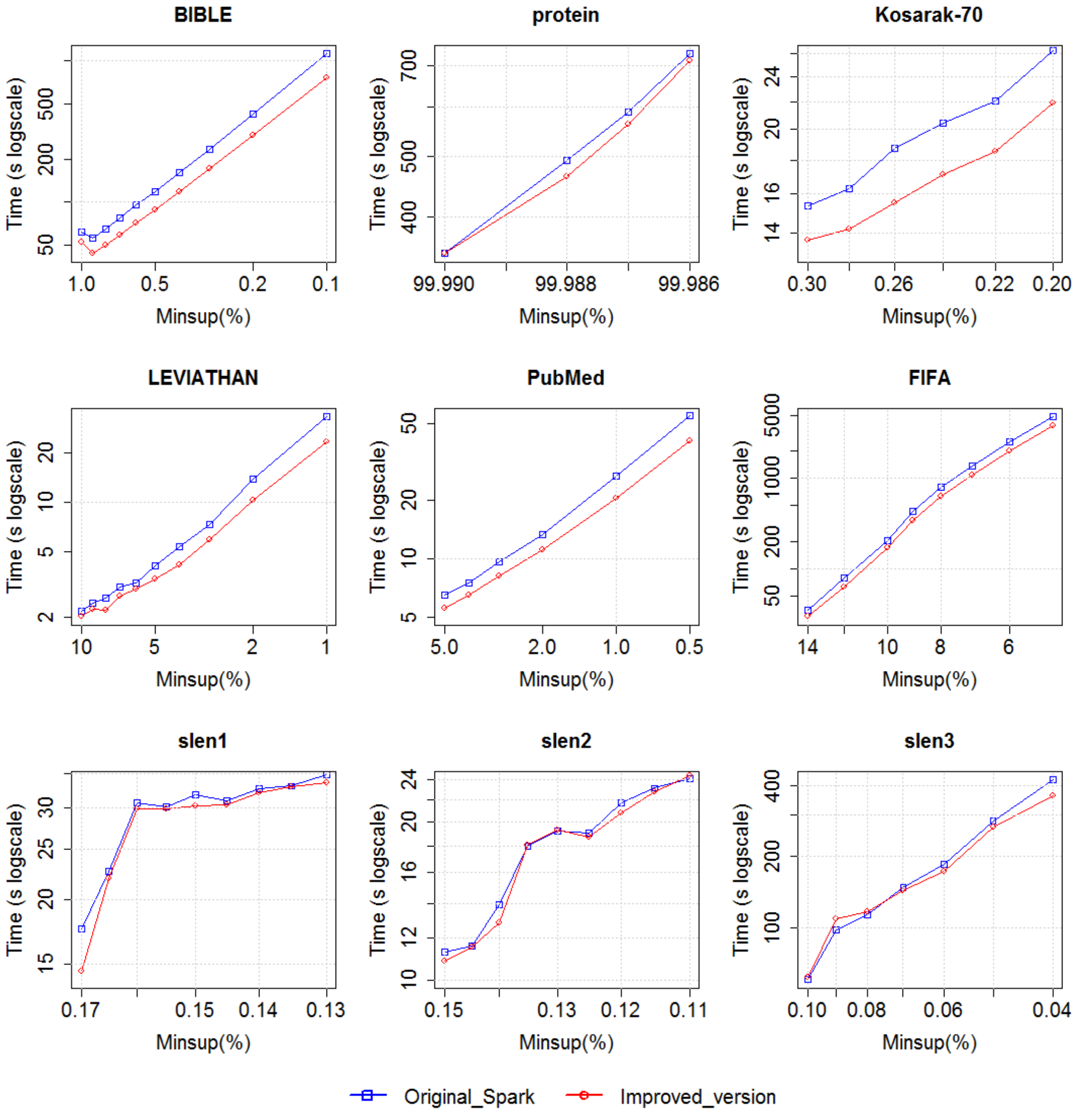


Figure 26: Performance improvement of fixing Spark's pre-processing

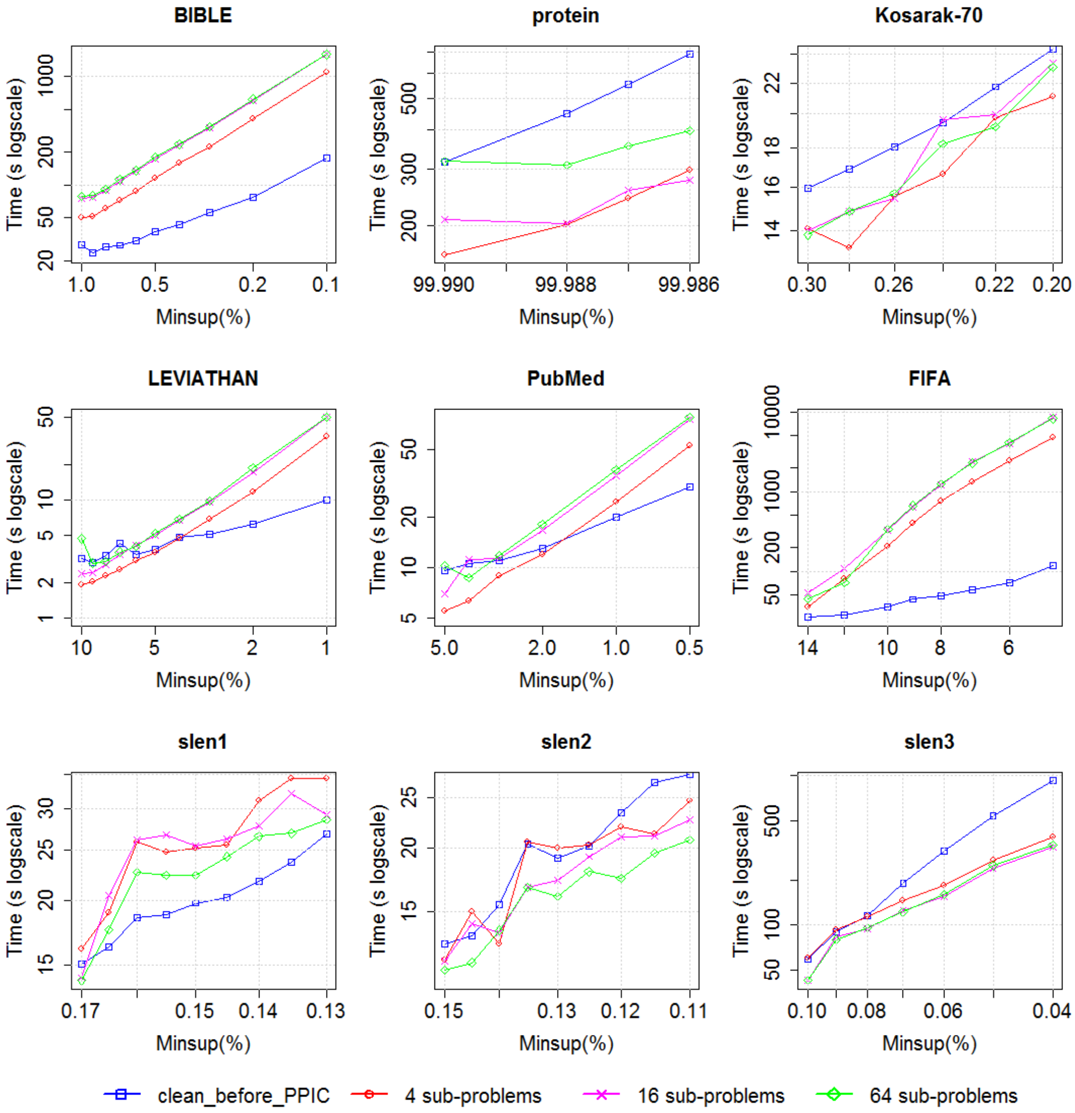


Figure 27: Performance improvement - soft limit on the number of created sub-problem. Tested on the reference (Clean Before Local Exec) implementation. (P

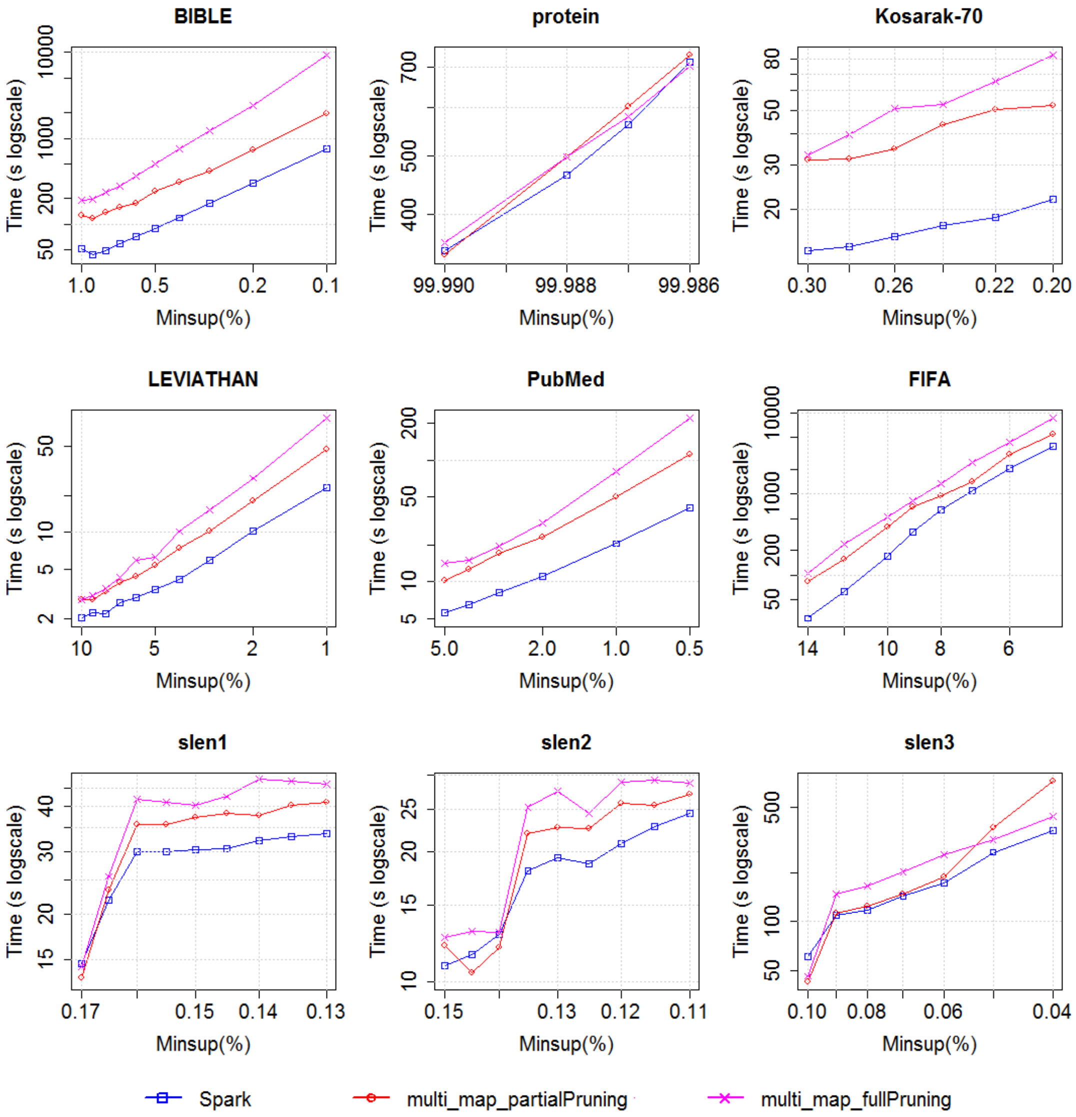


Figure 28: Performance comparison of a full pruning VS a partial pruning on the multi_map implementation.
(Please take notice that we are here comparing to the original Spark implementation instead of the reference algorithm. Since the featured implementation doesn't posses the early-start improvement and the comparison is, in this case, much more relevant.)

7.4 Algorithms:

Algorithm 2: Prefix Projection Incremental Counting propagator (PPIC)

```
Data: Given a sequence of symbol database (SDB), an array of reversible int representing
      our search space (P), and minimal number of support ($\theta$)
Result: PPIC return the set of frequent sequence of symbol contained in SDB, such that
        any returned pattern appear at least $\theta$ times in the database.
// Init global param:
sids = [0, ..., len(SDB)]; poss = [0] * len(SDB);
$\phi = 0$; $\varphi = len(SDB)$
projFreq = Array such that projFreq[i] contains the number of sequence initially
  supporting symbol i;
Function propagate($SDB, P, i, \theta$):
    // espilon == special value that ends a valid pattern
    if $P_i$ isBoundTo(epsilon) then
        // A valid pattern has been found, fill remaining space to collect
        // $P_0$ cannot be bound to epsilon
        foreach $j$ in $i + 1, ..., L$ do
            P(j).assign(epsilon)
        end
        /* When success is returned, the current pattern is collected a
           solution and we backtrack to the previous node of the search
           tree.                                                      */
        return Success
    else
        while $P_i$ isBound and $i <$ len(P) do
            nSupport = projectAndGetFreqs($SDB, Pi, \theta, sids, poss, \phi, \varphi$)
            if $nSupport < \theta$ then
                /* When failure is returned, we backtrack to the previous node
                   of the search tree.                                     */
                return Failure
            end
            // projFreq has been updated by the projectAndGetFreqs method.
            // Prune domain of $P_{i+1}$
            pruneDomain($i + 1, projFreq$)
            i++
        end
    end
    /* When suspend is returned, we continue down the search tree.      */
    return Suspend
end
```

Algorithm 3: PPIC Continued

```
Function projAndGetFreqs($SDB, itemToProject, \theta, sids, poss, \phi, \varphi$):
    projFreq[a] = 0 $\forall a \in \{1, ..., nSymbols\}$
    i = $\phi$; j = $\phi + \varphi$; sup = 0
    while $i < \phi + \varphi$ do
        sid = sids[i]; pos = poss[i]; seq = SDB[sid]
        /* pos => first position in a sequence, such that the previous
           prefix was supported.  To project the new item, we only need to
           search the remains of the sequence.                          */
        /* sid => The sequence id of a sequence which supported the previous
           prefix.                                                       */
        /* sids => A vector starting at position $\phi$ and containing $\varphi$ elements,
           allowing to only search sequence that supported the previous
           pattern.                                                      */
        /* seq => A sequence where the new item should be projected.     */
        if lastPosMap(sid)(itemToProject) - 1 > pos then
            // Find next position of itemToProject in seq
            if pos < firstPosMap(sid)(itemToProject) then
                // Jump to first occurrence of the item in the sequence
                pos = firstPosMap(sid)(itemToProject)
            else
                // Search for the next position of itemToProject
                while pos < len(seq) and itemToProject != seq[pos] do
                    pos = pos + 1
                end
            end
            // item supported -> update projected database
            sids[j] = sid; poss[j] = pos + 1; j = j + 1; sup = sup + 1;
            posToCheck = interestingPosMap(sid)(pos) while $posToCheck! = 0$ do
                // Faster than checking lastPosMap
                symbol = seq[posToCheck - 1] projFreqs[symbol] = projFreqs[symbol] + 1
                 posToCheck = interestingPosMap(sid)(posToCheck - 1)
            end
        end
        i = i+1
    end
    $\phi = \phi + \varphi$; $\varphi$ = sup
    return projFreqs
end
```

Algorithm 4: Spark's original implementation: Pre-Processing

```
Data: Given a sequence of sets of symbol database (SSDB), a minimal number of
      supporting sequence ($\theta$), a maximal pattern length (maxPatLen), and a maximal
      local projected database size (maxLocalSize)
Result: Return the set of frequent sequence of sets of symbol contained in SDB. Such that
        any returned pattern appear at least $\theta$ times in the database.
Function PreProcessing($SSDB, maxPatLen, maxLocalSize, \theta$):
    // Clean database through a map reduce phase
    freqItems = findFrequentItem(SSDB, $\theta$)
    cleanedSequences = cleanSequenceAndRenameItem(SSDB, freqItems)
    // Find frequent sequences of sets of symbols
    solutionPatterns = scalableExecution(cleanedSequences, maxPatLen, maxLocalSize, $\theta$)
    // Translate to original items name and Return
    return translateBackToOriginalItemsName(solutionPatterns)
end
```

Algorithm 5: Spark's original implementation: Scalable execution

```
Function scalableExecution($SSDB, maxPatLen, maxLocalSize, \theta$):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.        */
    postfixes = SSDB.map(seq => PostFix(seq))
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = [prefix.empty]
    // Start scalable execution
    while largePrefixes is not empty do
        mapReduceResults = postifixes.flatMap( postfix =>
            largePrefixes.flatMap( prefix =>
                extensions = postfix.project(prefix).findPrefixExtension()
                extensions.map( (item, postfixSize) =>
                    // Return a (key, value) pair
                    ((prefix.id, item), (1, postfixSize))
                )
            )
        ).reduceByKey($(v1.1 + v2.1, v1.2 + v2.2)$) // Aggregate values by key
        .filterByValue($v.1 \geq \theta$) // Keep key-value pair with enough support
        // Empty larger prefixes list
        largePrefixes.clean()
        // Fill it with new, extended, prefixes
        foreach ((prefiID, extendingItem), (support, projDBSize) ∈ mapReduceResults do
            // Create prefix, add it as solution
            newPrefix = largePrefixes.getByID(prefixID) += extendingItem
            solutionPatterns += newPrefix
            if len(newPrefix) < maxPatternLength then
                // len(newPrefix) == number of non-zero item in prefix
                // zero == separator between item-sets
                if projDBSize > maxLocalProjDBSize then
                    largePrefixes += newPrefix
                else
                    smallPrefixes += newPrefix
                end
            end
        end
    end
    if len(smallPrefixes) > 0 then
        // For each prefix, project the whole database and solve locally
        toExecuteLocaly = postifixes.flatMap( postfix =>
            smallPrefixes.flatMap( prefix =>
                (prefix.ID, postfix.project(prefix).compress())
            )
        ).groupByKey().flatMap(sequences =>
            localExecution(sequences, maxPatLen - len(prefix), $\theta$)
        )
    end
end
```

Algorithm 6: Spark's original implementation: Local execution

```
Function localExecution($SSDB, maxPatLen, \theta$):
    if maxPatLen == 0 then
        return Array.empty
    end
    // Find extending item that are sufficiently frequent
    counts = Map[Int, Int].empty.withDefaultValue(0)
    foreach postfix $\in$ SSDB do
        foreach (extendingItem, size) $\in$ postfix.findPrefixExtension() do
            counts(extending) += 1
        end
    end
    counts = counts.filterByValue(val $\geq \theta$)
    // Extend for each item and continue searching
    solutions = ArrayList.empty[Prefix]
    foreach key $\in$ counts do
        projectedDB = postfixes.project(key)
        foreach extension $\in$ localExecution(projectedDB, maxPatLen-1, $\theta$) do
            solution += key + extensions
        end
    end
    return solutions
end
```

Algorithm 7: Spark's original implementation: Project and findPrefixExtension

```
Class Postfix(sequence):
    start = 0
    partialProjections = Array.empty
    Function project (Item):
        if start > len(sequence) then
            return
        end
        newPartialProjection = Array.empty
        if Item extends current Item-set in Prefix then
            foreach partial ∈ partialProjections do
                newPos = findNextItemInCurrentItemSet(sequence, partial, Item)
                if Found Item in current ItemSet then
                    newPartialProjection += newPos
                end
            end
            start = findSmallestPosition(newPartialProjection) + 1
        else if Item start a new Item-set in Prefix then
            for $Pos \in \{start, ..., len(sequence) - 1\}$ do
                if sequence[Pos] == Item then
                    if first found in loop then
                        start = Pos + 1
                    else
                        newPartialProjection += Pos
                    end
                end
            end
        end
        partialProjections = newPartialProjection
    end
    Function project (Prefix):
        foreach Item ∈ Prefix do
            this.project(Item)
        end
    end
    Function project (findPrefixExtension):
        extendingItems = Set.empty
        foreach partial ∈ partialProjections do
            // Find items that extend current itemSet
            extendingItems ++= findAllItemUntilNextSeparator(sequence, partial)
        end
        for $Pos \in \{start, ..., len(sequence) - 1\}$ do
            // Find items that start a new itemSet
            extendingItems += sequence[Pos]
        end
    end
end
```

Algorithm 8: First scalable CP based implementation

```
Function
 scalableExecution($SSDB, maxPatLen, maxLocalSize, \theta, hasSetsOfSymbols$):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.        */
    postfixes = SSDB.map(seq => PostFix(seq))
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = [prefix.empty]
    // Start scalable execution
    while largePrefixes is not empty do
        ... // Same as Spark's original implementation
    end
    // Start local execution
    if len(smallPrefixes) > 0 then
        // For each prefix, project the whole database and solve locally
        toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
            (prefix.ID, postfix.project(prefix).compress())
        )).groupByKey().flatMap(sequences =>
        // Use argument to determine which local execution to use
        if hasSetsOfSymbols then
            localExecution(sequences, maxPatLen - len(prefix), $\theta$)
        else
            sequences = removeSeparator(sequences)
            matrices = generatePPICMatrices()
            /* PPIC will generate the P vector, add constraint to enforce
               maxPatLen and then launch the execution, calling propagate in
               the process.                                                   */
            PPIC(sequences, maxPatLen - len(prefix), $\theta$, matrices)
        end
        )
    end
end
```

Algorithm 9: New functionalities: Scalable execution

```
Function scalableExecution($SSDB, minPatLen, maxPatLen, maxLocalSize, \theta,$
$limitItemPerItemset, softSubProblemLimit$):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.         */
    postfixes = SSDB.map(seq => PostFix(seq))
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = [prefix.empty]
    // Start scalable execution
    while largePrefixes is not empty do
        if len(largePrefixes) + len(smallPrefixes) $\geq$ softSubProblemLimit $>$ 0 then
            smallPrefixes ++= largePrefixes // Switch to local execution
            break
        end
        mapReduceResults = postifixes.flatMap( postfix =>
            ... // See First Scalable CP based Implementation
        ).reduceByKey(($v1.1 + v2.1, v1.2 + v2.2$)) // Aggregate values by key
        .filterByValue($v.1 \geq \theta$) // Keep key-value pair with enough support
        // Empty larger prefixes list
        largePrefixes.clean()
        // Fill it with new, extended, prefixes
        foreach ((prefiID, extendingItem), (support, projDBSize) $\in$ mapReduceResults do
            // Create prefix, add it as solution
            newPrefix = largePrefixes.getByID(prefixID) += extendingItem
            if respectConstraints(newPrefix, minPatLen, limitItemPerItemset) then
                solutionPatterns += newPrefix
            end
            if canExtendedItemRespectConstraint(newPrefix, maxPatLen,
             limitItemPerItemset then
                // Search extensions for this prefix
                if projDBSize $>$ maxLocalProjDBSize then
                    largePrefixes += newPrefix
                else
                    smallPrefixes += newPrefix
                end
            end
        end
    end
    if len(smallPrefixes) $>$ 0 then
        /* Similar local execution than in First Scalable CP based
           Implementation, but with additional checks to guarantee that the
           new constraints are respected.                                   */
        ...
    end
end
```

Algorithm 10: Quick - Start: Scalable execution

```
Function scalableExecution($SSDB, minPatLen, maxPatLen, maxLocalSize, \theta,$
$limitItemPerItemset, softSubProblemLimit, frequentItemAndCount$):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.        */
    postfixes = SSDB.map(seq => PostFix(seq))
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = Array.empty
    foreach (item, count) ∈ frequentItemAndCount do
        largePrefixes += Prefix(item, count) // count == nbSupport for item
    end
    // Start scalable execution
    while largePrefixes is not empty do
        // Project and extend prefix
    end
    if len(smallPrefixes) > 0 then
        // Project and extend prefix locally
    end
end
```

Algorithm 11: Clean database before local execution of PPIC:

```
Function scalableExecution($SSDB, minPatLen, maxPatLen, maxLocalSize, \theta,$
$hasSetsOfSymbols, limitItemPerItemset, softSubProblemLimit, freqItemAndCount$):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.         */
    postfixes = SSDB.map(seq => PostFix(seq))
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = Array.empty
    foreach (item, count) $\in$ freqItemAndCount do
        largePrefixes += Prefix(item, count)// count == nbSupport for item
    end
    // Start scalable execution
    while largePrefixes is not empty do
        // Project and extend prefix
    end
    if len(smallPrefixes) $>$ 0 then
        // For each prefix, project the whole database and solve locally
        toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
            (prefix.ID, postfix.project(prefix).compress())
        )).groupByKey().flatMap(sequences =>
        // Use argument to determine which local execution to use
        if hasSetsOfSymbols then
            localExecution(sequences, maxPatLen - len(prefix), $\theta$)
        else
            sequences = cleanUnFrequentItemsAndRename(sequences)
            matrices = generatePPICMatrices()
            /* PPIC will generate the P vector, add constraint to enforce
               maxPatLen and then launch the execution, calling propagate in
               the process.                                                */
            PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, $\theta$, matrices,
             prefix)
        end
        )
    end
end
```

Algorithm 12: Automatic Local Execution Selection:

```
Function scalableExecution(SSDB, minPatLen, maxPatLen, maxLocalSize, $\theta$,
limitItemPerItemset, softSubProblemLimit, freqItemAndCount):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.          */
    postfixes = SSDB.map(seq => PostFix(seq))
    // Recalculate MaxItemPerItemset
    calculatedMaxItemPerItemset = findLargestItemSetSize(SSDB)
    if limitItemPerItemset == 0 or limitItemPerItemset > calculatedMaxItemPerItemset
     then
        limitItemPerItemset = calculatedMaxItemPerItemset
    end
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = Array.empty
    foreach (item, count) $\in$ freqItemAndCount do
        largePrefixes += Prefix(item, count) // count == nbSupport for item
    end
    // Start scalable execution
    while largePrefixes is not empty do
        // Project and extend prefix
    end
    if len(smallPrefixes) > 0 then
        // For each prefix, project the whole database and solve locally
        toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
            (prefix.ID, postfix.project(prefix).compress())
        )).groupByKey().flatMap(sequences =>
        // Determine which local execution to use
        if calculatedMaxItemPerItemset > 1 then
            localExecution(sequences, maxPatLen - len(prefix), $\theta$)
        else
            sequences = cleanUnFrequentItemsAndRename(sequences)
            matrices = generatePPICMatrices()
            /* PPIC will generate the P vector, add constraint to enforce
               maxPatLen and then launch the execution, calling propagate in
               the process.                                                */
            PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, $\theta$, matrices,
             prefix)
        end
        )
    end
end
```

Algorithm 13: Clean database before any local execution:

```
Function scalableExecution(SSDB, minPatLen, maxPatLen, maxLocalSize, $\theta$,
limitItemPerItemset, softSubProblemLimit, freqItemAndCount):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.        */
    postfixes = SSDB.map(seq => PostFix(seq))
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = Array.empty
    foreach (item, count) $\in$ freqItemAndCount do
        largePrefixes += Prefix(item, count) // count == nbSupport for item
    end
    // Start scalable execution
    while largePrefixes is not empty do
        // Project and extend prefix
    end
    if len(smallPrefixes) > 0 then
        // For each prefix, project the whole database and solve locally
        toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
            (prefix.ID, postfix.project(prefix).compress())
        )).groupByKey().flatMap(sequences =>
        // Clean sequences
        (sequences, canUsePPIC) = removeUnFrequentItemsEfficiently(sequences) // NB:
           Partial starts must also be modified to remain correct
        // Determine which local execution to use
        if ! canUsePPIC then
            localExecution(sequences, maxPatLen - len(prefix), $\theta$)
        else
            sequences = removeSeparators(sequences)
            matrices = generatePPICMatrices()
            /* PPIC will generate the P vector, add constraint to enforce
               maxPatLen and then launch the execution, calling propagate in
               the process.                                                */
            PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, $\theta$, matrices,
             prefix)
        end
        )
    end
end
```

Algorithm 14: Positions lists: Project and findPrefixExtension

```
Class Postfix(sequence):
    start = 0; partialProjections = Array.empty
    firstPosList = getFirstPosList(sequence); lastPosList = getLastPosList(sequence)
    Function project (Item):
        if start > len(sequence) or start > lastPosList(Item) then
            return
        end
        // Here, we know Item is present in remains of sequence
        newPartialProjection = Array.empty
        if Item extends current Item-set in Prefix then
            foreach partial ∈ partialProjections do
                newPos = findNextItemInCurrentItemSet(sequence, partial, Item)
                if Found Item in current Item-set then
                    newPartialProjection += newPos
                end
            end
            start = findSmallestPosition(newPartialProjection) + 1
        else if Item start a new Item-set in Prefix then
            if start == 0 then
                start = firstPosList(sequence)
            end
            for $Pos \in \{start, ..., len(sequence) - 1\}$ do
                if sequence[Pos] == Item then
                    if first found in loop then
                        start = Pos + 1
                    else
                        newPartialProjection += Pos
                    end
                end
            end
        end
        partialProjections = newPartialProjection
    end
    Function project (Prefix):
        foreach Item ∈ Prefix do
            this.project(Item)
        end
    end
    Function project (findPrefixExtension):
        extendingItems = Set.empty
        foreach partial ∈ partialProjections do
            // Find items that extend current itemSet
            extendingItems ++= findAllItemUntilNextSeparator(sequence, partial)
        end
        for (Item, lastPos) ∈ lastPosList, such that lastPos ≥ start do
            extendingItems += Item
        end
    end
end
```

Algorithm 15: Specialised Execution: Pre-Processing

```
Function PreProcessing($SSDB, maxPatLen, maxLocalSize, \theta$):
    // Clean database through a map reduce phase
    freqItems = findFrequentItem(SSDB, $\theta$)
    cleanedSequences = cleanSequenceAndRenameItem(SSDB, freqItems)
    /* Find database type.  If any item-set of size greater than one found
       in sequences, database type will be SSDB, else SDB                */
    databaseType = SSDB.map(sequence => findType(sequence)).reduce()
    // Find frequent sequences of sets of symbols
    solutionPatterns = scalableExecution(cleanedSequences, databaseType, maxPatLen,
      maxLocalSize, $\theta$)
    // Translate to original items name and Return
    return translateBackToOriginalItemsName(solutionPatterns)
end
Function scalableExecution(...):
    /* Separate postfix object in two sub-objects, one which search for
       item-sets extension (type == SSDB) and one where it never does so
       (type == SDB). Initialise all sequences with one object or the other,
       depending on database type.                                        */
    ...
end
```

Algorithm 16: Sorting sub-problems on the reducer:

```
Function scalableExecution($SSDB, minPatLen, maxPatLen, maxLocalSize, \theta,$
$hasSetsOfSymbols, limitItemPerItemset, softSubProblemLimit, freqItemAndCount$):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.        */
    postfixes = SSDB.map(seq => PostFix(seq))
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = Array.empty
    foreach (item, count) $\in$ freqItemAndCount do
        largePrefixes += Prefix(item, count)// count == nbSupport for item
    end
    // Start scalable execution
    while largePrefixes is not empty do
        // Project and extend prefix
    end
    if len(smallPrefixes) > 0 then
        // For each prefix, project the whole database and solve locally
        toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
            (prefix.ID, postfix.project(prefix).compress())
        )).groupByKey().SortByDBSize().flatMap(sequences =>
        // Use argument to determine which local execution to use
        if hasSetsOfSymbols then
            localExecution(sequences, maxPatLen - len(prefix), $\theta$)
        else
            sequences = cleanUnFrequentItemsAndRename(sequences)
            matrices = generatePPICMatrices()
            /* PPIC will generate the P vector, add constraint to enforce
               maxPatLen and then launch the execution, calling propagate in
               the process.                                               */
            PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, $\theta$, matrices,
             prefix)
        end
        )
    end
end
```

Algorithm 17: Sorting sub-problems on the mapper:

```
Function scalableExecution($SSDB, minPatLen, maxPatLen, maxLocalSize, \theta,$
$hasSetsOfSymbols, limitItemPerItemset, softSubProblemLimit, freqItemAndCount$):
    /* Encapsulate each sequence in a postfix, allowing to keep the current
       position and partial starts without copying the sequences.        */
    postfixes = SSDB.map(seq => PostFix(seq))
    // Init prefix list and solution list
    solutionPatterns = Array.empty
    smallPrefixes = Array.empty
    largePrefixes = Array.empty
    foreach (item, count) ∈ freqItemAndCount do
        // For quick-started prefixes, set projected DB size to max allowed
        largePrefixes += Prefix(item, count, Long.MaxValue)
    end
    // Start scalable execution
    while largePrefixes is not empty do
        // Project and extend prefix
    end
    if len(smallPrefixes) > 0 then
        // For each prefix, project the whole database and solve locally
        // Prefix are modified to hold their projected database size
        toExecuteLocaly = postifixes.flatMap( postfix =>
         smallPrefixes.SortByDBSize().flatMap( prefix =>
            (prefix.ID, postfix.project(prefix).compress())
        )).groupByKey().flatMap(sequences =>
        // Use argument to determine which local execution to use
        if hasSetsOfSymbols then
            localExecution(sequences, maxPatLen - len(prefix), $\theta$)
        else
            sequences = cleanUnFrequentItemsAndRename(sequences)
            matrices = generatePPICMatrices()
            /* PPIC will generate the P vector, add constraint to enforce
               maxPatLen and then launch the execution, calling propagate in
               the process.                                               */
            PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, $\theta$, matrices,
             prefix)
        end
        )
    end
end
```

Algorithm 18: PPIC with a Map based Structure (partial pruning):

```
// Init global param:
...
Function propagate($SSDB, P, i, \theta$):
    ... // Same propagate as PPIC, but the database is an SSDB instead
end
Function projAndGetFreqs($SSDB, itemToProject, \theta, sids, poss, \phi, \varphi$):
    projFreq[a] = 0 $\forall a \in \{1, ..., nSymbols\}$; i = $\phi$; j = $\phi + \varphi$; sup = 0
    while $i < \phi + \varphi$ do
        sid = sids[i]; pos = poss[i]; seq = SSDB[sid]
        // Get positions of itemToProject in sequence
        listSoughtItemPos = seq.get(itemToProject).discardItemsBefore(pos)
        if not listSoughtItemPos.isEmpty() then
            // Find next position of itemToProject in seq
            if lastItemInPwasSeparator()) then
                // Jump to first occurrence of the item in the sequence
                pos = listSoughtItemPos.pop() + 1
                // item supported -> update projected database
                sids[j] = sid; poss[j] = pos; j = j + 1; sup = sup + 1;
            else
                /* Search the sequence Item-set by Item-set until all elements
                   of P's current item-set match                              */
                for position $\in$ listSoughtItemPos do
                    if curItemSetInSeqContainsAllItemOfCurrentItemSetInP(seq, position)
                     then
                        // Jump to first occurrence of the item in the sequence
                        pos = listSoughtItemPos.pop() + 1
                        // item supported -> update projected database
                        sids[j] = sid; poss[j] = pos; j = j + 1; sup = sup + 1;
                        Break;
                    end
                end
            end
            if itemToProject supported in sequence then
                // Count support for items (partial pruning)
                foreach (item, positionsList) $\in$ seq do
                    positionList.discardItemsBefore(pos)
                    if not positionList.isEmpty() then
                        projFreqs[item] = projFreqs[item] + 1
                    end
                end
                /* For full pruning, find all item-set in sequence that
                   support current Item-set in P, augment the projected
                   frequency of all item present in those item-sets           */
            end
        end
        i = i+1
    end
    $\phi = \phi + \varphi$; $\varphi$ = sup
    return projFreqs
end
```

Algorithm 19: PPIC with partial projections:

```
Function projAndGetFreqs($SSDB, itemToProject, \theta, sids, poss, \phi, \varphi, partialProj$):
    projFreq[a] = 0 $\forall a \in \{1, ..., nSymbols\}$; i = $\phi$; j = $\phi + \varphi$; sup = 0
    while $i < \phi + \varphi$ do
        sid = sids[i]; pos = poss[i]; seq = SSDB[sid]
        if itemToProject == separator then
            partialProj = Array.empty
            pos = findNextSeparatorInSequence(seq, pos)
            if pos < len(seq) then
                // item supported -> update projected database
                sids[j] = sid; poss[j] = pos + 1; j = j + 1; sup = sup + 1;
                projFreq = findFrequencyOfEachItemFromLastPosList(sid, pos)
            end
        else if partialProj.isEmpty then
            pos = firstPositionOfItemToProjectInRemainOfSequence(seq, pos)
            partialProj = AllSuccessivePosOfItemToProject(seq, pos)
            if pos < len(seq) then
                // item supported -> update projected database
                sids[j] = sid; poss[j] = pos + 1; j = j + 1; sup = sup + 1;
                projFreq = findFrequencyOfEachItemFromLastPosList(sid, pos)
            end
        else
            newPartialProj = Array.empty
            foreach posToCheck $\in$ partialProj do
                posToCheck = findPosOfNextItemToProjectInCurrentItemSet(seq,
                  posToCheck)
                if posToCheck < len(seq) then
                    newPartialProj += posToCheck
                end
            end
            partialProj = newPartialProj
            pos = partialProj.min
            if not partialProj.isEmpty then
                // item supported -> update projected database
                sids[j] = sid; poss[j] = pos + 1; j = j + 1; sup = sup + 1;
                projFreq = searchItemSetOfPartialProjForExtendingItem(seq, partialProj)
            end
        end
        i = i+1
    end
    $\phi = \phi + \varphi$; $\varphi$ = sup
    return projFreqs
end
```

8 Acknowledgment

Computational resources have been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11

I would like to express my gratitude to my supervisor Pierre Schaus for the useful comments and remarks that steered this thesis back in the right the direction when it was most needed. Furthermore, I would like to thank John Aoga for the continuous and unfailing support provided through the duration of this master thesis, and my readers: Guillaume Derval, Nijssen Siegfried, and Roberto D'Ambrosio, for their time and their valuable comments.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**