UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

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

Academic year 2016-2017

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

TODO - Half a page to a page of content should be enough

1 Introduction

TODO Prefix Projection Incremental Counting propagator mathematics sequence of symbols sequence of sets of symbols

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 ItemSets : A symbol (s) 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 group into sets of symbol.
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 will refer to sets of items as ItemSets.
In the remainder of this paper, we will generally 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 = $\{s_1, s_2, ..., s_n\}$ of length N, is an ordered list of potentially repeating symbols. Those symbols can either be grouped into sets (in which case each set will be clearly separated from others using delimiters) or each symbol could be part of it's own independent set. In the remained of this paper, we will distinguish between those two type of sequence by calling them either sequence of sets of symbols (SoSS) in the first case or sequence of symbols (SoS) in the second.

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

- *<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.*
- *<(A B)(C A)> is a sequence of sets of symbols (SoSS), where parenthesis act as delimiters between symbols sets.*
- *<(A)(C)(A)> is a sequence of symbols (SoS), since no sets of symbol contain more than one element.*

Definition 3. Sub-Sequence / Super-Sequence: A sequence alpha = $\{\alpha_1, \alpha_2, ..., \alpha_m\}$ is a sub-sequence of seq = $\{s_1, s_2, ..., s_n\}$ and seq is a super-sequence of alpha if and only if : m $\leq$ n and $\forall i \in \{1, ..., m\} \quad \exists j_i \quad such \quad that \quad 1 \leq j_1 \leq ... \leq j_m \leq n \quad and \quad \alpha_i = seq_{j_i}$

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 contains only 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 with 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 specialized algorithm may never be used on SSDB.
NB : SDB are SSDB but the opposite is not true !

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

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

Definition 5. Cover, Support, Pattern, Frequent Pattern : The **cover** of a sequence seq in SSDB, denoted by cover(seq), is the subset of sequences in SSDB that are a super-sequence of seq. The **support** of a sequence seq in SSDB, denoted nbSupportSDB(seq), is the number of sequence in the cover. Any sequence seq can be a **pattern**, but we call **frequent pattern**, pattern where nbSupportSSDB(seq) $\geq \theta$, where θ is a given minimum support threshold.

Definition 6. Sequential Pattern Mining (SPM) Given an minimum sup-port threshold θ and a sequence database SSDB, the SPM problem is to find all frequent patterns of the SSDB.

Definition 7. Prefix, Suffix Let α be a pattern. If a sequence β is a super-sequence of α then the prefix of α in β is the smallest sequence of symbol in β that is still a super-sequence of α. The remains of β 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.

2.2 Existing specialised approaches

2.2.1 apriori

The Apriori algorithm, created in 1995 [1], was designed for frequent item-sets mining in a transactional databases.

This breath first search (BFS) algorithm, whose performance are now surpassed by more modern techniques, finds all frequent item-sets by iteratively growing it's sequential patterns.

First, at the start of each iteration, all length-N candidate will be generated (N being the number of the iteration) using the result of the previous iteration and the length-1 candidates as basis. The support of those candidates will then be checked with the databases and un-frequent pattern will be deleted. Finally frequent length-N 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 candidate using the length N-1 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 it's good point however, lied in it's ability to be used to detect association rules (Example : When A is present, B has 75% chance of also being present), which would give indications about the general trends in the database and allowed human operators to get a general understanding of the in-putted dataset.

2.2.2 GSP

The Generalized Sequential Pattern (GSP) algorithm [2] 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 length-1 pattern, it then proceed with generating all length-2 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 item 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 length-1 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 length-N found in the previous iterations, we detect it's first and last N-1 item, this creating two sub-sequential pattern by omitting an element either at the end or start. We will then create new 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 the 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 length-2 sequential pattern due to being un-frequent, we can delete all candidate sequential pattern containing BB, as they similarly cannot be present.
Are thus left with : A(AB), (AB)A, ABA, AAB, BAA

Finally, we will count the number of support for each candidate having passed the pruning phase, and remove unsupported sequential patterns. Another iteration of generating candidate, 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 length-4 pattern, no length-5 pattern will be generated and the execution will stop having determined that all solutions where 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.

2.2.3 Prefix-Span

The Prefix-Span approach [3, 4] 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.

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 length-1 sequential patterns & final list of sequential patterns eventually extended from those length-1 prefixes

Figure 1: Prefix-Span example

The main selling point of the algorithm lies in it's 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 it's 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 Prefix-Span.

However, techniques exists to break past this bottle-neck and mitigate it's effects. Techniques among which we should mention the 'Bi-level projection' technique, which consist in creating a triangular matrix (S-matrix [21]) registering the number of supports for all length-2 sequences that can be assembled from length-1 prefixes. A quick scan of the S-matrix then allows the detection of supported length-2 prefixes. They can then be extended again through the same technique after their projection on the database, thus allowing a reduction of the number of necessary prefix projection during the algorithm.

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.

2.2.4 cSPADE

The cSPADE algorithm [5] uses combinatorial properties to divide the original sequence pattern mining problem in smaller sub-problems that can be solved independently using efficient lattice search techniques, and simple id-list join operations. This algorithm is thus parallelizable, furthermore the parallelization has a linear scalability with respect to the size of the inputted database.

cSPADE's first step is to compute all length-1 sequential pattern using a simple database scan. Then, we generate all length-2 sequential pattern and count the number of supporting sequences for each pair of item in a bidimensional matrix. Counting the number of supporting sequence being realised through a single database scan by first transforming the original vertical representation of the database into an horizontal representation (see Figure **??**).

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

(a) Vertical representation of the database

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)

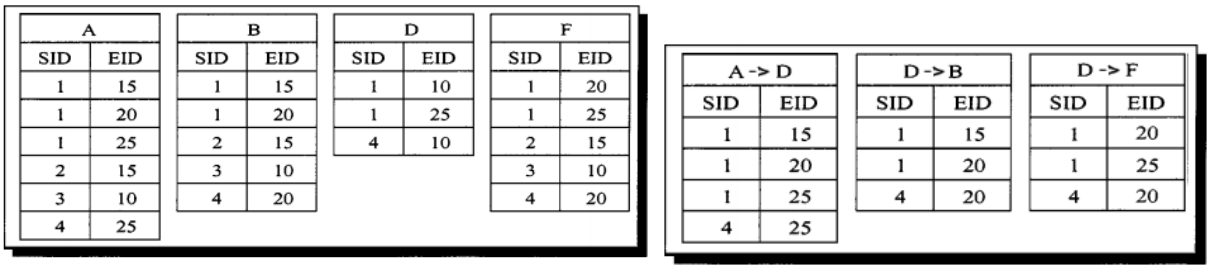
(b) Horizontal representation of the database

Figure 2: cSPADE database representation

Subsequent length-n sequence patterns can then be formed by joining length-(n-1) 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 length-3 patterns onward, as ID-lists for length 1 and 2 patterns can be extremely large and potentially won't fit in memory.

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



A	
SID	EID
1	15
1	20
1	25
2	15
3	10
4	25

B	
SID	EID
1	15
1	20
2	15
3	10
4	20

D	
SID	EID
1	10
1	25
4	10

F	
SID	EID
1	20
1	25
2	15
3	10
4	20

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

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

D -> B	
SID	EID
1	15
1	20
4	20

D -> F	
SID	EID
1	20
1	25
4	20

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

Figure 3: cSPADE's ID-lists

2.3 Existing CP Based approaches

2.3.1 CPSM

The CPSM approach [11] 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 allowing a search for frequent pattern over the database. Thanks to it's integration in CP solvers, it also easily support additional constraints such as size constraint or regular-expressions constraints.

The global constraint, which form of the basis of this approach, is called the global exists-embedding constraint. Based on an incremental propagator (DFS), 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 it's 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 it's 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 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, symbol that are considered unfrequent won't be branched over during the execution. This pruning is done by counting the number of support for each extensions during the embedding existence verification.

Although impressive for it's modularity, the efficiency of this algorithm didn't quite caught up to specialised non-CP algorithm such as Prefix-Span or cSPADE. It however surpassed their efficiency at searching for specific solution thanks to it's modularity.

2.3.2 PP

This second approach based on CPSM proposed a slightly different global constraint by taking idea from prefix-projection [9]. Similarly to it's 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 it's 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 sequence, making it extremely memory and time efficient.

During the extensions pruning phase, only supportive sequences will also be checked, as they are the only relevant sequence to find extensions with.

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

2.3.3 Gap-Seq

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

Similarly to it's predecessor, the algorithm was designed to solve sequential patterns mining problems involving sequences of sets 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 constraint cannot be taken into account.

2.3.4 PPIC

Largely based on it's predecessor, CPSM and GAP, PPIC is an algorithm designed for solving sequence pattern mining problems involving sequences of sets of symbols. Unlike Gap-seq, constraint over time gaps are not supported.

This new algorithm speciality laying in it's record breaking efficiency, surpassing even state-of-the-art non-CP solver, generally by a wide margin (see Figure 22).

PPIC's execution can be separated in two stages :

1. **Pre-processing :** In this first stage, we first clean the received sequences from unfrequent items, renaming them into unique ID's. 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 are 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 realise that to achieve the same goal a whole columns 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 constraint, depending on the wishes of the user. Thus restraining the solution space as desired and improving performances at reaching 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 prefixe while projecting the current extension. For further extensions of the prefix, only those sequence will thus have to be searched to find projection or to prune un-frequent extensions. Since the algorithm also keep track of the number of support for each extensions, an improvement was also made to stop searching for projections once all such that number of supporting sequences have been found during the prefix's projection. Since the further sequence 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 items name corresponding to the recorded ID's 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 [1, 2].

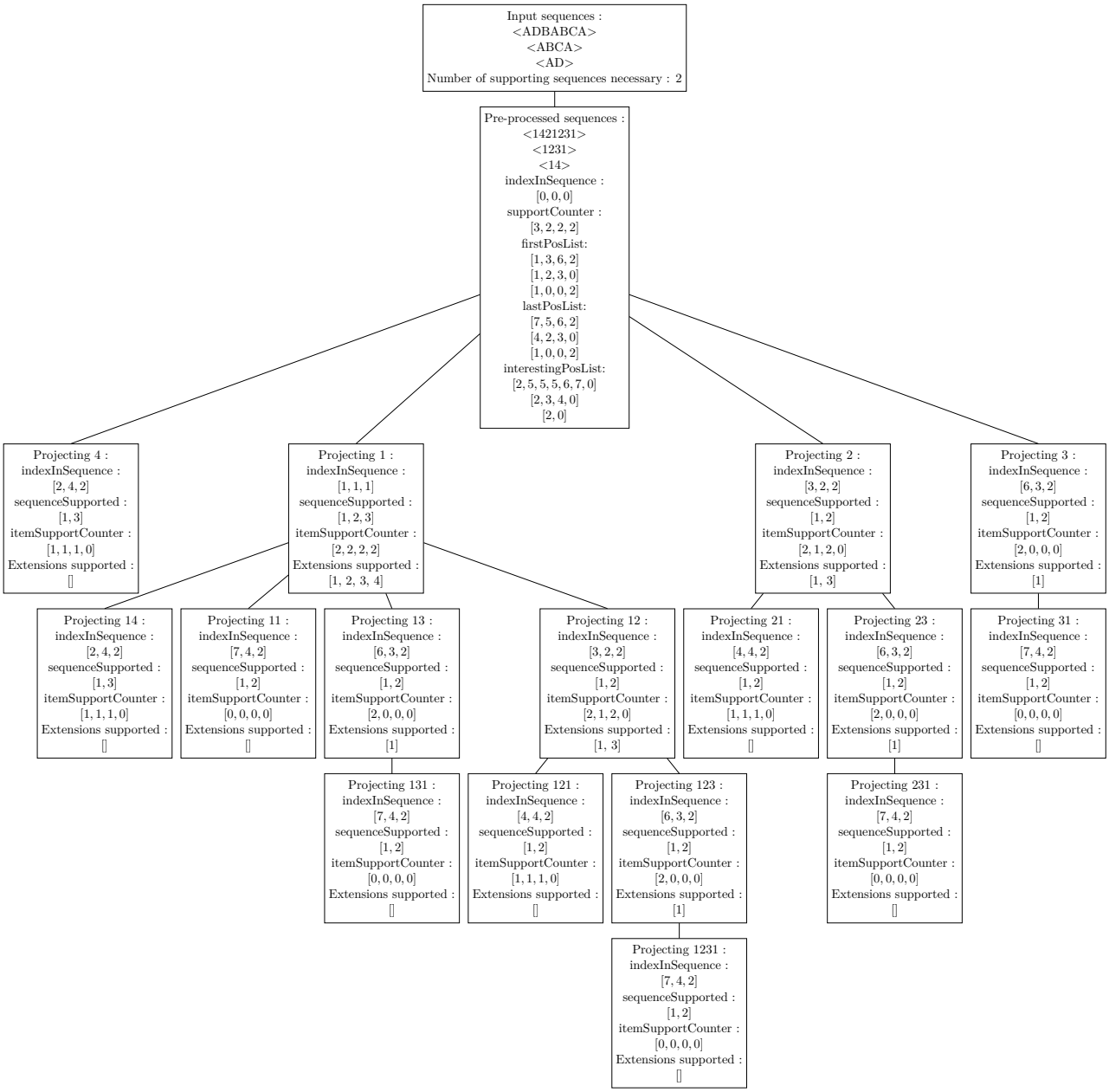


Figure 4: A simple execution of PPIC's algorithm.
The solutions patterns are the projected prefixes

2.4 Parallelisation

2.4.1 The Benefits of Parallelisation

2.4.2 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

2.4.2.1 Hadoop

Hadoop is an open-source framework that allows large-scale data processing across clusters of machine. Based on a Map-reduce programming model, this framework allows larger scale iterative computation on humongous quantities of data. At each iteration, the data is read from the distributed file systm (HDFS), modified through a MapReduce, then stored back on the file system.

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

Widely used since it's Initial release in December 10, 2011. Hadoop slowly climbed to become 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 Prefix-Span on Hadoop was already available on the internet.

2.4.2.2 Spark

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

Originally developed at UC Berkeley in 2009, It's 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 problem in scalable environments.

Furthermore, Spark's has been designed to make use of it's clusters RAM memory efficiently, allowing the engine 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 Prefix-span available in Spark's machine learning library.

2.4.2.3 Final choice

As said earlier, both of those libraries already disposing of a scalable PrefixSpan implementation. 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 [**?**]. Those performances are presented in Figure 5

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

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 implementations where only required

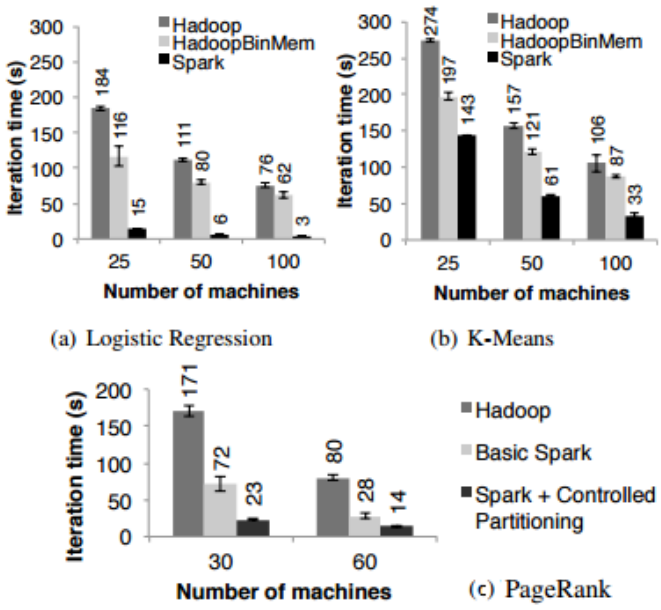


Figure 5: Performance comparison of Hadoop and Spark

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. Be it through the addition of CP based algorithms and techniques, or more directly.

The performance of those algorithm will however be tested in a later section.

3.1 Spark's original implementation

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

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

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

   (a) First, a large prefix is projected on the database. If there is no prefix to project, the scalable execution comes to an end and the solutions will be returned.

   (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 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 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 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 databases needed for future extensions small enough. Depending on the parameters inputted by the user, this stage may be skipped. TODO: add performance difference of keeping database on shuffle file.

4. **Post-processing :** During the post-processing step, we translate back the unique IDs into the item 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 symbol 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 a single-item pattern mining problem, no partial start would be created and kept in memory, since the algorithm automatically detect that those item-sets cannot be extended.

If the item-set end, the full remains of the sequence, from the earliest partial position recorded, will have to be searched. During this search, 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 itemset quicker. The remainder of the database will also be searched, but only for extension that starts new itemsets.

An example of a fully scalable execution from this algorithm can be found in figure 6. A pseudo can also be found in Algorithm [3, 4, 5].

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 itemsets were fully cleaned of their item, the algorithm had a tendency of creating sequences of zero delimiters, as the algorithm still delimited empty itemsets.

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 **??**.

Later, this small correction was proposed to Spark's community and quickly accepted into the default implementation. In the remains of this paper, we will 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.

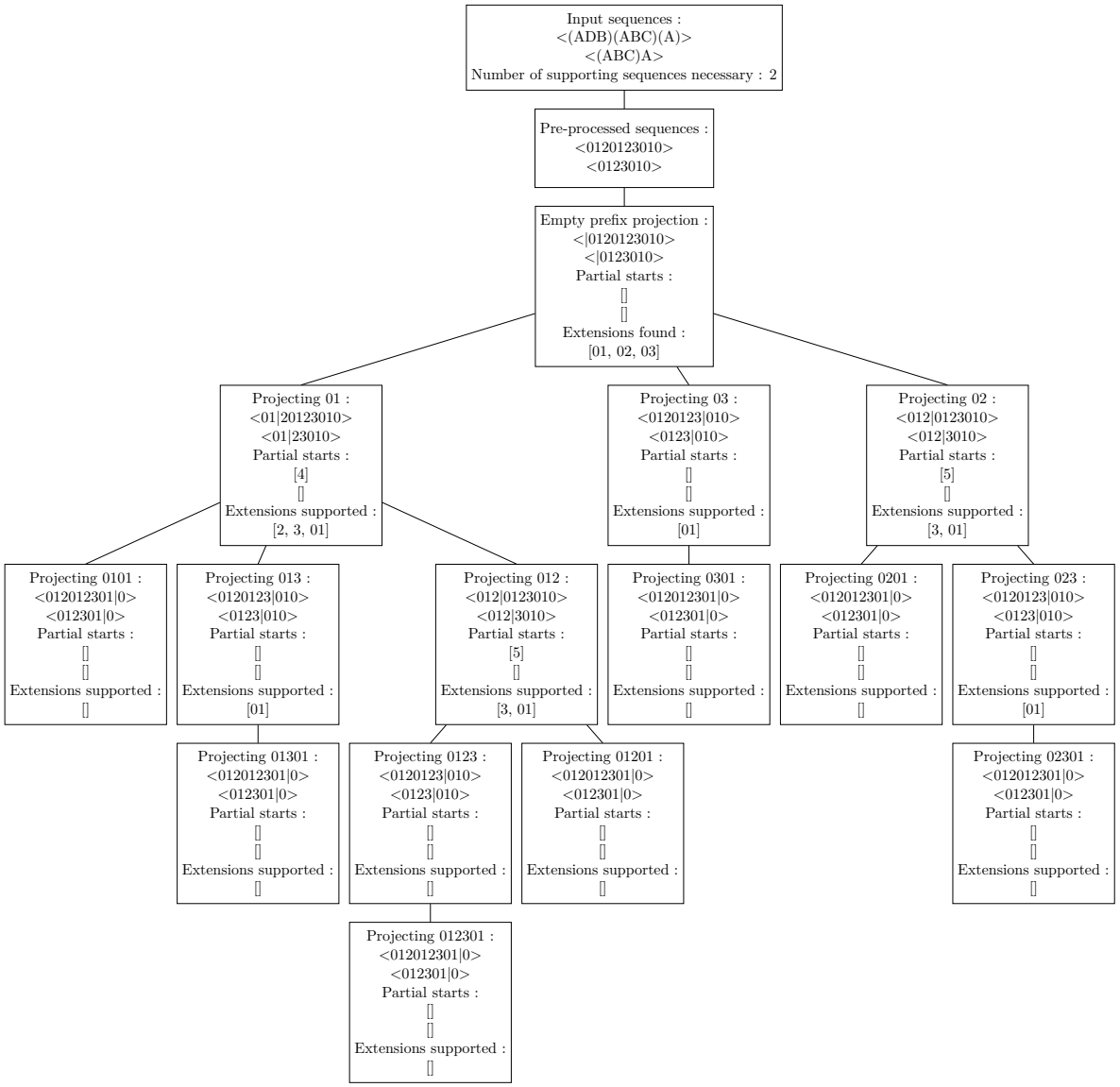


Figure 6: A simple execution of Spark's algorithm.
The solutions are the projected prefixes (except the empty prefix)

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 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 user could specify whether PPIC could be used on the input dataset.

This implementation's pseudo-code can be found below, in Algorithm 7.

3.3 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 algorithm have been separately optimised in terms of performances.

2. Improve Spark's performances by incorporating more idea's 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.

3.4 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 follow :

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 pattern containing less non-zero item that the specified input wouldn't be in-putted, we decided to set the default value of this parameter at 1, so that solution wouldn't be restricted.

3. Limit on the maximal number of item per item-set : This functionality was added so that user could, once again, better control their outputted results. Supposing an hypothetical business would like to find all sequences of item bought in pair 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 parameters 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-problem 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 you can see in Figure 24, 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-problems sizes appearing due to the forced local execution. Since the largest problems tend to be created and processed last, only a few executor 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 will consecrate a section to studying possible improvement of its performances later in this paper. TODO : add link to that section.

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 8.
We then looked into improving our performances, leading the the discovery of two potential inefficiencies.

3.4.1 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-item directly to the scalable stage, instead of discarding them before searching for them once more through a complete iteration over the database.

The code was modified accordingly, as shown in Algorithm 9.

3.4.2 Cleaning Sequence before the Local Execution

The next inefficiency we found was that, during PPIC's local execution, three matrices were build 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 databases of the local execution.

We thus modified our code accordingly, producing a code similar to Algorithm 10

Since, as we will 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 improvement were thus added separately to this version, later allowing us to better compare the performance gains brought by each implementation.

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

3.5 Improving the Switch to a CP Local Execution

In this section, we will 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 will explain its nature and the trade-off's its implementation may encompass.

3.5.1 Automatic Choice of the Local Execution Algorithm

Our next attempted improvement was to automatise to 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.

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

Additionally, should maxItemPerItemSet be left to it's default value or should it have been put to a much too high value, we could theoretically slightly improve performance by refraining from searching extensions to solution pattern 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 do not vary too much.

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

Of course, should the non-recalculated parameters be in use, only the requested solutions would be computed.

A pseudo-code demonstrating the implementation of this Automatic choice can be found in Algorithm 11

3.5.2 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 SoS 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 it's performance gain to our previous maxItemPetItemSet based implementation.

While creating this new implementation, we also discovered more than a few inefficiencies 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 type of problems and, hopefully, for every dataset.

A pseudo-code representing this implementation can be found in Algorithm 12.

3.6 Improving the Scalable Execution

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

3.6.1 Position lists

The first idea we had to improve Spark's performances, was to use LAPIN'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 solution 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 import use of memory, as the positions list would need to be kept on RDD along side sequences. To adopt this solution, the improvement would thus need to be important enough to motivate the benefits of the trade-off.

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

In Algorithm 13, you will find a pseudo-code of the implementation including both positions lists together.

3.6.2 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 internal representation could be seriously improved by removing unnecessary separator. Effectively reducing the size of their internal representation by two.

We thus implemented a new scalable stage specialised for sequence of symbols problems. It's 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. We needed to create a different database for each type and simply removing the delimiters after a check wouldn't have been 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 we could efficiently detect whether each sequence could be use the shortened representation.

The problem being that sequence which could be solved by a sequence of symbols problems after cleaning would never be detected. However, since this case rarely appear over a whole database, we decided checking before cleaning would be sufficient for our purposes.

We thus put performance forward, implementing the code found in Algorithm 14.

3.6.3 Priority Scheduling for Sub-Problems

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

In the original code, problem 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 created sub-problem.

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 would rarely appear in the original version. Coincidentally, we also realised that major drops in performance were experienced if those large problem were solved last, since some executor would be left with nothing to do while other would be stuck with atomic 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.

3.6.3.1 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).

3.6.3.2 Sort sub-problem on the reducer

We thus concluded 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 15.

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 executor.

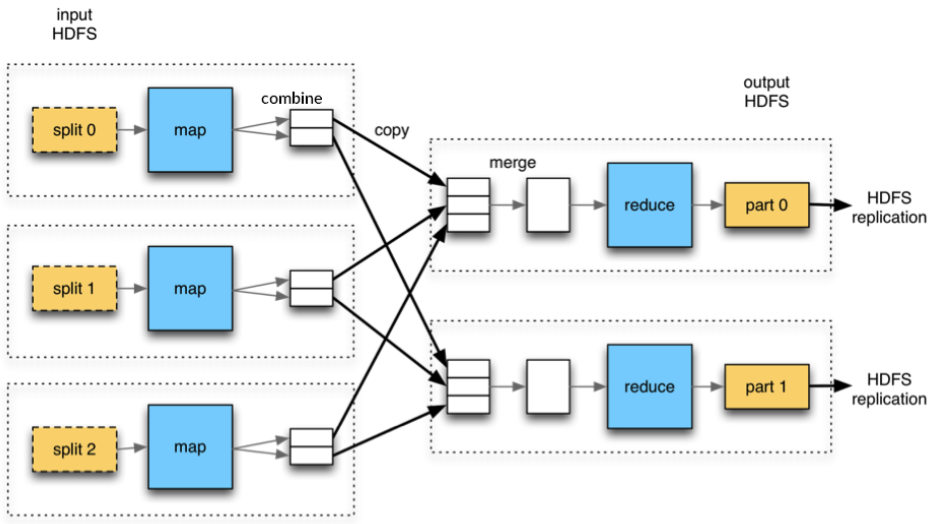


Figure 7: Spark's mapReduce

As it turns out, to sort the various sub-problems depending on their size, Spark's obviously needs to evaluate and hold in memory all sub-problems assigned to an executor. Only then could Spark sort them, all the while while keeping those sub-problems either on disk or memory.

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-problem and keeping them in memory..

3.6.3.3 Sort sub-problem on the mapper

We thus realised we needed to sort our sub-problems during the mapping phase of the mapReduce. After a few trial and error, we realised that the mapping function of Spark created sub-problem 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 RDD 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 16

3.7 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 it's predecessor.

3.7.1 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 key symbol as value.

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

This new structure's purpose was to allow us to make distant jumps and checks more efficiently, at the cost of an higher memory consumption. Theoretically, finding the next position of a given symbol would be O(1), be this next position the last or first position of the symbol. Should the 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 such symbol were contained in the remains of the sequence.

The trade-off lied in the number of reversible point 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 17.

3.7.2 Adding Partial Projections to PPIC

We 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 to be used in a PPIC like algorithm.

First we realised that keeping all three matrices wouldn't be efficient. In a set of symbol 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 positions lists, as they remained relevant. 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 position before the next separator, thus we expect greater performance improvement the longer item-sets could extended.

TODO : Pseudo code

4 Performances

To compare the performances of our various implementations, we first need to discuss how those performance 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 you 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 performance 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 much 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 performances performance can remain comparable while guaranteeing a complete absence of shuffler crash due to large partitions.

The number of partitions created for each dataset are displayed in Table 2. As you 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 client deploy mode (not locally). More specifically, during our tests, we will run a simple architecture composed single master and a worker with four executor. 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, scalability performances will also be tested in larger, memory-restricted, architecture. The architecture then used shall be specified at the beginning of the concerned section.

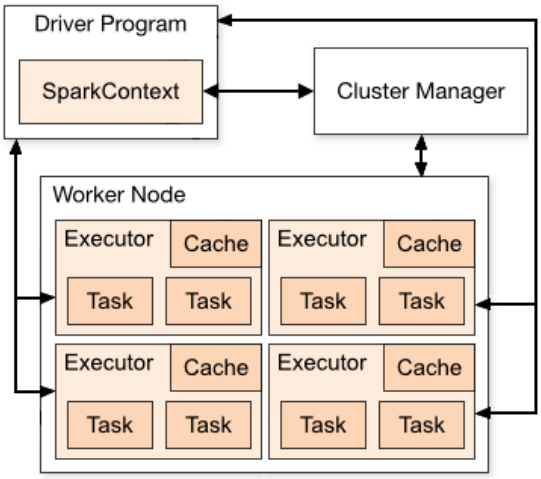


Figure 8: The simple architecture used during the majority of our tests

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 prevent 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.

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.

NB: As you will see further down in our performances tests, a section has been dedicated to comparing the performance of our final algorithm under default and optimal parameters. As you will see, performance improvement can be quite substantial.

TODO : Prove it

4.3.3 Measurement Span

For our performance tests, we will 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

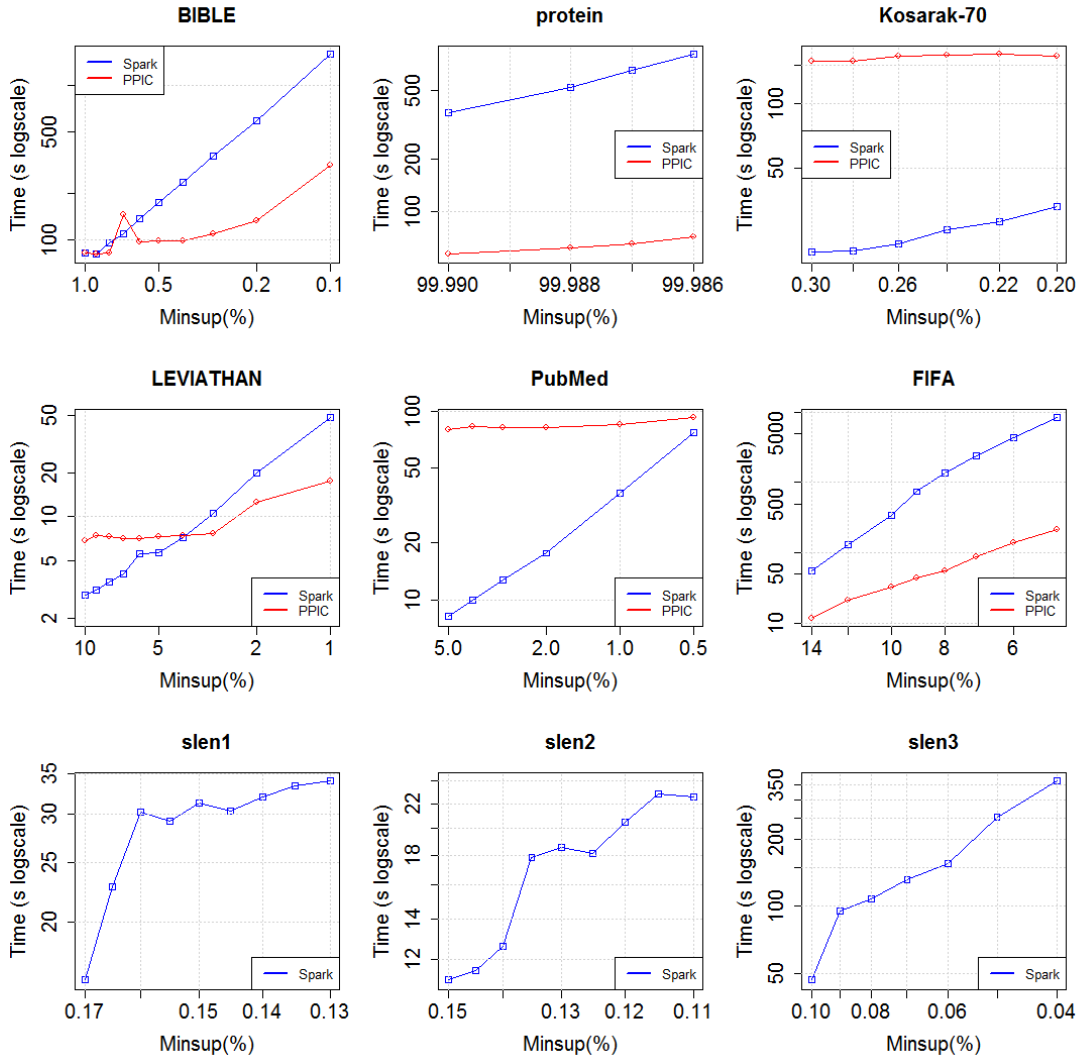
4.4.1 Comparing Spark and PPIC's original implementations

Since PPIC's is neither scalable nor concurrent, we will exceptionally use and 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 you can see in Figure 9, given the same resources, PPIC greatly overcomes Spark on most datasets. But, although it completely outperforms Spark's implementation on sparser dataset like protein, it fails to do so on denser datasets like Kosarak-70 or PubMed. Especially when the

minimal number of supporting sequence is large, and much of the dataset can be cleaned during the pre-processing stage.

We can also notice that PPIC's performance are far more stable, and that running additional tests for a smaller amount of minimal support may even show on inversion on denser datasets like PubMed.



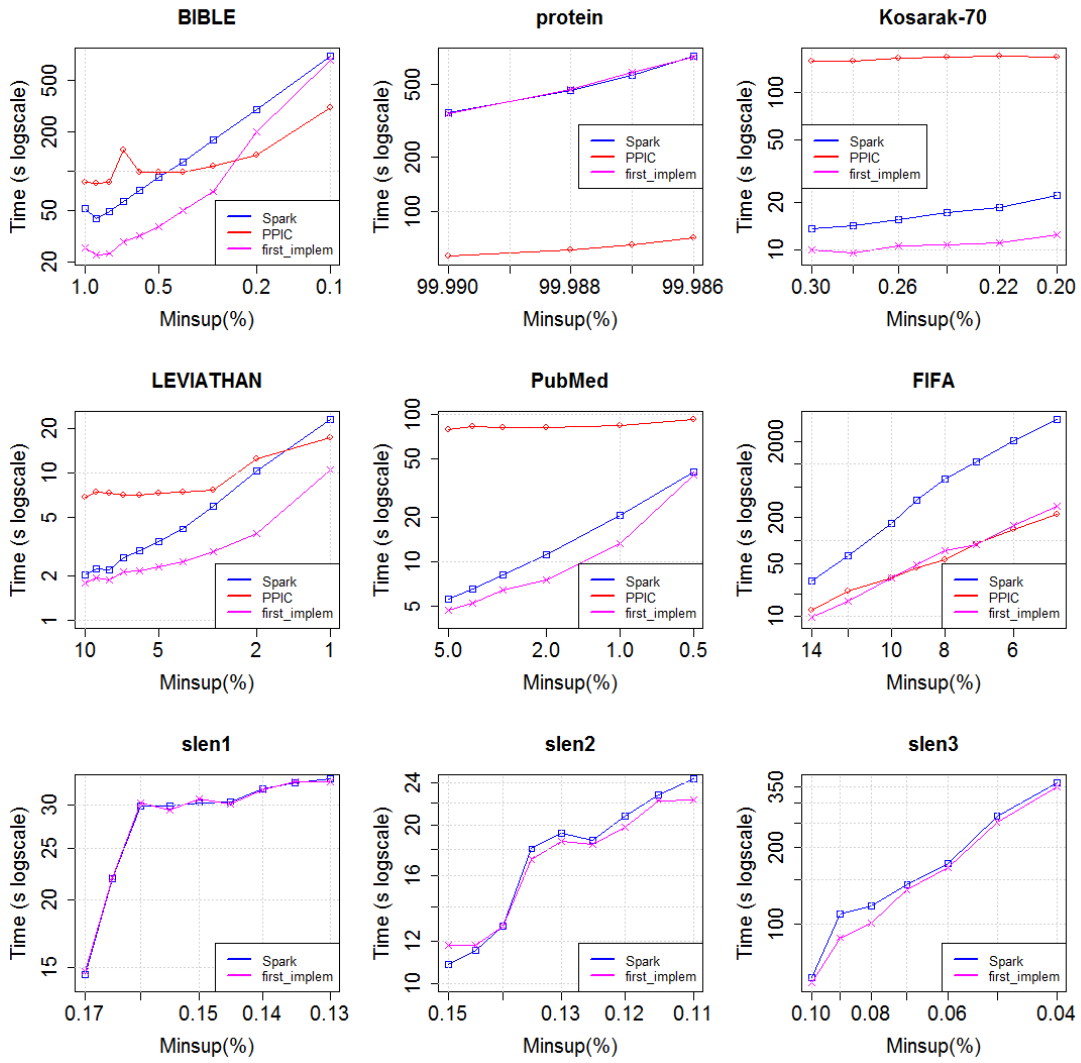
⚠ Test realised on a simplified architecture ⚠

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

4.4.2 Performances of our First CP Based Implementation

As you can see in figure 10, this first implementation, which simple merges the two algorithm 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 amount of resources of which it disposes. The worst such case being protein, a dataset on which



⚠ PPIC's performance 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.

Figure 10: A first implementation that makes PPIC scalable

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 amount of required support isn't really felt by the algorithm. On Kosarak and PubMed, their performance remain worse by a 10x factor which cannot be explained by the lack of resources alone.

In fact, those performances actually result from three major factors :

1. First, the pre-processing of the original PPIC algorithm wasn't implemented efficiently, probably since only execution time were 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.

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 item present in the input database.

   However, our first-implementation surprisingly 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 explain further our new implementation's efficiency gain on dataset with large amount of distinct items.

3. Finally, our third major factor is a major inefficiency that comes as a by-product of scalability. It appear mostly on smaller datasets composed of a small number of repeated symbols, such as our protein dataset. In those cases, since the various sub-problem created before our local execution are actually very similar to each other, our new implementation will lose and 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 comparison, 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 unpractical 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 you can see, very small gain in time are generally made. At 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 loss of performance 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.

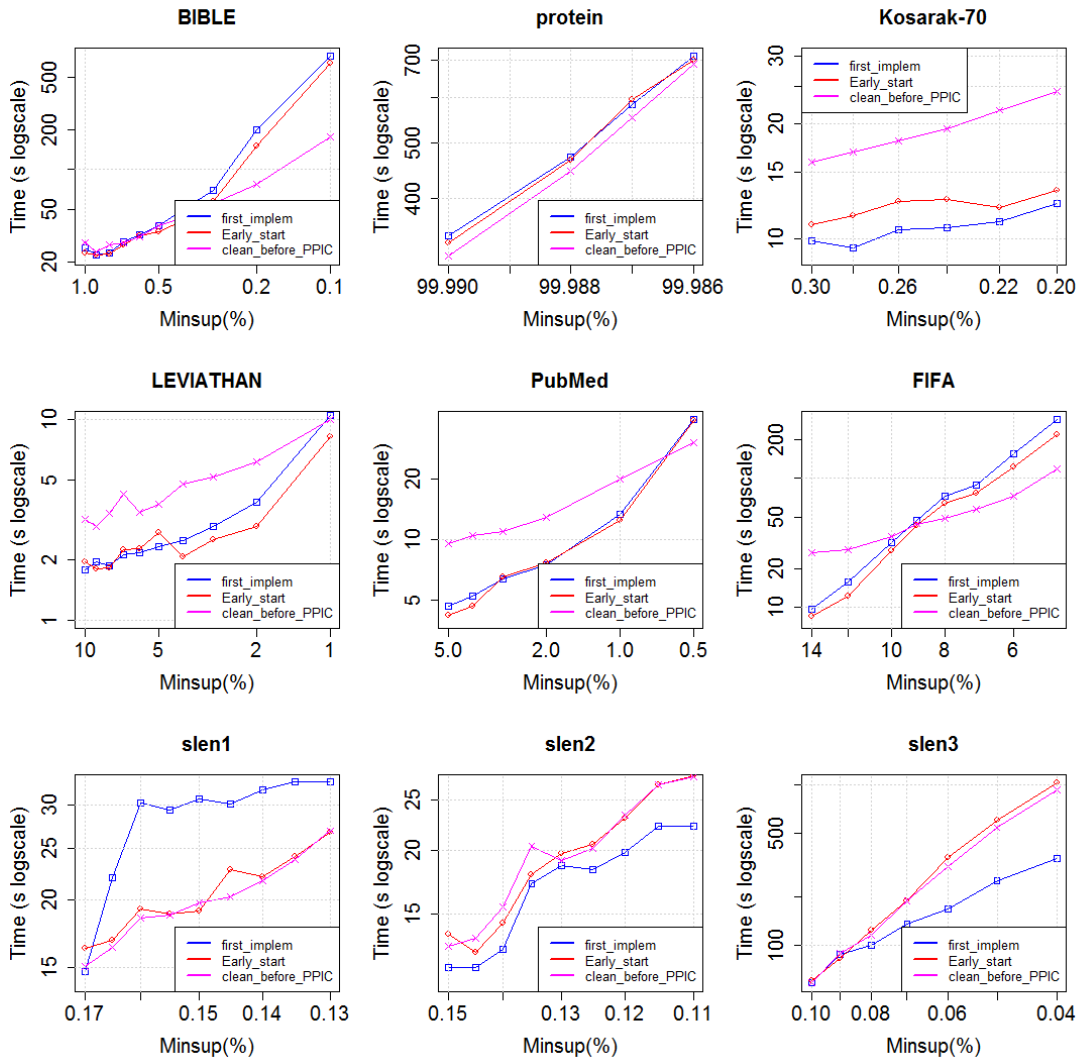


Figure 11: The performance impact of not re-calculating frequent items

GRAPH ONLY WITH QUICKER-START

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 object 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 received the items and their frequencies and create the prefixes. Especially when, as in our slen3 dataset which is a combination of multiple generated datasets, every partition generally concern 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 inneficiency 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

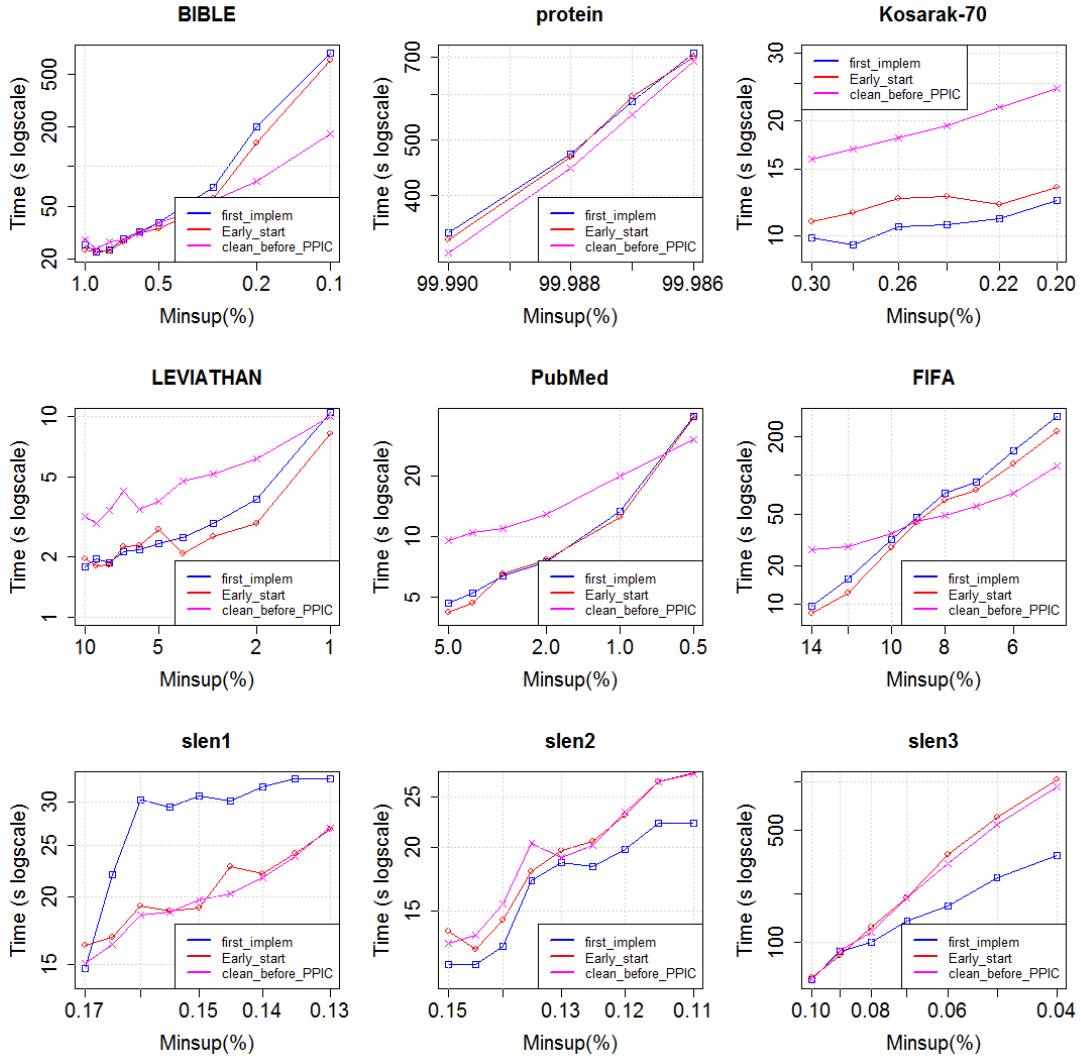


Figure 12: The performance impact of Database Pre-processing Before PPIC's Execution

As you 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 gain were also observed on protein and the latter stages of PubMed.

However, we can observe worsening performances in smaller datasets and in executions with larger minsup values. Theses loss 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 can be counted in hundred of seconds on Bible and Fifa.

The reason being those performance gain 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 remain of those very sequences for patterns. Finally, if less 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 gain/losses, all further improvement 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. Of whose the performances will be tested.

4.4.5 Performances with Automatic choice of Local Execution

As you 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 loss where 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.

4.4.6 Performances of Adding Pre-processing Before Any Local Execution

As you 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 much advantage of this additional cleaning step. The various improvement we made for cleaning sequence of symbols, on the other hand, are quite strongly reflected in these performance measurement, surpassing our expectation on their effectiveness.

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

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

4.4.7 Performances with Position lists

As you can see in Figure 15, the resulting performance of those three implementation 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

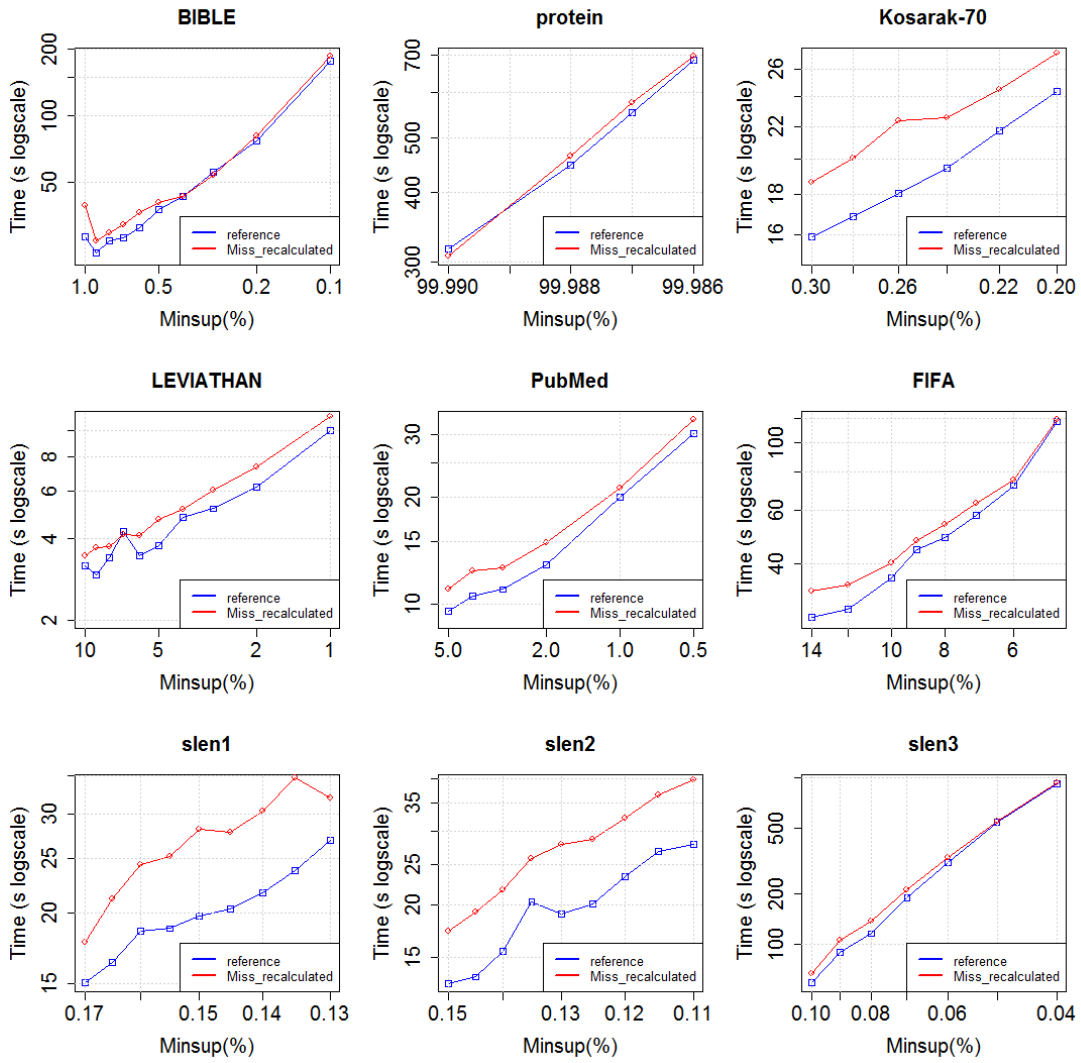


Figure 13: Performance improvement of recalculating maxItemPerItemset to avoid search for multi-item extension when unnecessary

than our reference algorithm.

Those worsening performance appear 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 database 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 performance improve significantly.

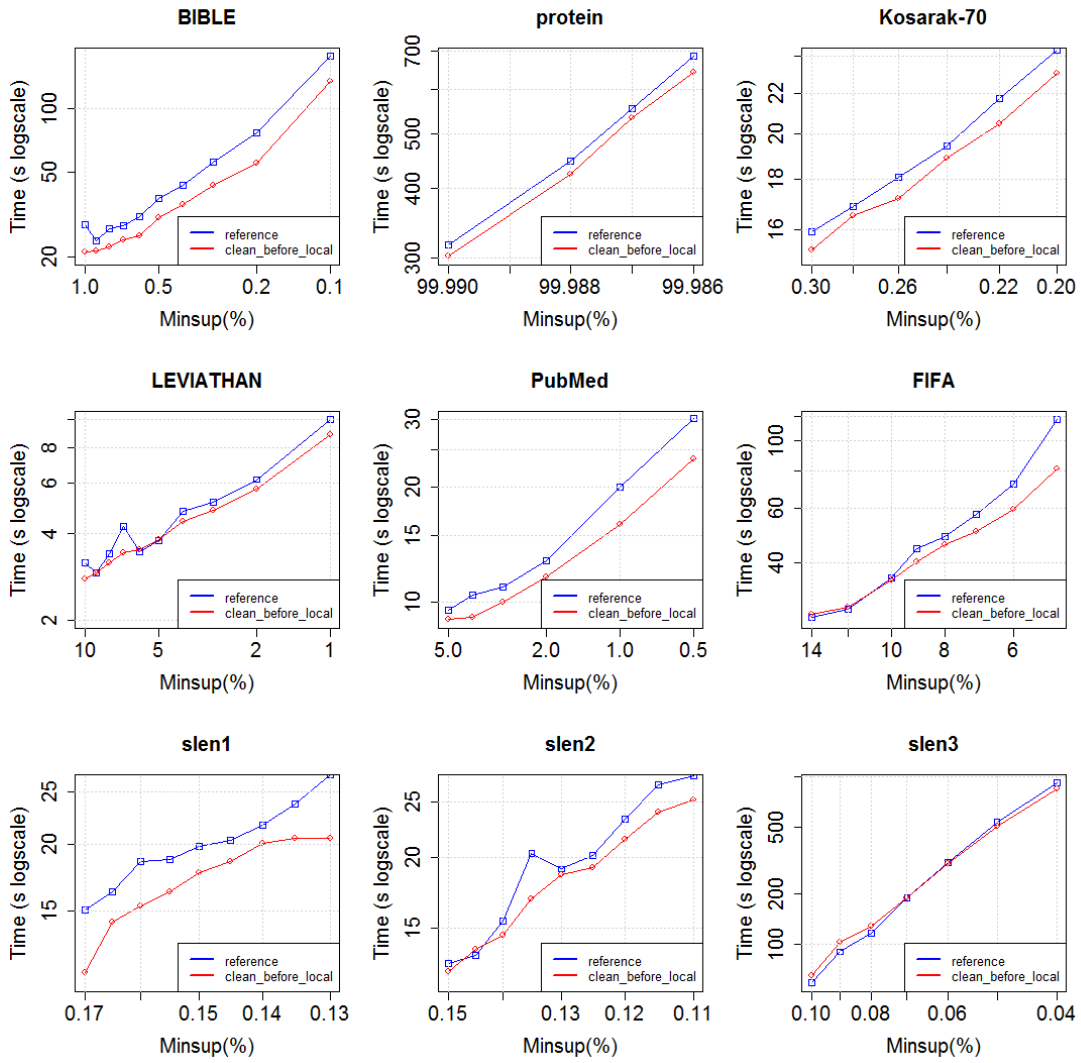


Figure 14: Efficiency gain of cleaning the sequence database before any local execution

However, at the exception of the slen1 dataset, the measured performance do not justify a doubled memory consumption (tripled in the case of the 'first_last_pos' improvement). Should no better implementation be found, the improvement could be restrained to only appear in Spark's local execution. Although performance improvement would be lesser, they would make more sense for a memory consumption point of vue.

4.4.8 Performances with Specialized scalable execution

As you can see on the slen datasets of Figure 16, this improvement 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.

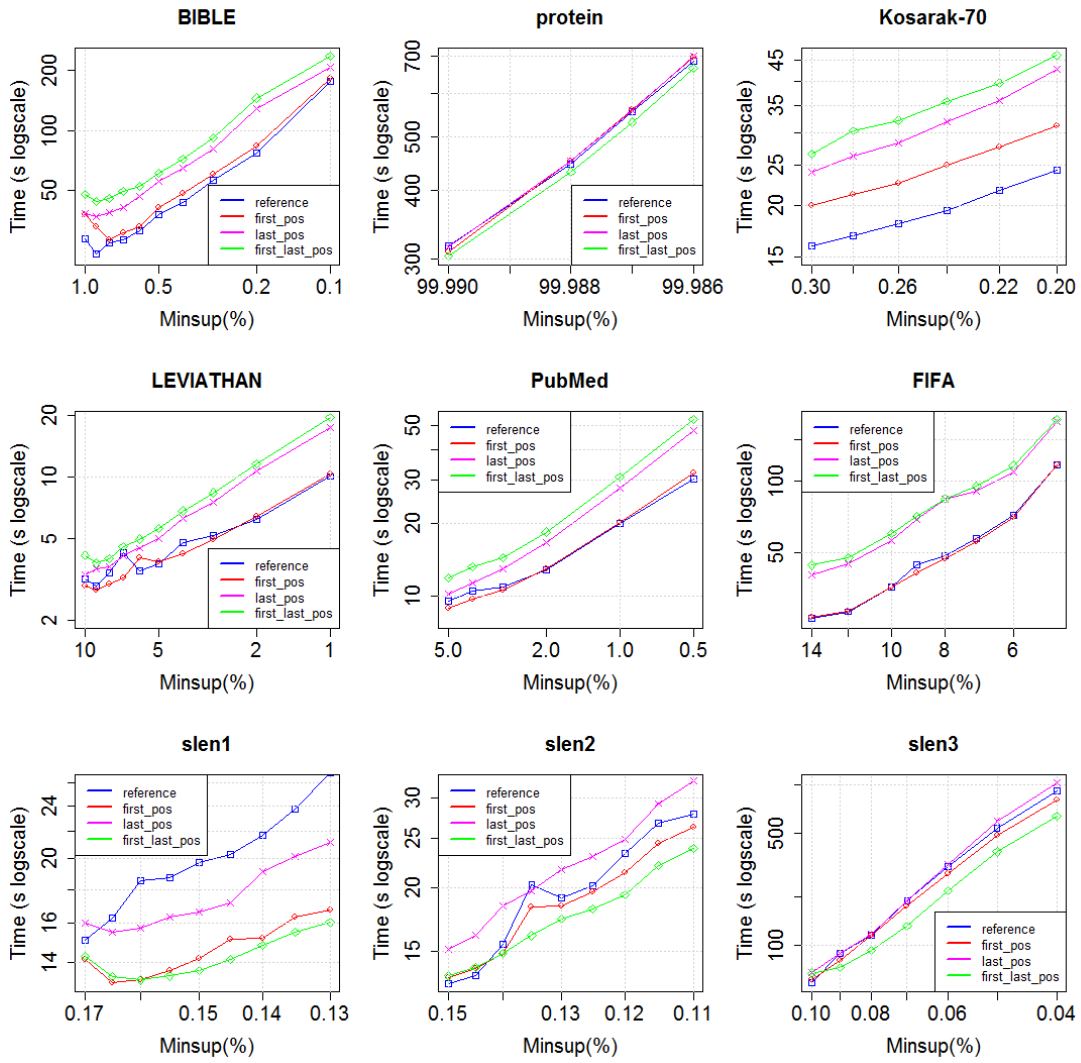


Figure 15: Performance gain of first/last position lists

4.4.9 Performances of Using Priority Scheduling for the Local Execution

4.4.9.1 Performances of Sorting Sub-Problems on the Reducer

As you 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 loss 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 you can see on protein's measurements, the black line lack its fourth point. The reason being a constant crash of the algorithm due to a lack of memory !

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

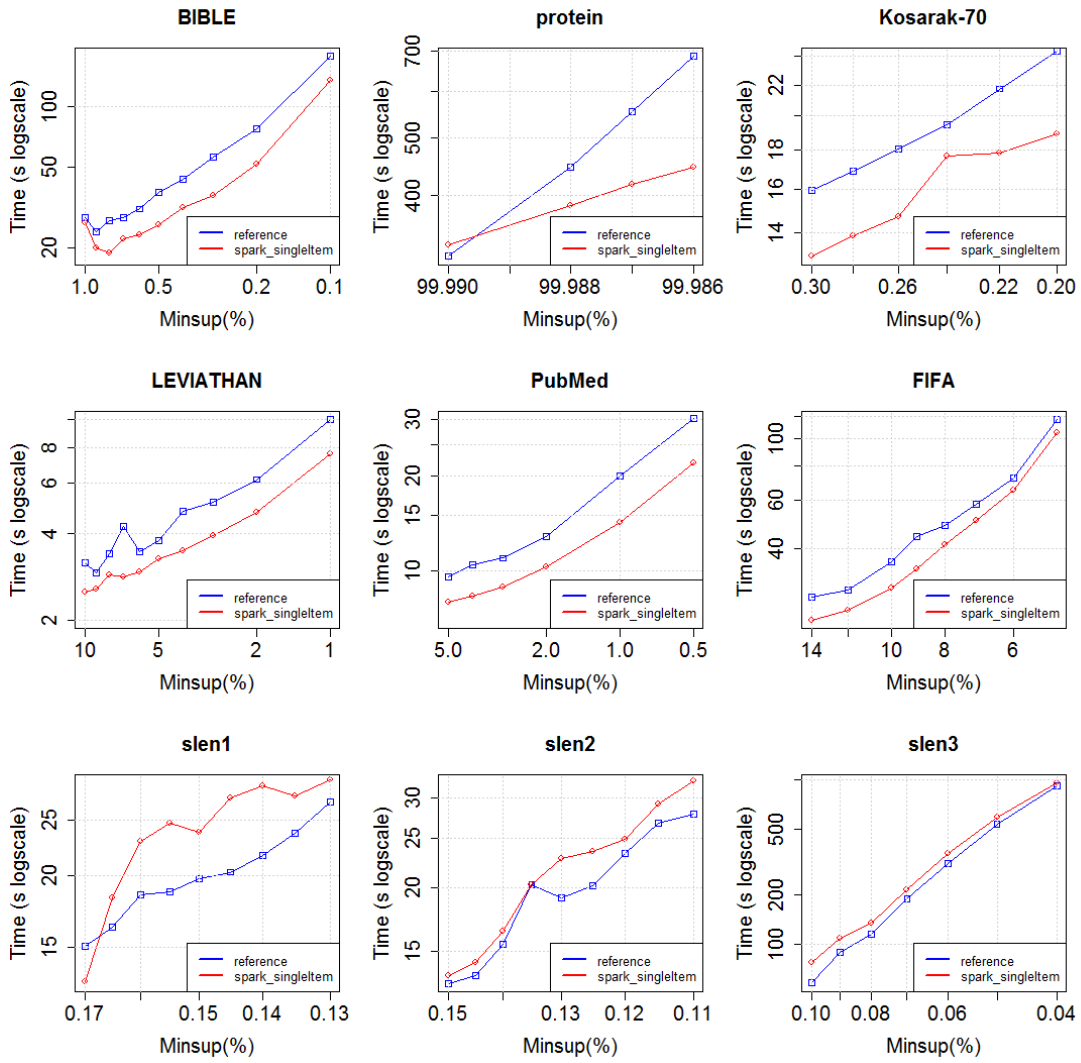


Figure 16: Performance improvement of specialising spark's scalable stage.

4.4.9.2 Performances of Sorting Sub-Problems on the Mapper

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

Not only have we obtained our previous boost in performance, we also 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

As you can see in Figure 19, the performance of this implementation were terrible.

Profiling our algorithm revealed it came from two majors factor. 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 period of time were our algorithm did nothing but backtrack the various sequences of its database.

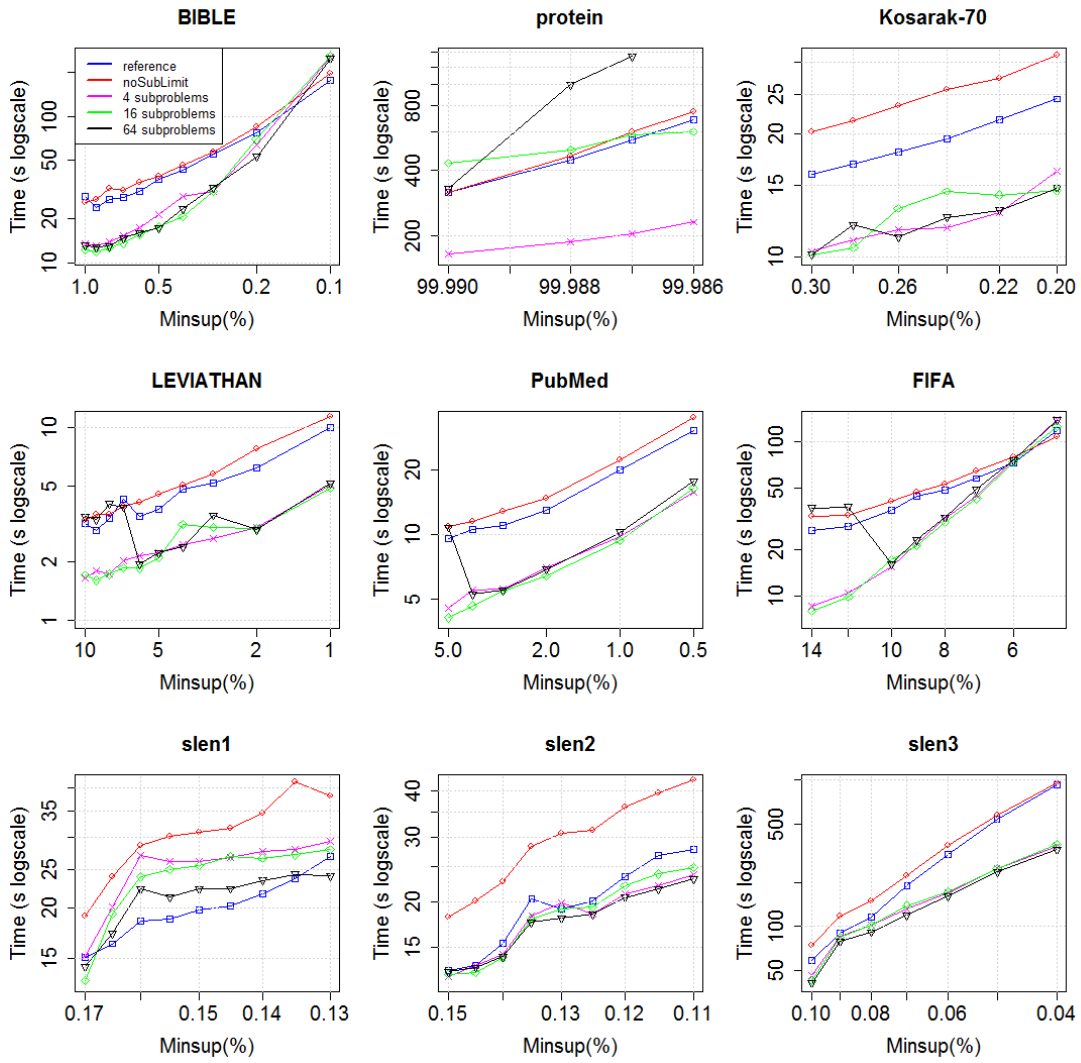


Figure 17: Performance of a 'biggest problems first' scheduling executed through the default sortBy() function of Spark.

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 last positions was no longer sufficient. To prune efficiently, each itemSet now needs to be checked so that its possible extensions could be deemed supported.

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 25.

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

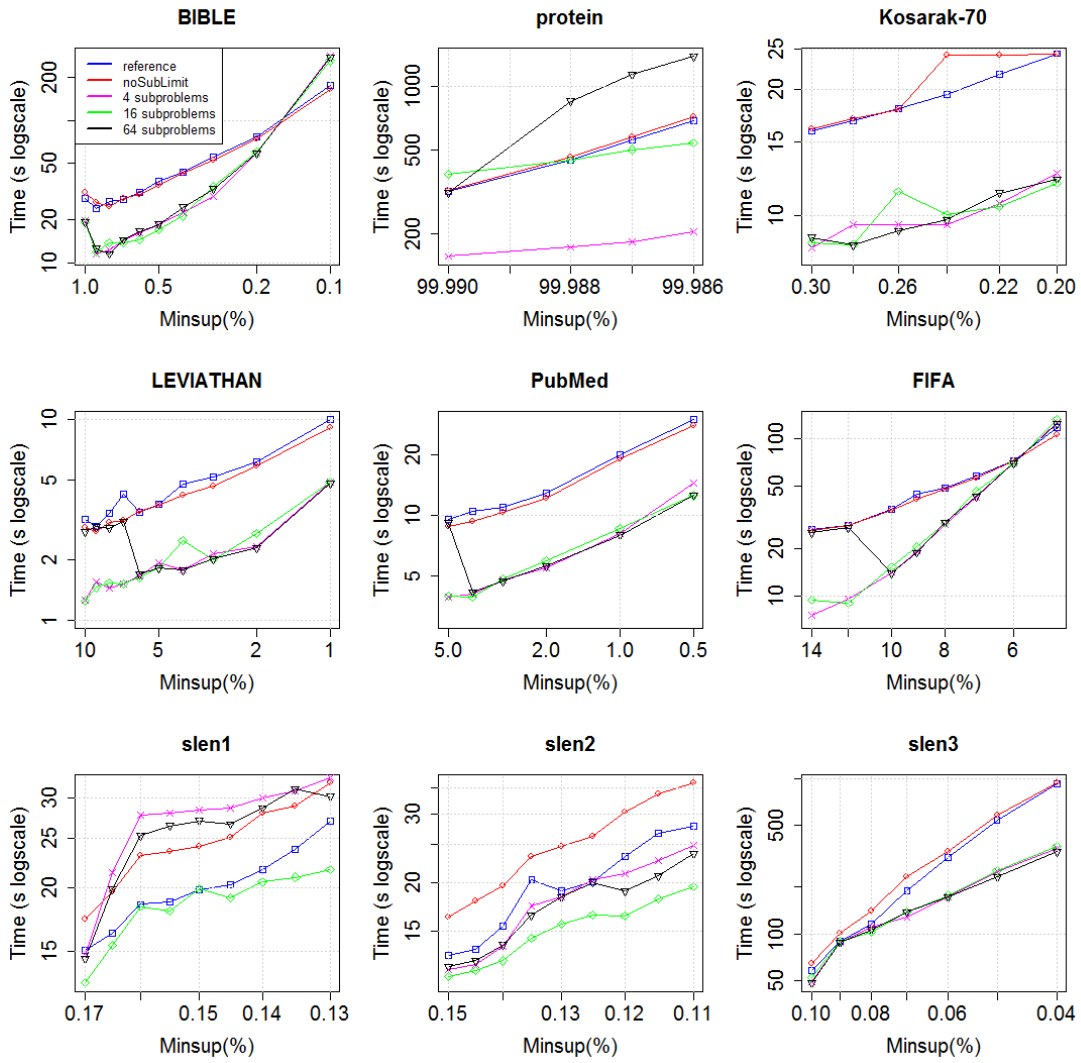


Figure 18: Performance improvement of sorting sub-problems during map stage.

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 original algorithm and our previously created map algorithm on sequence of sets of symbols. On slen3, this implementation was even much more efficient than anything obtained earlier through positions lists.

These performance however came at the cost of an inefficiency on dataset of sequences of symbols, where performances fails 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.

4.5 Scalability Tests

5 Conclusion

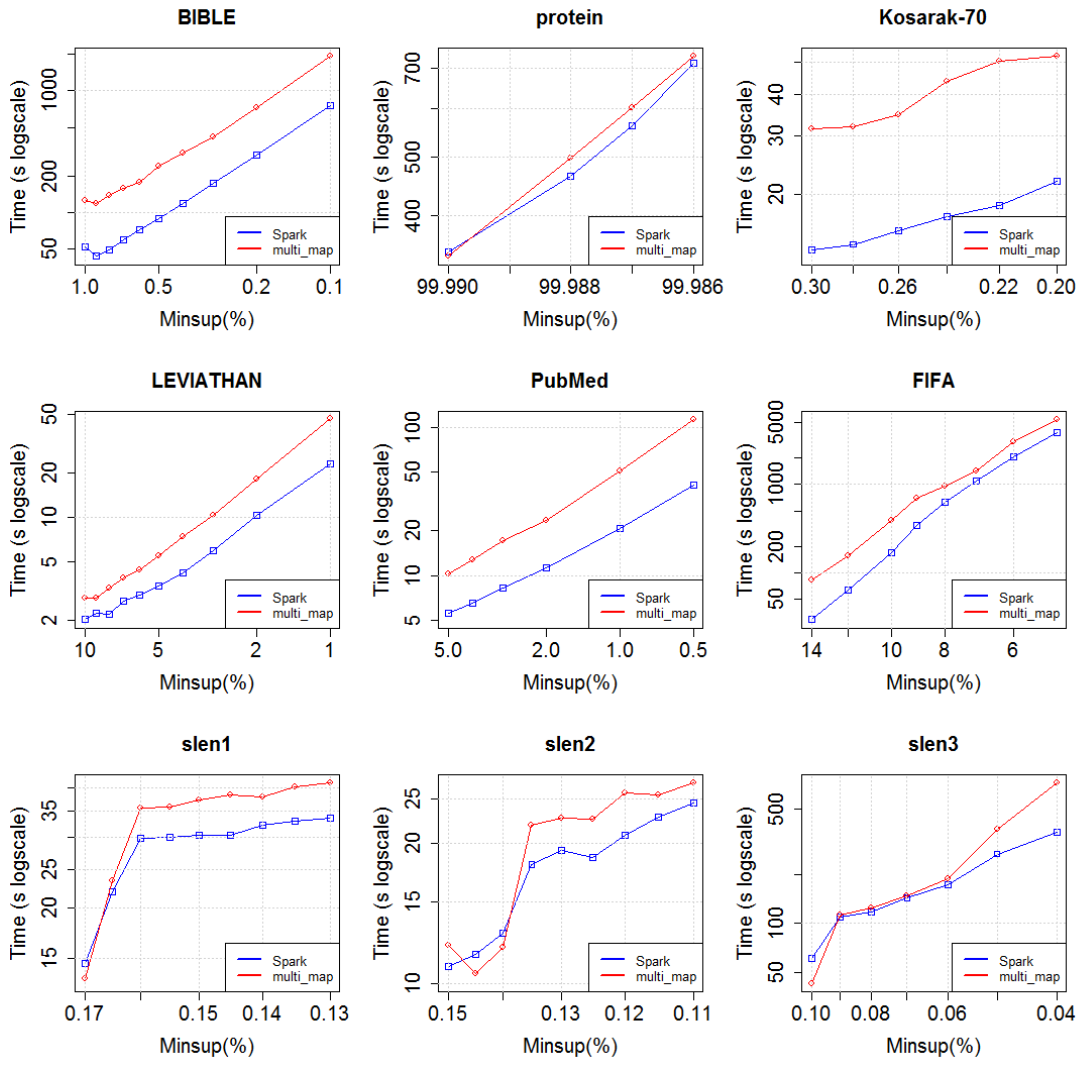


Figure 19: Performance of a CP based algorithm replacing all three pre-processed matrices from PPIC by a Map structure
(Please take notice that we are here comparing to the original Spark implementation instead of the reference algorithm. Since the featured implementation doesn't posses the early-start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

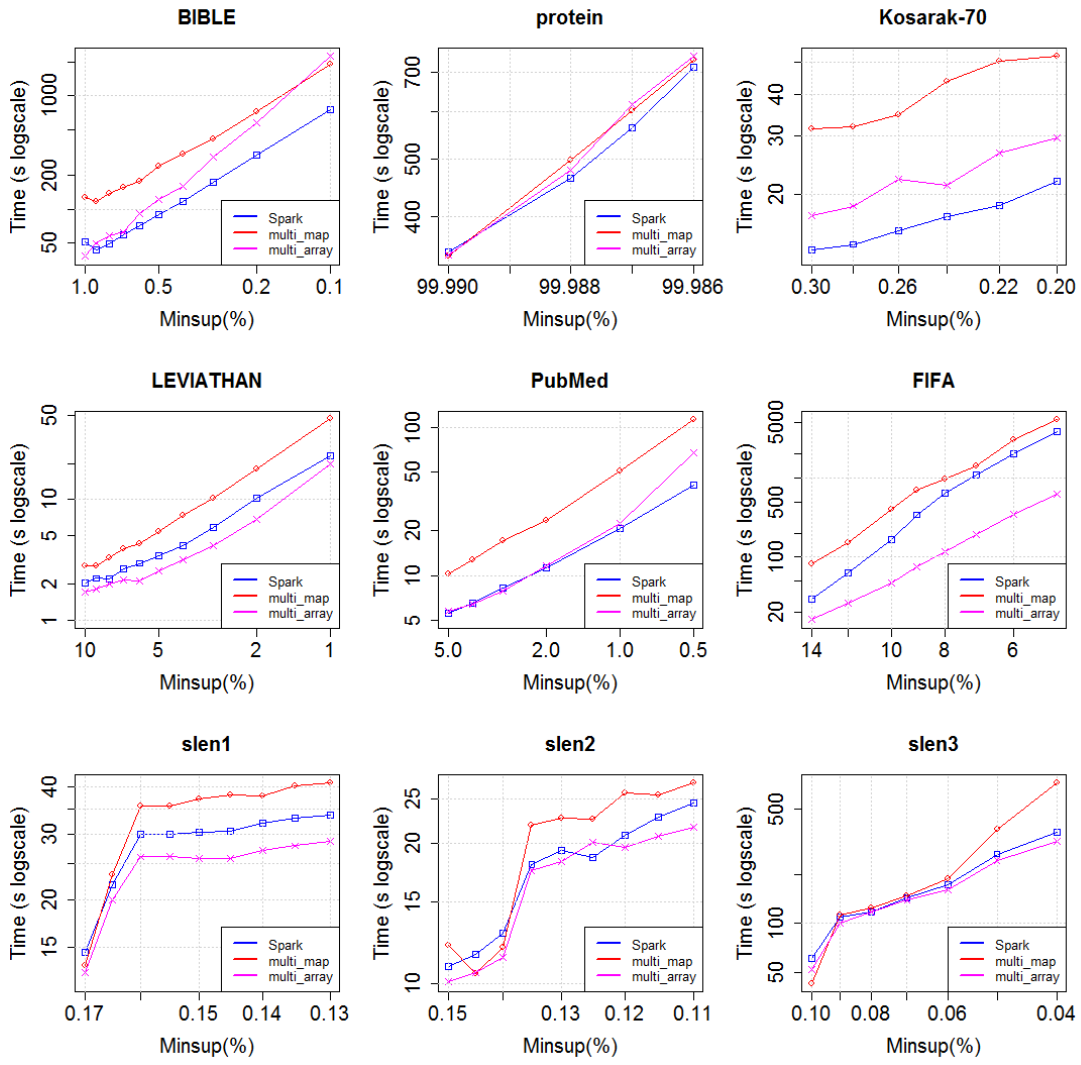


Figure 20: Performance of a CP based algorithm extending
PPIC's implementation by maintaining partial starts.
(Please take notice that we are here comparing to the original Spark implementation instead of the reference algorithm. Since the featured implementation doesn't posses the early-start improvement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

TODO - ADD REFERENCES, CA MANQUE ENCORE

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

6 Annexes

6.1 Glossary :

Acronyms

PPIC Prefix Projection Incremental Counting propagator. 5

SoS sequence of symbols. 5, 13, 15–17, 20, 23

SoSS sequence of sets of symbols. 5, 13, 15, 17, 20, 21

6.2 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	
$(\_c)$	$(\emptyset, 2, \emptyset)$	$(\emptyset, 1, \emptyset)$	$\emptyset$
	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 21: Example of an S-matrix for Prefix-Span bi-level projection

6.3 Additional Performance Comparisons :

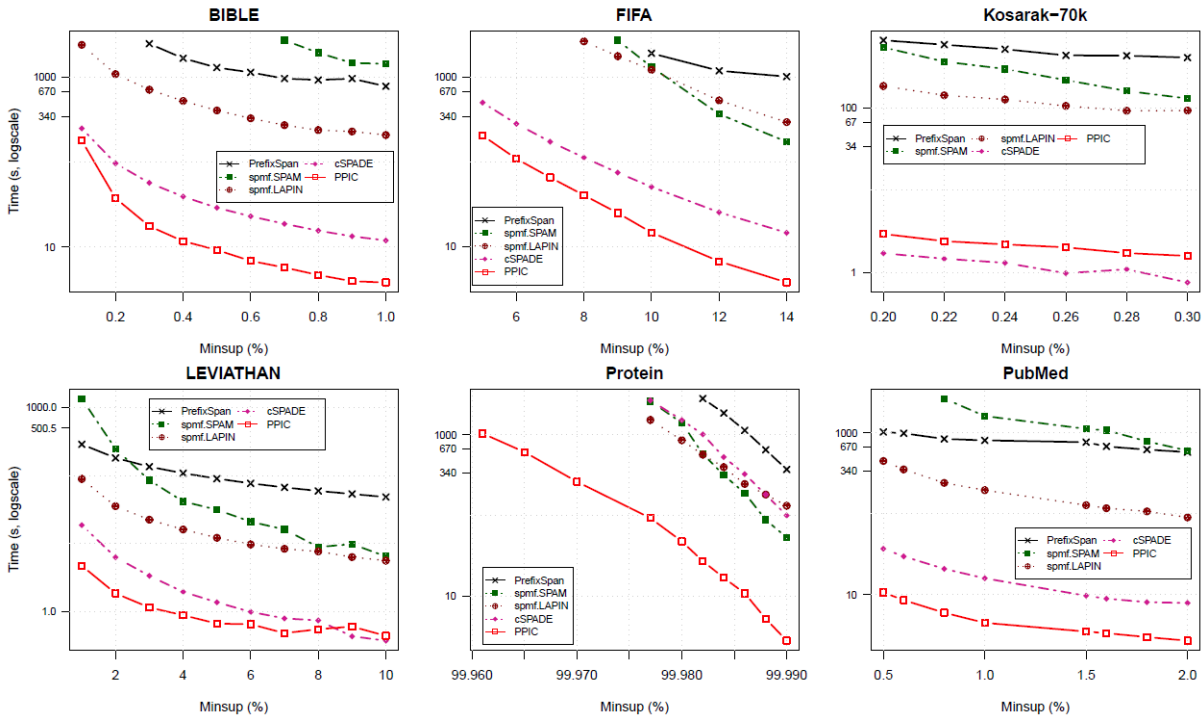


Figure 22: PPIC's performances VS other specialized algorithm

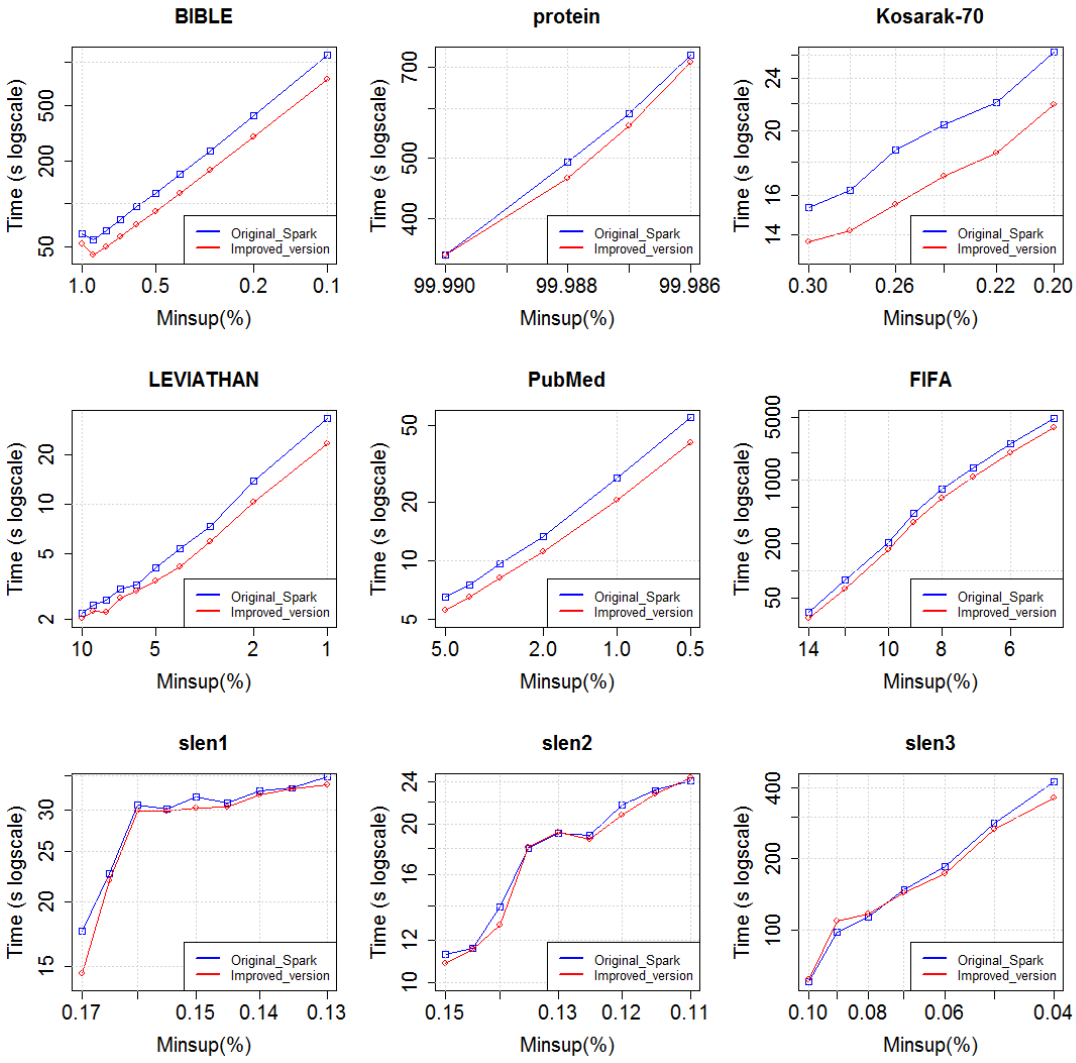


Figure 23: Performance improvement of fixing Spark's pre-processing

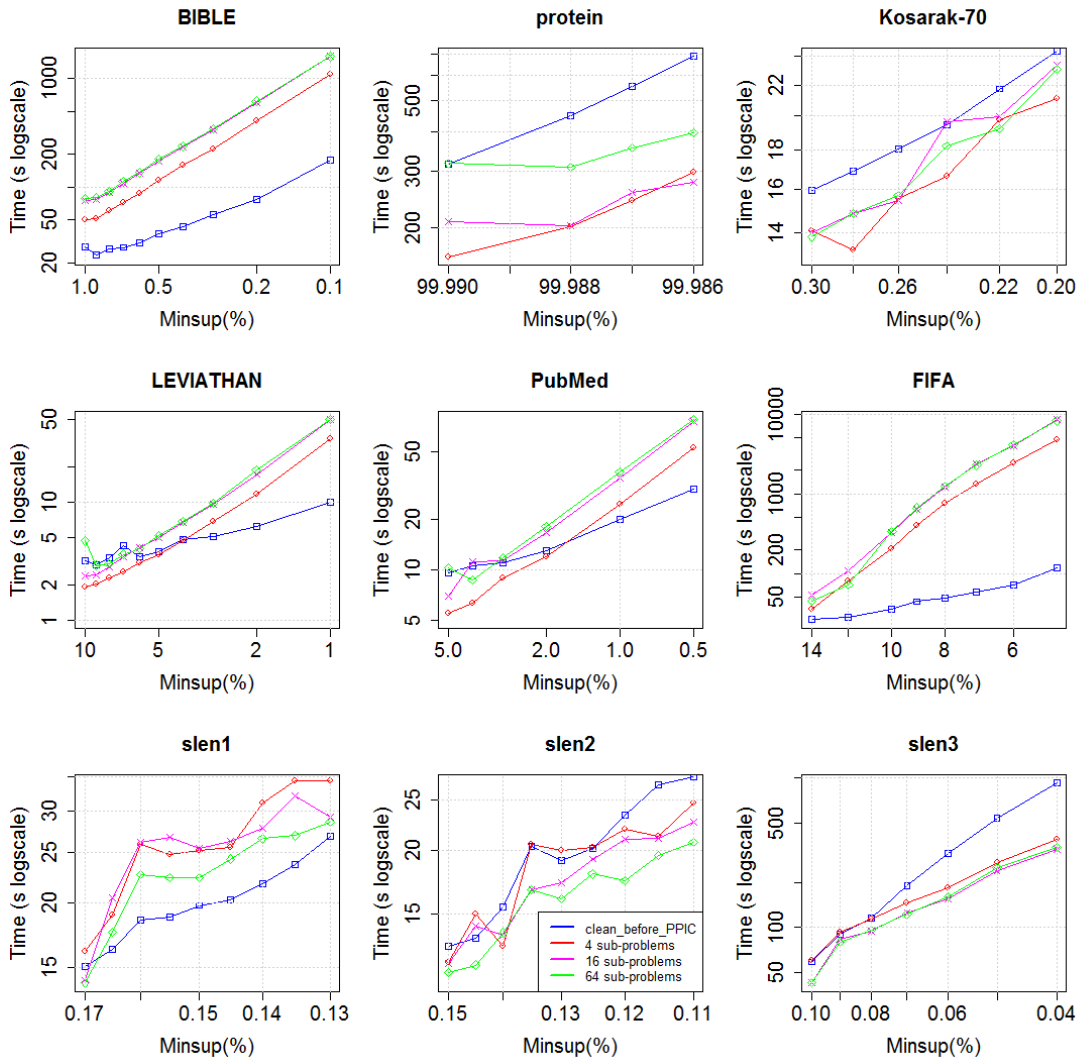


Figure 24: Performance improvement - soft limit on the number of created sub-problem. Tested on the reference (Clean Before Local Exec) implementation.
(P

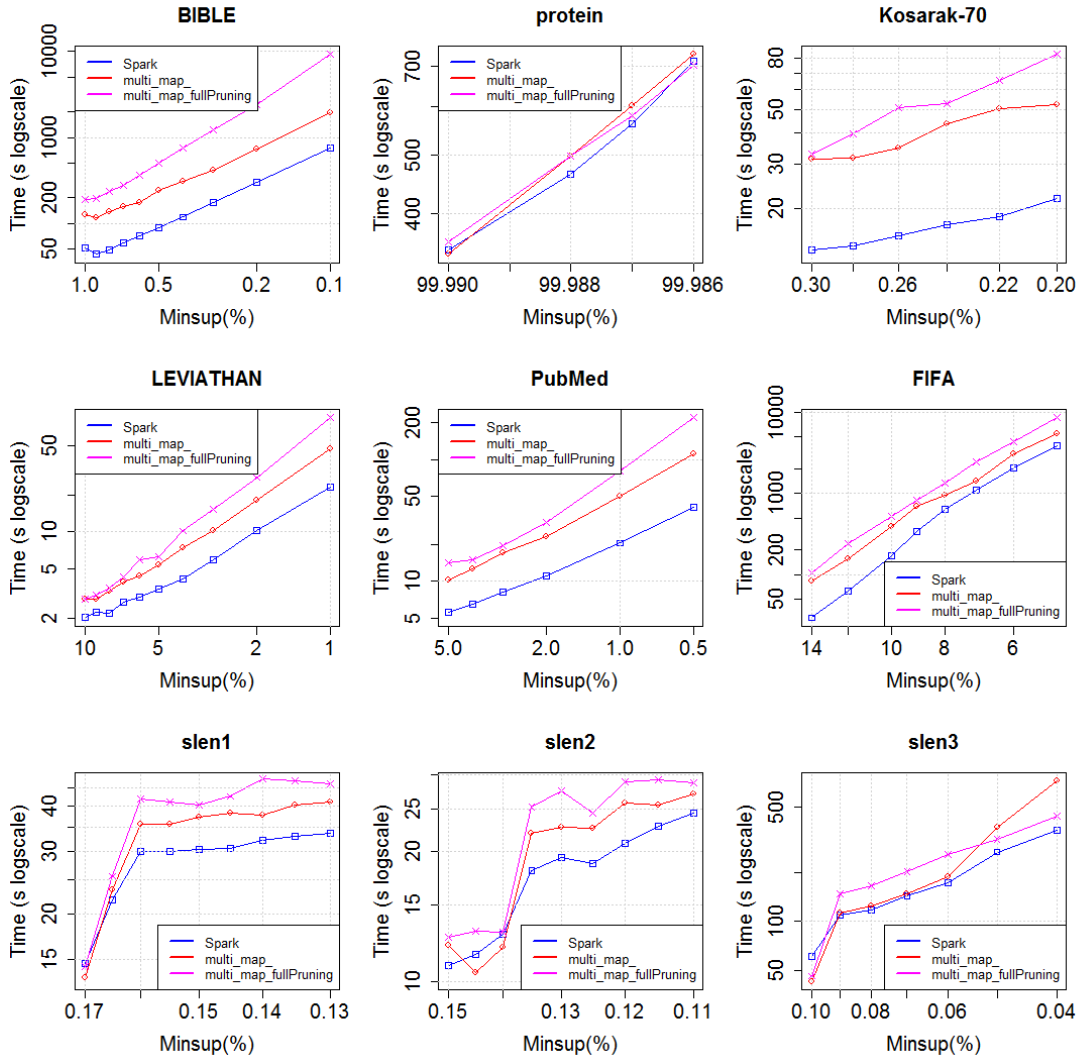


Figure 25: Performance comparison of a full pruning VS a partial pruning on the multi_map implementation.
(Please take notice that we are here comparing to the original Spark implementation instead of the reference algorithm. Since the featured implementation doesn't posses the early-start improvement and the comparison is, in this case, much more relevant.)

6.4 Algorithms :

Algorithm 1: 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 (θ)
Result: PPIC return the set of frequent sequence of symbol contained in SDB, such that any returned pattern appear at least θ 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, θ`) :`
  `// 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 2: PPIC Continued

```
1  Function projAndGetFreqs(SDB, itemToProject, θ, sids, poss, ϕ, φ) :
2      projFreq[a] = 0 ∀a ∈ {1, ..., nSymbols}
3      i = ϕ; j = ϕ + φ; sup = 0
4      while i < ϕ + φ do
5          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 ϕ and containing φ elements,
              allowing to only search sequence that supported the previous
              pattern.                                                     */
           /* seq => A sequence where the new item should be projected.    */
6          if lastPosMap(sid)(itemToProject) - 1 > pos then
               // Find next position of itemToProject in seq
7              if pos < firstPosMap(sid)(itemToProject) then
                   // Jump to first occurrence of the item in the sequence
8                  pos = firstPosMap(sid)(itemToProject)
9              else
                   // Search for the next position of itemToProject
10                 while pos < len(seq) and itemToProject != seq[pos] do
11                     pos = pos + 1
12                 end
13             end
               // item supported -> update projected database
14             sids[j] = sid; poss[j] = pos + 1; j = j + 1; sup = sup + 1;
15             posToCheck = interestingPosMap(sid)(pos) while posToCheck! = 0 do
                   // Faster than checking lastPosMap
16                 symbol = seq[posToCheck - 1] projFreqs[symbol] = projFreqs[symbol] + 1
                    posToCheck = interestingPosMap(sid)(posToCheck - 1)
17             end
18         end
19         i = i+1
20     end
21     ϕ = ϕ + φ; φ = sup
22     return projFreqs
23 end
```

Algorithm 3: Spark's original implementation : Pre-Processing

Data: Given a sequence of sets of symbol database (SSDB), a minimal number of supporting sequence (θ), 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 θ times in the database.

1 **Function** `PreProcessing(`$SSDB, maxPatLen, maxLocalSize, \theta$`) :`
  `// Clean database through a map reduce phase`
2 freqItems = findFrequentItem(SSDB, θ)
3 cleanedSequences = cleanSequenceAndRenameItem(SSDB, freqItems)
  `// Find frequent sequences of sets of symbols`
4 solutionPatterns = scalableExecution(cleanedSequences, maxPatLen, maxLocalSize, θ)
  `// Translate to original items name and Return`
5 **return** translateBackToOriginalItemsName(solutionPatterns)
6 **end**

Algorithm 4: Spark's original implementation : Scalable execution

```
1  Function scalableExecution(SSDB, maxPatLen, maxLocalSize, θ) :
       /* Encapsulate each sequence in a postfix, allowing to keep the current
          position and partial starts without copying the sequences.          */
2      postfixes = SSDB.map(seq => PostFix(seq))
       // Init prefix list and solution list
3      solutionPatterns = Array.empty
4      smallPrefixes = Array.empty
5      largePrefixes = [prefix.empty]
       // Start scalable execution
6      while largePrefixes is not empty do
7          mapReduceResults = postifixes.flatMap( postfix =>
8              largePrefixes.flatMap( prefix =>
9                  extensions = postfix.project(prefix).findPrefixExtension()
10                 extensions.map( (item, postfixSize) =>
                       // Return a (key, value) pair
11                     ((prefix.id, item), (1, postfixSize))
12                 )
13             )
14         ).reduceByKey((v1.1 + v2.1, v1.2 + v2.2)) // Aggregate values by key
15         .filterByValue(v.1 ≥ θ) // Keep key-value pair with enough support
           // Empty larger prefixes list
16         largePrefixes.clean()
           // Fill it with new, extended, prefixes
17         foreach ((prefiID, extendingItem), (support, projDBSize) ∈ mapReduceResults do
               // Create prefix, add it as solution
18             newPrefix = largePrefixes.getByID(prefixID) += extendingItem
19             solutionPatterns += newPrefix
20             if len(newPrefix) < maxPatternLength then
                   // len(newPrefix) == number of non-zero item in prefix
                   // zero == separator between itemSets
21                 if projDBSize > maxLocalProjDBSize then
22                     largePrefixes += newPrefix
23                 else
24                     smallPrefixes += newPrefix
25                 end
26             end
27         end
28     end
29     if len(smallPrefixes) > 0 then
           // For each prefix, project the whole database and solve locally
30         toExecuteLocaly = postifixes.flatMap( postfix =>
31             smallPrefixes.flatMap( prefix =>
32                 (prefix.ID, postfix.project(prefix).compress())
33             )
34         ).groupByKey().flatMap(sequences =>
35             localExecution(sequences, maxPatLen - len(prefix), θ)
36         )
37     end
38 end
```

Algorithm 5: Spark's original implementation : Local execution

```
1  Function localExecution(SSDB, maxPatLen, θ) :
2      if maxPatLen == 0 then
3          return Array.empty
4      end
       // Find extending item that are sufficiently frequent
5      counts = Map[Int, Int].empty.withDefaultValue(0)
6      foreach postfix ∈ SSDB do
7          foreach (extendingItem, size) ∈ postfix.findPrefixExtension() do
8              counts(extending) += 1
9          end
10     end
11     counts = counts.filterByValue(val ≥ θ)
       // Extend for each item and continue searching
12     solutions = ArrayList.empty[Prefix]
13     foreach key ∈ counts do
14         projectedDB = postfixes.project(key)
15         foreach extension ∈ localExecution(projectedDB, maxPatLen-1, θ) do
16             solution += key + extensions
17         end
18     end
19     return solutions
20 end
```

Algorithm 6: Spark's original implementation : Project and findPrefixExtension

```
1  Class Postfix(sequence) :
2      start = 0
3      partialProjections = Array.empty
4      Function project (Item) :
5          if start > len(sequence) then
6              return
7          end
8          newPartialProjection = Array.empty
9          if Item extends current ItemSet in Prefix then
10             foreach partial ∈ partialProjections do
11                 newPos = findNextItemInCurrentItemSet(sequence, partial, Item)
12                 if Found Item in current ItemSet then
13                     newPartialProjection += newPos
14                 end
15             end
16             start = findSmallestPosition(newPartialProjection) + 1
17         else if Item start a new ItemSet in Prefix then
18             for Pos ∈ {start, ..., len(sequence) − 1} do
19                 if sequence[Pos] == Item then
20                     if first found in loop then
21                         start = Pos + 1
22                     else
23                         newPartialProjection += Pos
24                     end
25                 end
26             end
27         end
28         partialProjections = newPartialProjection
29     end
30     Function project (Prefix) :
31         foreach Item ∈ Prefix do
32             this.project(Item)
33         end
34     end
35     Function project (findPrefixExtension) :
36         extendingItems = Set.empty
37         foreach partial ∈ partialProjections do
               // Find items that extend current itemSet
38             extendingItems ++= findAllItemUntilNextSeparator(sequence, partial)
39         end
40         for Pos ∈ {start, ..., len(sequence) − 1} do
               // Find items that start a new itemSet
41             extendingItems += sequence[Pos]
42         end
43     end
44 end
```

Algorithm 7: First scalable CP based implementation

```
1  Function
     scalableExecution(SSDB, maxPatLen, maxLocalSize, θ, hasSetsOfSymbols) :
       /* Encapsulate each sequence in a postfix, allowing to keep the current
          position and partial starts without copying the sequences.        */
2      postfixes = SSDB.map(seq => PostFix(seq))
       // Init prefix list and solution list
3      solutionPatterns = Array.empty
4      smallPrefixes = Array.empty
5      largePrefixes = [prefix.empty]
       // Start scalable execution
6      while largePrefixes is not empty do
7          ... // Same as Spark's original implementation
8      end
       // Start local execution
9      if len(smallPrefixes) > 0 then
           // For each prefix, project the whole database and solve locally
10         toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
11             (prefix.ID, postfix.project(prefix).compress())
12         )).groupByKey().flatMap(sequences =>
           // Use argument to determine which local execution to use
13         if hasSetsOfSymbols then
14             localExecution(sequences, maxPatLen - len(prefix), θ)
15         else
16             sequences = removeSeparator(sequences)
17             matrices = generatePPICMatrices()
               /* PPIC will generate the P vector, add constraint to enforce
                  maxPatLen and then launch the execution, calling propagate in
                  the process.                                               */
18             PPIC(sequences, maxPatLen - len(prefix), θ, matrices)
19         end
20         )
21     end
22 end
```

Algorithm 8: New functionalities : Scalable execution

```
1  Function scalableExecution(SSDB, minPatLen, maxPatLen, maxLocalSize, θ,
2  limitItemPerItemset, softSubProblemLimit) :
       /* Encapsulate each sequence in a postfix, allowing to keep the current
          position and partial starts without copying the sequences.          */
3      postfixes = SSDB.map(seq => PostFix(seq))
       // Init prefix list and solution list
4      solutionPatterns = Array.empty
5      smallPrefixes = Array.empty
6      largePrefixes = [prefix.empty]
       // Start scalable execution
7      while largePrefixes is not empty do
8          if len(largePrefixes) + len(smallPrefixes) ≥ softSubProblemLimit > 0 then
9              smallPrefixes ++= largePrefixes // Switch to local execution
10             break
11         end
12         mapReduceResults = postifixes.flatMap( postfix =>
13             ... // See First Scalable CP based Implementation
14         ).reduceByKey((v1.1 + v2.1, v1.2 + v2.2)) // Aggregate values by key
15         .filterByValue(v.1 ≥ θ) // Keep key-value pair with enough support
           // Empty larger prefixes list
16         largePrefixes.clean()
           // Fill it with new, extended, prefixes
17         foreach ((prefiID, extendingItem), (support, projDBSize) ∈ mapReduceResults do
               // Create prefix, add it as solution
18             newPrefix = largePrefixes.getByID(prefixID) += extendingItem
19             if respectConstraints(newPrefix, minPatLen, limitItemPerItemset) then
20                 solutionPatterns += newPrefix
21             end
22             if canExtendedItemRespectConstraint(newPrefix, maxPatLen,
                limitItemPerItemset then
                   // Search extensions for this prefix
23                 if projDBSize > maxLocalProjDBSize then
24                     largePrefixes += newPrefix
25                 else
26                     smallPrefixes += newPrefix
27                 end
28             end
29         end
30     end
31     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.                                  */
32         ...
33     end
34 end
```

Algorithm 9: Quick - Start : Scalable execution

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

Algorithm 10: Clean database before local execution of PPIC :

```
1  Function scalableExecution(SSDB, minPatLen, maxPatLen, maxLocalSize, θ,
2  hasSetsOfSymbols, limitItemPerItemset, softSubProblemLimit, freqItemAndCount)
   :
       /* Encapsulate each sequence in a postfix, allowing to keep the current
          position and partial starts without copying the sequences.          */
3      postfixes = SSDB.map(seq => PostFix(seq))
       // Init prefix list and solution list
4      solutionPatterns = Array.empty
5      smallPrefixes = Array.empty
6      largePrefixes = Array.empty
7      foreach (item, count) ∈ freqItemAndCount do
8          largePrefixes += Prefix(item, count)// count == nbSupport for item
9      end
       // Start scalable execution
10     while largePrefixes is not empty do
           // Project and extend prefix
11     end
12     if len(smallPrefixes) > 0 then
           // For each prefix, project the whole database and solve locally
13         toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
14             (prefix.ID, postfix.project(prefix).compress())
15         )).groupByKey().flatMap(sequences =>
           // Use argument to determine which local execution to use
16         if hasSetsOfSymbols then
17             localExecution(sequences, maxPatLen - len(prefix), θ)
18         else
19             sequences = cleanUnFrequentItemsAndRename(sequences)
20             matrices = generatePPICMatrices()
               /* PPIC will generate the P vector, add constraint to enforce
                  maxPatLen and then launch the execution, calling propagate in
                  the process.                                                 */
21             PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, θ, matrices,
                prefix)
22         end
23         )
24     end
25 end
```

Algorithm 11: Automatic Local Execution Selection :

```
1  Function scalableExecution(SSDB, minPatLen, maxPatLen, maxLocalSize, θ,
2  limitItemPerItemset, softSubProblemLimit, freqItemAndCount) :
       /* Encapsulate each sequence in a postfix, allowing to keep the current
          position and partial starts without copying the sequences.        */
3      postfixes = SSDB.map(seq => PostFix(seq))
       // Recalculate MaxItemPerItemset
4      calculatedMaxItemPerItemset = findLargestItemSetSize(SSDB)
5      if limitItemPerItemset == 0 or limitItemPerItemset > calculatedMaxItemPerItemset
        then
6          limitItemPerItemset = calculatedMaxItemPerItemset
7      end
       // Init prefix list and solution list
8      solutionPatterns = Array.empty
9      smallPrefixes = Array.empty
10     largePrefixes = Array.empty
11     foreach (item, count) ∈ freqItemAndCount do
12         largePrefixes += Prefix(item, count) // count == nbSupport for item
13     end
       // Start scalable execution
14     while largePrefixes is not empty do
           // Project and extend prefix
15     end
16     if len(smallPrefixes) > 0 then
           // For each prefix, project the whole database and solve locally
17         toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
18             (prefix.ID, postfix.project(prefix).compress())
19         )).groupByKey().flatMap(sequences =>
           // Determine which local execution to use
20         if calculatedMaxItemPerItemset > 1 then
21             localExecution(sequences, maxPatLen - len(prefix), θ)
22         else
23             sequences = cleanUnFrequentItemsAndRename(sequences)
24             matrices = generatePPICMatrices()
               /* PPIC will generate the P vector, add constraint to enforce
                  maxPatLen and then launch the execution, calling propagate in
                  the process.                                               */
25             PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, θ, matrices,
                prefix)
26         end
27         )
28     end
29 end
```

Algorithm 12: Clean database before any local execution :

```
1  Function scalableExecution(SSDB, minPatLen, maxPatLen, maxLocalSize, θ,
2  limitItemPerItemset, softSubProblemLimit, freqItemAndCount) :
       /* Encapsulate each sequence in a postfix, allowing to keep the current
          position and partial starts without copying the sequences.        */
3      postfixes = SSDB.map(seq => PostFix(seq))
       // Init prefix list and solution list
4      solutionPatterns = Array.empty
5      smallPrefixes = Array.empty
6      largePrefixes = Array.empty
7      foreach (item, count) ∈ freqItemAndCount do
8          largePrefixes += Prefix(item, count) // count == nbSupport for item
9      end
       // Start scalable execution
10     while largePrefixes is not empty do
           // Project and extend prefix
11     end
12     if len(smallPrefixes) > 0 then
           // For each prefix, project the whole database and solve locally
13         toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
14             (prefix.ID, postfix.project(prefix).compress())
15         )).groupByKey().flatMap(sequences =>
           // Clean sequences
16         (sequences, canUsePPIC) = removeUnFrequentItemsEfficiently(sequences) // NB :
              Partial starts must also be modified to remain correct
           // Determine which local execution to use
17         if ! canUsePPIC then
18             localExecution(sequences, maxPatLen - len(prefix), θ)
19         else
20             sequences = removeSeparators(sequences)
21             matrices = generatePPICMatrices()
               /* PPIC will generate the P vector, add constraint to enforce
                  maxPatLen and then launch the execution, calling propagate in
                  the process.                                               */
22             PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, θ, matrices,
                prefix)
23         end
24         )
25     end
26 end
```

Algorithm 13: Positions lists : Project and findPrefixExtension

```
1  Class Postfix(sequence) :
2      start = 0; partialProjections = Array.empty
3      firstPosList = getFirstPosList(sequence); lastPosList = getLastPosList(sequence)
4      Function project (Item) :
5          if start > len(sequence) or start > lastPosList(Item) then
6              return
7          end
           // Here, we know Item is present in remains of sequence
8          newPartialProjection = Array.empty
9          if Item extends current ItemSet in Prefix then
10             foreach partial ∈ partialProjections do
11                 newPos = findNextItemInCurrentItemSet(sequence, partial, Item)
12                 if Found Item in current ItemSet then
13                     newPartialProjection += newPos
14                 end
15             end
16             start = findSmallestPosition(newPartialProjection) + 1
17         else if Item start a new ItemSet in Prefix then
18             if start == 0 then
19                 start = firstPosList(sequence)
20             end
21             for Pos ∈ {start, ..., len(sequence) − 1} do
22                 if sequence[Pos] == Item then
23                     if first found in loop then
24                         start = Pos + 1
25                     else
26                         newPartialProjection += Pos
27                     end
28                 end
29             end
30         end
31         partialProjections = newPartialProjection
32     end
33     Function project (Prefix) :
34         foreach Item ∈ Prefix do
35             this.project(Item)
36         end
37     end
38     Function project (findPrefixExtension) :
39         extendingItems = Set.empty
40         foreach partial ∈ partialProjections do
               // Find items that extend current itemSet
41             extendingItems ++= findAllItemUntilNextSeparator(sequence, partial)
42         end
43         for (Item, lastPos) ∈ lastPosList, such that lastPos ≥ start do
44             extendingItems += Item
45         end
46     end
47 end
```

Algorithm 14: Specialised Execution : Pre-Processing

```
1  Function PreProcessing(SSDB, maxPatLen, maxLocalSize, θ) :
       // Clean database through a map reduce phase
2      freqItems = findFrequentItem(SSDB, θ)
3      cleanedSequences = cleanSequenceAndRenameItem(SSDB, freqItems)
       /* Find database type.  If any itemSet of size greater than one found
          in sequences, database type will be SSDB, else SDB                  */
4      databaseType = SSDB.map(sequence => findType(sequence)).reduce()
       // Find frequent sequences of sets of symbols
5      solutionPatterns = scalableExecution(cleanedSequences, databaseType, maxPatLen,
         maxLocalSize, θ)
       // Translate to original items name and Return
6      return translateBackToOriginalItemsName(solutionPatterns)
7  end
8  Function scalableExecution(...) :
       /* Separate postfix object in two sub-objects, one which search for
          itemSets 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.                                          */
9      ...
10 end
```

Algorithm 15: Sorting sub-problems on the reducer :

```
1  Function scalableExecution(SSDB, minPatLen, maxPatLen, maxLocalSize, θ,
2  hasSetsOfSymbols, limitItemPerItemset, softSubProblemLimit, freqItemAndCount)
   :
       /* Encapsulate each sequence in a postfix, allowing to keep the current
          position and partial starts without copying the sequences.          */
3      postfixes = SSDB.map(seq => PostFix(seq))
       // Init prefix list and solution list
4      solutionPatterns = Array.empty
5      smallPrefixes = Array.empty
6      largePrefixes = Array.empty
7      foreach (item, count) ∈ freqItemAndCount do
8          largePrefixes += Prefix(item, count)// count == nbSupport for item
9      end
       // Start scalable execution
10     while largePrefixes is not empty do
           // Project and extend prefix
11     end
12     if len(smallPrefixes) > 0 then
           // For each prefix, project the whole database and solve locally
13         toExecuteLocaly = postifixes.flatMap( postfix => smallPrefixes.flatMap( prefix =>
14             (prefix.ID, postfix.project(prefix).compress())
15         )).groupByKey().SortByDBSize().flatMap(sequences =>
           // Use argument to determine which local execution to use
16         if hasSetsOfSymbols then
17             localExecution(sequences, maxPatLen - len(prefix), θ)
18         else
19             sequences = cleanUnFrequentItemsAndRename(sequences)
20             matrices = generatePPICMatrices()
               /* PPIC will generate the P vector, add constraint to enforce
                  maxPatLen and then launch the execution, calling propagate in
                  the process.                                                 */
21             PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, θ, matrices,
                prefix)
22         end
23         )
24     end
25 end
```

Algorithm 16: Sorting sub-problems on the mapper :

```
1  Function scalableExecution(SSDB, minPatLen, maxPatLen, maxLocalSize, θ,
2  hasSetsOfSymbols, limitItemPerItemset, softSubProblemLimit, freqItemAndCount)
   :
       /* Encapsulate each sequence in a postfix, allowing to keep the current
          position and partial starts without copying the sequences.          */
3      postfixes = SSDB.map(seq => PostFix(seq))
       // Init prefix list and solution list
4      solutionPatterns = Array.empty
5      smallPrefixes = Array.empty
6      largePrefixes = Array.empty
7      foreach (item, count) ∈ freqItemAndCount do
           // For quick-started prefixes, set projected DB size to max allowed
8          largePrefixes += Prefix(item, count, Long.MaxValue)
9      end
       // Start scalable execution
10     while largePrefixes is not empty do
           // Project and extend prefix
11     end
12     if len(smallPrefixes) > 0 then
           // For each prefix, project the whole database and solve locally
           // Prefix are modified to hold their projected database size
13         toExecuteLocaly = postifixes.flatMap( postfix =>
            smallPrefixes.SortByDBSize().flatMap( prefix =>
14             (prefix.ID, postfix.project(prefix).compress())
15         )).groupByKey().flatMap(sequences =>
           // Use argument to determine which local execution to use
16         if hasSetsOfSymbols then
17             localExecution(sequences, maxPatLen - len(prefix), θ)
18         else
19             sequences = cleanUnFrequentItemsAndRename(sequences)
20             matrices = generatePPICMatrices()
               /* PPIC will generate the P vector, add constraint to enforce
                  maxPatLen and then launch the execution, calling propagate in
                  the process.                                                 */
21             PPIC(sequences, maxPatLen, minPatLen, maxItemPerItemSet, θ, matrices,
                prefix)
22         end
23         )
24     end
25 end
```

Algorithm 17: PPIC with a Map based Structure (partial pruning) :

```
  // Init global param :
1 ...
2 Function propagate(SSDB, P, i, θ) :
3 |  ... // Same propagate as PPIC, but the database is an SSDB instead
4 end
5 Function projAndGetFreqs(SSDB, itemToProject, θ, sids, poss, ϕ, φ) :
6 |  projFreq[a] = 0 ∀a ∈ {1, ..., nSymbols}; i = ϕ; j = ϕ + φ; sup = 0
7 |  while i < ϕ + φ do
8 |  |  sid = sids[i]; pos = poss[i]; seq = SSDB[sid]
  |  |  // Get positions of itemToProject in sequence
9 |  |  listSoughtItemPos = seq.get(itemToProject).discardItemsBefore(pos)
10|  |  if not listSoughtItemPos.isEmpty() then
  |  |  |  // Find next position of itemToProject in seq
11|  |  |  if lastItemInPwasSeparator()) then
  |  |  |  |  // Jump to first occurrence of the item in the sequence
12|  |  |  |  pos = listSoughtItemPos.pop() + 1
  |  |  |  |  // item supported -> update projected database
13|  |  |  |  sids[j] = sid; poss[j] = pos; j = j + 1; sup = sup + 1;
14|  |  |  else
  |  |  |  |  /* Search the sequence ItemSet by ItemSet until all elements
  |  |  |  |     of P's current itemSet match                              */
15|  |  |  |  for position ∈ listSoughtItemPos do
16|  |  |  |  |  if curItemSetInSeqContainsAllItemOfCurrentItemSetInP(seq, position)
  |  |  |  |  |   then
  |  |  |  |  |  |  // Jump to first occurrence of the item in the sequence
17|  |  |  |  |  |  pos = listSoughtItemPos.pop() + 1
  |  |  |  |  |  |  // item supported -> update projected database
18|  |  |  |  |  |  sids[j] = sid; poss[j] = pos; j = j + 1; sup = sup + 1;
19|  |  |  |  |  |  Break;
20|  |  |  |  |  end
21|  |  |  |  end
22|  |  |  end
23|  |  |  if itemToProject supported in sequence then
  |  |  |  |  // Count support for items (partial pruning)
24|  |  |  |  foreach (item, positionsList) ∈ seq do
25|  |  |  |  |  positionList.discardItemsBefore(pos)
26|  |  |  |  |  if not positionList.isEmpty() then
27|  |  |  |  |  |  projFreqs[item] = projFreqs[item] + 1
28|  |  |  |  |  end
29|  |  |  |  end
  |  |  |  |  /* For full pruning, find all itemSet in sequence that support
  |  |  |  |     current ItemSet in P, augment the projected frequency of
  |  |  |  |     all item present in those itemSets                        */
30|  |  |  end
31|  |  end
32|  |  i = i+1
33|  end
34|  ϕ = ϕ + φ; φ = sup
35|  return projFreqs
36 end
```

Algorithm 18: PPIC with partial projections :

```
1  Function projAndGetFreqs(SSDB, itemToProject, θ, sids, poss, ϕ, φ, partialProj) :
2      projFreq[a] = 0 ∀a ∈ {1, ..., nSymbols}; i = ϕ; j = ϕ + φ; sup = 0
3      while i < ϕ + φ do
4          sid = sids[i]; pos = poss[i]; seq = SSDB[sid]
5          if itemToProject == separator then
6              partialProj = Array.empty
7              pos = findNextSeparatorInSequence(seq, pos)
8              if pos < len(seq) then
                   // item supported -> update projected database
9                  sids[j] = sid; poss[j] = pos + 1; j = j + 1; sup = sup + 1;
10                 projFreq = findFrequencyOfEachItemFromLastPosList(sid, pos)
11             end
12         else if partialProj.isEmpty then
13             pos = firstPositionOfItemToProjectInRemainOfSequence(seq, pos)
14             partialProj = AllSuccessivePosOfItemToProject(seq, pos)
15             if pos < len(seq) then
                   // item supported -> update projected database
16                 sids[j] = sid; poss[j] = pos + 1; j = j + 1; sup = sup + 1;
17                 projFreq = findFrequencyOfEachItemFromLastPosList(sid, pos)
18             end
19         else
20             newPartialProj = Array.empty
21             foreach posToCheck ∈ partialProj do
22                 posToCheck = findPosOfNextItemToProjectInCurrentItemSet(seq,
                    posToCheck)
23                 if posToCheck < len(seq) then
24                     newPartialProj += posToCheck
25                 end
26             end
27             partialProj = newPartialProj
28             pos = partialProj.min
29             if not partialProj.isEmpty then
                   // item supported -> update projected database
30                 sids[j] = sid; poss[j] = pos + 1; j = j + 1; sup = sup + 1;
31                 projFreq = searchItemSetOfPartialProjForExtendingItem(seq, partialProj)
32             end
33         end
34         i = i+1
35     end
36     ϕ = ϕ + φ; φ = sup
37     return projFreqs
38 end
```

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