CP based Sequence Mining on the cloud using spark

Cyril de Vogelaere, Pierre Schaus, John O.R. Aoga

UCL - Université catholique de Louvain

Objectif :

The objective of this master thesis is to develop a **scalable** and **efficient** CP based implementation of the prefix-span algorithm using Spark, and to release the developed algorithm for the benefit of the Spark community.

Research based upon :

- **PPIC** :
  Developed by John O.R. Aoga during his master thesis, this algorithm offer record breaking performances, **outperforming specialized system through the use of a generic constraint solver**. The main two improvement being the pre-computation of the positions at which a symbol becomes unsupported by a sequence and the usage of a backtracking-aware data structure that allows fast incremental storing and restoring of the projected database.
- **Spark** :

  An **open source cluster computing framework** providing programmers with an application programming interface centered on a data structure called the **resilient distributed dataset** (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way.

  Spark also disposes of a machine learning library, which already provides an efficient prefix-span based implementation (**BFS**). The idea is thus to build upon that implementation, to develop a, more efficient, CP based solution.

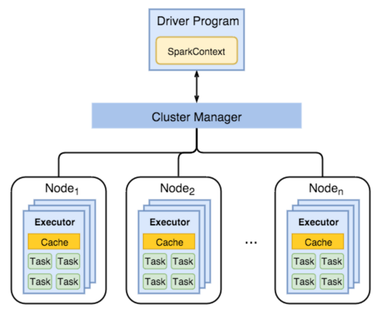


Fig. 1: Spark's architecture

Prefix-span algorithm:

PPIC is based on the **prefix-span algorithm**. It works by analysing the current sequence to find frequent items, then it extends the current prefix and repeat the process, backtracking when necessary (**DFS**), thus generating all postfixes incrementally.

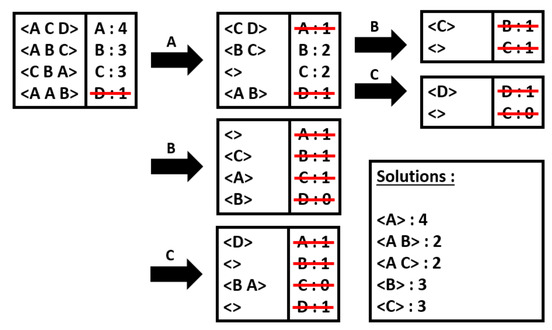
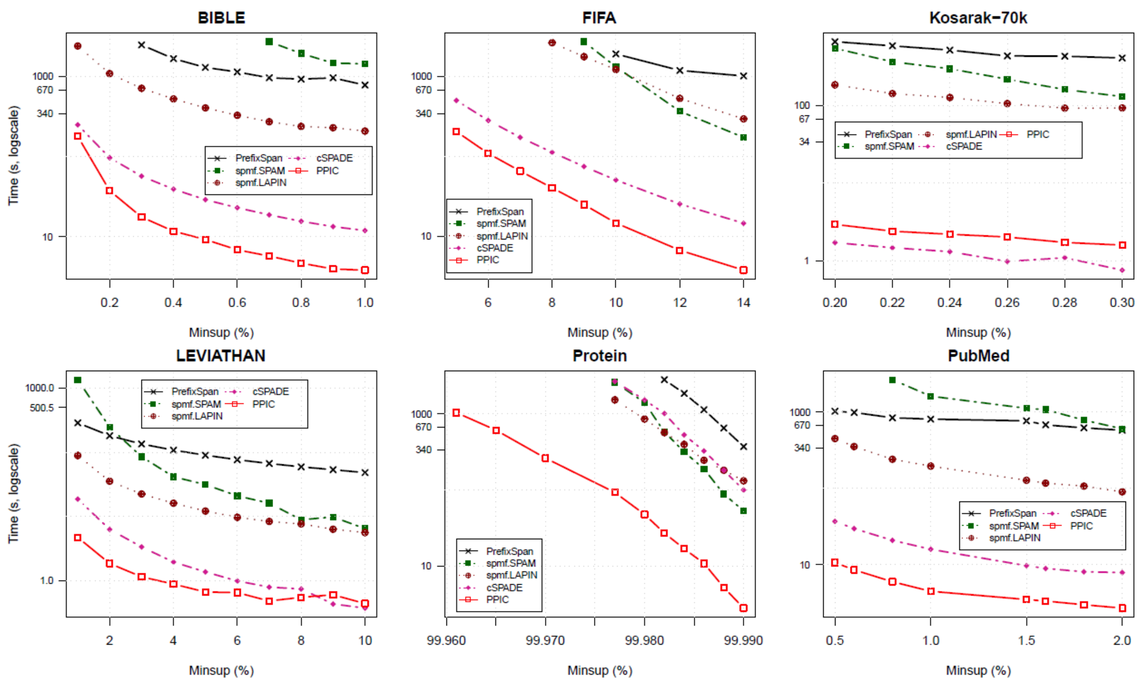
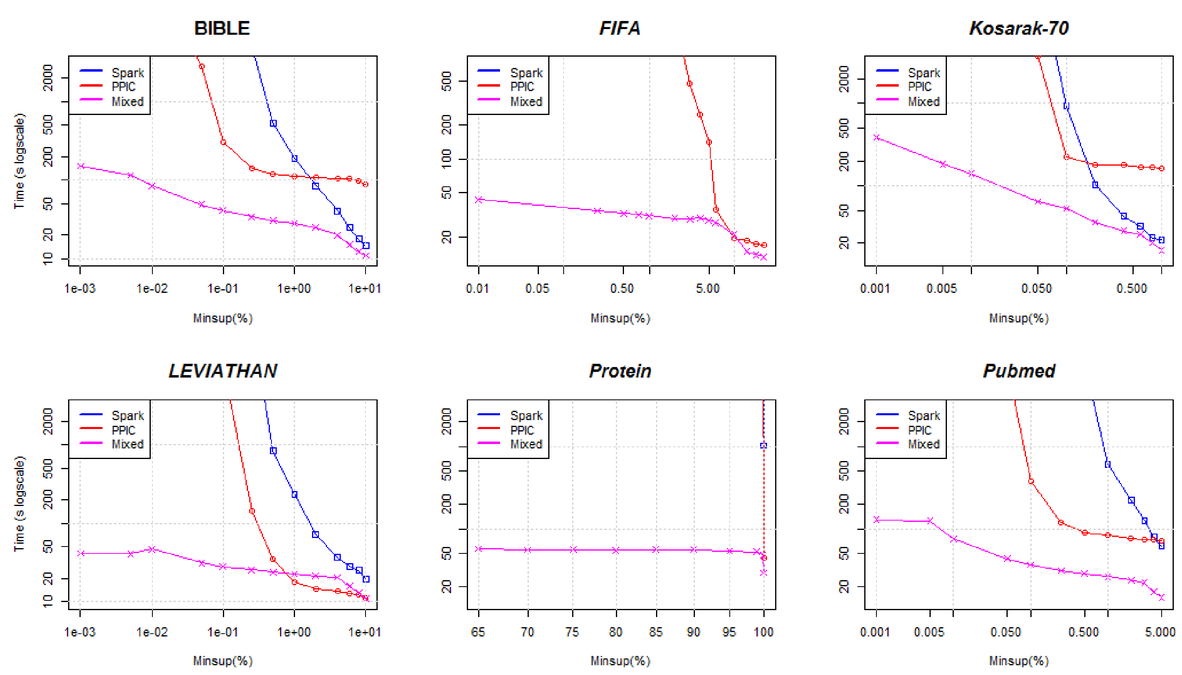


Fig. 2: Prefix-span, an easy example

PPIC vs Specialized solvers :



PPIC vs Spark's original implementation :



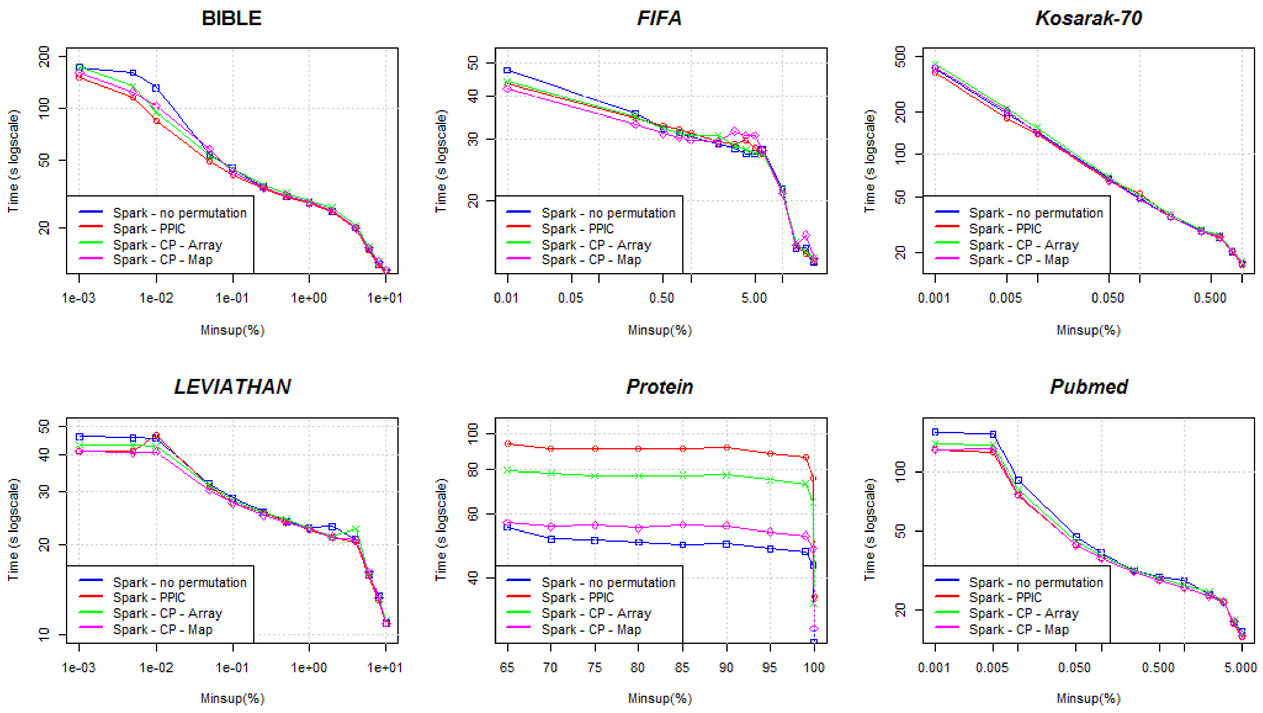
Flaws of the original algorithms :

- **PPIC** :
  - The algorithm is not scalable
  - Although PPIC is ridiculously fast, the **pre-computation** needed before the algorithm can be run are extremely slow and/or **consume so much memory that the algorithm crashes**.
- **Spark** :
  - The algorithm **searches for permutations even if there are none**, which is a **huge waste of time and space**.
  - While searching for permutations, the partialStarts array can grow exponentially large depending on the problem, leading to memory issues and massive slowdowns.

Merging the two algorithms, how it solves our problems :

- We **split the search tree using Spark's algorithm** then switch to **PPIC during the local execution**. This way, the preprocessing matrices are kept to manageable sizes, effectively speeding up the whole algorithm, while making PPIC scalable.
- We can modify Spark's original algorithm so it **doesn't search for permutations uselessly**, as shown in the no permutation algorithm (see graph on the right). We can also switch earlier to a local execution, reducing the cost of transferring the huge partialStart array around.

Performance comparison of various merged algorithm :



What's left to do :

- Finish tests on datasets with permutations !
- Perform scalability tests !
- Start working on the paper
- Implement an early local switch.
- Propose the end result to the Spark community.