Frequent itemset sampling of high throughput streams on FPGA accelerators

Maël Gueguen

▶ To cite this version:

Maël Gueguen. Frequent itemset sampling of high throughput streams on FPGA accelerators. Embedded Systems. Université Rennes 1, 2020. English. NNT : 2020REN1S053 . tel-03120148v2

HAL Id: tel-03120148
https://theses.hal.science/tel-03120148v2

Submitted on 7 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies
de l'Information et de la Communication
Spécialité : Informatique

Par

« Mael GUEGUEN »

« Improving the performance and energy efficiency of complex heterogeneous manycore architectures with on-chip data mining »

Thèse présentée et soutenue à RENNES, FRANCE, le 23 octobre 2020
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Frédéric PÉTROT, Professeur, Grenoble INP
Marc PLANTEVIT, Maitre de Conférence HDR, Université de Lyon 1

Composition du jury :

Examinateurs :	Laurence PIERRE Professeur, Université Grenoble Alpes
	Benjamin NEGREVERGNE, Maitre de Conférences, Université Paris-Dauphine
Dir. de thèse :	Olivier SENTIEYS, Professeur, Université de Rennes 1
Co-dir. de thèse :	Alexandre TERMIER, Professeur, Université de Rennes 1

TABLE OF CONTENTS

Introduction **6**

1 Background and Related Work **9**

1.1 Frequent Itemset Mining 10

1.2 Frequent Itemset Mining Algorithms 11

1.2.1 ECLAT 11

1.2.2 FP-Growth 12

1.3 Pattern Sampling 15

1.3.1 Input Space Sampling 16

1.3.2 Output Space Sampling 17

1.3.3 Output Space Sampling with random SAT constraints 21

1.4 Conclusion 23

2 Accelerating ECLAT on FPGA **25**

Introduction 25

2.1 General Introduction to FPGA Acceleration 25

2.2 Itemset Mining on FPGA 28

2.2.1 ECLAT Algorithm 28

2.2.2 FPGA Acceleration of ECLAT in [1] 30

2.3 Contribution to Memory Optimized FPGA Acceleration of ECLAT 36

2.4 Architecture of the ECLAT accelerator 41

2.4.1 Overview 41

2.4.2 Finding last item from an itemset 41

2.4.3 Support Counting 43

2.4.4 Correspondence List Address Cache 44

2.5 Results 46

3 Itemset Sampling acceleration on FPGA **49**

3.1 Sampling Itemsets with a System of Binary Constraints 49

3.1.1 Rewriting the Constraints with Gauss-Jordan Elimination 51
3.1.2 Reducing the Sampling Problem to a Regular Itemset Mining Task 55
3.1.3 Algorithmic implementation of ECLAT with dynamic XOR constraints 60
3.2 Accelerator Architecture 63
3.2.1 Overview 63
3.2.2 Linked list for results 63
3.2.3 Parameters' Influence on the Accelerator 63
3.3 Experiments 66
3.3.1 Impact of Sampling on Execution Time 67
3.3.2 Impact of Correspondence List Size 67
3.3.3 Comparison Against Flexics 68
3.4 Extension to Parallel Execution 68
3.4.1 Static Scheduling 69
3.4.2 Work Stealing 71
Conclusion 83

4 Stream sampling **85**
4.1 Context and related work 86
4.1.1 Itemset Mining in a stream 86
4.2 General Approach 90
4.2.1 Stream mining and Itemset sampling 90
4.3 Techniques 92
4.3.1 Navigating the FP-Tree during a scan 92
4.3.2 Maintenance of the FP-Tree with new constraints 100
4.3.3 Applying decay to the FP-Tree 106
4.3.4 Maintenance of the FP-Tree when removing constraints 107
4.3.5 Algorithmic implementation of the streaming sampler 111
4.4 Experiments on Pattern Quality 114
4.4.1 Synthetic Experiments 114
4.4.2 Experiments on a Semi-Real Dataset 121
4.5 Experiments on Hardware Acceleration 126

Conclusion **131**

Bibliography **134**

Bibliography **135**

INTRODUCTION

Context

When analyzing data, an important task is to discover *correlations*. Such correlations can be that Amazon customers that buy a laptop often buy a mouse simultaneously, or that a set of files are often accessed simultaneously in a cloud storage context. These correlations give powerful insights on the contents of the data, that may be used to take actions. For example Amazon will immediately show mouses to customers having put a laptop in their cart, and the cloud storage operator will position the files often accessed together on the same physical machine.

Discovering potentially complex correlations is handled by Pattern Mining algorithms, whose goal is to explore a huge combinatorial space efficiently. Most of them have been designed for an offline data analysis setting: a static dataset is input to the algorithm which performs pattern extraction (runtime varying from a few seconds to several days depending on the data and parameters). The results are then manually reviewed by a data analyst. However, many modern settings require an online loop, where the data is continuously arriving. In this context, patterns are extracted in few seconds at most, and are used to take automated decisions. For example, Amazon needs to be able to adapt quickly to changes in customer behavior or arrival of new products, and file access patterns will change upon completion of projects using those files. Two approaches are available for processing large dataset with real-time constraints: either with the high computation power of servers in the cloud, or via dedicated hardware for a greater energy efficiency.

Motivations

There exist multiple pattern mining algorithms that handle data streams, however they have two limitations: 1) they are likely to output thousands of results of mixed interest and 2) they are not designed to cope with a high throughput. The first limitation

is shared by most pattern mining algorithms: they extract patterns that are repeated *frequently* in the data, and in practice there are many of such repeated patterns. A large portion of these patterns contain redundant information. There are several ways to reduce the number of patterns output to a manageable size, one of the most drastic being *output space pattern sampling*, where a historical solution has been *input space pattern sampling*. The idea is to only compute a set of frequent patterns, and to have guarantees that these patterns are a representative sample of the complete set of patterns. While *input space pattern sampling* can only delay the pattern explosion caused by the redundancy in the results, *output space pattern sampling* offers a scalable solution with stronger statistical guarantees. To the best of our knowledge though, output space sampling approaches in the literature do not consider streaming data. Furthermore, academic research has been focused on software sampling techniques, while taking hardware into consideration can further enhance performance and energy efficiency. In this thesis, we show that pattern sampling algorithms are a promising approach for performing a pattern-based analysis of data streams, and we show that their implementation on hardware architectures such as FPGAs allows to process high throughputs with modest hardware resources.

Contributions

The main goal of this PhD is to provide efficient pattern sampling algorithms for FPGA architectures.

To this aim, our first contribution sets up an important building block, which is to have a flexible frequent itemset mining algorithm adapted for FPGA. While such approaches exist in the literature, they are designed solely for frequent itemset mining, and cannot easily be adapted to other variants of the problem such as pattern sampling. We thus started from an existing adaptation of the ECLAT algorithm on FPGA [1] and revisited the algorithm and the FPGA architecture of this approach, in order to have a solid algorithm basis on which to build our pattern sampling contributions. Our improvement helps in making the traversal of the enumeration tree of ECLAT more predictable, i.e. we provide a simple function that allows to determine exactly, from any candidate itemset, the next itemset to visit. In software approachs this is usually handled by backtracking, but on FPGA such stack-based approach is not well adapted.

This paves the way for our second contribution, which is a pattern sampling al-

gorithm on FPGA for static datasets. We base our approach on the state-of-the-art Flexics pattern sampling algorithm [2], which is itself based on Eclat for the mining part. The main challenge is that the sampling process makes the traversal of the search space even more irregular than classical pattern mining algorithms, which is ill-adapted for FPGA computation. Our main algorithmic contribution here is to propose a way to have a regular exploration of the search space of pattern despite of the random “jumps” introduced by the sampling approach.

Our third and last contribution is to adapt the previous approach to the case of data streams, in order to have a streaming pattern sampling algorithm. This requires to maintain a limited summary of past data so that sampling results reflect not only the immediate contents of the data, but also some trends from the recent past. This also requires to revisit the sampling process, which is based on adding random constraints to reduce the size of the pattern space: in a streaming case, such constraints can be added but also removed, a new case that we had to handle.

Outline

Chapter 1 introduces the main pattern mining concepts, with notable algorithms, and presents pattern sampling approaches. Chapter 2 starts with an introduction to FPGA acceleration, followed by the description of our own itemset mining accelerator and how it compares to the literature. Chapter 3 describes the state of the art Flexics algorithm for pattern sampling, as well as our algorithmic contribution to adapt Flexics to FPGA. The chapter follows with the architecture of our approach and the results. Section 4 presents our approach for pattern sampling on streaming data on FPGA. The algorithmic contribution as well as the FPGA architecture are detailed, and experiments show the efficiency of our approach. Last, Section 4.5 concludes the thesis and provides possible perspectives.

CHAPTER 1

BACKGROUND AND RELATED WORK

In recent years, the amount of data generated or exchanged between computing systems rose drastically. Due to their sheer size, it has become increasingly difficult to analyse datasets in practice. The field of data science proposes a variety of solutions to process and analyse huge amounts of data within a reasonable time or with limited resources. Some of the tasks involved in the analysis of large datasets are anomaly detection[1], phase detection[2], frequent subgraph mining[3], frequent itemset mining[4], or other options.

This document focuses heavily on frequent itemset mining, sometimes abbreviated as “itemset mining”, which is described in Section 1.1. Section 1.2 presents two itemset mining techniques from the literature, ECLAT [3] and FP-Growth [4], upon which our work is based. Finally, Section 1.3 introduces the notions of sampling in itemset mining, also called *pattern sampling*. Pattern sampling solves one of the biggest issues of itemset mining, called “pattern explosion”, where an itemset miner returns so many results that they cannot be interpreted in reasonable time by an expert or an automated system. Two methods are presented, “input space sampling” and “output space sampling”, the later being more effective. This distinction is important for this document, as we provide a new output space sampling implementation and its acceleration on FPGA platforms.

[1] Finding outliers in a database, rare patterns with a significantly different distribution.
[2] Finding shifts of distribution of time series.
[3] Finding recurrent subgraphs in a larger graph.
[4] Finding recurrent ensemble of items in a database.

1.1 Frequent Itemset Mining

We consider an alphabet $\mathcal{A} = \{X_1, ..., X_N\}$. The singleton elements of the alphabet $\mathcal{A}$ are called *items*, and any subset $I \subseteq \mathcal{A}$ is called an *itemset*. Itemsets are ensembles of items. For convenience purposes, an itemset is represented with its items sorted in ascending order, for any arbitrary order provided over $\mathcal{A}$. Using this order, itemsets can also be sorted lexicographicaly.

A database $\mathcal{D} = \{\mathcal{T}_1, ..., \mathcal{T}_m\}$ is a collection of *transactions*, where $\forall j \in [1, m]\ \ \mathcal{T}_j \subseteq \mathcal{A}$. The *support* of an itemset $I \subseteq \mathcal{A}$ in $\mathcal{D}$ corresponds to the number of transactions of $\mathcal{D}$ where I is included: $support_{\mathcal{D}}(I) = \{\mathcal{T}_j \mid \mathcal{T}_j \in \mathcal{D}$ and $I \subseteq \mathcal{T}_j\}$. An itemset is *frequent* in $\mathcal{D}$ if its support is above a user-defined minimum frequency threshold t_h. Notice that it is often inconvenient to express the support and minimum frequency threshold as a number of transactions, hence both can be expressed in a *relative* way: $supportRelative_{\mathcal{D}}(I) = \frac{support_{\mathcal{D}}(I)}{|\mathcal{D}|}$. This allows to express the support and minimum frequency threshold as a percentage of the size of the total database.

Example: Table 1.1 is a database $\mathcal{D}$ using the alphabet $\mathcal{A} = \{a, b, c, d\}$. Given a (relative) minimum frequency threshold t_h of 25% (itemsets must be in at least two transactions to be frequent), the frequent itemsets are listed in Table 1.2.

$\mathcal{T}_1$	$\{a, b\}$
$\mathcal{T}_2$	$\{b, c, d\}$
$\mathcal{T}_3$	$\{c\}$
$\mathcal{T}_4$	$\{a, c, d\}$
$\mathcal{T}_5$	$\{c\}$
$\mathcal{T}_6$	$\{a, c, d\}$
$\mathcal{T}_7$	$\{a, b\}$
$\mathcal{T}_8$	$\{a, d\}$

Table 1.1: Example database $\mathcal{D}$

Itemset	Support
$\{a\}$	5
$\{a, b\}$	2
$\{a, c\}$	2
$\{a, c, d\}$	2
$\{a, d\}$	3
$\{b\}$	3
$\{c\}$	5
$\{c, d\}$	3
$\{d\}$	4

Table 1.2: Frequent itemsets from $\mathcal{D}$ with $t_h = 2$

A simple way to find and list all frequent itemsets is called *generate and test*. A particular itemset is generated, its support in the database is measured, then a new

itemset is generated, its support measured, and so on. Starting with the itemset $\{a\}$, we look at all transactions in the database and check how many contain it. $\{a\}$ is present in $\mathcal{T}_1$, $\mathcal{T}_4$, $\mathcal{T}_6$, $\mathcal{T}_7$ and $\mathcal{T}_8$, so the support of the itemset is 5. The itemset is thus considered frequent, and will be stored in the results with its corresponding support.

The next itemset generated is dictated by the enumeration strategy of the pattern mining algorithm used. In this example, we use a depth-first algorithm to enumerate itemsets. One of such algorithms, ECLAT [3], will be presented in Section 1.2 and detailed in Chapter 2. In a depth-first algorithm, the itemset enumerated after $\{a\}$ is often $\{a, b\}$. Here $\{a, b\}$ is contained in two transactions and will be stored in the results.

If a generated itemset has a support lower than the threshold t_h, like the itemset $\{a, b, c\}$ with a support of 0, it is infrequent. Infrequent itemsets are discarded.

1.2 Frequent Itemset Mining Algorithms

1.2.1 ECLAT

There is a vast literature on algorithms for mining frequent itemsets. In this document, we focus on the Eclat algorithm [3]. Like most pattern mining algorithms, ECLAT follows a “generate and test” approach: potentially frequent itemsets are generated, then their frequency is tested in the database. ECLAT explores the search space of candidate itemsets in a depth-first manner, starting with small itemsets and augmenting them one item at a time. An example of a search tree is given in Figure 1.1. Note that the tree is asymmetric to avoid exploring twice the same itemset.

ECLAT relies on the *Apriori* principle [5] to prune the search space. If an itemset is infrequent in the database, any itemset that contains it will have a lower frequency and it will be infrequent too: it does not need to be explored.

Figure 1.1 represents the complete search space of itemsets explored by a depth-first search algorithm like ECLAT for the alphabet $\{a, b, c, d\}$. Consider again the example database of Table 1.1 with $t_h = 2$. Itemset $\{a, b\}$ has a support of 2 and is thus frequent. On the other hand, its augmentation $\{a, b, c\}$ has a support of 0: it is infrequent and will be discarded. By application of the *Apriori* principle, there will be no further augmentation of $\{a, b, c\}$, so the itemset $\{a, b, c, d\}$ will not be enumerated.

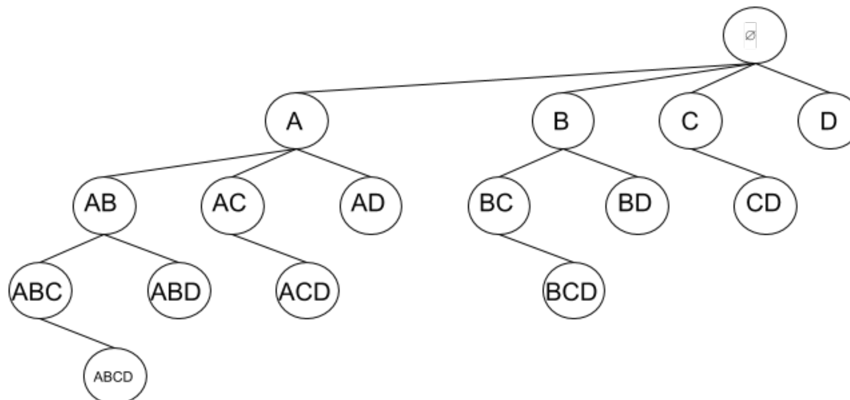

Figure 1.1: Example of a depth-first tree for the alphabet $\{a, b, c, d\}$ in ECLAT

1.2.2 FP-Growth

FP-Growth [4], or Frequent Pattern Growth, is a frequent itemset mining algorithm that relies on a compact data structure to represent the database and the patterns, called an *FP-tree*. This data structure is designed to improve the depth-first search of the algorithm in several ways:

- Thanks to a double way of connecting the nodes of the FP-tree, the traversal of the FP-tree can follow the same order as a standard depth-first search of itemsets, without redundancies. In particular, for any given itemset, it is easy to have access to all its immediate super-itemsets;
- Computing the support of an itemset can be done efficiently, as the necessary information are already stored in the tree.

Another improvement brought by FP-Growth is that for each frequent itemset found (each node of the depth first search), a *conditional FP-Tree* is constructed, with corresponds to the projection of the database on the transactions that contain that itemset. This allows, as the search deepens, to manipulate smaller and smaller databases, for which computations are more efficient. As the search tree of frequent itemset mining is generally "bottow-wide", i.e. with a few high level nodes relatively to a huge number of leaves, it is important to make computations as efficient as possible in the bottom parts of the search tree.

The first step of the FP-Growth algorithm consists in building the FP-Tree. This requires a rewritting of the database:

- Compute individual items frequencies;
- Remove infrequent items;
- Rewrite each transaction with items in decreasing frequency order;
- Sort lexicographically the rewritten database.

The FP-tree is then built as a prefix tree over this rewritten database. The nodes contain individual items, and following a path from the root (empty symbol) to a leaf gives a transaction. Each node also contains a number, which correspond to the number of transactions of the database that have the prefix given by the path from the root to this node.

In addition to the traditional prefix tree chaining, the FP-tree also maintains linked lists for each distinct item. These lists connect all the nodes of the FP-tree where the item appears.

As an example, consider again the database of Table 1.1, to be mined with a relative threshold of 25%. The order of items by descending frequency order is: $\{a:5, c:5, d:4, c:3\}$, there are no infrequent items. After rewriting, the database thus becomes the database shown in Table 1.3, which gives the FP-tree shown in Figure 1.2. The

$\mathcal{T}_4$	$\{a, c, d\}$
$\mathcal{T}_6$	$\{a, c, d\}$
$\mathcal{T}_8$	$\{a, d\}$
$\mathcal{T}_1$	$\{a, b\}$
$\mathcal{T}_7$	$\{a, b\}$
$\mathcal{T}_3$	$\{c\}$
$\mathcal{T}_5$	$\{c\}$
$\mathcal{T}_2$	$\{c, d, b\}$

Table 1.3: Rewriting of database $\mathcal{D}$

black edges represent the prefix tree chaining, while the green edges represent the transversal chaining of items.

Once the FP-tree is constructed, the FP-Growth search can start. It begins with singleton items, in increasing frequency order. For each item i (which are by construction frequent, the infrequent ones having been discarded), it is first output, as it is a frequent itemset. Then a conditional FP-tree restricted to the transactions containing

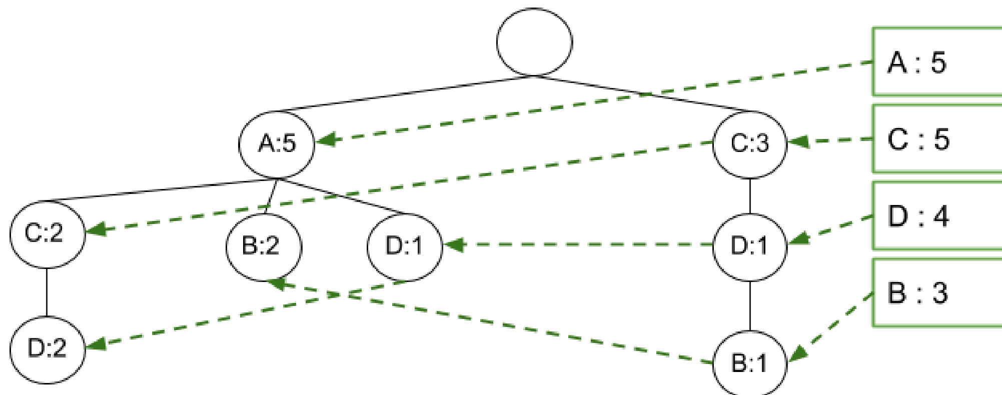

Figure 1.2: FP-Tree for the database of Table 1.1

item i is built. This is easy to do thanks to the transversal linking of the occurrences of the item. In this conditional FP-tree, i is removed as it is no longer needed.

In our example, the first item considered is b, which is output with a frequency of 3. The conditional FP-tree computed for b is shown in Figure 1.3. Note that the frequency counts have been update for all nodes, by removing frequencies of paths that do not contain b.

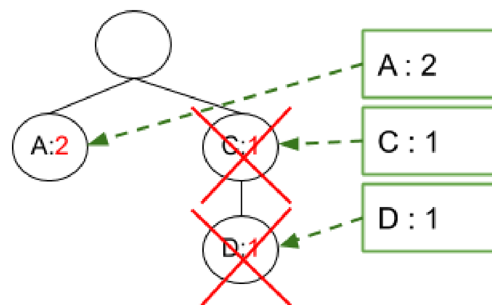

Figure 1.3: Conditional FP-tree for item b

Then the procedure is recursively applied, which we quickly sum-up below:

- Item d is selected. As we are in a conditional FP-tree for b, this corresponds to a candidate itemset $\{d, b\}$. It is not frequent ($d : 1$ in the conditional FP-tree) so it is removed and this branch of the recursion terminates;

- Item c is selected (candidate $\{c, b\}$). Here also the frequency threshold is not met so the recursion terminates for this branch;

- Item a is selected (candidate $\{a, b\}$). This time the frequency threshold is met, hence $\{a, b\}$ is output. A conditional FP-tree for $\{a, b\}$ is generated but it is empty, so the recursion stops for this branch also;
- All recursive calls possible for item b are finished, so the algorithm gets back at the top level and does the same steps with items d, c and finally a.

1.3 Pattern Sampling

An important drawback of pattern mining techniques such as frequent itemset mining is that they usually return a huge number of patterns, due to the pattern explosion phenomenon.

A solution proposed to remedy this issue is to do away with exhaustive listing of the results, and use sampling methods to return reduced but statistically representative results. There exist two common sampling methods for itemset mining: *input space sampling* (ISS) and *output space sampling* (OSS). Input space sampling refers to itemset mining on a downsampled database, whereas output space sampling returns a portion of the results a standard itemset mining algorithm would return.

Figure 1.4: Frequent Itemset Mining with no sampling

Figures 1.4, 1.5 and 1.6 illustrate how both forms of sampling differ from each other and from regular itemset mining. Figure 1.4 represents the classical process of frequent itemset mining, using a database as input, and ouputing its results as a (huge) list of frequent itemsets with their support. In Figure 1.5 we present Input Space Sampling. The database is downsampled before proceeding to the analysis, then the frequent itemset mining is executed as in conventional itemset mining. In Figure 1.6, we

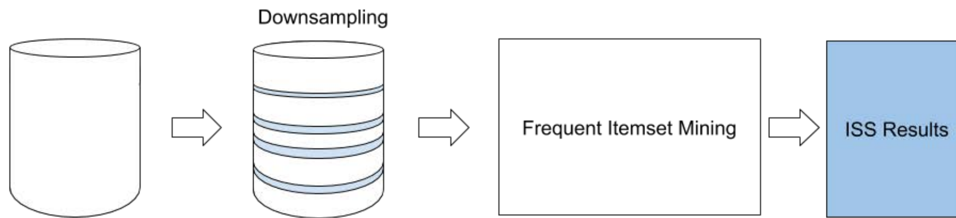

Figure 1.5: Frequent Itemset Mining with Input Space Sampling on a downsampled database

Figure 1.6: Frequent Itemset Mining with Output Space Sampling on an unaltered database

present a simple Output Space Sampling example. The frequent itemset mining is executed on the original database, and its results are randomly sampled at the very end. In real applications, this process is optimized to generate the final samples directly, without generating all of the frequent itemset mining results. We will present such techniques in a later section.

1.3.1 Input Space Sampling

The historically first sampling solutions introduced to tackle the issues of pattern explosion were focused on reducing the size of the input database by downsampling it [6], [7]. These sampling methods have been labeled by later works, such as in [8], as "*Input Space Sampling*" (ISS), since the sampling mechanism occurs on the input Database, but at their time of publication, they were labeled simply as "sampling".

A simple ISS technique is to draw a defined number of transactions from the database, say 10% or 1% of the total number of transactions for example, and organise it as a sampled database. This sampled database will be the one the itemset mining algorithm will

be applied to in order to generate the final results.

Three major issues arise with this type of technique. First, the number of final results are correlated but not directly linked to the size of the database. There are no strong guarantees on the upper bound of the final results with this type of sampling.

Second, as the downsampling gets more aggressive, the distribution of the sampled database has higher probability to deviate from the original.

Finally, the sizes of databases in real life application is ever increasing, meaning the sampling must be more and more aggressive to stay effective. This introduces larger and larger risks of bias in a sampled database's distribution leading to results of bad quality when compared to the original database.

1.3.2 Output Space Sampling

As opposed to input space sampling, output space sampling solves the pattern explosion issue by returning only a sampled set of all the results, with a target size provided by the user. The input database used for the output space sampling is the original database, and no modifications are made to it whatsoever.

As this approach is more recent, fewer works are present in the literature, and we focused our attention on three of them. Earlier works have adapted standard statistical sampling methods to the case of frequent itemsets [9], or exploit simple statistical observations on the structure of the pattern space [8], [10].

Monte Carlo Markov Chain The method in [9] proposes a random walk using a Monte Carlo Markov Chain approach to find suitable samples. The authors use the method to mine complex patterns such as graphs, but here we will explain it in the context of itemsets, as in this context itemsets can be considered as a particular case of graphs. A random walk of $T > 0$ steps proceeds as follow:

- Step 1: generate a random itemset I_0 that is frequent in the database (support $\geq t_h$)

- Step 2: list all neighbour itemsets of I_t $(t \geq 0)$. A neighbour is an itemset that can be generated by either removing a single item from I_t, or adding a single item of the alphabet not already present in I_h (infrequent neighbours are discarded)
- Step 3: draw randomly the next itemset I_{t+1} using the transition probabilities as defined in [9]
- Step 4: repeat from step 2 if $t < T$, else return I_t

The number of steps T dictates how long the random walk will be. T has to be big enough to allow the random walk to converge and return a proper sample. A random walk has to be executed for every sample itemset we desire to generate.

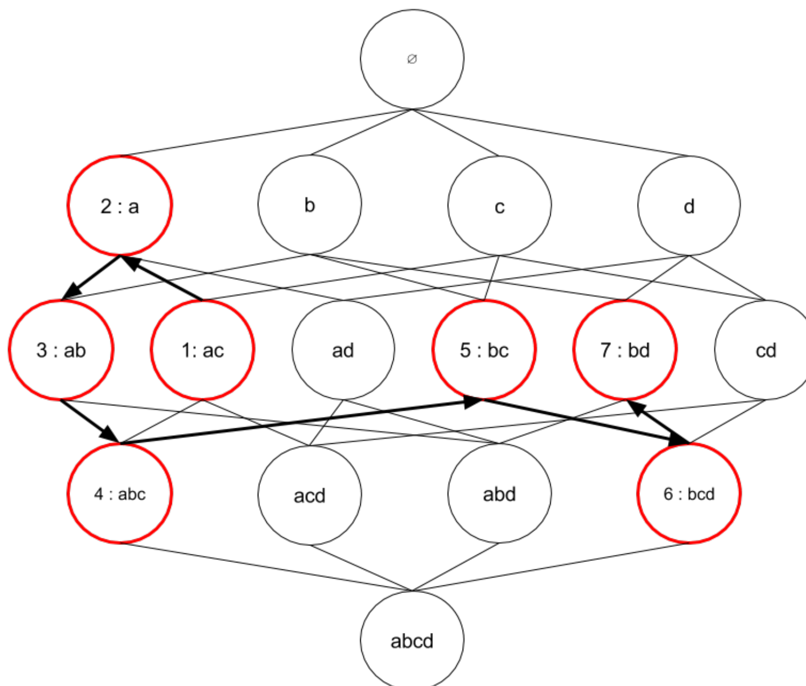

Figure 1.7: Random walk across the itemset lattice

Figure 1.7 shows an example of a random walk using a Monte Carlo Markov Chain. First, a random itemset is picked in the lattice, $\{a, c\}$. Using the transition probabilities from [9], a neighbour is chosen at random, $\{a\}$. A new neighbour is randomly chosen until the number of steps reaches the maximum number of steps, 7 in the example. At

$\mathcal{T}_1$	$\{a, b\}$
$\mathcal{T}_2$	$\{b, c, d\}$
$\mathcal{T}_3$	$\{c\}$
$\mathcal{T}_4$	$\{a, c, d\}$
$\mathcal{T}_5$	$\{c\}$
$\mathcal{T}_6$	$\{a, c, d\}$
$\mathcal{T}_7$	$\{a, b\}$
$\mathcal{T}_8$	$\{a, d\}$

Table 1.4: First step: draw a transaction from the database

$\{a, c, d\}$
$\{a\}$
$\{c\}$
$\{d\}$
$\{a, c\}$
$\{a, d\}$
$\{c, d\}$
$\{a, c, d\}$

Table 1.5: Second step: draw an itemset contained in the picked transaction

the 7th step, the random walk is on the itemset $\{b, d\}$. As it is the final step, the itemset $\{b, d\}$ is outputed as the sample.

Two-Steps The Two-Steps procedure is described in [8]. All transactions in the database are given a probability according to a given distribution for a future random drawing. Two simple probability distributions are given by the authors: the uniform distribution, where all transactions are as likely to be picked. And area function, where the probability to pick a transaction is proportional to its length. The first step is then to draw randomly a transaction w.r.t to these probability. The second step randomly draws an itemset from the powerset of the items of the transaction.

Tables 1.4 and 1.5 illustrate the Two-steps procedure with a simple example. We consider the uniform case where all transactions have the same probability $\frac{1}{8}$ to be picked. In Table 1.4, the 6th transaction, $\{a, c, d\}$, is randomly drawn in the first step. For the second step, all itemsets present in the picked transaction are enumerated, as shown in Table 1.5. As we used a uniform distribution during the first step, the

probabilities of each itemset will also be uniform during the second step, so $1/7$ for all itemsets. A random itemset is picked, for example $\{a, d\}$, and is the final sample.

The two steps have to be executed for every sample itemset we desire to generate. However, the two-steps procedure requires a non negligible initialization time to compute the probability to pick each transaction of the database if the distribution is not uniform. The sampling itself is very fast and can be called multiple times to return a desired number of samples. The time and resources needed to draw a random sample with the two-steps procedure is way faster than the random walk used by the Monte Carlo method, that can take more time than the initialisation of the two-steps procedure. An itemset in the database has the same probability to be outputed by the two-steps procedure as it would with exhaustive itemset mining followed by a random sampling on its results with the same probabilistic density.

Flexics The third technique we present is to use a regular itemset mining algorithm and use well designed random constraints to mine only a sub-space of all the possible itemset patterns. This method allows to output a number of itemsets within a user-defined range, and saves on computation time when compared to the itemset mining without sampling.

In this aim, two algorithmes were proposed by Dzyuba et al. in [2]. GFlexics uses a general purpose itemset miner that can operate on user defined constraints (e.g., closed itemsets, itemsets lengths within a range). EFlexics uses the standard ECLAT algorithm to simply mine frequent itemsets, but is faster than the general purpose itemset miner. Both rely on constraints developed by the SAT community.

Instead of returning one single sample (frequent itemset) per user query, the number of desired samples is within the range $[\![V_l, V_h]\!]$ with high probability, where V_l and V_h are respectively the low and high thresholds provided by the user.

This form of output space sampling is very efficient, and its algorithms have a good potential to be adapted to FPGA. Hence we based our work on pattern sampling on FPGA on this approach. More details are provided in the following section on OSS techniques based on random SAT constraints.

1.3.3 Output Space Sampling with random SAT constraints

We start this section with a refresher on the SAT problem. Boolean satisfiability problem, abbreviated SAT, is a field of computer science dedicated to finding logic interpretations that satisfy given boolean formulas. Given a system of boolean formulas, a SAT problem can be to either find any solution that satisfies the formulas, find all possible solutions, or find the total number of solutions (among others). $\neg a \wedge (b \vee c)$ is an example of a boolean formula, and assigning *false* to a and *true* to b and c is a correct solution for the problem of finding an assignement that satisfies the formula.

There may be many possible assignements satisfying a set of logical formulas. Works have been done to return unbiased parts of SAT solutions or estimate the total number of solutions. One of such SAT solving techniques is Weightgen [11]. Weightgen [11] is a toolchain that estimates the total weight of solutions of a SAT problem, where a weight function is given by the user, and finds a representative sample of all the solutions. The final sample has a weight lower than a user given bound. When the weight function is uniform, the total weight is equivalent to the number of solutions. This means the user specifies a number of solutions it is willing to receive, and Weightgen finds a bounded number of solutions that is statistically representative of the entire set of solutions.

In order to return a bounded number of solutions, Weightgen divides uniformly the space into sub-spaces, called cells, and keeps the solutions from a single sub-space. Weightgen starts by considering the entire space of solutions as its starting cell. Whenever the weight of all the solutions found so far in a cell exceeds the user threshold, it is divided again in a smaller cell.

A cell is divided in smaller cells using particular hash functions. These hash functions take the form of SAT constraints, that an element from a cell either satisfies or not. With this SAT constraint, it is possible to divide the cell into two sub-spaces, the sub-space of elements that satisfy the constraint, and the sub-space of elements that do not. To follow the method used in [11], the sub-space of elements that satisfy the SAT constraint is chosen as the new cell[5].

[5] If we desired to choose the sub-space of elements that do not satisfy the constraint, we can build a SAT constraint satisfied only by the elements that do not satisfy the previous constraint

The family of SAT constraints used by Weightgen are randomly generated XOR constraints. Each variable in the SAT problem has a probability $\frac{1}{2}$ to be in a XOR constraint. The general form of XOR constraint has the form:

$$\bigoplus_{i \in [\![1,N]\!]} b_i.X_i = b_0, \tag{1.1}$$

Where N is the number of variables, X_i ($i \in [\![1,N]\!]$) is the ith variable, and b_i ($i \in [\![0,N]\!]$) are the randomly generated coefficients of the constraint. Each coefficient has a probability $\frac{1}{2}$ to be 1 (variable is present) or 0 (variable is not present). b_0 is called the parity bit, and allow to either generate the constraint "$\bigoplus_{i \in [\![1,N]\!]} b_i.X_i$ is true" or "$\bigoplus_{i \in [\![1,N]\!]} b_i.X_i$ is false".

XOR operator

The boolean operator XOR (e**X**clusive **OR**) when used with two variables outputs **true** if exactly one of its input is **true**. If both of its inputs are **false** or both of its inputs are **true**, then the output is **false**. A XOR operator with $n \in \mathbb{N}$ is equivalent as using $n-1$ cascading XOR operators using two inputs. It can be simplified as a "parity check", the output of the XOR operator with n inputs is **true** if an even number of inputs is **true**, and **false** if an odd number of inputs is **true**.

In the case of pattern sampling, the *SAT problem* is to find frequent patterns, and a cell is a particular set of patterns that can be tested instead of the complete set of patterns. The variables used to generate constraints are the items from the alphabet. The *SAT problem* of finding frequent patterns can be solved in any particular way, for example with a dedicated pattern mining algorithm, as long as it is restricted to the cell considered.

This is the strategy used by Flexics: use the Weightgen algorithm (with a bounded oracle to ensure its execution is limited in time) to compute a cell with a weight lower than the user bound; then, use a pattern mining algorithm (general algorithm for con-

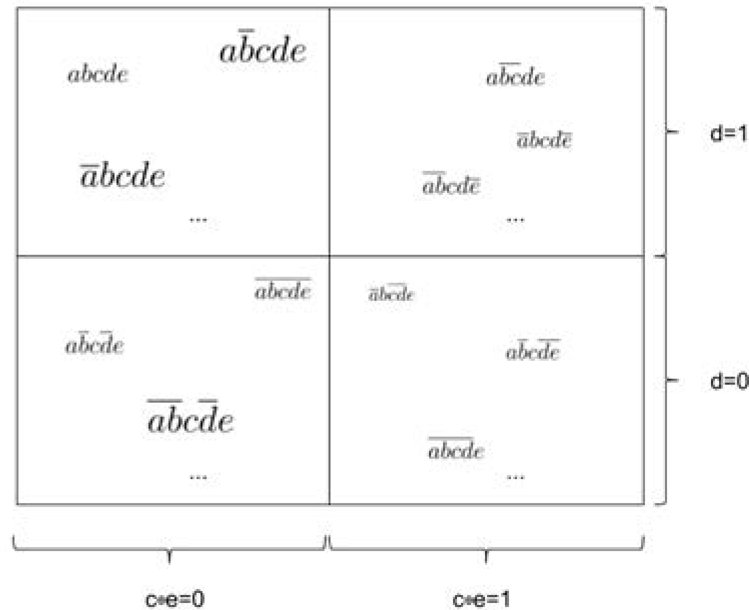

Figure 1.8: Itemset space divided into four cells using two SAT constraints

straint based patterns like closed itemsets, maximal itemsets, etc., or ECLAT for faster frequent itemset mining) on the cell.

Figure 1.8 illustrates how the output space can be divided uniformly to produce samples. In this case, we randomly generated the constraints $d = 1$ and $c \oplus e = 0$. By using these constraints and their complement (resp. $d = 0$ and $c \oplus e = 1$), we can display all itemsets as belonging to one of four cells. In itemset sampling, we are only interested in finding a sub-space representative of the whole space, so we pick a single cell, the one in the upper-left corner. All cells are statistically equivalent, so we pick the cell that contains the itemsets that satisfy all constraints generated so far.

1.4 Conclusion

The Flexics framework is one of the most efficient for principled pattern sampling. Furthermore, the original paper shows how to build it using Eclat, an FPGA-friendly algorithm (compared to other frequent itemset miners). And the approach is based on XOR constraints, which are well handled by hardware, it thus seems especially relevant to base an FPGA-based pattern sampling method on the Flexics framework. To the best

of our knowledge, this is the first pattern sampling accelerator on FPGA architectures. This is our major contribution in this thesis, presented in the next chapters.

CHAPTER 2

ACCELERATING ECLAT ON FPGA

Introduction

The Flexics framework for itemset sampling [2] propose two implementations: one is a general purpose sampler, whereas the second is based on the ECLAT algorithm [3] to provide better performance. Our contribution aimed to provide an itemset sampler algorithm and architecture that can be accelerated on FPGA boards, while ensuring strong statistical guarantees similar to Flexics. As the ECLAT algorithm is rather standard in the itemset mining community, there exist already several FPGA implementations of it. Our implementation is heavily based on [1], providing some improvements. We then describe how to add the output space sampling feature to the accelerator, and how well its implementation on FPGA compare against the current state of the art. To end the chapter, we propose an extension to a more distributed computation of the itemsets, though this work was still in development at the time of writing and no experiments have been performed.

2.1 General Introduction to FPGA Acceleration

In Computer Science, the most common computer architectures are centered around a central processor unit (**CPU**). A typical CPU reads and executes instructions from a binary code in memory one by one, and reads/writes data in memory. With such methods as out-of-order execution, multi-threading, or by using multiple CPUs, it is possible to process more than one operation at a time. Graphical Processor Units (**GPU**) and multicore CPUs are examples of architectures that use such techniques.

The boost in performance gained with parallel execution comes at a cost, either more hardware has to be allocated on the chip to hold multiple processors, or the

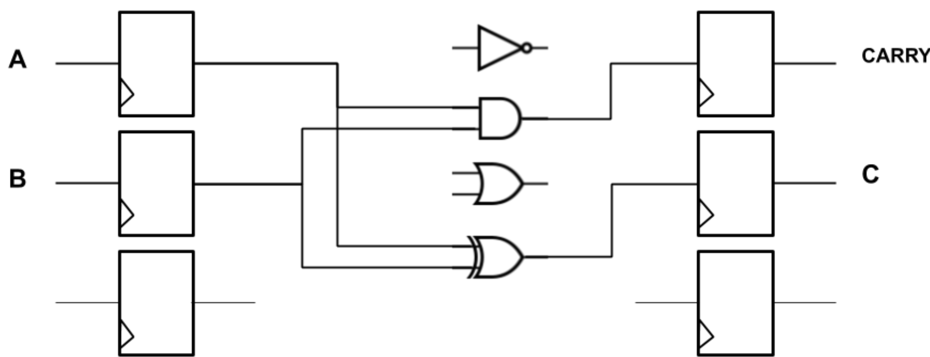

Figure 2.1: Example implementation of 1 bit adder with carry

processors have to be scaled down. For example a GPU is comprised of an array of processor clusters, each containing a defined number of processors. In a cluster, all the processors execute the same instruction at a given time (organisation called "single instruction multiple data", **SIMD**), and provide good performance for parallel tasks such as image processing, but perform worse than general purpose CPU otherwise.

A Field-Programmable Gate Array (**FPGA**) is an integrated circuit holding reconfigurable logic. Instead of holding of one or more processing units that read and execute binary codes, it consists of many smaller electronic components whose logic function can be configured and whose input and output can be rewired. Application Specific Integrated Circuits (**ASIC**s) have architectures similar to FPGAs, but provide better performance and cannot be reconfigured once built. FPGAs are very good candidates to prototype an architecture before mass producing a new product with ASICs.

Figure 2.1 shows a toy example of a 1-bit adder with carry. A, B, C and $CARRY$ are input and output signals. The result of the addition $A + B$ is send to C, and the $CARRY$ signal holds 1 if the addition overflows. In this example, the signal are stored temporarily in flip-flops (boxes in the figure) surrounding the input and output. Flip-flops are electronic components that store the value of its input signal whenever it receives a rising edge of a clock signal (symbolized by the triangle). Wires carrying clock signals are not shown for better readability. Each clock cycle, the input flip-flops update the values of A and B, and transmit these values for the whole duration of the next clock

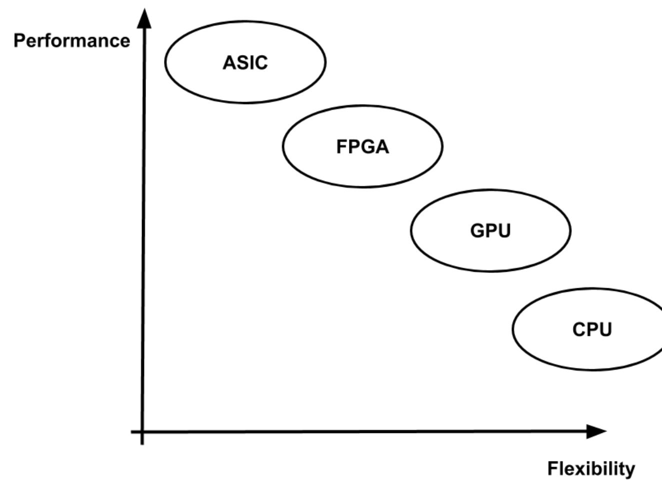

Figure 2.2: Trade-off of performances Vs. flexibility of multiple architectures

cycle. Similarly, the values of C and $CARRY$ are updated at each clock cycle using the latest values of A and B. Due to the flip-flops, it takes an entire clock cycle for the outputs to be up to date after the input change.

Figure 2.1 displays unused components (two latches and two logic gates are unconnected). On a real FPGA, the specifics of the board dictates what type and how many gates are available on the chip, while the design tool picks and chooses which component to use and how to route them depending on the design provided by the developer.

Figure 2.2 illustrates the four previously mentioned architectures and their relative performance/flexibility. CPUs are general purpose and can run any type of applications. GPUs can offer better performance, but are not always a viable option compared to CPUs, as in practice GPUs are only efficient when the time spent on data transfer with the memory is amortized. FPGAs and ASICs can outperform other architectures on specialized tasks, but can only process one task once configured, and that configuration is definitive for an ASIC.

Once configured, the FPGA is able to handle specific tasks without the need of a program in memory. A FPGA usually runs a slower clock than standard CPU, but allows

for massive parallelism; FPGAs usually outperform standard CPUs both in timing and energy consumption when the application is suited for parallel computations.

The internal logic of an FPGA can be configured using hardware description language, that "describes" the layout of the circuit, or high-level synthesis to allow for more abstraction. High-Level Synthesis (**HLS**) consists in a toolchain that allows user to configure FPGA architectures by writing C/C++ code. HLS is not able to convert any C/C++ code to FPGA designs, as it must work with operations realisable without a CPU on the board and limited hardware. This imposes restrictions on the code used for HLS, and although it is possible to compile a C/C++ code meant for HLS and run it on a regular CPU, it is not guaranteed to offer the best performance compared to a a code written with usual architectures in mind.

2.2 Itemset Mining on FPGA

Two families of methods are described in the literature for itemset mining on FPGA. A first method is to use a controller like a CPU allowing for complex controls, and restrict the FPGA to accelerate the support counting of the itemsets, as seen in [12]. The other methods accelerate the whole application on hardware [1], [13]. The main difficulty is to streamline the exploration of the tree-shaped search space for an efficient execution on FPGA. The asymmetry of the tree is a first problem. A second issue is that pruning by the Apriori property, although strictly mandatory for performance, makes the search tree unpredictably irregular, which is not good for getting performance out of FPGA acceleration. The approach proposed by [1] elegantly solves both problems, and we used it as a starting point in the design of our approach for FPGA-based pattern sampling.

2.2.1 ECLAT Algorithm

As already mentioned, ECLAT is a frequent itemset mining algorithm, that generates and tests itemsets following a depth-first method. Figure 2.3 represents the itemsets organized in a depth-first tree. Figure 2.4 is a second, more compact way to represent the same tree, by only using the last item in the itemset in a node. To retrieve the itemset represented by a node in a suffix tree such as presented in Figure 2.4, we

congregate all the items present in the nodes on the path between the node we are interested in, and the root of the tree.

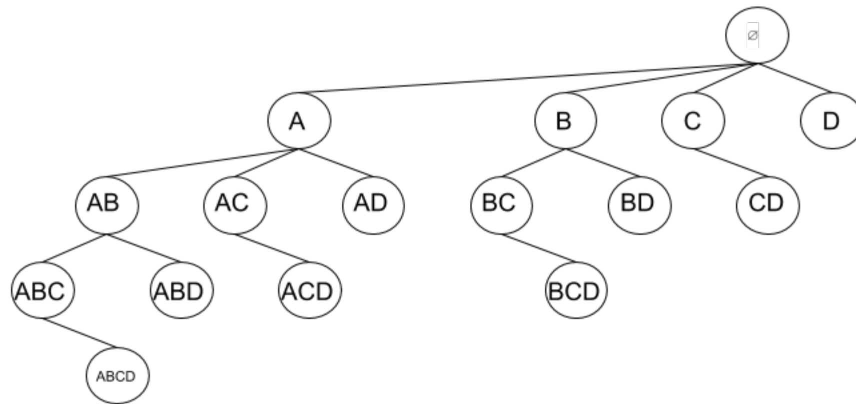

Figure 2.3: *Depth first* tree for the alphabet A,B,C,D

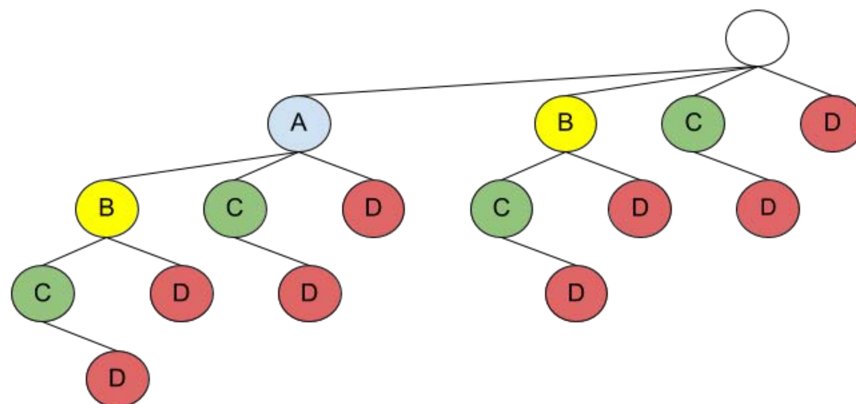

Figure 2.4: Suffix tree for the alphabet A,B,C,D

ECLAT scans the tree using the Apriori principle. If an itemset is not frequent, all itemsets that contain it are infrequent, and do not need to be tested. In the case of a depth-first tree, this means the algorithm can discard all children and their descendants[1] of the current itemset. This pruning technique allows to reduce greatly the execution without heuristics.

[1] In this thesis, we consider a descendent node in a tree as a child or child of another descendent.

The generation of itemsets in the ECLAT algorithm is recursive. If an itemset $\mathbf{X}$ is frequent, $\mathbf{X} \cup Y$ will be tested for each item Y higher than all items of $\mathbf{X}$ in the lexicographic order. The itemset tree structure illustrates this recursion. As it is possible to see in Figure 2.3, the structure of the sub-tree of all the descendants of an itemset depends only of the last item. For example, an itemset having C has last item will have a sub-tree of alphabet size exactly 1 (before pruning by Apriori property) containing a single node with last item D.

2.2.2 FPGA Acceleration of ECLAT in [1]

The implementation proposed in [1] shows that the ECLAT algorithm can be rewritten with a *while* loop instead of recursive calls like in the usual software implementation of ECLAT. The idea of this *while* loop is the following: the algorithm tests the nodes in a depth-first fashion as shown in Figures 2.3 and 2.4, by testing the "leftmost" branches first in the tree (A, AB, ABC, ABCD, then ABD, then AC, ACD, ...), with the caveat that branches might be killed early when itemsets are infrequent. We illustrate another representation of the depth-first tree in Figure 2.5 to highlight the different branches for the alphabet $\{A, B, C, D\}$. The representation of the branches can be useful to give some intuitions on the algorithm, but has several shortcoming for in-depth details explanations. The 'branch representation' will sometimes be used in this section, while the rest of the document will primarily focus on the nodes of the trees. The end condition of the *while* loop is met when all nodes/branches are tested; for the depth-first algorithm, this means the last branch containing only the singleton of the last item (D in our case). This node will always be tested last, and there are no prior itemset that can potentially kill it, no matter the alphabet.

The strategy described by [1] divides itemsets in two parts called FIFO A and FIFO B, to compute the next candidate itemset depending on the last parent itemset. FIFO B, the prefix, contains the entirety of the last frequent parent tested. FIFO A, the suffix, contains the unique item that differs between the candidate and the last parent itemset.

The column *Result* in Table 2.1 corresponds to the candidate itemset, equals to $FIFO\ A\ \&\ FIFO\ B$. Columns *Valid* and *Output* signal if an itemset is frequent. When a frequent itemset is found, the values of FIFO A and B are stored in two separate

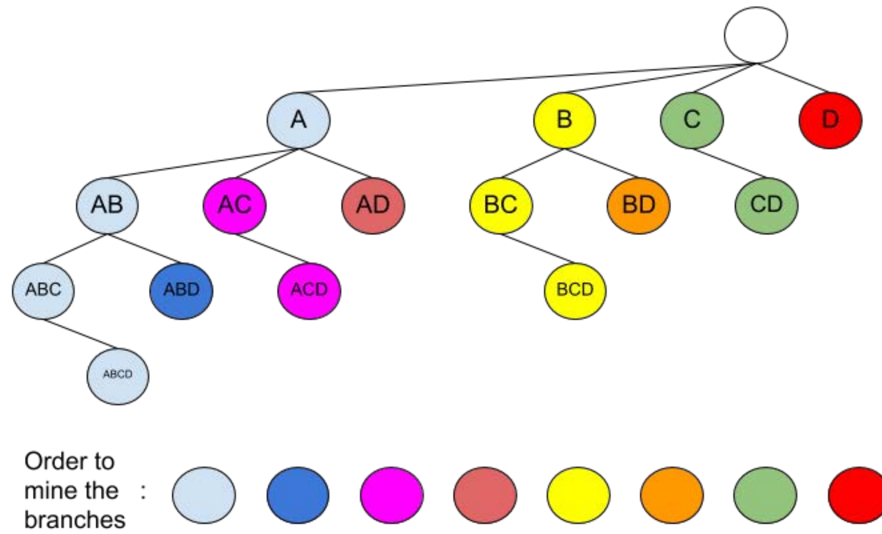

Figure 2.5: *Depth first* tree represented as a collection of branches and their order in the scan

memories. *Flush* indicates if the current itemset is infrequent or a leaf of the tree. In these cases, the algorithm has to start a new branch, by loading from memory the values of FIFO A and FIFO B.

A state machine is in charge of selecting what values are loaded of removed from the FIFO memories. The authors of [1] did not present this state machine in their paper, and did not shared their code. We thus re-implemented their approach in two different ways: one way where we tried to be as faithful as possible to the paper [1], and another way where we added our own improvements. When describing in details the algorithm of [1] below, the design choices are those of our faithful implementation, which may have minor differences with the original implementation. We will present after our improved approach.

We reuse the database $\mathcal{D}$ in Table 2.2 used in the previous chapter with the results for $t_h = 2$ provided in Table 2.3 for the example which is illustrated in Table 2.1.

Step 1 The algorithm starts with no itemset in *'FIFO B'*, used to store the prior itemset. *'FIFO A'* is initialised with the first item *'A'*. *'Result'* is set with the concatenation of *'FIFO A'* and *'FIFO B'*: the itemset *'A'* and its frequency are tested. Since *a* is frequent

Step	FIFO B	FIFO A	Result	Valid	Output	Flush
1		a	a	Yes	Yes	No
2	a	b	a&b	Yes	Yes	No
3	a&b	c	a&b&c	No	No	No
4	a&b	d	a&b&d	No	No	Yes
5	a	c	a&c	Yes	Yes	No
6	a&c	d	a&c&d	Yes	Yes	Yes
7	a	d	a&d	Yes	Yes	Yes
8		b	b	Yes	Yes	No
9	b	c	b&c	No	No	No
10	b	d	b&d	No	No	Yes
11		c	c	Yes	Yes	No
12	c	d	c&d	Yes	Yes	Yes
13		d	d	Yes	Yes	Yes

Table 2.1: "FIFO Control Sequence" [1], adapted for $\mathcal{D}$

with a support of *5*, the flag *'Valid'* is set to *'Yes'*, and *a* is added to the list of frequent itemsets (flag *'Output'* will always be set to the same value as *'Valid'* during the run). The flag *'Flush'* is set to *'No'* as the current branch is not yet over (detected when *'FIFO A'* contains the last item of the alphabet or *'Valid'* contains *'No'*). The values of *'FIFO A'* and *'FIFO B'* are stored in memory as intermediate results.

Step 2 As the first branch is still tested, *'FIFO B'* is assigned with the prior value of *'Result'*, *'a'*. *'FIFO B'* is then assigned with the itemset following its last value in

$\mathcal{T}_1$	$\{a, b\}$
$\mathcal{T}_2$	$\{b, c, d\}$
$\mathcal{T}_3$	$\{c\}$
$\mathcal{T}_4$	$\{a, c, d\}$
$\mathcal{T}_5$	$\{c\}$
$\mathcal{T}_6$	$\{a, c, d\}$
$\mathcal{T}_7$	$\{a, b\}$
$\mathcal{T}_8$	$\{a, d\}$

Table 2.2: Example database $\mathcal{D}$

Itemset	Support
$\{a\}$	5
$\{a, b\}$	2
$\{a, c\}$	2
$\{a, c, d\}$	2
$\{a, d\}$	3
$\{b\}$	3
$\{c\}$	5
$\{c, d\}$	3
$\{d\}$	4

Table 2.3: Frequent itemsets from $\mathcal{D}$ with $t_h = 2$

the alphabet: *'b'*. *'Result'* is set with the concatenation of *'FIFO A'* and *'FIFO B'*: the itemset *'a&b'* and its frequency are tested. Since *a,b* is frequent with a support of *2*, the flag *'Valid'* is set to *'Yes'*, and *a,b* is added to the list of frequent itemsets. The flag *'Flush'* is set to *'No'* as the current branch is not yet over. The values of *'FIFO A'* and *'FIFO B'* are stored in memory as intermediate results.

Step 3 As the first branch is still tested, *'FIFO B'* is assigned with the prior value of *'Result'*, *'a&b'*. *'FIFO B'* is then assigned with the itemset following its last value in the alphabet: *'c'*. *'Result'* is set with the concatenation of *'FIFO A'* and *'FIFO B'*: the itemset *'a&b&c'* and its frequency are tested. This time, *a,b,c* is not frequent with a support of *0*, the flag *'Valid'* is set to *'No'*, and *a,b,c* is discarded. The flag *'Flush'* is set to *'No'* as the current branch is not yet over, even if it is killed. The values of *'FIFO A'* and *'FIFO B'* are not stored in memory as intermediate results since the *'Valid'* value is *'No'*.

Step 4 As the second branch started, *'FIFO B'* value is not assigned because *'Valid'* was set to *'No'* and *'Flush'* was set to *'No'*, and it keeps the value *'a&b'*. *'FIFO B'* is assigned with the itemset following its last value in the alphabet: *'d'*. *'Result'* is set with the concatenation of *'FIFO A'* and *'FIFO B'*: the itemset *'a&b&d'* and its frequency are tested. *a,b,d* is not frequent with a support of *0*, the flag *'Valid'* is set to *'No'*, and *a,b,d* is discarded. The flag *'Flush'* is set to *'Yes'*, since *'FIFO A'* contains 'd', meaning we reached the end of the branch. The values of *'FIFO A'* and *'FIFO B'* are not stored in memory as intermediate results since the flush flag is set to 'Yes' (without too many details, even if *'Output'* is set to *'Yes'*, a flush implies the values will never be used later, so it can be safely discarded).

Step 5 As the third branch is started, *'FIFO B'* value is loaded from its memory, *'a'* (determined by a state machine after a flush). *'FIFO B'* value is computed using a value loaded from memory by the state machine. 'b' is loaded, but the actual value will be the following item in the alphabet: *'c'*. *'Result'* is set with the concatenation of *'FIFO A'* and *'FIFO B'*: the itemset *'a&c'* and its frequency are tested. *a,c* is frequent with a support of *2*, the flag *'Valid'* is set to *'Yes'*, and *a,c* is added to the list of frequent itemsets. The flag *'Flush'* is set to *'No'*. The values of *'FIFO A'* and *'FIFO B'* are stored in memory as intermediate results.

Step 6 As the third branch is still tested, *'FIFO B'* is assigned with the prior value of *'Result'*, *'a&c'*. *'FIFO B'* is then assigned the itemset following its last value in the alphabet: *'d'*. *'Result'* is set with the concatenation of *'FIFO A'* and *'FIFO B'*: the itemset *'a&c&d'* and its frequency are tested. *a,c,d* is frequent with a support of *2*, *'Valid'* is set to *'Yes'*, and *a,c,d* is added to the list. The flag *'Flush'* is set to *'Yes'* since *'FIFO A'* contains 'd' meaning we reached the end of the branch. The values of *'FIFO A'* and *'FIFO B'* are not stored in memory as intermediate results because of the flush flag.

Step 7 The fourth branch is started, *'FIFO B'* value is loaded from its memory, *'a'* (determined by a state machine after a flush). *'FIFO B'* value is computed using a value loaded from memory by the state machine. 'c' is loaded, but the actual value will be the following item in the alphabet: *'d'*. *'Result'* is set with the concatenation of *'FIFO A'* and *'FIFO B'*: the itemset *'a&d'* and its frequency are tested. *a,d* is frequent with a support of *3*, the flag *'Valid'* is set to *'Yes'*, and *a,d* is added to the list of frequent itemsets. The flag *'Flush'* is set to *'Yes'*, since *'FIFO A'* contains 'd' meaning we reached the end of the branch. The values of *'FIFO A'* and *'FIFO B'* are not stored in memory as intermediate results because of the flush flag.

Step 8 The fifth branch is started, *'FIFO B'* value is loaded from its memory, this time it is the empty itemset (determined by a state machine after a flush). *'FIFO B'* value is computed using a value loaded from memory by the state machine. 'a' is loaded, but the actual value will be the following item in the alphabet: *'B'*. *'Result'* is set with the concatenation of *'FIFO A'* and *'FIFO B'*: the itemset *'B'* and its frequency are tested. *b* is frequent with a support of *3*, the flag *'Valid'* is set to *'Yes'*, and *a,d* is added to the list of frequent itemsets. The flag *'Flush'* is set to *'No'*, as the current branch is not yet over. The values of *'FIFO A'* and *'FIFO B'* are stored in memory as intermediate results.

Issues with the implementation While many in-depth details of the implementation are provided by [1], several important parameters are left unexplained and require reverse-engineering. Mainly, the state machine is a black box, and several points (why the value loaded from memory for *'FIFO A'* requires to compute its next itemset instead of directly using the stored value after a flush in steps 5, 7, 8; or why it is not required to store the FIFOs' values during a flush) are not present in the article. For this reason,

we preferred to provide more explanations of our own implementations based on [1], to avoid providing wrong information about this one.

In [1], *flush* is used as a signal for FIFO A to load a previously stored value from a memory called *intermediate results*, different from the final results to be outputed at the end of the run. As we explained previously, we developed a solution based on this approach, and a second of our own, with an in-depth explanation in the next section.

Heap solution based on [1] When an itemset is not a leaf and has at least one child itemset left to test, its value will be kept in memory so it can be retrieved by FIFO B and FIFO A. As a branch is only investigated as long as the tested itemsets are frequent, there is no need to store intermediate results of FIFO A and FIFO B when an itemset is infrequent. When an itemset is frequent, but not a leaf, its children itemset will be investigated, meaning any of these children might need to access its information at a latter point in the execution. When an itemset tested is a leaf of the tree, not only it has no potential child to use its intermediate results, but it also means the parent if the current itemset had all its children tested (when an itemset is a leaf, it must contain the last letter of the alphabet, thus there are no letter left to create another itemset from its parent with a latter letter). This means that when an itemset is a leaf, neither its information, nor the information of its parent, will be loaded from the intermediary results, and can be removed safely if they exist. This type of *Last in, First Out* read/write access in a memory in a typical use case for a heap, an itemset only requires the values of FIFO A and FIFO B of its previous sibling, as its reuses its FIFO B value directly (containing their parent itemset) and the value following the value of its FIFO A (explaining the operations of steps 5, 7, 10 and 13 in the previous example). When the algorithm goes down one level in the tree, it stores data in one stack of the heap, and removes a stack of the heap when it backtracks by one level. Lateral movements in the tree do not add/remove data from the intermediate results in the heap. The maximum size of the heap to be allocated in memory is determined only by the size of the alphabet, up to n stacks in the heap for an alphabet of n items, while the values stored by FIFO B in the tree can consist of itemset containing up to n letters. A naive implementation would thus results in a $O(n^2)$ memory allocation size.

2.3 Contribution to Memory Optimized FPGA Acceleration of ECLAT

This section aims to present our own ECLAT accelerator for FPGA architecture. While inspired by [1], we tried to give a self-sufficient explanation of our contribution that do not require a full understanding of previous ECLAT architectures. Our strategy uses a separation of itemsets in FIFO A and FIFO B from [1], translated to prefix and suffix. The flushing mechanism is totally reworked, as there are three distinct cases when scanning the tree: current itemset is a leaf, current itemset is not a leaf but is frequent, and current itemset is neither a leaf nor frequent.

To simplify a bit future explanations and proofs, we introduce the concept of *I-step*. I-step is a term used in sequence mining, a field of data mining that encompasses itemset mining. Applying I-step on an itemset[2] means to add an item to the itemset to generate a new itemset. The item to be added during the I-step must satisfy two conditions:

- The item must not be present in the original itemset.
- The item must be after the last item of the itemset in the lexicographic order used for the alphabet.

For example, a valid I-step for the itemset AC is to add the item D. A and C are not valid items as they are already present. B is not a valid item either because the itemset contains the item C, which is after B in the lexicographic order.

We also consider for any itemset its last item in the lexicographic order to be called its *suffix*. We define the *prefix* of an itemset the itemset itself if stripped of the suffix.

A leaf in the tree is a node with no child node. Such nodes correspond to itemsets that cannot be used for an I-step to generate a child itemset. Itemsets corresponding to leaf nodes in the tree will be referred to as leaves, without necessarily mentioning the tree. All itemsets containing the last item of the alphabet are leaves, and all leaves contain the last item of the alphabet.

[2]The original definition for sequence is out of the scope of this thesis, so we propose a simplified version for itemsets in a depth-first algorithm.

Proof. If an itemset contains the last item of the alphabet, there exist no suitable item for an I-step, so the itemset has no children. If an itemset does not contain the last item of the alphabet, this item is a candidate for an I-step, and there exist at least one itemset that can be generated as a child of the original itemset. □

Generating a new itemset from a current itemset

When an itemset is frequent and is not a leaf, the next itemset to test is its first child (generated by an I-step).

When an itemset is not frequent and is not a leaf, it is guaranteed to have at least a sibling left to test. In this case the next itemset to test is its first sibling.

Proof. If an itemset I is not a leaf, it means that it does not hold the last item of the alphabet. So there exists at least one item of the alphabet that can be used to perform an I-step on the parent of I. Thus, we can create an itemset by removing the last item (suffix) of I and adding an item that was candidate for I-step. The new itemset I' and original itemset I share the same prefix, but have a different suffix, so they are represented by siblings in the tree. □

When an itemset is a leaf, it is guaranteed to not have a sibling, but its parent has at least one. There does not exist any item greater than all the items in the itemset, otherwise it would have at least one child because such item could be used for an I-step. The current itemset's parent has at least another sibling. In this case the next itemset to test is the first sibling of the current itemset's parent.

Proof. If an itemset is a leaf, it contains the last item of the alphabet, so it is not possible to generate a new itemset by replacing its suffix with a suffix bigger in the alphabet. For this reason, no leaf in the tree can have a sibling to its right. For the parent itemset of the current itemset, there exists at least one item candidate for I-step (the current itemset is itself a child of its parent). So the parent of the current itemset has at least one sibling to its right. □

Memory optimized solution There exists a mathematical relation between the values of FIFO A before and after a *flush* that can be computed. As we will explain in the following paragraphs, when the current itemset is a leaf of the tree, and the FIFO

A is flushed, the next itemset is the first sibling of the current itemset's parent. This particular itemset can be computed using information stored in FIFO B. This alleviates the need to store the values of FIFO A for later use. Using the relation "the next itemset is the first sibling of the current itemset's parent" has also a significant advantage over retrieving previously stored items, namely it can be used by our sampler described in the next chapter. As the alphabet and the size of the tree change dynamically for our sampler, we did not find a simple workaround to use the technique proposed by [1], while the relation "the next itemset is the first sibling of the current itemset's parent" is still straightforward when the alphabets used are dynamic. While this solution removes the need for a heap to store intermediary results, it requires a function able to find the last letter in an itemset efficiently, which is implementation dependant. The solution we propose finds the letter of any itemset in $O(ln(n))$, n being the number of letters in the alphabet, fixed at compile time, thus constant during an execution. Details will be shown in the *Architecture* section.

The rest of this section explains in more details our solution that do not require a heap, and both solutions will be compared in the results section.

For any itemset, there are three possibilities:

- The itemset is neither frequent nor a leaf: the next itemset is its first sibling. Example Figure 2.6.
- The itemset is frequent but not a leaf: the next itemset is its first child. Example Figure 2.7.
- The itemset is a leaf: the next itemset is the first sibling of the current itemset's parent. Example Figure 2.8.

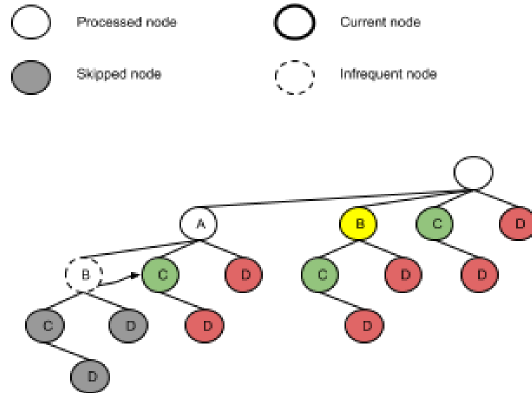

Figure 2.6: Itemset is infrequent; next itemset is it right sibling

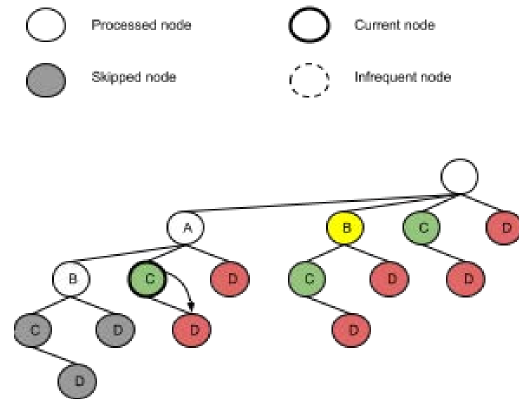

Figure 2.7: Itemset is frequent; next itemset is it first child

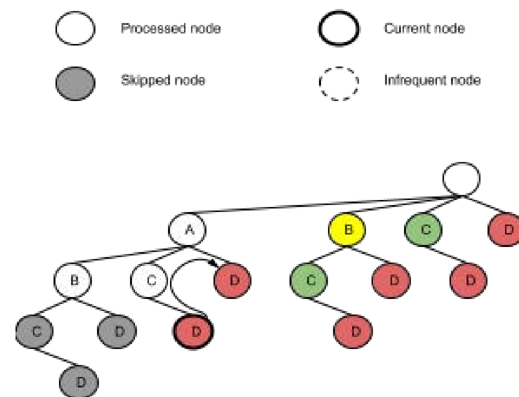

Figure 2.8: Itemset is a leaf; next itemset is the first sibling of its parent

The prefix value (stored in FIFO B) can be computed depending on the three cases:

- The itemset is frequent but not a leaf: the suffix value is added to the prefix (FIFO B = FIFO A & FIFO B as in [1]).
- The itemset is neither frequent nor a leaf: prefix stays the same.
- The itemset is a leaf: the highest item from the prefix is removed (the new prefix value becomes an older value corresponding to the itemset's parent).

The suffix value for the next itemset corresponds to the item added to its parent itemset.

When the next itemset is the current itemset's children, the value has to hold the first item higher than the highest item in the current itemset. As the highest item in the current itemset is currently held in the suffix value, it is easy to update.

When the next itemset is the current itemset's first sibling, the suffix value has to hold the first item higher than the highest item in the current itemset.

When the next itemset is the current itemset's parent first sibling, the suffix value has to hold the first item higher than the highest item in the current itemset's parent. This item is not present in the suffix value, but can be found in the prefix. As the prefix holds the current itemset's parent, this item is the last item in the prefix when using the lexicographical order. In the hardware implementation we will develop in later section, itemsets are represented as presence bitvectors (the nth of the vector is set to 1 if the nth item is present. In that case, the highest item corresponds to the highest leading bit of the prefix.

The suffix value (stored in FIFO A) depending on the current itemset is computed as follow (where the "+1" operator means fetching the next item in the alphabet):

- The itemset is frequent but not a leaf: FIFO A = FIFO A + 1
- The itemset is neither frequent nor a leaf: FIFO A = FIFO A + 1
- The itemset is a leaf: FIFO A = leading_bit(FIFO B)+1

These cases are shown in Figures 2.6, 2.7 and 2.8.

2.4 Architecture of the ECLAT accelerator

2.4.1 Overview

Figure 2.9 shows an overview of the accelerator architecture implemented in the programmable logic fabric of a Zynq FPGA from Xilinx. The *zc706* evaluation board used for the experiments holds a *System on Chip*, containing two embedded Cortex A-9 CPU cores and a XC7Z045FFG900-2 FPGA, connected to an off-chip 1GB DDR3 memory. Our design uses communications between the FPGA and the CPUs only when the hardware application starts and stops, in order to interface with the user. All accesses from the accelerator in the FPGA to the main memory are achieved using the *High Performance AXI* port, without cache coherency protocols. These settings speedup dramatically the communication between the accelerator and the RAM, since there are no other application than the accelerator on the board to concurrently access the memory, and the CPU cores remain in stand-by while the FPGA is working. This section details the support counting acceleration of Figure 2.9 and an additional feature we called correspondence list.

2.4.2 Finding last item from an itemset

Our accelerator represents itemsets in an alphabet of size n as bitvectors of size n. The ith bit of an itemset X is set to 1 if the ith item of the alphabet is present in X, and set to 0 if it is absent. Finding the last item of an itemset is equivalent to finding the highest leading bit of a bitvector of size n. A naive implementation will use a while loop that check the nth bit to see if its value is set to 1, if not, check the $(n-1)^{th}$, and so on. This would results in a variable execution time bounded by a $O(n)$ factor, but also would require n bitshifting operations that can be very costly if the compiler is given bishift operations with non-constant shifts.

Instead, we first cut the bitvector in half with a bitshift of $\frac{n}{2}$, and check if the leading half of the vector is equal to zero. If it is not zero, then the leading bit is contained in this half. If it is equals to zero, then the leading bit is in the other half. The resulting containing the leading bit is recursively cut in half, until the vector is of size 1, after $\frac{ln(n)}{ln(2)}$ bitshifts. The execution time is thus constant for any input, and the bitshift used are always $\frac{n}{2}$, $\frac{n}{4}$, $\frac{n}{8}$, 2, 1, and can be hardcoded in the FPGA.

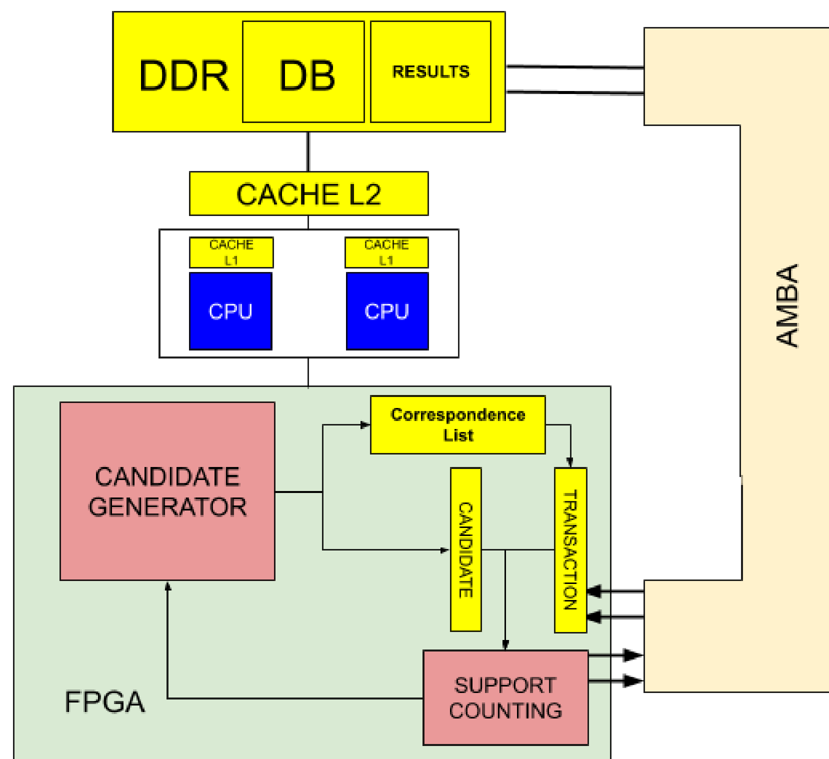

Figure 2.9: Overview of the ECLAT accelerator architecture

Table 2.4 shows an example bivector of size 16 and how to find its leading bit position, returned as *offset*. At first, the vector is split in two vectors of size 8. Its leading vector, 10101100 is not equal to zero, thus contains the leading bit of the whole vector. the other half can be discarded for the rest of the function. The offset of the leading is increased by 8, the number of bits we discarded this far. Then the leading vector is split in two vector of size 4. 1100 is not equal to zero, so we keep this vector and increase the offset by 4, now with a value of 12. The vector 1100 is split in two vectors, 11 and 00. The leading vector is equal to zero, so we have to keep the 11 vector, and the offset is not increased. Finally, the last vector is split in two bits, and the leading bit is non-zero, meaning the final offset is 13, and the leading bit of 1010010110101100 is the 13th bit.

Table 2.4: Finding the leading bit of a bitvector of size 16

	bitvector	offset
Original bitvector:	1010010110101100	0
Split bivector with bishift of 8:	10100101 10101100	8
Split bivector with bishift of 4:	1010 1100	12
Split bivector with bishift of 2:	11 00	12
Split bivector with bishift of 1:	1 1	13
Final bitvector:	1	13

2.4.3 Support Counting

The support counting loop of the architecture takes most of the execution time, given the finite memory bandwidth. The loop takes in transactions from the database, checks if the current candidate is present, and if so, increments a counter. Given a transaction bitvector and a candidate itemset bitvector, for the candidate to be present in the transaction, all bits set to 1 in the candidate have to be set to 1 in the transaction too. If we call the current candidate itemset to test c, and the current transaction from the database t, one way to implement this proposition with FPGA logic is

$$c \subset t \iff t\& \, c = c. \tag{2.1}$$

This proposition can be simplified, since all bits set to 1 are set to 0 in $\bar{t}$. Thus for all items in c, they have to be absent from $\bar{t}$ for c to be present in t:

$$c \subset t \iff \bar{t} \;\&\; c = 0. \tag{2.2}$$

Since this feature only uses simple logic gates and a counter increment, its impact on the area cost is very low.

2.4.4 Correspondence List Address Cache

Many dataset reduction techniques have been developed for pattern mining in order to exclude redundant or unnecessary operations. Several of them can be seen as pre-processing techniques to optimize the ensuing pattern mining, such as removing non-frequent items from the database, or grouping identical transactions together. Some others aim to optimize the mining of patterns using knowledge discovered during the pattern mining.

Such works include the method of *Database projection*, first introduced by FP-Growth [14]. Database projection is a method that divides a frequent itemset mining problem into multiple smaller itemset mining problems. Using an alphabet $\{A, B, C, D\}$ for example, it could result in four itemset mining problems, one in which all itemsets start with A, a second where all itemsets start with B, a third with C and a last with D. At this point, it is possible to use *projected Databases*, one corresponding to each itemset mining problem. In the first problem, all itemsets start with A, so no transaction in the database that do not contain A can possibly give useful information for the itemset mining problem. It is then possible to use a reduced database, called projected database, containing only itemsets with A, instead of the full database. A projected database is generated for each itemset mining problem, potentially reducing by orders of magnitude the number of transaction read for each itemset. The contribution we provide in this thesis proposes a similar technique for depth-first pattern mining algorithms executed on FPGA.

In the special case of a depth-first algorithm like ECLAT, if the set of transactions containing a frequent itemset is registered in the memory of the FPGA, it is guaranteed

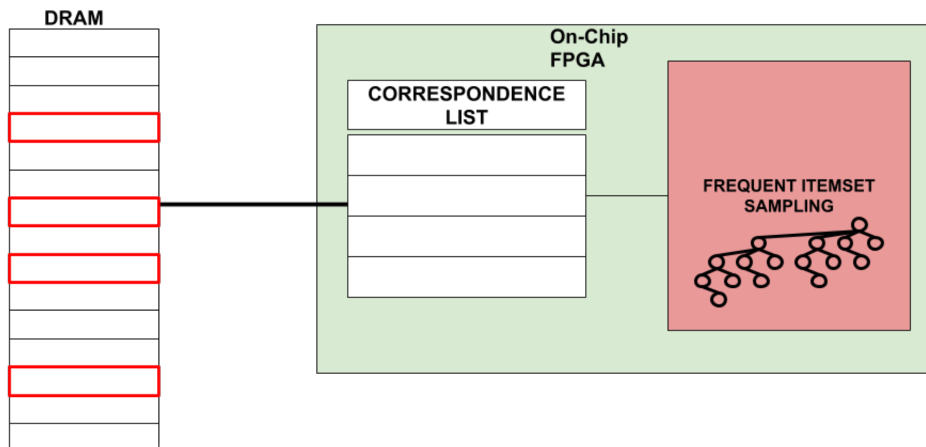

Figure 2.10: Example of dataset reduction using correspondence list

the next candidate itemset is a child of the itemset (when it is not a leaf of the tree), and its own set of transactions is contained in the registered set of transactions. By allocating in memory the addresses of transaction containing an itemset, the accelerator is able to scan a reduced portion of the database, as illustrated in Figure 2.10. If the set of addresses cannot fit in the allocated memory, the accelerator scans the whole database.

This technique can only work if the dataset uses a frequency threshold sufficiently low to fit in the FPGA memory. If the accelerator is tasked to find itemsets present in at least 90% of a large database, the correspondence list would not fit in the FPGA. The pattern explosion implies that a regular itemset miner could spend the majority of its execution scanning for itemsets with very low support (sometimes less than thousands or hundred of transactions depending on the database). Our proposition is to speedup the computation of these itemsets with low support, but with support still superior to the frequency threshold, when the density of the dataset favors it.

If we allow to register only the sets of transactions of several frequent itemsets, it is possible to store the histories of frequent itemsets at different depths in the tree. This reduces the number of transactions accessed in memory. In order to only keep track of sets of transactions of itemsets that can be reused by future candidate itemsets, the addresses of transactions of an itemset is evicted from memory depending on its depth in the tree.

The speedup brought by the correspondence list is greatly dependant on the database being mined and its density. For instance, for a frequent itemset mining algorithm looking for itemsets present in at least 10 000 transactions, a correspondence list would need more than 10 000 transactions worth of memory space allocated per itemset stored in the list. Any lower amount of memory allocated would not allow to store the complete support of frequent itemsets, rendering the dataset reduction impossible with the proposed method. However, the exact same database processed by a frequent itemset mining algorithm looking for itemsets present in at least 100 transactions could use a correspondence list with only a percent of that required memory.

As the database density and, consequently, the support of frequent itemsets are usually known at run time, the size of the correspondence list is already fixed when these information are available. If the accelerator detects the correspondence list cannot be used, due to the frequency support being too high for the correspondence list, it will not use the dataset reduction at all. The obvious drawback of this solution is that memory has to be allocated before the user knows the real requirements, leading to some guesswork at compile time on what amount of memory could be useful for most use cases.

2.5 Results

Figure 2.11 represents the requirements for the FPGA accelerator we designed using both techniques, with intermediate results stored locally, and with our proposed memory optimisation. The costs of Look-Up Tables and 18Ko BRAM Banks are shown on the two axis for different values of the maximum size of alphabet supported by the accelerator.

The memory optimisation allows to reduce the use of BRAM Banks from 1208 to 978 for alphabets of size 2048, which is a 19% gain. The LUT allocation goes from 17835 to 21929, which is a 18% increase.

This trade-off can be really interesting, as memory is a critical component for this type of application, while there are more than enough resources available to implement

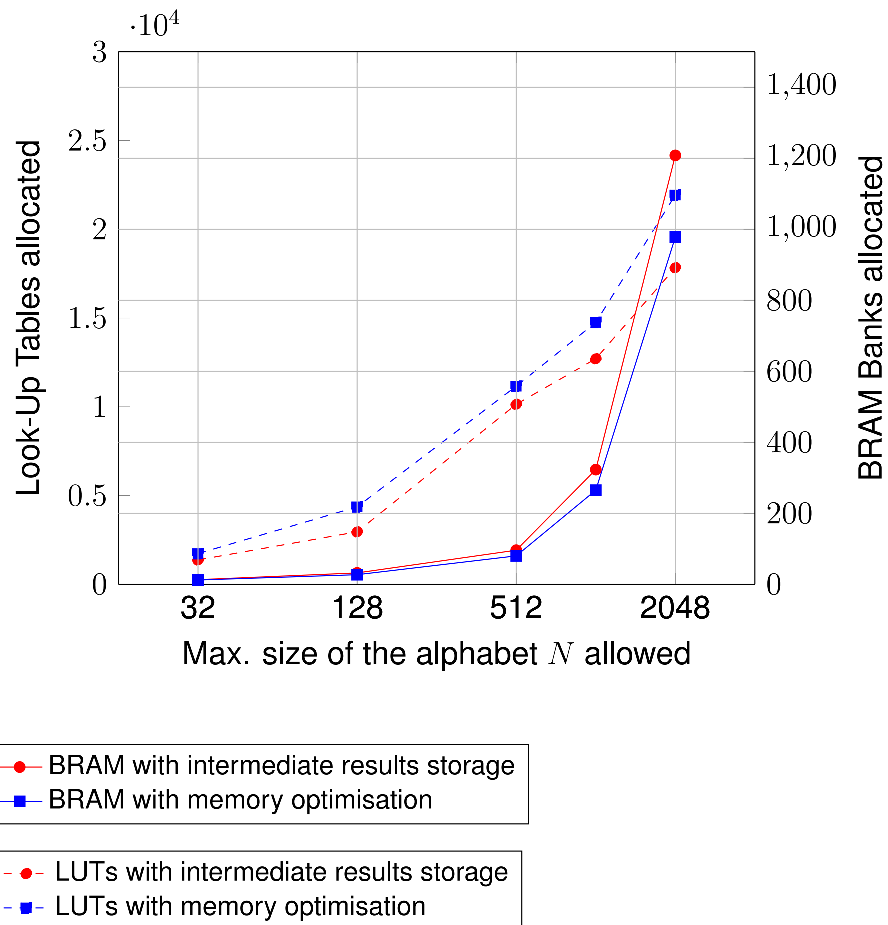

Figure 2.11: Impact of alphabet size on allocated resources

logic operators with Flip-Flops and Look-Up Tables. Even with an alphabet of size 2048, the accelerator uses around 10% of a zc706's FPGA LUTs and 3% of the Flip-Flops. The Flip-Flop usage is not shown on Figure 2.11, as it is one of the least used resources of the FPGA, and the relative increase is even lower than the one of the LUTs.

Out of the 1090 18Ko BRAM Memory Banks present on a zc706 board, the storage of intermediate results would require 1208 Banks, around 10% more than what is available. Our optimisation allows the design to go back under this limit by using 978 BRAM banks, allowing for the design to be implemented on a board as small as a zc706.

CHAPTER 3

ITEMSET SAMPLING ACCELERATION ON FPGA

In the previous chapter we presented our contribution for a frequent itemset mining accelerator on FPGA, using the ECLAT algorithm. While this type of itemset mining is quite common in the literature, it is nonetheless a prerequisite for the rest of the contributions presented in this document, all focused around the acceleration of frequent itemset sampling on FPGA. This chapter starts by introducing a method of output space sampling adapted to frequent itemset, used by works such as [2]. The chapter then submits mathematical tools and representations that we used to develop our own frequent itemset sampler accelerator. At last, the chapter contains an algorithmic description, a description of a hardware architecture, and quantitative evaluations. While there exist hardware accelerator architectures for frequent itemset mining, and software solutions for frequent itemset sampling in the literature, this is, to the best of our knowledge, the first architecture for FPGA that support frequent itemset sampling.

3.1 Sampling Itemsets with a System of Binary Constraints

As introduced in the first chapter, the main idea of Flexics is to use random XOR constraints to cut the itemset space in regions of equivalent size and distribution, called *cells*. The original Flexics algorithm is a two-stage process, where first the *Weightgen Oracle* is called, followed by the actual itemset mining step. The Weightgen Oracle, described in [11], estimates the *weight* of the solution (related to the number of results found) of a SAT problem. Weightgen can also be used to generate random arbitrary constraints on the variables of the SAT problem that will reduce the number of results, until the estimated weight is lower than or equal to a user specified threshold. In

the case of Flexics, the SAT problem given to Weightgen is finding frequent itemsets in the database. The random constraints generated to sample the results are XOR constraints using the alphabet $\mathcal{A}$ of size N from the database. The generated XOR constraints have the form

$$\bigoplus_{i \in [\![1,N]\!]} b_i.X_i = b_0, \tag{3.1}$$

where b_i are randomly generated independent variables in $[\![0,1]\!]$ with a uniform distribution. For any $i \in [\![1,N]\!]$, a b_i dictates if the item X_i will be present in the XOR constraint (1 if present, 0 otherwise). The b_0 coefficient is called the *parity bit*, because a XOR with multiple inputs returns the parity of the sum of its boolean inputs. These XOR constraints act as a system of Boolean equations, where a given pattern can either satisfy all the equations for the system, or not satisfy at least one of them.

For example, with a database using an alphabet $\mathcal{A}' = \{a, b, c, d, e, f\}$, and given two randomly generated XOR constraints $a \oplus c \oplus e = 1$ and $b \oplus e \oplus f = 0$, we can check if two frequent candidate itemsets $abde$ and $abdf$ are to be returned in the samples. For $abde$, the first XOR constraint gives $(1 \oplus 0) \oplus 1 = 1 \iff (1) \oplus 1 = 1 \iff 0 = 1$, which is false. Thus, $abde$ does not satisfy all XOR constraints and is not part of the subspace to be returned. For $abdf$, the first XOR constraint gives $(1 \oplus 0) \oplus 0 = 1 \iff (1) \oplus 0 = 1$, which is true, and the second one gives $(1 \oplus 0) \oplus 1 = 0 \iff (1) \oplus 1 = 0 \iff 0 = 0$, which is also true. Thus, the frequent itemset $abdf$ satisfies all XOR constraints and will be returned as a sample.

The two-stage approach (Weightgen then mining) is not relevant on an FPGA. First, implementing Weightgen would require a lot of estates, that would be unused most of the time. Second, in a streaming context one would like to be able to quickly react to changes in the pattern distribution, which is difficult to do when pre-computing the number of constraints. **Our first contribution is thus to propose a single-step approach for pattern sampling**, that does not use Weightgen. Instead, the mining starts immediately, and during the mining, our approach dynamically generates new XOR constraints if the potential output size goes above the user-given threshold.

This way, the number of interesting patterns does not need to be estimated in advance, but the list of interesting patterns discovered is maintained when new con-

straints are generated by dynamically pruning patterns than do not satisfy the newly added constraints. The principles of our dynamic constraint generation is as follows: Whenever the current number of results exceeds the user threshold, new XOR constraints are generated randomly and unsatisfiable results are pruned though maintenance. Our algorithm ensures that no constraint in the system of equations can be derived from a linear combination of the other constraints. If a newly generated XOR constraint causes the system of equations to be unsolvable, it is rejected. If it is equal to a linear combination of already generated constraints, the new constraint is also rejected. After a constraint is rejected, new constraints are randomly generated, until a generated random XOR constraint respects the criteria. In practice, however, there is a probability of $\frac{1}{2^{N-K}}$ for a new random XOR constraint to be rejected in a system already holding K XOR constraints. Thus, this process of rejecting constraints does not impact significantly average performance.

3.1.1 Rewriting the Constraints with Gauss-Jordan Elimination

The problem we try to solve is the following: scan the depth-first tree with the ECLAT algorithm, but skip the itemsets that do not satisfy the generated XOR constraints. However, when scanning the sparse samples in the tree, there is no simple relation like the prefix/suffix separation used in [1] between a sample and the next.

It is harder to compute what is the next sample to test given any position in the tree, and subsequently, to know if there exist satisfying itemsets down its branch or if the branch no longer holds itemsets to mine. We developed a solution to reduce the problem of mining satisfying itemsets randomly distributed in the tree to a simpler yet equivalent problem that uses ECLAT on a reduced tree that represents *at least* all frequent satisfying itemsets.

At any given time with K generated XOR constraints, for an alphabet of size N, the corresponding system of equations would be in the form of

$$\forall k \in [\![1, K]\!],\ \left(\bigoplus_{i \in [\![1,N]\!]} b_{i,k}.X_i \right) = b_{0,k}, \tag{3.2}$$

where $b_{i,k} \in [\![0, 1]\!]$, $\forall k \in [\![1, K]\!]$ and $\forall i \in [\![1, N]\!]$. With this notation, the $b_{i,k}$ can be considered as matrix coefficients for an array of size $K \times (N + 1)$. The parameter K is dynamic and data dependant (it increases during the execution as new samples are found) and is only theoretically bounded by N.

Using linear combinations, the matrix corresponding to the system of equations can be transformed using Gaussian elimination, to generate a row echelon matrix. The solutions of the original system of equations and after the Gaussian elimination are strictly the same. Flexics showcases the use of Gaussian elimination on the XOR constraints to enable recursive calls in ECLAT. In our FPGA implementation however, such recursive calls are not a viable strategy. For this reason, we introduce a new technique to implement ECLAT with sampling using XOR constraints, by relying on the Gauss-Jordan elimination.

The Gauss-Jordan elimination is a variant of Gaussian elimination which results in a unique normalized row echelon matrix. This unique matrix has two types of rows: one type can contain any value, whereas the other type of row contains exactly one 1 that delimits the echelon and 0 everywhere else. After a Gauss-Jordan elimination, our algorithm rewrites the system of equations in the form of

$$\forall k \in [\![1, K]\!],\ X_{c_k} = \left(\bigoplus_{i \in [\![1, N-K]\!], f_i < c_k} b'_{f_i,k}.X_{f_i} \right) \oplus b'_{0,k} \tag{3.3}$$

where $\{c_i\}_{i \in [\![1,K]\!]}$ denote the indexes of the rows corresponding to the echelons. There are K indexes c_i as there are K rows in the Gauss-Jordan matrix. We refer to the items X_{c_i} as *constrained items*. All the other indexes corresponding to rows with any values allowed are noted as $\{f_i\}_{i \in [\![1,N-K]\!]}$. There are $N - K$ indexes f_i as there are N indexes in total. The items with indexes X_{f_i} are called *free items*.

In our case, the normalized form of Equation 3.3 will result with all $b'_{c_k,i}$ coefficients from equations from the right side with linear combinations to be ones if $k = i$, and zeroes otherwise (see Table 3.3 for an example). This way, we can ensure that $\forall i, k \in [\![1, N]\!], [\![1, K]\!]$, $b_{i,k}$ coefficients on the right side of the equations, $\exists k' \in [\![1, K]\!], i = a_{k'} \implies b'_{i,k} = 0$.

The introduction of the distinction between **free items** and **constrained items** will be the basis of our contribution in frequent itemset sampling on FPGA. We defined as

free alphabet the part of the alphabet that contains the free items, and *constrained alphabet* the part of the alphabet that contains the constrained items. The intersection of the free and constrained alphabet is empty, and their union is the entire alphabet.

Example Given an alphabet $\mathcal{A} = \{A, B, C, D, E\}$, three XOR constraints are randomly generated:

- $C \oplus E = 1$
- $A \oplus B \oplus C \oplus E = 1$
- $A \oplus B \oplus C \oplus D \oplus E = 0$

These three constraints are then displayed in a matrix format as in Eq. 3.4. Table 3.1 shows the implementation of a matrix containing the random XOR constraints suitable for an FPGA implementation.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \tag{3.4}$$

Table 3.1: Example of a matrix containing the random XOR constraints

0	0	1	0	1	1
1	1	1	0	1	1
1	1	1	1	1	0

The constraints after linear combinations that result in a *row-echelon matrix* obtained with Gaussian elimination are illustrated in Eq. 3.5.

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \tag{3.5}$$

Its correponding implementation is depicted in Table 3.2. This type of matrix is characterized by having the first nonzero numbers from the left (called pivot) of the rows that are strictly to the right of the pivot of the above row.

Table 3.2: Example of a row echelon matrix after Gaussian elimination

1	1	1	0	1	1
0	0	1	1	1	0
0	0	0	1	0	1

Gaussian elimination can result in different row-echelon matrices depending on the linear combinations used. The result of Gauss-Jordan elimination that normalizes the row-echelon matrix is illustrated in Eq. 3.6.

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \tag{3.6}$$

The normalized matrix has all its pivots with a value of 1 (in our case, the matrices can only hold 0 and 1 values, so it was already the case), and the values above the pivots are all zeroes.

Table 3.3: Example of a row echelon matrix after Gauss-Jordan elimination

1	1	0	0	1	0
0	0	1	0	1	1
0	0	0	1	0	1

The set of constrained indexes (corresponding to the pivots) is $\{c_i\}_{i\in[\![1,K]\!]} = \{1,3,4\}$, and the set of free indexes is $\{f_i\}_{i\in[\![1,N-K]\!]} = \{2,5\}$. More precisely, $c_1 = 1$, $c_2 = 3$, $c_3 = 4$, $f_1 = 2$, $f_2 = 5$.

NOTE To explain the linear combinations of Gauss-Jordan, the previous examples take the form of row-echelon matrices. These result in the XOR constraints $A = B$, $C = E \oplus 1$ and $D = 1$. In practice, we will prefer to write the matrices the other way, as

in Eq. 3.7, to match the form of Eq. 3.3.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \tag{3.7}$$

This gives a different, yet equivalent, system of equations: $E = C \oplus 1$, $D = 1$ and $B = A$. With these equations, it will be easier to make use of the free items and constrained items in later sections.

3.1.2 Reducing the Sampling Problem to a Regular Itemset Mining Task

For any system of equations resulting from XOR constraints, our approach is to split the alphabet into two parts:

- the set of constrained items X_{c_k} that appear in only one XOR constraint after Gauss Jordan elimination, and
- the set containing the rest of the alphabet, denoted free alphabet, whose items X_{f_i} can appear in any XOR constraint.

In this way, there are two complementary alphabets, that can be non-contiguous, splitting items from the initial alphabet in two categories. Then, the right hand of Equation 3.3 is computed using the $b'_{f_i,k}$ in the Gauss-Jordan Matrix and the X_{f_i} are equal to 1 if the f_i are present in the free itemset and to 0 otherwise. This formula dictates if the items X_{c_k} are present in the constrained itemset or not.

For two different itemsets constructed from the original alphabet $\mathcal{A}$ that satisfy the XOR constraints, their free itemsets are necessarily different. Reciprocally, there exists a single itemset constructed from $\mathcal{A}$ that satisfies the XOR constraints for each free itemset. Thus, there is a bijection between the pattern space of the database alphabet and the subspace of patterns satisfying the XOR constraints: a scan of all free itemsets can be used to scan all satisfying itemsets. Furthermore, a sample constructed from

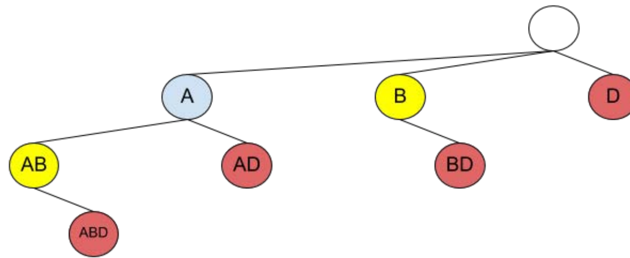

Figure 3.1: Tree used by ECLAT on the free alphabet $\{A, B, D\}$

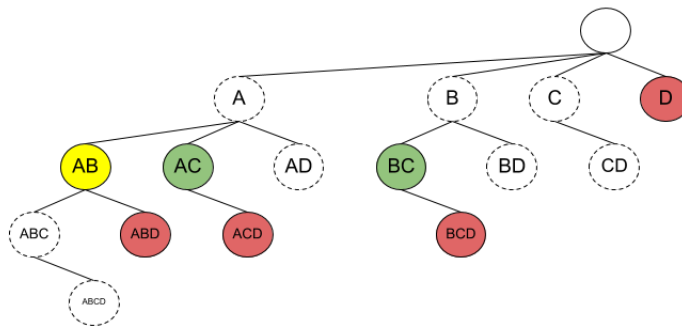

Figure 3.2: Sampled tree to be mined with the SAT constraint $A \oplus B \oplus C = 0$

the full alphabet $\mathcal{A}$ will always be a super-set of its reduced free itemset. Thus, the Apriori principle dictates that if a sample itemset is frequent, its corresponding free itemset is also frequent. Thus: a scan of all **frequent** free itemsets can be used to scan all **frequent** satisfying itemsets. Frequent free itemsets can lead to infrequent satisfying itemsets, but the algorithm is guaranteed to find all samples.

Figure 3.1 shows an example tree that use the free alphabet $\{A, B, D\}$, while the entire alphabet is $\{A, B, C, D\}$. Figure 3.2 shows the entire tree using the alphabet $\{A, B, C, D\}$, but keeps colored only the nodes that satisfy the constraint $A \oplus B \oplus C = 0$. As shown in Figure 3.3, there exist a bijective relation between the nodes of the *reduced tree* that use the free alphabet, and the nodes that satisfy the XOR constraint in the regular tree.

Finding all frequent free itemsets can be done using ECLAT on the reduced free alphabet, and for each frequent free itemset, the algorithm will test its satisfying coun-

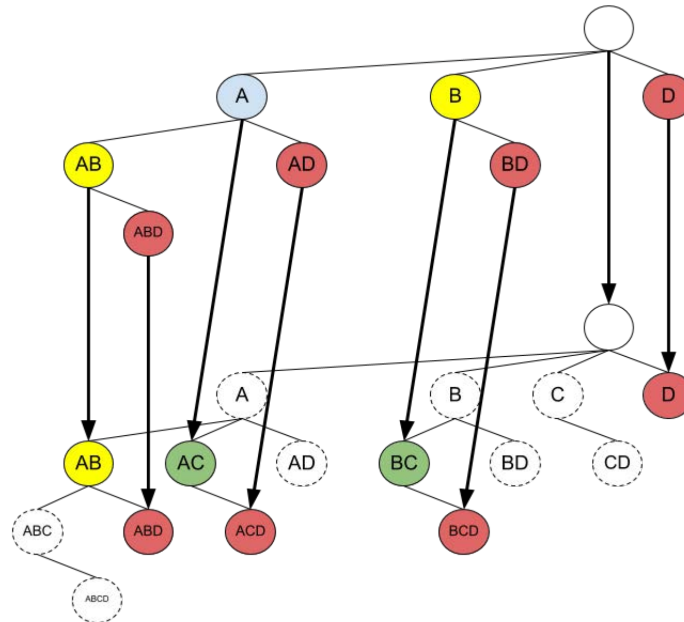

Figure 3.3: Use of the bijection $C = A \oplus B \oplus 0$ between the tree with free alphabet and the sampled tree

terpart. By looking at Figure 3.3, we can spot an issue with the application of the Apriori principle if this sampling method is used as is.

Example Let's say A is an infrequent itemset. Without sampling, A is tested, found infrequent, so the whole branch is skipped. If we only test itemsets that satisfy $A \oplus B \oplus C = 0$, we do not test A, but AC. AC is found infrequent, but the next itemset in the depth-first sampled tree is AB. Knowing AC is infrequent does not give knowledge about AB, so it has to be tested, and is ultimately found infrequent (because A is infrequent).

Testing only satisfiable itemsets allow to skip over itemsets that could hold information that preemptively prune large portions of the tree, and can result in more itemset being tested. To resolve this problem, we propose to mine additional itemsets, in parallel of the itemsets already mined for sampling. When mining itemsets, the sampler both mine a sample itemset that satisfy the XOR constraints, and the corresponding free itemset. This decision could impact the performance of a CPU implementation, but the solution we propose for FPGA architecture results in a marginal resource allo-

cation, without increasing the execution time.

Example Going back with the previous example, this time mining two itemsets at once, AC is mined at the same time as A. AC is found infrequent, and it will not be returned as a frequent sample. A is found infrequent, and this time, it is possible to deduce that all itemsets from its branch in the tree using the free alphabet are infrequent. For all itemsets in this tree, the corresponding itemsets in the sampled tree are also infrequent, because of the Apriori principle.

For each itemset in the *reduced* depth-first tree, the accelerator tests the frequency of two itemsets. The free itemset has to be tested to allow pruning with the Apriori principle, and the sample itemsets have to be tested since they will be returned in the final results if it is frequent.

One last consideration we have to take is the fact that we allow for dynamic sampling. We specifically do not estimate in advance the number of XOR constraints needed for the number of samples to be lower than a user threshold t_h. Instead, we generate a new XOR constraint whenever the current number of sample exceeds t_h, pruning around half of the results on average[1]. The maintenance of the results is rather trivial, the samples kept so far are organized in a linked list, we scan through the list when a constraint is generated and remove results that do not satisfy it. However, any new constraint will modify the row-echelon matrix, altering the free and constrained alphabets. The Gauss-Jordan matrix can be maintained in $O(N)$, N being the number of constraints generated, with linear combinations. When a new constraint is added, one new letter from the alphabet goes to the constrained alphabet, meaning it no longer is in the free alphabet. We have to take this into consideration as the control sequence we use to mine the trees is based on the free alphabet.

Example We start from the tree with the free alphabet $\{A, B, D\}$ shown Figure 3.4. A new constraint, let's say $A \oplus B = 1$, is generated. B is now in the constrained alphabet and no longer in the free alphabet, giving the tree shown Figure 3.5. If the sampler was mining an itemset such as A in the free alphabet, its node is in the new tree and its is rather straightforward for the algorithm to continue. If the sample was mining an

[1]Each itemset has a probability $\frac{1}{2}$ to satisfy any XOR constraint generated

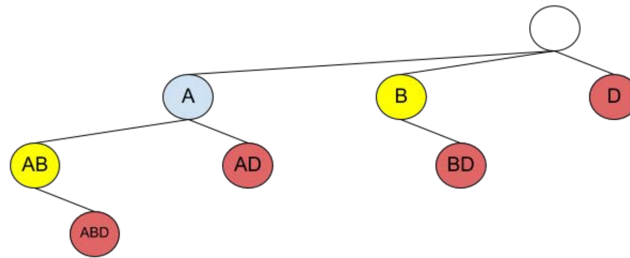

Figure 3.4: Tree used by ECLAT on the free alphabet $\{A, B, D\}$

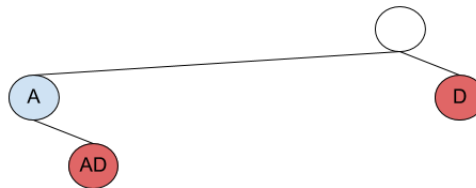

Figure 3.5: Tree used by ECLAT on the free alphabet $\{A, D\}$

itemset containing B, the letter removed from the free alphabet, we must determine where in the new tree it should jump to proceed, without skipping or mining redundant itemsets. In this case, the node containing the next itemset to mine is in fact the first sibling of the first node containing B in the branch.

In the general case, if the sampler was just mining an itemset containing the letter to be removed from the free alphabet, that letter is either the last letter of the alphabet or not.

If the letter is not the last in the free alphabet, the next itemset to mine is the first sibling of the first itemset that holds the letter in the branch. In practice, all items after the letter removed in the alphabet are deleted from the itemset, providing the parent of the itemset we need to mine. The next itemset to mine is that itemset, using the first free letter after the letter removed in an I-step. Such an item exists for an I-step, since the letter removed is not the last in the free alphabet.

If the letter removed is the last letter in the free alphabet, the next itemset is the same next itemset we would choose if we were mining its parent without the letter.

In summary: we now have a bijection between the desired sample space and a regular itemset space using a reduced alphabet. An itemset from the sample space is frequent only if the corresponding itemset from the reduced space is frequent. However the inverse is not true, and finding frequent reduced itemsets can lead to scanning non-frequent samples. To avoid unnecessary computations, the itemsets to scan for the control sequence shall be as close from the samples as possible, to reduce the number of tested samples that can be deduced as non-frequent. A new bijection can be created for this purpose: use the biggest common subset of all samples derived from the subsequent branches of current itemset. Instead of scanning the reduced itemset, the optimized algorithm can scan the biggest common itemset, reducing the proportion of scanned itemset for non-frequent samples while keeping the Apriori principle true. The biggest common itemset can be computed from the reduced itemset using the XOR constraints and the suffix of the reduced itemset. In short, the new bijection is very similar, the reduced itemsets are still used. However, the reduced itemset are not scanned directly, but built upon to create biggest common itemsets.

3.1.3 Algorithmic implementation of ECLAT with dynamic XOR constraints

Algorithm 1 eclat_pseudo_code

```
while not finished do
    compute_next_itemset(free itemset)
    sample = compute_corresponding_sample(free_itemset)
    count_in_DB(free_itemset, sample)
    if is_frequent(sample) then
        add_to_results(sample, results)
    end if
    while number_of_results > threshold do
        create_constraint(XOR_matrix)
        update_free_alphabet_and_free_itemset(XOR_mat, free_itemset)
        prune_results_with_new_constraint(XOR_mat, results)
    end while
end while
```

The Algorithm 1 can be applied to software architectures, but was designed with hardware in mind, specifically FPGAs: it uses constant time to generate a candidate

Algorithm 2 compute_next_itemset

```
if is_leaf(itemset) then
  suffix = leading_bit(prefix)+1
  prefix = prefix - leading_bit(prefix)
else
  if is_frequent(itemset) then
    prefix = prefix & suffix
    suffix = suffix +1
  else
    prefix = prefix
    suffix = suffix +1
  end if
end if
```

Algorithm 3 count_in_DB

```
count = 0
for all transaction ∈ Database do
  if itemset ∈ transaction then
    count = count +1
  end if
end for
```

sample, two itemsets are tested at the same time, many XOR operations are meant to be done in parallel. The *"compute_next_itemset"* function shown in Algorithm 2 sums up the explanation of the Figures 2.6, 2.7 and 2.8 in the Chapter 2. When an itemset is processed, it is either a leaf, a frequent itemset or an infrequent itemset, and the next itemset is generated accordingly. The "+1" operation is used to simplify the pseudo code, and just means the next letter in the alphabet is used. for example, "A"+1 is "B". The function *"count_in_DB"* shown in Algorithm 3 is rather simple, as it reads all transactions in the database to check if the current itemset to mine is in it, and its support is computed accordingly. The *"update_free_alphabet_and_free_itemset"* function is rather implementation specific. In our accelerator, we use a linked list to store the letters of the free alphabet, as any of them might be removed randomly. This affects the "+1" operator in *"compute_next_itemset"* that has to make sure it only uses letters from the current free alphabet. Letters no longer in the free alphabet must be removed from the current itemset, if any exist.

The portion "while(number of results $>$ threshold)" of the algorithm cannot be reduced to a constant time of execution (only statistical bounds with high probabilities, but can theoretically be infinite). This portion of the algorithm is only called once every time the number of frequent samples is higher than the user bound, so this part of the algorithm is not called too often (cannot be invoked more times than the size of the alphabet) thus is not critical. The update and prune parts are rather fast ($O(1)$ and $O(user_bound)$), but it is the *create_constraint* function call that has probabilistic bounds. Gauss-Jordan elimination is bounded in execution time but might not be negligible.

The critical part of the algorithm is "count_in_DB", because of the large database and the I/O bound nature of the problem. The two major propositions for this issue are a dataset reduction technique and parallelizing the algorithm, and will be presented below.

3.2 Accelerator Architecture

This section presents the architecture of our accelerator on an actual FPGA board, and gives more details on how the design is implemented.

3.2.1 Overview

Figure 3.6 shows an overview of the accelerator architecture implemented in the programmable logic fabric of a Zynq FPGA from Xilinx. The *zc702* evaluation board used for the experiments holds a *System on Chip*, containing two Cortex A-9 CPUs and a XC7Z020-CLG484-1 FPGA, connected to an off-chip 1GB DDR3 memory. Our design uses communications between the FPGA and the CPUs only when the hardware application starts and stops, in order to interface with the user. All accesses from the accelerator in the FPGA to the main memory are achieved using the *High Performance AXI* port, without cache coherency protocols. The correspondence list is an optional feature that can also be implemented in the sampler, and was showcased in the previous chapter. It is only absent from Figure 3.6 for readability purposes. Most of the architecture details provided in the previous chapter are also valid for the sampler architecture

3.2.2 Linked list for results

In order to store frequent itemsets and, to be able to prune them when the number of results exceeds the threshold, a linked list is allocated with a bounded size. This way, insertions of new results at the head of the list are achieved in constant time.

When a new XOR constraint is created, insatisfiable results have to be pruned from the linked list. For each frequent itemset in the list, if it does not satisfy the XOR constraints, the pruning mechanism will remove it and rejoin the list in constant time.

3.2.3 Parameters' Influence on the Accelerator

In order to analyze the theoretical performance of the accelerator, we introduce two parameters: DB_SIZE, the number of transactions in the database, and NB_BLOCK, the number of memory accesses to get a transaction. During the support counting, the accelerator processes chunks of $\frac{N}{NB_BLOCK}$ bits wide data in a pipelined fashion.

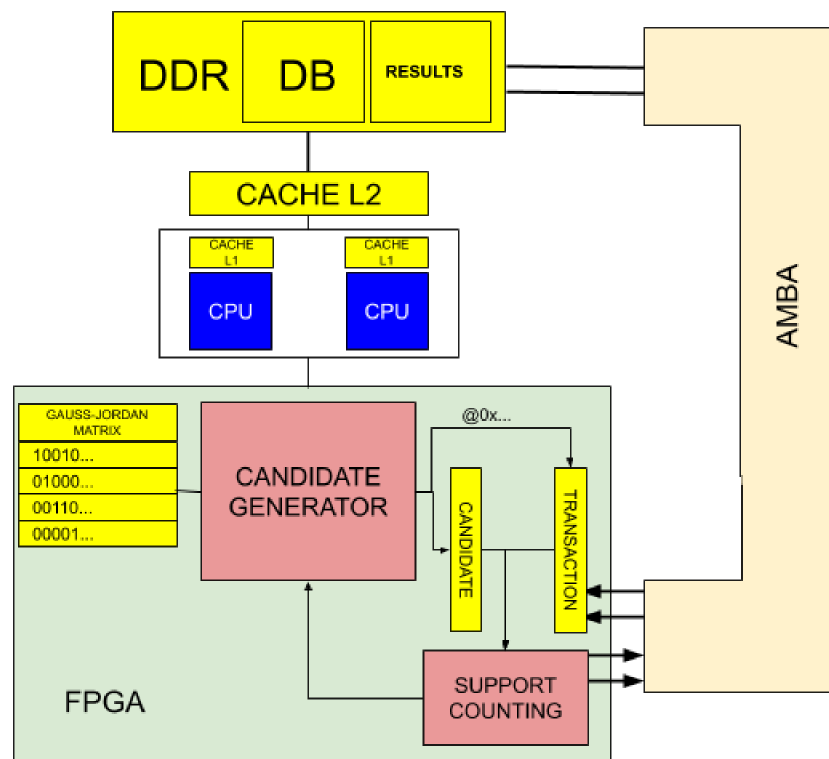

Figure 3.6: Overview of the itemset sampling accelerator architecture

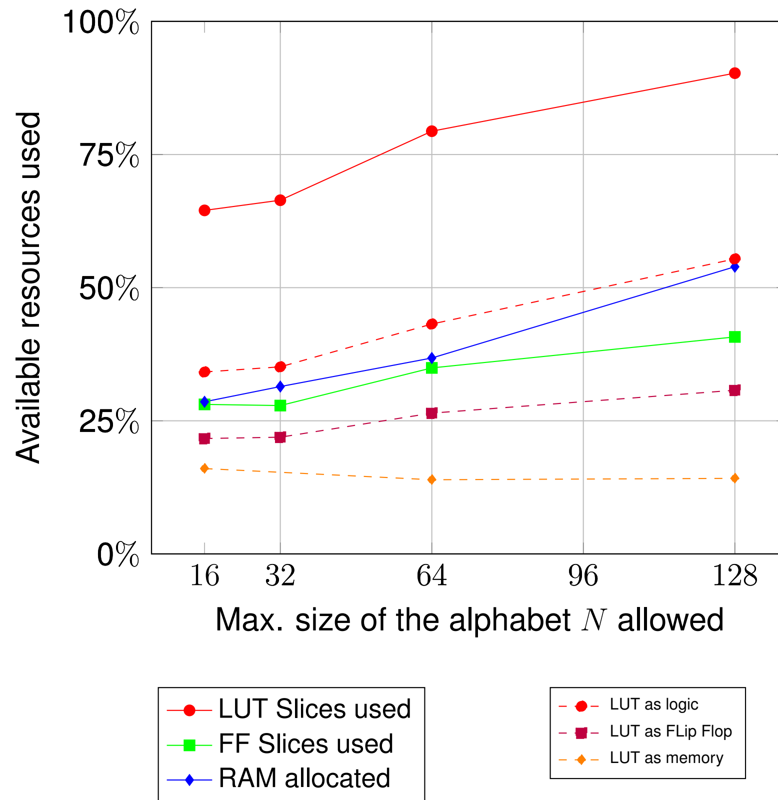

Figure 3.7: Impact of alphabet size on allocated resources

If the memory is able to feed the accelerator without bandwidth limitation, the pipeline can process a chunk every four cycles at 100MHz. The number of cycles needed to compute the support of an itemset is $4 \times DB_SIZE \times NB_BLOCK$.

If we consider an example where the memory has to deliver chunks of size $^{N}/_{NB_BLOCK} = 128$ bits, the internal logic of the FPGA can process $128/4 \times 100Mb/s = 400MB/s$. Thus, the accelerator will be stalled by the memory if it is unable to reach this throughput. This is often the case in I/O bound applications such as data mining. A simple way to tackle this issue is to encode the transactions inside the memory, and decode it back to bit vectors once in the FPGA. The naive approach to store potentially long bit vectors can be a huge waste of space and bandwidth, especially with sparse datasets containing many zeroes.

Figure 3.7 shows the amount of resources allocated on the XC7Z020 SoC, a FPGA used by the low-cost Zybo Z7 board, for different values of N. An accelerator with a fixed value N can process any database with an alphabet of size less or equal to N. At the moment only rather small alphabets can be processed because of the high usage

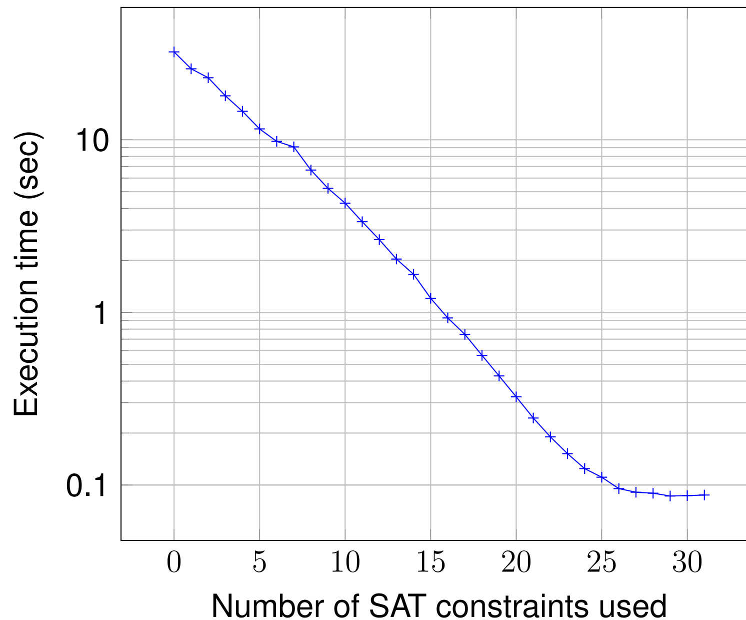

Figure 3.8: Impact of number of XOR constraints on execution time

of Look-Up Tables (LUT), as nearly all of the slices are allocated for an alphabet of size 128. For the same alphabet size, only half of the available RAM and register slices are used. The parameter N is thus linked to two limiting factors: the allocated logic and the memory bandwidth. It is however noteworthy that the FPGA used in our experiments is rather small and our approach will scale efficiently with larger FPGA devices.

3.3 Experiments

The dataset used for the performance analysis of our accelerator is available publicly from [15]. This dataset uses an alphabet of size 32. We impose $N/NB_BLOCK = 32$ bits. The FPGA thus processes $100MB/s$, which is close to the bandwidth between the memory and the FPGA. The dataset consists of around 13000 transactions, and the frequency threshold used is 1%. This fits very well the use case of a low-frequency support, and allows for our design to be tested with a wide range of correspondence list sizes.

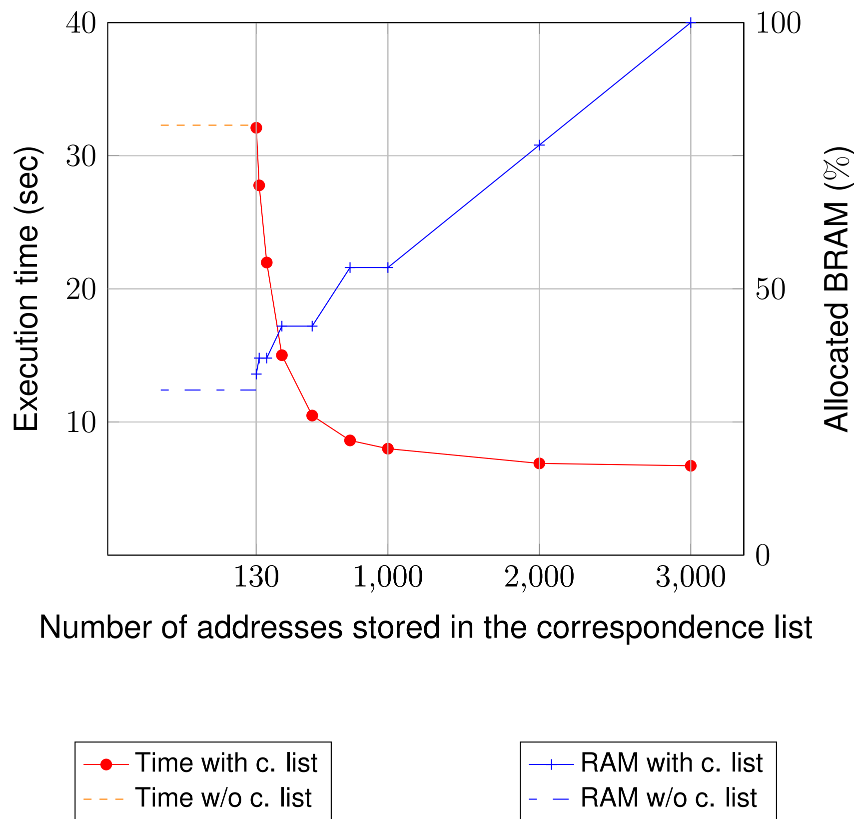

Figure 3.9: Execution time wrt. correspondence list size

3.3.1 Impact of Sampling on Execution Time

Figure 3.8 shows the effects of the sampling on the execution time. In this scenario, the accelerator can return any number of samples (no user bound), but the number of random XOR constraints is fixed at compilation. Each new constraint reduces the execution time by a factor close to 1.2, until it reaches the point where there are so few itemsets left to scan, the initialization becomes non-negligible.

3.3.2 Impact of Correspondence List Size

Figure 3.9 compares the execution time of the accelerator with and without a correspondence list, presented in the last chapter. The correspondence list brings a very noticeable speedup with rather low amounts of memory allocated. Allocating memory to store 300 addresses (roughly twice the support threshold in this case) results in a speedup of 2.1. The maximum speedup corresponding to 3000 addresses reaches 4.7.

Figure 3.9 also displays the amount of memory allocated in the FPGA for the tested values. The relation is not stricly linear because only fixed amounts of block RAM can be used. The correspondence list with a size of 3000 results in all the BRAM (140) being allocated.

Table 3.4: Time needed (in seconds) to compute 1000 samples

	EFlexics (CPU)	Our FPGA
german	25	11,7
heart	45	11,6
kr-vs-kp	9	12,7
primary	10	1,2
vote	19	5,8

3.3.3 Comparison Against Flexics

Table 3.4 shows results taken from [2], where EFlexics (the fastest implementation of Flexics) is tasked to return 1000 samples. These results were obtained on an Intel Xeon CPU running at 3.2GHz and with 32Gb of RAM on the CP4IM dataset [16]. We compare the execution of our accelerator running on the *zc702* board when tasked to return a thousand samples to the software execution time, without allocating a correspondence table. Our accelerator achieves a speedup between 2 and 10, except for the *kr-vs-kp* dataset. The accelerator performs best with small alphabet sizes, as these problems require a lot less memory bandwidth.

3.4 Extension to Parallel Execution

In this section, we present a detailed approach for extending the proposed algorithm to parallel execution on FPGA. Due to delays and implementation issues, we were unfortunately not able to have an FPGA implementation stable enough to propose experiments. The approach is thus only presented from a theoretical point of view, and no experiments have been realised at the time of writing this thesis.

The ECLAT-based itemset sampler described in this chapter scans itemsets in the enumeration tree one by one in a depth-first fashion. However, to determine if an

itemset is frequent or not during a scan, the algorithm only needs to count its number of occurrences in the database. Since it is a read-only task on the database, it is possible to count the occurrences of as many itemsets in parallel as the memory bandwidth allows for it.

In theory, all itemsets in a cell (or any subspace of the ensemble of all itemsets) can be tested, but the Apriori principle explicited in the Apriori algorithm [17] proposes a very simple optimization that drastically reduces the number of itemsets to compute. Our contribution proposes an implementation of the parallelisation of the ECLAT-based sampler, accounting for limited bandwidth between the FPGA and its external memory, while taking advantage of the Apriori principle as much as possible to reduce the number of unnecessary computations.

The general design we propose is as follows: the accelerator holds several small *processing units*, or PU, responsible for generating candidate itemsets, and all candidates are counted during a scan of the database on a shared module.

In this section we first present a basic static scheduling of ECLAT on 2^N processing units ($N \in \mathbb{N}$), then an improved dynamic solution with *work stealing*.

3.4.1 Static Scheduling

Any two itemsets that are not descendent of each other can be mined in parallel without inducing additional computation ("true" task parallelism), while the descendant of an itemset not mined yet can be speculatively mined, at the risk of unnecessary computations. Thus, when given a choice, it is best to process itemsets with no obvious relations like being a descendent of the other, and speculate when it is no longer possible.

Because the depth-first tree is built with a recursive geometry, a tree resulting from an alphabet of size N can be split into two sub-trees of equal shape and size. One of the two trees is the depth-first tree that would result from the alphabet minus its last item (size $N-1$). The second tree contains the same itemsets as the other tree for alphabet of size $N-1$, with the insertion of the last item of the original alphabet.

Figure 3.10 shows this phenomenon where it is possible to build a tree for an alphabet of size N using the tree for an alphabet of size $N-1$, and a copy of it where each itemset is expanded with the Nth item. This operation can be repeated up to N times

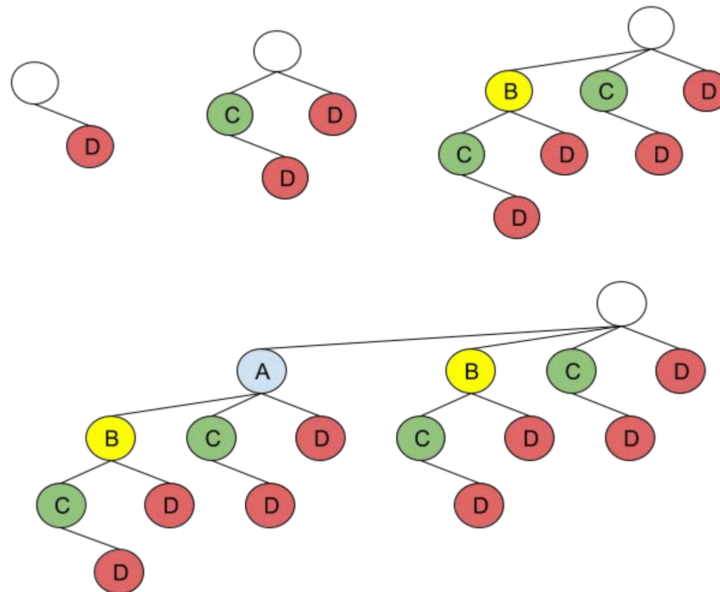

Figure 3.10: Depth-first tree for alphabet of size 1, 2, 3 and 4

The two sub-trees are not independent, the root of one is the parent of the root of the other tree, but when the roots of the tree are processed, any itemset of a tree can be processed in parallel of any itemset of any other tree.

Two components of the design have to be adapted for distributing ECLAT: the support counting, and the candidate generation. Thanks to the data parallelism, the new design loads transactions in a similar way to the sequential version, and the presence of multiple itemsets in a transaction are tested at the same time. This technique does not increase the memory bandwidth needed, and ensures that processing units (PU) start and finish the support counting in a synchronized manner. Transactions are loaded from memory in the same manner as they would in the iterative algorithm, but multiple support counting components are allocated on the FPGA, allowing to test the presence of multiple itemsets in a transaction at the same time. This increases the amount of logic allocated, but not the time needed to perform a database scan.

The candidate generation is harder to parallelize. Because the results are data-dependant, it is not possible to predict the load imbalances, and the PU should be affected dynamically to portions of the tree. Even if the tree can be divided in 2^k sub-trees of equal size (k< size of the alphabet), the probability of itemsets in the sub-trees to be frequent are different, and the execution time depends greatly on it.

The static approach therefore provides strong limitation. To overcome these limitations, we introduce a dynamic approach based on work stealing in the next section.

3.4.2 Work Stealing

Our dynamic strategy for parallel computation of ECLAT is based on work stealing. The whole tree is assigned to a PU, and each processing unit will split the tree already assigned to a PU into two, take half of it, and leave the other half to the original PU. The choice of the tree to split would be as follows: split larger sub-trees first, and split the sub-trees that are higher in the tree first. These choices are made to reduce the load imbalance as much as possible, by computing sub-trees of similar size, and because itemsets high in the tree are more likely to be frequent and develop branches.

A parallelisation of ECLAT on FPGA is proposed in [1], using a method that seems to be very close of our proposed work stealing approach. The article does not provide much details however, which is why we mention it without comparing it to our own method.

The challenge of this method is to ensure that an itemset from a partially scanned sub-tree will not be scanned multiple times after it is split in two.

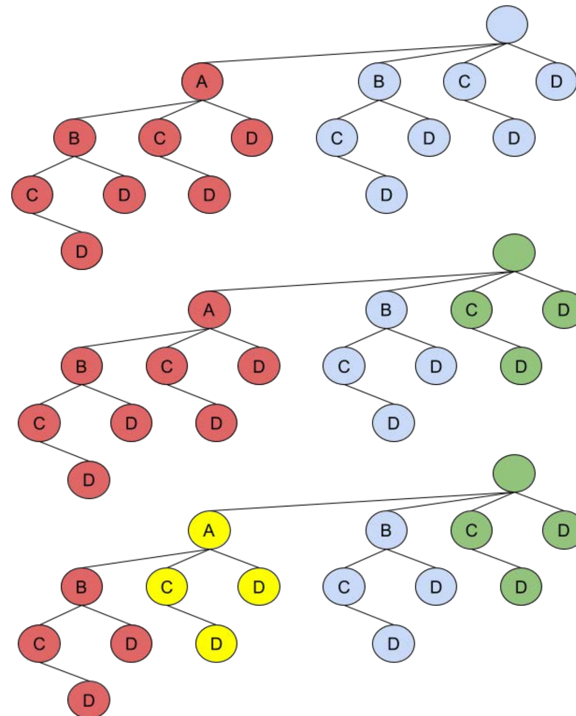

Figure 3.11: Tree computation distributed on 2, 3 and 4 PUs

At any given time during the execution, each PU is assigned a sub-tree with a root node, a sub-tree alphabet size, and a current node. Each PU mines its sub-tree of size $K > 0$ in the same manner as ECLAT mines an entire tree using an alphabet of size K. When a PU finishes its sub-tree, it can proceed to work stealing in order to share the workload of another PU still in execution.

Because each sub-tree of alphabet size $K > 1$ can be split into two sub-trees of alphabet size $K - 1$, it is possible to steal the work of any PU that is not working on a sub-tree of alphabet size 1. When a PU steals work from another PU, the sub-tree is split in two sub-trees of equal alphabet size, and each PU is assigned to a sub-tree. The PU that got part of its work stolen can continue its execution on the sub-tree from where it was. Some information, such as the alphabet size of the sub-tree and the condition that dictates when the PU reaches the end of the sub-tree, are updated.

Example Figure 3.12 and Figure 3.13 illustrate this process on a sub-tree of alphabet size 4. PU 1 is in the process of mining its sub-tree, currently mining ABD, when PU 2

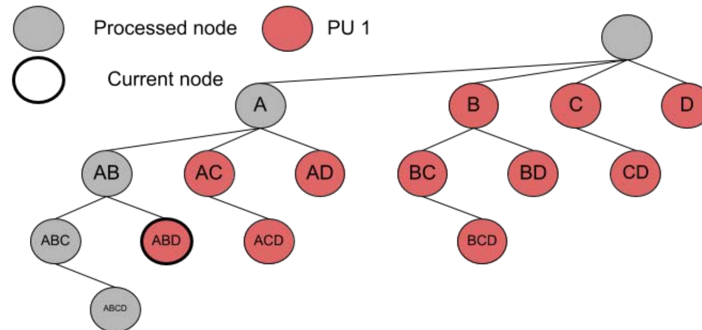

Figure 3.12: Sub-tree of alphabet size 4 mined on PU 1

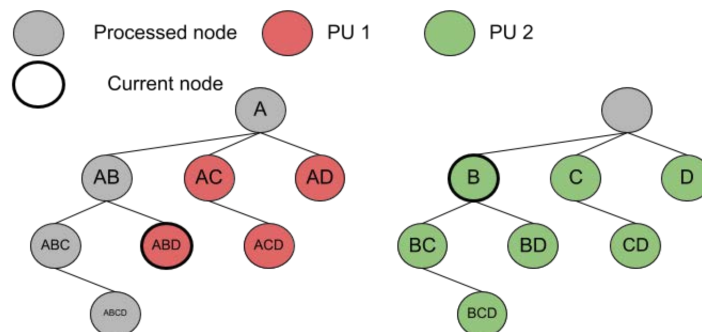

Figure 3.13: Sub-tree of alphabet size 4 split into two sub-trees of alphabet size 3 after work stealing

initiates work stealing. After work stealing, PU 1 is affected to the half of its original sub-tree that contains the work it was processing, and can resume its work as normal. PU 2 is affected to a sub-tree that no other PU has mined, except for the itemset at its root, processed by PU 1.

When a sub-tree of alphabet size $K > 1$ gets split into two sub-trees of alphabet size $K - 1$, it means both sub-trees have one "less degree" of freedom than the original. As the alphabet size of an entire tree corresponds to the number of letters of its alphabet, the alphabet size of a sub-tree dictates how many letters can be used in I-step on its root or other itemsets of the sub-tree. When looking at Figure 3.12, the original sub-tree has an empty itemset $\emptyset$ as its root, and an alphabet $\{A, B, C, D\}$. In Figure 3.13, both sub-trees are of alphabet size 3, one has A as its root, the other uses $\emptyset$. For both the sub-trees, the set of letters they can add or remove to itemsets is $\{B, C, D\}$, making it their *local* alphabet.

In the general case, a sub-tree of alphabet size $K > 0$ in a tree of alphabet size $N > K$ always uses the last K letters of the tree's alphabet as its own local alphabet. Part of the prefix of the itemsets in a sub-tree can include letters from the tree's alphabet that are not present in the local alphabet. For example, all itemsets start with A in PU 1's sub-tree in Figure 3.13. For this reason, we split the prefix into two parts, the local prefix, holding only letters from the local alphabet, and the root, constant in the whole sub-tree. In later examples, we consider the root of the sub-trees to be $\emptyset$ for readability purposes, but any itemset using letters from the tree's alphabet can be used as root as long as none of its letters are in the local alphabet (for example, $X, Y, Z,$ etc., in an alphabet $\{X, Y, Z, A, B, C, D\}$).

We can note that the root of a sub-tree being split into two sub-trees gets assigned to the PU that initiated the work stealing. In practice, we ensure that the root of a sub-tree is not mined by a PU if it was already mined previously with a flag during the work stealing that signals the PU should start at the itemset following the root.

For optimization purposes, we aim to reduce load imbalance, because the process of work stealing generates an overhead. Due to mutual exclusions (PU should not work on the same sub-tree), assigning the work to steal is not trivial to parallelize, so our FPGA implementation can only assign a PU to steal the work of a single other PU at a time. For these reasons, the work stealing routine should be called the least amount of time possible in order to amortize its cost with the parallel execution of the PUs.

A simple metric we propose to minimize the number of time the work stealing routine is invoked to determine which work to steal is the alphabet size of the sub-tree. Because the number of itemset to mine in a tree of alphabet size $K > 0$ can be up to 2^{K-1}, this method steals the work of the PU with the biggest estimated amount of work (number of itemsets to mine).

When there exist multiple PUs with the same sub-tree alphabet size, we can use another metric: the depth of the root of the sub-tree in the tree. The depth of a node in the tree is equal to the number of items its itemset holds. Both in theory and in practice, itemsets with a larger number of items have lower probabilities to be frequent, thus, are less likely to be mined by algorithms such as ECLAT (because of the Apriori principle

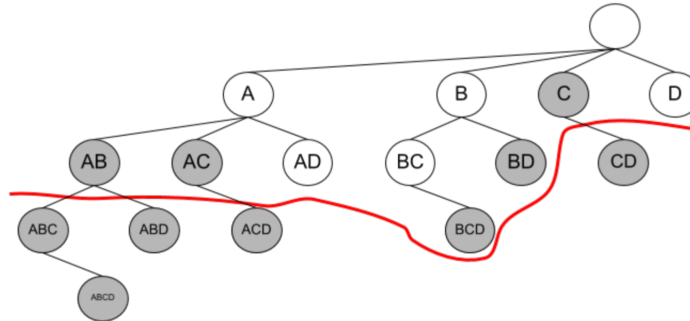

Figure 3.14: Example of a frontier between frequent itemset and infrequent itemsets

that prunes most infrequent itemsets). For this reason, it is reasonable to think two sub-trees of equal alphabet size will not share an equal amount of work if the depths of their roots are significantly different.

Another heuristic we could use instead of the depth of the root of a sub-tree is the frequency of the root of a sub-tree. This should provide better results that using the depth, as the frequencies of the children of an itemset is more directly correlated with the actual frequency of the said itemset, rather than its number of items. This technique is harder to implement efficiently, as the root of a PU can change after a work stealing. For this reason we did not use it, as we did not find an efficient way to implement it without a huge cost in memory accesses or extra computations, while keeping track of the alphabet size of the sub-trees is relatively costless.

Figure 3.14 shows an example tree where AB, BD, C and all their children itemsets are infrequent. The red curve represents the frontier between itemsets that a frequent itemset algorithm will test (in this case ECLAT), and the itemsets it skipped. This frontier does not match the real theoretical frontier between frequent itemsets and infrequent itemsets, as the algorithm has to find an infrequent itemset in a branch before deducing its children are infrequent with the Apriori principle. What this frontier illustrates is that the longer an itemset is, and the deeper it is in the tree, the less likely it is to be frequent and/or to be tested by the frequent itemset mining algorithm. This is the reason we favor sharing the work of sub-trees higher in the tree, as sub-trees lower in the tree are more likely to have a lot of itemsets skipped.

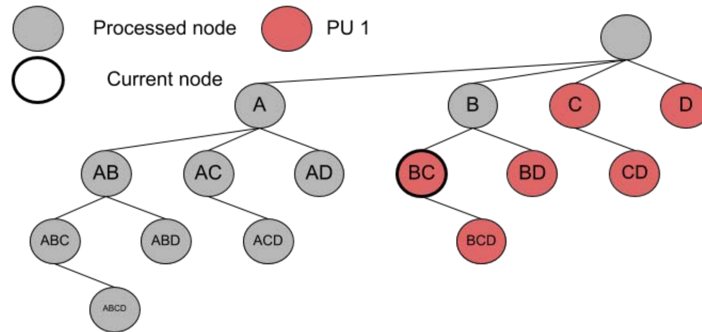

Figure 3.15: Sub-tree of alphabet size 4 mined on PU 1, mined halfway

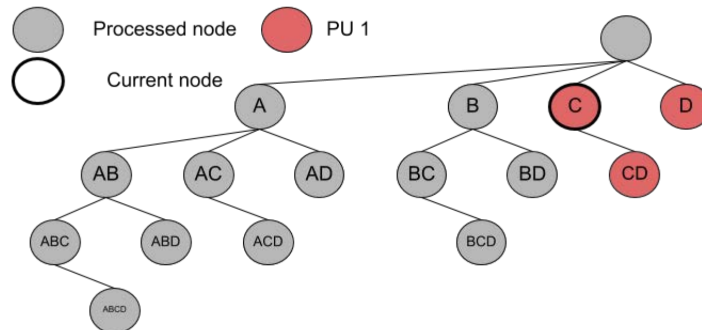

Figure 3.16: Sub-tree of alphabet size 4 mined on PU 1, $\frac{3}{4}$ of the sub-tree mined

Our implementation chooses randomly in the pool of PUs with the largest sub-tree and the smallest depth whenever it needs to proceed to work stealing, since we have no particular reason to choose one over the others.

After the decision to steal work from a PU is made, the actual redistribution of the work has to be done. As stated earlier, any sub-tree of alphabet size $K > 1$ can be divided in two sub-trees of alphabet size $K - 1$. It is however important to ensure a PU is not affected to a sub-tree that was already mined.

Example We can see by looking at Figure 3.15 and 3.16 that if we were to split a sub-tree into two smaller sub-trees of equal alphabet size, it would lead to affecting a PU with work that was already processed. Two naive "lazy" strategies are represented in Figure 3.17. The presented Strategy 1 would result in PU 1 (already working on the sub-tree) continuing its work on its half of the sub-tree, and the other PU being assigned a sub-tree with no work left, and immediately calling for work stealing. The

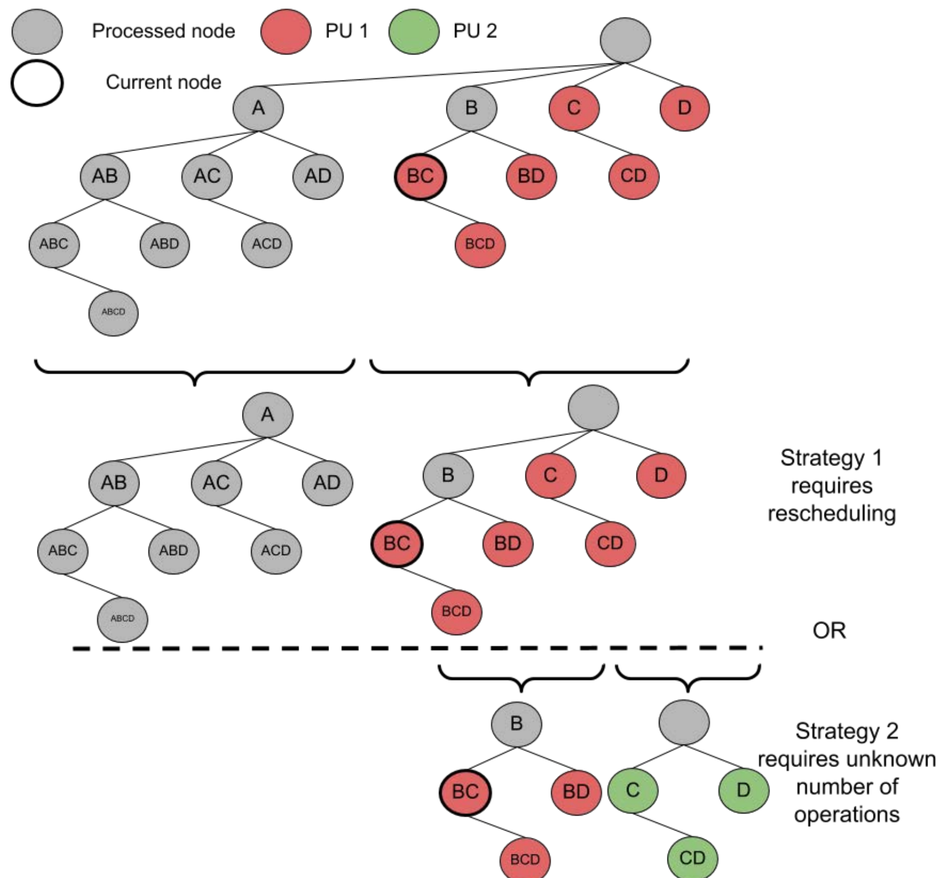

Figure 3.17: Two naive strategies for work stealing and their issues

solution we propose aims to simplify it without calling multiple times the work stealing routine back to back. We thus introduce a new mechanic to ensure that a PU is ready to split its sub-tree in two at any time, in case work stealing was to happen. Another solution, as presented in Strategy 2, would be to divide the half with remaining work into two new equal parts, or quarters of the original sub-tree. However this strategy can lead to an unspecified number of divisions when work stealing is called. In Figure 3.16, the sub-tree would have to be divided three times in a row (first division: subtree with root A and subtrees with roots B, C, D; second division: subtree with root B and substrees with roots C, D; third division: subtree with root C and subtree with root D). More generally any sub-tree with an alphabet size of $K > 0$ can require up to K subdivisions, leading to computations with a linear complexity.

Instead, the solution we provide is able to make all work stealing operations in constant time, with simple operations. Whenever a PU has processed the first half of its sub-tree 3.15, i.e. when it no longer processes itemsets with the first letter in its alphabet, it keeps only the half not yet processed, and discard the other, already finished half, to reach a state illustrated in Figure 3.18. When the first half of this new sub-tree is processed, the sub-tree is divided again in a recursive fashion to reach a state illustrated in Figure 3.19.

This operation is similar to the already implemented work stealing, affecting the PU to the half of the sub-tree to be processed, and discarding the already processed sub-tree. In practice this update of the sub-tree is done as soon as the PU reaches the itemset whose prefix is the root of the sub-tree, and suffix is the second letter of the sub-tree's alphabet. We also note that the root of a sub-tree is a special case, as it might already be processed when the rest of the sub-tree is not. Since there is only one root per sub-tree, we use a boolean flag during work stealing to handle such cases and start the itemset mining on the next node of the tree.

This way, if a PU tries to steal work from another PU, it can immediately divide its sub-tree into two halves with itemsets to process, without the risk of affecting a half that was already processed. It has also the benefit of updating the the alphabet size of the PU to signal other which sub-tree should be divided first during work stealing. With this proactive technique, the case displayed in Figure 3.17 is simpler. In this example, the part of the sub-tree containing the letter A is no longer present, and Strategy 1 for the work stealing can take effect immediately without further computation to find the resulting sub-trees.

Example This time, PU 1 will update its status whenever it finishes half of its sub-tree. In Figure 3.18, it means the sub-tree PU 1 decreases its alphabet size to become a sub-tree of alphabet size 3 when it reaches the itemset B. In Figure 3.19, the sub-tree becomes a sub-tree of alphabet size 2 when PU 2 reaches the itemset C. When work stealing happens in Figure 3.20 and 3.21, it can proceed the same way as it would if PU 1 was still working on the first half of its tree.

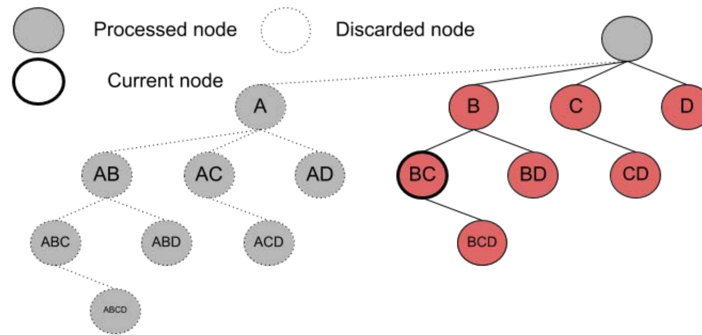

Figure 3.18: Sub-tree of alphabet size 4 mined on PU 1, mined halfway, with alphabet size updated to 3

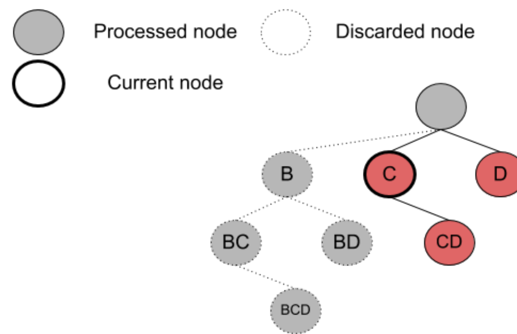

Figure 3.19: Sub-tree of alphabet size 4 mined on PU 1, $\frac{3}{4}$ of the sub-tree mined, with alphabet size updated to 2

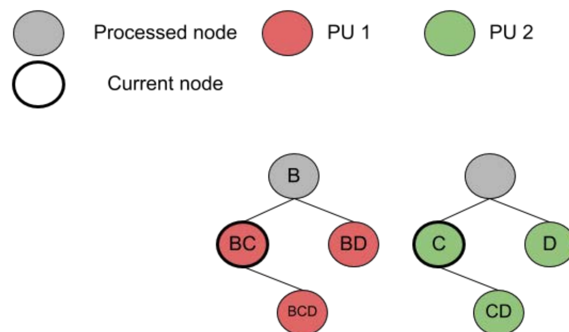

Figure 3.20: Work stealing on sub-tree after it reduced its alphabet size to 3

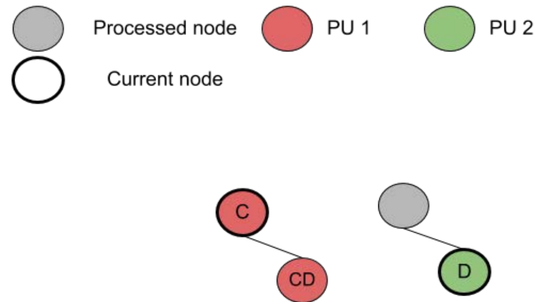

Figure 3.21: Work stealing on sub-tree after it reduced its alphabet size to 2

This way, a PU assigned with an initially large sub-tree can reduce dynamically its alphabet size by mining its itemsets. When a PU calls the work stealing routine, it will steal work from a PU currently mining one of the largest sub-trees, instead of looking for the sub-trees with largest initial alphabet size. This implies that a PU that rapidly processes its sub-tree (because there are many infrequent itemsets) will be less likely to get its work stolen than a PU that spends a lot of time on its sub-tree.

When sampling with random SAT constraints is introduced, then the alphabet used by the tree corresponds to the free alphabet we defined earlier in the chapter. Any subsequent local alphabet used by a sub-tree is thus a subset of the free alphabet. When a new constraint is generated, all local alphabets are updated if need be, in the same way it is done in the iterative algorithm. If the local alphabet of a PU becomes empty, the PU proceeds to work stealing. If a PU is in a position where it was going to mine an itemset that no longer satisfies the XOR constraints, it has to look for the first sibling of the itemset whose suffix is the letter that was removed from the free alphabet (as shown previously in this chapter). It is however possible this sibling is not in the current sub-tree, in which case, the entire sub-tree belongs to an unsatisfiable branch of the tree. This can be detected by comparing the root of the sub-trees with the new SAT constraint.

Interaction with correspondence list With multiple PUs mining each a sub-tree, the use of a single correspondence list is no longer possible, as each PU can go up and down their own sub-trees independently. A possible solution is to allocate a dedicated correspondence list per PU, and each PU manages its list the same way a

correspondence list is managed in the iterative algorithm. During the support counting phase, the union of all correspondence lists has to be checked in the database. The number of transactions pulled from the database in the memory will depend on this union of correspondence lists. Contrary to the iterative algorithm, that needs to pull all transactions in the correspondence lists once, if a transaction is present in multiple lists at once, the parallel algorithm will only pull the transaction once. For the support counting, the theoretical speedup is thus:

$$1 \leq Speedup = \frac{\sum\limits_{PU} |Corr.\ List_{PU}|}{|\bigcup\limits_{PU} (Corr.\ List_{PU})|} \leq Nb.\ of\ PU$$

The minimum speedup possible of 1 is obtained if the intersection of the correspondence lists is empty, all transactions in the lists will be pulled once from the database. The maximum speedup possible is obtained if all correspondence lists are equal across all PU. In that case, pulling the transactions from the union of the list is equivalent to pulling the transactions from a single list. In order to maximize this speedup, itemsets mined at a given time across all PU should share as many transactions as possible. In practice, this means the PU should work on sub-trees close to one another so they share as many items in their itemsets as possible. This contradicts the work stealing strategy we described earlier that distributes PUs on sub-trees of equivalent alphabet sizes. Depending on the database, and if the correspondence list mechanism reduces considerably the number of transactions pulled from the external memory, then another work stealing policy should be privileged, by grouping PUs on smaller sub-trees in a local area. This introduces a trade-off between how local we want the mining to be, and how often should we want to reassign PUs to new sub-trees.

Algorithm 4 shows the pseudo-code of our parallelized version of ECLAT (with sampling). The main differences with the original algorithm are

- the generation of several candidate itemsets in parallel on multiple PUs,
- a support counting function able to measure the support of multiple itemsets at the same time,
- a work stealing routine,

Algorithm 4 eclat_parallel_pseudo_code

```
while not finished do
  for all PU do
    compute next itemset of PU
    update alphabet size of PU's sub-tree
    compute corresponding sample of PU
  end for
  count in DB({free itemset_i, sample_i | i ∈ ⟦1, NB_PU⟧})
  if number of results > threshold then
    create constraint
    for all PU do
      update local free alphabet and free itemset
      if PU sub-tree is over then
        steal the work of another PU
      end if
    end for
    prune results with new constraint
  end if
  for all PU do
    if PU sub-tree is over then
      steal the work of another PU
    end if
  end for
end while
```

- an edge-case scenario to update the information about a PU's sub-tree when it processed half of it, and
- an edge-case scenario to enable work stealing if a sub-tree no longer hold satisfiable itemsets when a new constraint is generated.

Conclusion

We proposed a first approach for performing pattern sampling on FPGA. This approach is promising: despite running on a modest FPGA clocked at 100 MHz, it can be up to an order of magnitude faster than a software version running on a server class Xeon CPU clocked at 3.2 GHz. We also proposed a dataset reduction technique applicable to depth-first algorithms as ECLAT to improve execution time, at the cost of allocated memory on the FPGA.

The current approach is sequential: an exciting research direction is to parallelize the implementation in order to explore several branches of the search space simultaneously. Since itemsets access to the same database, the transactions can be used for the support counting of multiple itemsets. These two properties allow to divide the whole tree in sub-trees that can be processed in parallel without requiring more communication between the FPGA and the memory.

This parallel extension of the proposed accelerator will allow to make good use of higher-end FPGA and reach throughputs compatible with real-time analysis on demanding streaming data. This thesis focused on the FPGA algorithm and its performance. In order to process actual streams, another important element is to have efficient communications between the RAM holding the data and the FPGA. In this regard, a solution is to compress the database in RAM, which will cost a small allocation of resources on the FPGA to decode the transactions, but will allow a much better use of the bus bandwidth.

CHAPTER 4

STREAM SAMPLING

The rate of data generation in many fields can be so high that it becomes too difficult to store and process it offline. Stream mining techniques are designed specifically to analyse data incoming from a stream with unique methods to avoid problems associated with streams, such as consecutive reruns of pattern mining on almost identical databases.

Stream Mining is the sub-field of itemset mining where the database is a continuous unbounded stream of transactions that cannot be fully stored in memory. (extend motivations here and add a figure)

Every transaction now has a timestamp information in addition to its itemset in the form of a tuple $(timestamp, itemset)$. In our case, the transactions arrive in batches that have to be processed at the same time, meaning all transactions that contain a timestamp contained within a certain interval will be in the same batch. The size of the batches can be fixed or not depending on the problem, and can be any positive integer. Batches are processed one by one as they arrive. Data might arrive at a regular pace or not depending on the type of data processed, or the user specification. Table 4.1 shows an example stream of four batches, each batch containing two transactions. Parameters such as the number of transactions per batch, or the frequency at which batches arrive for processing are dependant on the stream to be processed and/or user settings.

$\mathcal{B}_1$	$\mathcal{T}_1$	$\{a, b\}$
	$\mathcal{T}_2$	$\{c, d, b\}$
$\mathcal{B}_2$	$\mathcal{T}_3$	$\{c\}$
	$\mathcal{T}_4$	$\{a, c, d\}$
$\mathcal{B}_3$	$\mathcal{T}_5$	$\{c\}$
	$\mathcal{T}_6$	$\{a, c, d\}$
$\mathcal{B}_4$	$\mathcal{T}_7$	$\{a, b\}$
	$\mathcal{T}_8$	$\{a, d\}$

Table 4.1: Example of stream $\mathcal{S}$

4.1 Context and related work

4.1.1 Itemset Mining in a stream

The immediate issue of streaming data for itemset mining is the fact that typical algorithms like ECLAT process all data at once to produce their results. New data must be stored at all time to be taken into account, which leads to huge amount of allocated memory, or not taking into account much information from the stream. These algorithms also need to start from scratch each time the dataset changes, meaning they are usually not able to use previous mining results even if the new dataset is very close from the current one. For this reason, algorithms not designed to handle data streams have particularly poor time performance, since they run multiple times on highly overlapping data.

Stream mining algorithms, on the other hand, are designed to be continuously running over streaming data: they don't need to be restarted, and they don't take into account all the data. After some defined time delay, raw past data is removed from the stream mining algorithm working memory, avoiding constant growth of memory usage. However, as this past data contains useful information about the "background frequency" of the most frequent itemsets, a summary of this information is maintained by the algorithm.

The key strategy of stream mining algorithms is how they handle the memory of past events in their results. This strategy will dictate how they can free memory by discarding old batches and update it with new streaming data. To differentiate these

strategies, most approaches rely on *windows*. The window used by a stream mining algorithm is the set of batches from the stream that have a direct impact on the results. For instance, an algorithm that only keeps in memory the last seen n batches will use at time $t \in \mathbb{N}$ the window $[batch_{t-n+1}, ..., batch_t]$.

Three classes of algorithms are described in the literature that differ on how they handle the memory of past events: using a landmark window, a damped window, or a sliding window. Landmark windows typically keep memory of all batches seen since the beginning of the itemset mining, while sliding windows only recall the last n batches. The damped window is a variation of the landmark window that puts more emphasis on newer batches.

Figures 4.1 through 4.3 show an example that differentiate the three types of windows. In these examples we skipped the arrival of the first three batches to highlight the general case without considering the initialisation phase of the stream. Note that in this example, we considered the stream mining algorithm to be perfect, and its results without false positives or false negatives, as it can happen in typical algorithms that aim to process high throughputs of data streams.

Landmark window The landmark window showcased Figure 4.1 uses a relative support threshold of 25%. At any point of the stream, its results holds the itemsets present in more than 25% of the total number of transactions received since the beginning of the run. This means that the absolute support threshold, i.e. the total number of transactions required for an itemset to be frequent, will increase with time. For this reason, it is possible for itemsets to be frequent at one point in the stream, then to be evicted from the results if their frequency drop. This type of window can be ill suited for very long streams, as there usually are not not many itemsets to return if the threshold is too high, making applications such as phase detection troublesome. Additionally, as the window covers the whole stream, it is not easy to determine if a frequent itemset is recent, old, or scattered all over the stream.

Sliding window With the sliding window model shown in Figure 4.2, a batch of transaction is removed from the window each time a new batch is received (after the initialisation phase is over). In this case, the relative threshold of 25% and the absolute

threshold remain constant during the run. This type of window requires a user defined parameter, the window size, that will greatly impact the results. The main issue of this method is the need to buffer the batches in the stream as long as it is present in the window. Algorithms that use a sliding window require to access information from a batch to effectively remove it from the window

Damped window The damped window shown Figure 4.3 is rather similar to the landmark window, but uses a user defined parameter called decay, or decay rate, called $d \in$[0,1]. Each time a batch arrives and is processed like it would with a landmark window, the algorithm using the damped window model will proceed to applying decay to its results. The support of frequent itemsets are replaced with a "weighted support". When decay is applied to the results, the weighted support of every itemset in the results is multiplied by d. In the example of Figure 4.3, the weighted supports are computed by counting the occurrence of itemsets in the leftmost batch, applying a decay, adding to it the occurrence of the itemset in the second batch, applying a decay, and so on. As decay is applied to the past frequent itemsets, parts of them may become infrequent, especially if they no longer occur in recent windows. While this type of window theoretically has the same size as the landmark window (both received all the batches of the stream without explicitly removing any of them) its border is not hard-defined and is rather fuzzy. For instance, after receiving N batches, if an itemset X was last seen k batches ago, with a decay rate of d and t transactions per batch and a frequency threshold of t_h, it is impossible for X to be present in the results if $t_h > (t * \sum_{i=1}^{N-k} d^i) * d^k$, where $t * \sum_{i=1}^{N} d^i$ is the maximum possible weighted support of X the last time it was recorded (meaning it was in every possible transaction up to this point). However, if the itemset X reappears at any point during the k last batches, its weighted support might still be impacted by transactions that appeared at the very beginning of the stream, albeit with very little incidence due to the exponential decay. As the size of a damped window is thus very hard to estimate, the support threshold are given as absolute values, instead of relative to the window's size.

Figure 4.4 illustrates the difference between the three types of windows, showing the scope and "weight" of an itemset that would appear during the stream. With the landmark window model, all $N+1$ transactions are given the same importance, as we discussed previously. The sliding window allows to focus on exactly $N-M$ transactions,

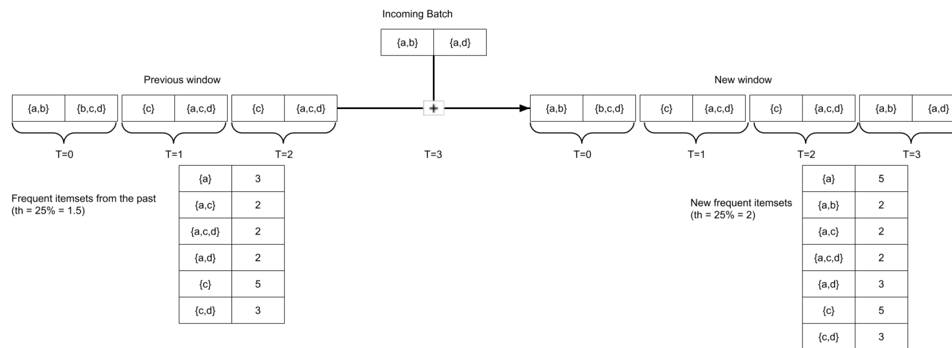

Figure 4.1: Use of the landmark window on the toy database with batches of two transactions

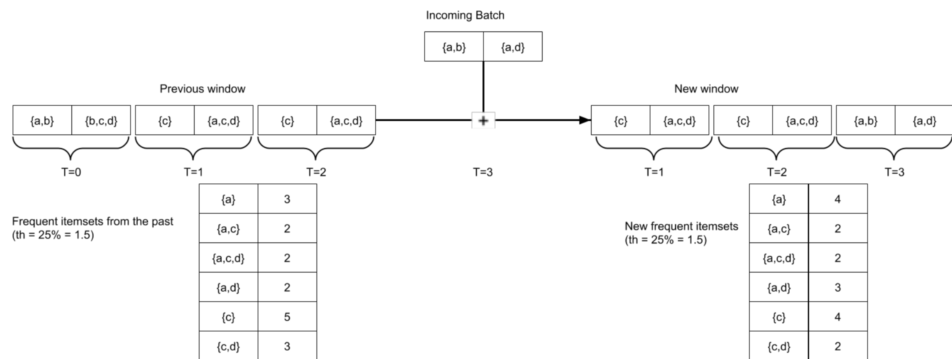

Figure 4.2: Use of the sliding window on the toy database with batches of two transactions and window size of three batches

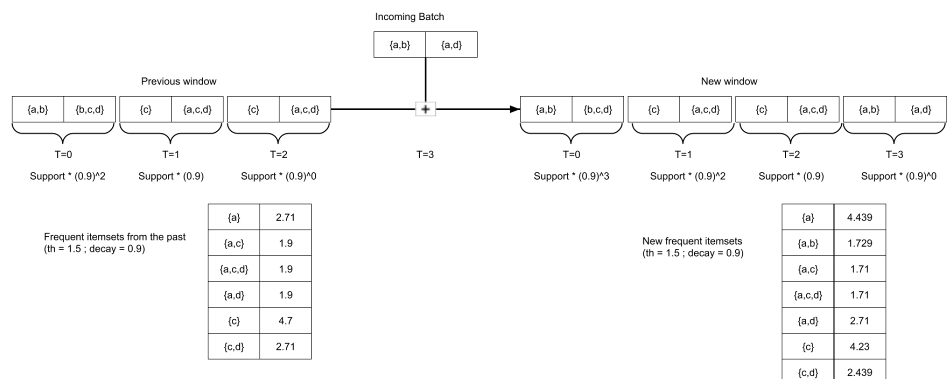

Figure 4.3: Use of the damped window on the toy database with batches of two transactions and decay rate of 0.9

making it easy to find frequent itemsets in the past hour/day/week for instance. For the damped window model, the weights are adjusted every time a batch arrive to give more emphasis on newer data, giving a simple approximation of a sliding window by changing the "slope" of the weights when tuning the decay rate.

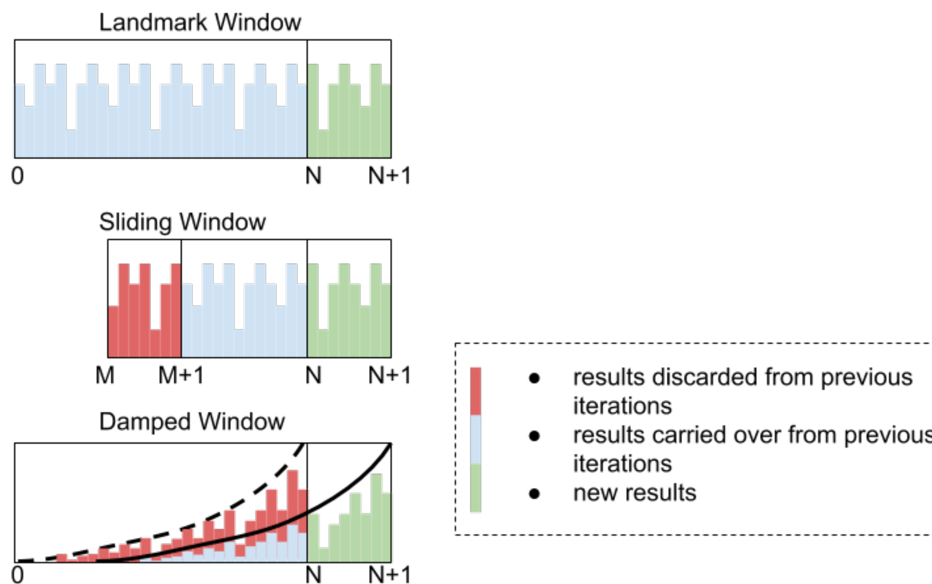

Figure 4.4: Landmark window, sliding window, and damped window

Our approach uses the damped window approach, as it can provide a weighted window that approximates the sliding window, but does require to buffer all batches in the window, allowing for a less costly implementation based on the landmark window model.

4.2 General Approach

4.2.1 Stream mining and Itemset sampling

In this section, we introduce our stream sampler that processes new batches of transactions as they arrive and updates its results accordingly. The inner mechanisms used to process the batches heavily reuses the ECLAT-based sampler described in the previous chapter, and adds new functionalities to support streaming data. This implemen-

tation leads to a stream mining solution with output space sampling, and is, to the best of our knowledge, the first output space sampler for streaming itemset.

The adaptation of our itemset sampler to streaming data require several adjustments. It is for example possible that the frequency of a frequent itemset in the results needs to be updated, while the same itemset is infrequent in the latest batch, thus making it unseen by the modified ECLAT algorithm. For this reason, the ECLAT-based sampler will now also look in previous results each time it tests a new itemset to check if it was already frequent. The decay mechanism used for the damped window also introduces a new feature in the sampling: the number of frequent itemsets might decrease, and pass below the minimum number of results. Existing constraints might be removed from the sampler to allow the number of results to be in the range requested by the user.

This approach requires to keep in memory a possibly large number of itemsets found in the previous windows, and to have the ability to quickly query the presence/absence as well as the frequency value of any of these itemsets. In our case, we use a FP-Tree (described in Chapter 1.2) as a storage mechanism for the results of our ECLAT-based algorithm, without using the original FP-Growth algorithm for the enumeration (ECLAT being more adapted to execution on a FPGA).

For our algorithm, the FP-Tree is simply a structure holding all results at a point in the stream, organized in such a way that it will be faster to retrieve any particular information from it. Using such a FP-Tree containing all previous frequent itemsets, we demonstrate that it is possible to retrieve all information about an itemset seen in the past stream in $O(1)$ time, compared to $O(n)$ with a regular FIFO data structure.

In the next section are described the three main new features of the sampler to handle streaming data: Navigating through the FP-Tree to find previous frequent itemsets, maintaining the FP-Tree when the sampling requires to add or remove a random constraint, and the handling of the decay between batches.

4.3 Techniques

4.3.1 Navigating the FP-Tree during a scan

In the litterature, FP-Trees are n-ary trees with nodes linking to up to $n-1$ children, where n is the size of the alphabet. While this structure is suited for a software implementation, this would be challenging to port efficiently to a FPGA. For this reason, we opted for a systolic architecture, where all nodes in a FP-Tree are identical structure-wise. Instead of having any number of children, a node in our FP-Tree has a first child and a first sibling.

To assign $k \in \mathbb{N}$ children to a node, $k \leq n$, the node is given a link to its first child node. Then, this first child is given a link to the second child node as its first sibling. Every i^{th} node, $0 \leq i \leq k-1$, is assigned the $(i+1)^{th}$ node as its first sibling. The siblings/children are arranged in the lexicographical order of the alphabet used.

With this structure, it is possible to represent a n-ary by using a binary tree in which it is easier to navigate, given all nodes have the same number of pointers.

In our implementation, we allocate two more pointers in the node to allow backtracking in the FP-Tree. The two pointers *first child* and *next sibling*, are given counterparts *parent* and *previous sibling*.

When scanning for itemsets, the ECLAT-based sampler can use three transformations to go from an itemset to the next. In terms of operation in the enumeration tree, the next itemset can either be the first child of the current node, its next sibling, or the next sibling of the parent of the current node. There exists a mapping between these operations in the enumeration tree and operations having the same effect in the FP-tree. Our streaming algorithm exploits this mapping to navigate simultaneously in the ECLAT enumeration tree of the current window (for discovering new frequent itemsets) and in the FP-tree holding frequent itemsets from past windows as well as those found in the new window. This simultaneous exploration avoids to re-traverse the FP-tree at each lookup operation and allows to have a $O(1)$ lookup complexity.

Example run Figures 4.5 through 4.11 showcase an example where most all possible cases are present, using a simple stream of two batches shown Table 4.2. The first

batch has already been processed, and results in the FP-Tree shown in Figure 4.5. The following steps describe how the second batch will be processed. Two pointers are used in the figures, one representing the current candidate itemset in the theoretical enumeration tree containing all the itemsets the algorithm might test (left side of the figures), a real pointer used to memorise a position inside the FP-Tree stored in memory (right side of the figures). Keep in mind the enumeration tree is not stored anywhere during the computation, as the algorithm only computes the value of the current candidate on the fly.

t_0	a ac ac ac ac ac
t_1	ab ab ab abd abd abcd abcd acd acd

Table 4.2: Example stream of two batches

Figure 4.5 starts with the assumption that prior data was already found in previous batches of the stream, namely, $\{a\}$ was found with a support of 8, and $\{a, c\}$ with a support of 5 at the start of the new batch. Only a portion of both trees will be shown for the sake of space and simplicity.

Later in the section, we explain in more details all possible cases (if an itemset is frequent or not, if it previously had children or not, if it is a leaf or not) to ensure all operations that add a node to the tree are correct. This example might skip over some heavy explanations to give a better intuition of the algorithm.

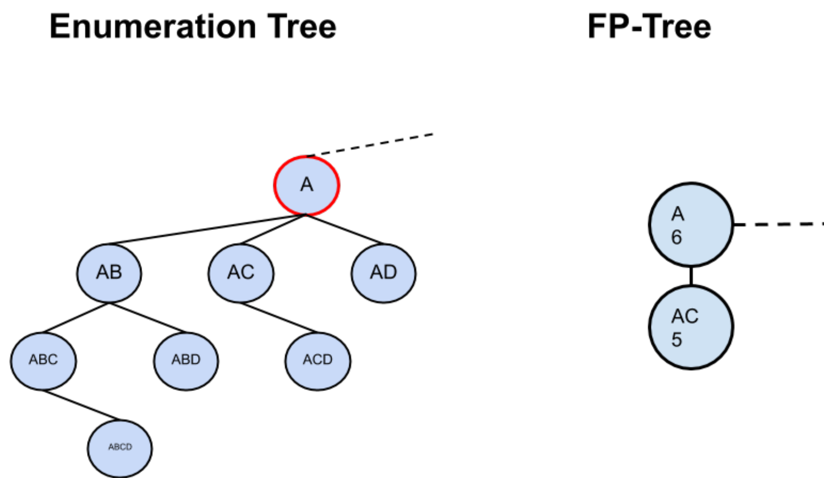

Figure 4.5: Part of the tree to be mined (left) and part of the FP-Tree before a new batch is processed (right)

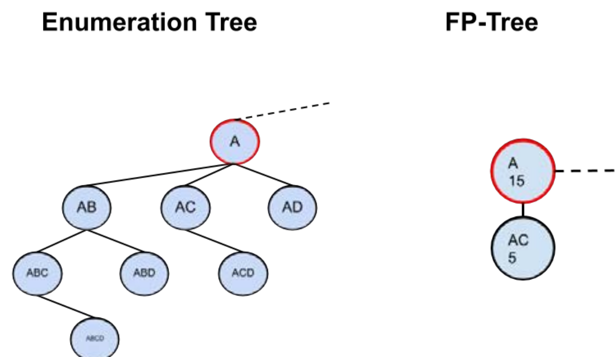

Figure 4.6: Step 1: mining itemset A

Step 1 In Figure 4.6 the algorithm tests the support of the itemset $\{a\}$ in the stream. As $\{a\}$ is already present in the FP-Tree, its support is updated by adding the number of new occurrences of the itemset.

Step 2 In Figure 4.7, the algorithm generates another itemset $\{a, b\}$. If this itemset existed in the FP-Tree, it would be the first child of the last itemset $\{a\}$ (due to the lexicographical order of the alphabet), but the first child is $\{a, c\}$. Therefore, $\{a, b\}$ cannot exist anywhere in the FP-Tree prior to this batch. $\{a, b\}$ is tested, and since it is a frequent itemset with a support of 6, it must be added to the results in the FP-Tree. Its place in the tree has to be as the first child of $\{a\}$ (for the previously mentioned lexicographic reasons), and the node that was previously the first child of $\{a\}$ becomes the first sibling of $\{a, b\}$. The FP-Tree pointer is moved to the node containing $\{a, b\}$.

Step 3 In Figure 4.8, the algorithm generates another itemset $\{a, b, c\}$. If this itemset existed in the FP-Tree, it would be the first child of the last itemset $\{a, b\}$, but it has no children so $\{a, b, c\}$ is not in the FP-Tree. $\{a, b, c\}$ is tested but found infrequent, so it will not be added to the FP-Tree, and the FP-Tree pointer stays on $\{a, b\}$.

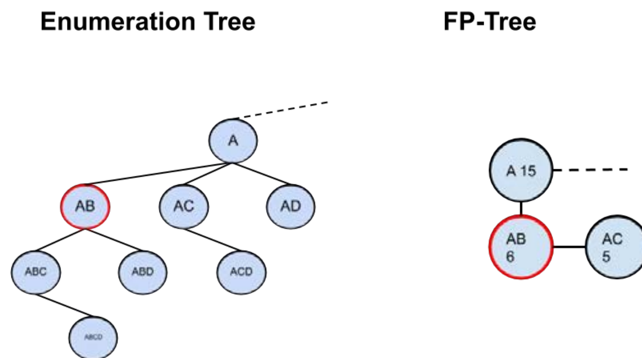

Figure 4.7: Step 2: mining itemset AB

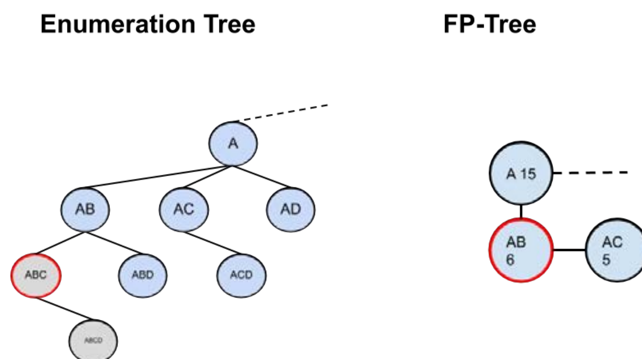

Figure 4.8: Step 3: mining itemset ABC

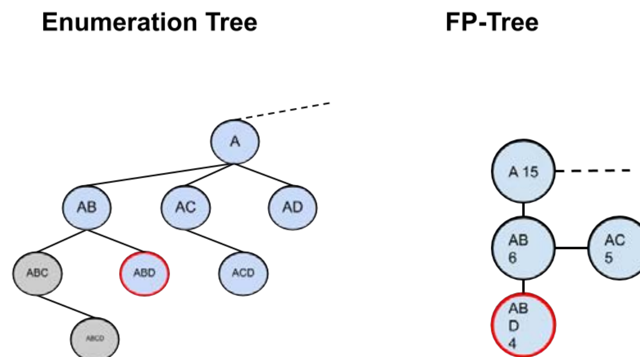

Figure 4.9: Step 4: mining itemset ABD

Step 4 In Figure 4.9, the algorithm generates another itemset $\{a, b, d\}$. If this itemset existed in the FP-Tree, it would be a child of the last itemset $\{a, b\}$, but it still has no children. Therefore, $\{a, b, d\}$ is not in the FP-Tree prior to this batch. $\{a, b, d\}$ is tested, and since it is a frequent itemset with a support of 4, it must be added to the results in the FP-Tree. Its place in the tree has to be as a child of $\{a, b\}$, and since it has no children, it will be its first child. The FP-Tree pointer is moved to the node containing $\{a, b, d\}$.

Step 5 In Figure 4.10, the algorithm generates another itemset $\{a, c\}$. Because the last itemset $\{a, b, d\}$ contains *d*, the last letter of the alphabet, it is a leaf in the enumeration tree, and the next itemset tested ($\{a, c\}$) is the first sibling of its parent ($\{a, b\}$). For this reason, the first operation is to move the FP-Tree pointer to $\{a, b, d\}$'s parent. From here, if $\{a, c\}$ is present in the FP-Tree, is must be the first sibling of $\{a, b\}$ (due to lexicographic reasons). $\{a, c\}$ is in the FP-Tree with a prior support of 5, and testing its support in the new batch updates it to 10. The FP-Tree pointer can then be moved to $\{a, c\}$.

Step 6 In Figure 4.11, the algorithm generates another itemset $\{a, c, d\}$. If this itemset existed in the FP-Tree, it would be a child of the last itemset $\{a, c\}$, but it has no children.

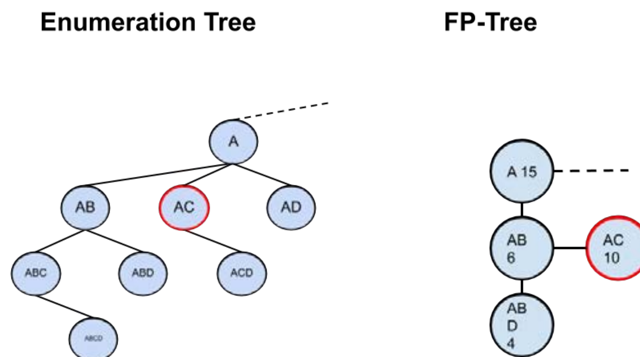

Figure 4.10: Step 5: mining itemset AC

Therefore, $\{a, c, d\}$ is not in the FP-Tree prior to this batch. $\{a, c, d\}$ is tested, and since it is a frequent itemset with a support of 5, it must be added to the results in the FP-Tree. Its place in the tree has to be as a child of $\{a, c\}$, and since it has no children, it will be its first child. The FP-Tree pointer is moved to the node containing $\{a, c, d\}$. This concludes this small example, which has given the general intuition on how our approach operates.

We now explain in detail the simultaneous traversal of the ECLAT enumeration tree and of the FP-tree storing frequent itemsets frequencies.

If the sampler generates a candidate that is the first child of the last tested itemset in the enumeration tree, then the last tested itemset was frequent (otherwise the current candidate would be skipped). The pointer in the FP-Tree is updated to match the last tested itemset, and if the new candidate exists in the FP-Tree, it must be the first child of the pointed node of the FP-Tree. If the new candidate does not exist in the FP-Tree, the last tested node does not have children (example in **steps 3**, **4** and **6**), or the first child is different from the new candidate (example in **step 2**).

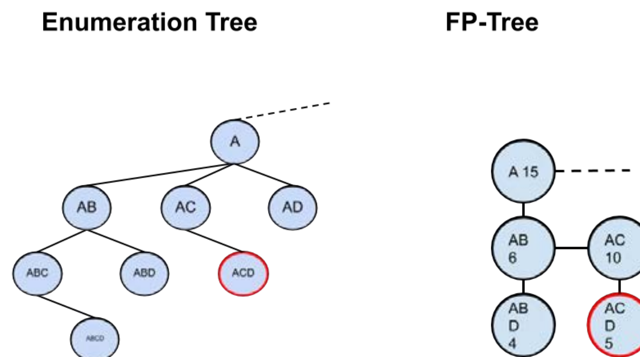

Figure 4.11: Step 6: mining itemset ACD

If the candidate existed in the FP-Tree, then its frequency is updated by the sampler (example in **steps 1** and **5**). Otherwise, it can be either frequent or infrequent. If it is infrequent, it is ignored (example in **step 3**). If it is frequent, a new node is created in the FP-Tree as the first child of the pointed node (examples in **steps 2**, **4** and **6**). Any previous first child of the pointed node will become the sibling of the new first child node (example in **step 2**).

If the sampler generates a candidate itemset that is the next sibling of the last tested itemset, then the last tested itemset was not frequent (otherwise, the new candidate would be one of its children). The pointer in the FP-Tree is still up-to-date, as no new information has been added to the FP-Tree in the last step. If the new candidate exists in the FP-Tree, the candidate is the sibling of the last frequent node in the general case. Another case can arise when the parent of the current candidate had no previous frequent child in the current batch. In that case, the pointed node in the FP-Tree is still the parent node of the current candidate (example in **step 3**). For this reason, finding the candidate in the tree can require to check the sibling and/or the first child of the pointed node. The update/adding of the candidate in the FP-Tree is done in a similar fashion to the previous case, where the updated node can be either the first child or next sibling of pointed node.

If the last itemset tested by the sampler is a leaf, the next candidate is the sibling of the parent of the last tested itemset. First, the pointer in the FP-Tree is updated to be the previous sibling of the candidate. Two possible cases arise, either the pointer points to a previous sibling of the last tested itemset, and the pointer needs to be updated as its parent (example in **step 4**); or there were no frequent child, and the last update to the pointer is still the previous sibling. Once the pointer is updated, the procedure is the same as when the candidate was the sibling.

Finding the next itemset in the FP-Tree and updating or adding new results are done in $O(1)$, as the pointer in the FP-Tree always move of at most two nodes in the tree.

4.3.2 Maintenance of the FP-Tree with new constraints

In order to keep the coherency between the tree scanned by the ECLAT-based sampler and the FP-Tree, they must both use the same alphabet. Recall from Chapter 2 that adding or removing constraints modifies the alphabet used by the sampler (the “free” alphabet), thus it is necessary to maintain the FP-Tree whenever changes are made to the constraints.

For this subsection, an example FP-Tree shown Figure 4.12 will be used to display the different maintenance operations. Firstly a XOR constraint, $C \oplus D = 0$, is randomly generated, the satisfiable and unsatisfiable nodes are shown in Figure 4.13, respectively blue and white. The algorithm will scan the FP-Tree in a depth-first fashion to check if nodes should be kept, removed or even displaced.

When a new constraint is generated, one random letter is no longer usable in the free alphabet used by the sampler. In the example, the generated constraint was equivalent to $D = C$, D is thus the letter removed from the free alphabet. To maintain the FP-Tree, all nodes are checked. A node in the FP-Tree contains an itemset that satisfies or not the new constraint, and that itemset contains or not the letter that will be removed from the alphabet.

In order to keep the FP-Tree coherent, it is necessary to ensure that itemsets are placed in a correct way with respects to the depth-first algorithm followed by the ECLAT-

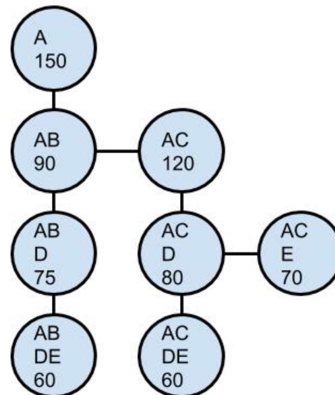

Figure 4.12: FP-Tree before new constraint is added

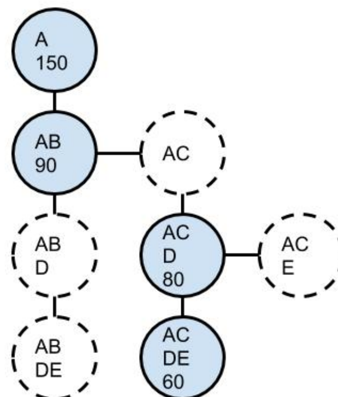

Figure 4.13: Satisfiable (blue) and unsatisfiable (white) nodes

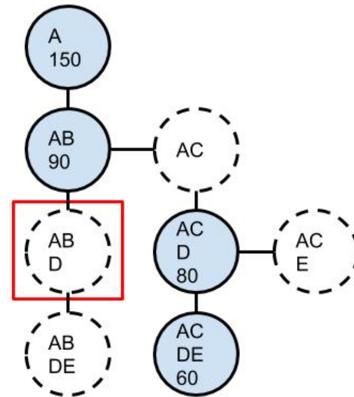

Figure 4.14: ABD is unSAT and contains the letter to be removed, it and all its children can be removed

based sampler. Given the Apriori principle, it is guaranteed that a node containing the "reduced itemset" (its corresponding itemset in the alphabet where the letter has been removed) already exists in the FP-Tree (as it contains a subset of the frequent itemset). It is then possible to migrate the information about the current itemset to the already existing node (that currently contains an unsatisfiable itemset that will be discarded).

Case 1 If a node does not contain the letter to be removed, and satisfies the new constraint, it is left unchanged. In Figure 4.13, we can see that the two nodes containing A and AB correspond to this case. In the depth-first traversal they are the two first node checked and will remain unaffected for the rest of the maintenance.

Case 2 If the node contains the letter to be removed, and does not satisfy the new constraint, it can safely be removed from the FP-Tree. Because the letter to be removed in the alphabet is the last letter present in the constraint using the lexicographical order, it is guaranteed that all children of the current node contain the node to be removed. For this reason, none of the children and their subsequent children satisfy the new constraint, and all of them can be safely removed. For example in Figure 4.14, the itemset ABD does not satisfy the constraint $D = C$, and since D is in ABD, D is present in all children itemset of ABD, and none of the letters prior to D that are not present in ABD are present in its children (since a child of an itemset is generated

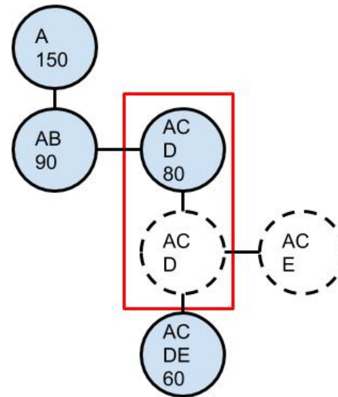

Figure 4.15: AC does not contain the letter to be removed, but is unSAT, it will be replaced by the node ACD

via the I-steps defined in the earlier chapter). The nodes containing ABD and all its children, in this case $ABDE$, can be removed from the tree.

Case 3 If a node does not contain the letter to be removed, but does not satisfy the new constraint, the current itemset will be removed from the tree. However, this particular place in the tree might be used by another itemset that satisfies the constraint, but currently contains the letter to be removed. Figure 4.13 shows this particular case. AC does not contain D but does not satisfy the constraint $D = C$, the information about the item AC can thus be safely discarded. However, we aim to maintain the FP-Tree such that the depth-first order is always in place. For each itemset that does not satisfy a XOR constraint, there exists exactly one itemset that satisfies it. The algorithm will perform a $O(n)$ search to try to find that itemset whenever it reaches this case. Here the corresponding itemset is ACD, that contains D and satisfies the constraint. Because it contains D, its node cannot remain in the FP-Tree after we remove D from the free alphabet. Our policy is the following: the node containing an unsatisfiable itemset without the letter will have its information replaced by its corresponding satisfiable itemset that necessarily contains the letter. The node that contained the corresponding itemset is then removed from the tree as its information has been displaced. In Figure **??** we temporarily put ACD in the node that contained AC

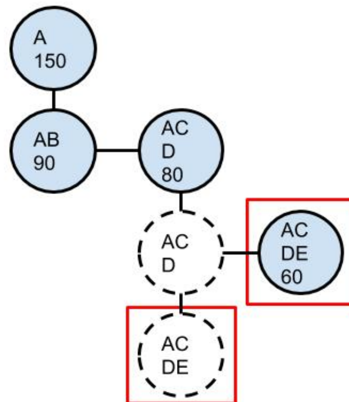

Figure 4.16: ACE does not contain the letter to be removed, but is unSAT, it will be replaced by the node ACDE

Case 4 If a node does contain the letter to be removed, but satisfies the new constraint, its information must be kept in the FP-Tree, but the node itself will not be able to exist with the new alphabet. For this reason, the information will have to be transposed to another node, namely, to the node containing the itemset that correspond with the bijection generated by the new set of constraints. This case is the dual of the previous one. Following the depth-first scan in the FP-Tree shown Figure **??** the algorithm comes accros the node containing $ACDE$ before the node containing ACE. The goal of the current operation will be the same as previously, looking for the corresponding node that contains the itemset ACE with a $O(n)$ search. Once it is found, the information of the unsatisfiable itemset ACE is replaced by the information of $ACDE$, and the node previously containing $ACDE$ is removed from the tree.

Case 3 and 4 a dual cases, the operations invoked are almost the same (case 3 searches in its children, case 4 searches in the siblings of its parent), and both nodes are processed at once. One of the node is removed, the other one will fall in case 1 if the scan ever scans it later.

In practice, a node is removed from the tree only if all its children and subsequent children can also be removed. Nodes to be removed in the case 2 can be safely removed immediately, but it is not the case for cases 3 and 4, that might involve children

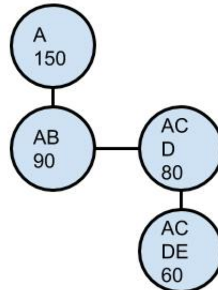

Figure 4.17: FP-Tree after all unSAT have been removed

node to process. Instead of removing the nodes to be removed in cases 3 and 4, they are marked to be removed. Similarly to case 2, it is possible to show that all the children of a node marked to be removed in cases 3 and 4 will have to be removed too, however their information have not been displaced yet to their corresponding nodes. The depth-first scan can thus check all nodes before removing them in a second depth-first traversal that will remove the marked nodes. A possible optimisation is to remove the marked nodes as the algorithms backtracks the depth-first tree.

Figure 4.17 shows the FP-Tree once all the nodes containing unsatifiable itemsets are removed. As we can see, the remaining nodes involved in cases 3 and 4 contains the letter D. From now on, we change representation, and use the new free alphabet. This way, referring to AC and ACE in the free alphabet will refer to ACD and $ACDE$ when results are output. This results in Figure 4.18, where the letter D has been removed from all nodes. This is simply a representation detail, as in practice the letter to be removed is taken care of during the first scan. While several $O(n)$ searches per node in a full scan of the FP-Tree might seem costly, in an output sampling context, the number of result nodes is bounded by a user defined parameter, and so are the searches in the FP-Tree.

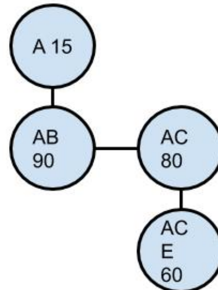

Figure 4.18: FP-Tree after complete removal of the letter

4.3.3 Applying decay to the FP-Tree

Between batches, decay is applied to all results in the FP-Tree. If the frequency of an itemset in a particular node gets lower than the frequency threshold, the node and all its children nodes contain infrequent itemsets, and can be removed from the tree.

Once the decay has been applied to the FP-Tree, it is possible that the number of frequent itemsets is lower than the lower threshold. This means too many constraints are currently being applied, and one or more constraint must be removed from the current set of constraints to allow the number of results to be between the bounds.

While our ECLAT-based sampler is able to generate new constraints dynamically as new frequent itemsets are found, no strategy have been specified when it comes to choosing which constraint to remove from the whole set of already existing constraints. According to the original WeightMC, all constraints are equivalent from a statistic point of view, so the number of frequent itemset that satisfy them or not are asymptotically equal. Which constraint to be removed is thus arbitrary to some extent, and we propose several policies to determine which constraints to remove first.

The first policy, inspired by WeightMC, is to remove the last generated constraint. This Last In First Out strategy ensures that the last frequent itemsets that were removed

from the tree by a constraint now satisfy the new set without the last constraint. If these frequent itemsets are cached in a separate "backup" memory, they can be retrieved for when the constraint is removed. This mechanism would only store itemsets removed by the last generated constraint, so if two or more constraints are removed, not all itemsets will be retrieved.

The second policy uses a First In First out strategy by removing the oldest generated constraint. With this method, the set of constraints evolves with the stream of batches, enabling the samples to be different when the distributions of the data batches are similar.

The third policy is selecting a random constraint from the set to remove it.

Improving constraint generation and removal A possible improvement to avoid generating constraint that remove too many itemset, or removing an existing constraint that allows too many itemsets to come back in the FP-Tree is to count the number of itemsets impacted by the constraint, and choosing another constraint to generate/remove if the current one results in too many itemsets removed/retrieved in the FP-Tree.

4.3.4 Maintenance of the FP-Tree when removing constraints

After a constraint has been removed, the alphabet has been modified and the FP-Tree has to be maintained in a similar fashion to when a new constraint is generated. We continue from the FP-Tree after the constraint $D = C$ was generated, shown Figure 4.19. For any integer n, the results in a FP-Tree that satisfy $n+1$ generated constraints will satisfy any subset of n constraints. For this reason, when we reintroduce the letter D after removing the constraint $D = C$, all results in the FP-Tree are kept. In Figure 4.20, we first reintroduce the letter D to the nodes that contain itemsets that would not satisfy $D = C$. For instance, when $D = C$ was still a constraint used, the itemset AC would not be in the FP-Tree, thus, the node containing the itemset "labeled" AC really contains information about ACD (and same for $ACDE$).

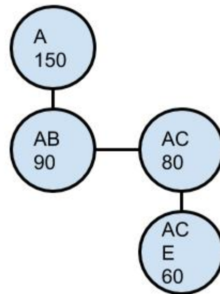

Figure 4.19: FP-Tree before new constraint is removed

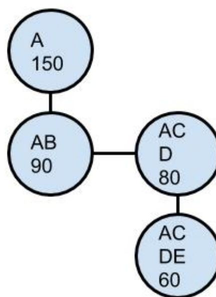

Figure 4.20: FP-Tree after the letter D has been reintroduced

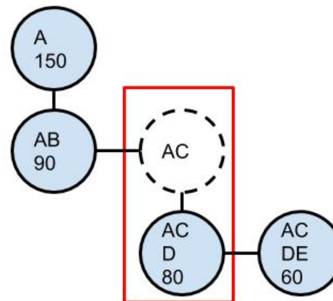

Figure 4.21: Creating a new node in the FP-Tree in the correct position to hold ACD

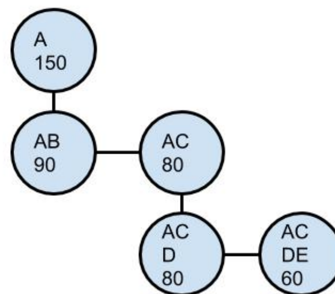

Figure 4.22: Estimating AC frequency

Case 1 If a node contains an itemset that does satisfy the constraint to be removed, it is left unchanged. This case is similar to the case 1 when generating a constraint. The itemsets A and AB fall in this category and remain unchanged.

Case 2 If a node contains an itemset that does not satisfy the constraint to be removed, it means it must hold the letter that was removed by the mentioned constraint, as we shown in Figure 4.20. The information about the itemset will be displaced somewhere else in the tree, to preserve the depth-first order. A new node has to be created to hold the current itemset, while the current node will hold the corresponding itemset that does not contain the letter reintroduced in the free alphabet. Figure 4.21 shows

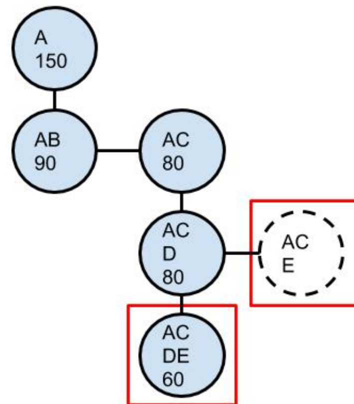

Figure 4.23: Creating a new node in the FP-Tree in the correct position to hold ACDE

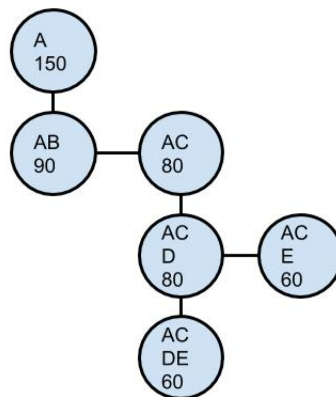

Figure 4.24: Estimating ACE frequency

the creation of the node containing AC, and the displacement of ACD. While it is possible to infer that the itemset AC is frequent, by its position in the FP-Tree, but it is also possible to estimate its frequency. Using the Apriori principle, it is possible to know the lower bound of the support of AC, by looking at all its children's supports. As shown in Figure 4.22, we can see that ACD has a support of 80, while $ACDE$ has a support of 60. This means the support of AC is at least 80. Since there is no further information in the tree, we use this bound as an estimate of the support.

In Figures 4.22 and 4.23 we can see that sometimes the case 2 results in creating a new node that is not a child of the current node. In the general case, the new node

is either a child of the current node, or a child of its sibling that contains the letter depending on if the last letter in the itemset is after the letter removed by the constraint. The parent of the created node has to contain the first parent of the itemset to displace in the lexicographic order. This requires a $O(n)$ search when creating a child of the current node, and two $O(n)$ searches in the other case, one to find the sibling containing the letter of the constraint removed, and another $O(n)$ to find where to create the new node. Once again, in an output sampling context, the number of result nodes is bounded, and so are the searches in the FP-Tree.

We can note that by adding (Figure 4.12 and 4.18), then removing (Figure 4.19 and 4.24) the same constraint ($C \oplus D = 0$) some information has been lost. The frequencies of AC and ACE in the FP-Tree are inferior to their real frequencies (prior to adding the constraint), and the information about ABD and $ABDE$ have been completely lost with the sampling. This loss of information is to be expected if stream features changes of density, of in the case of concept drifts. As the accelerator is running using a limited memory budget, it is not possible to retain a perfect memory from the past, causing the definitive loss of information with some sampling operations. In the case of concept drifts for example, the lost information should lose relevance once the sampler gets enough new batches, and the stored samples will converge back to a representative sample.

4.3.5 Algorithmic implementation of the streaming sampler

As in previous the chapter, section 3.1.3, the Algorithm 5 can be applied to software architectures, but was designed with hardware in mind, specifically FPGAs. The candidate sample is generated in constant time, and a pointer is added to keep track of a sample in the FP-Tree. This sample is, at all time, either the same sample as the candidate when it exists in the tree, or it would be parent/sibling.

The *"compute_next_itemset"* function shown in Algorithm 6 is adapted to both update the candidate sample from the enumeration tree, and the pointer pointing to the node to be used from the FP-Tree. The update of the candidate is explained in previous chapters, and detailed in section 3.1.3. The update to the pointer is detailed in

Algorithm 5 eclat_stream_pseudo_code

```
while not finished do
    compute_next_itemset(free_itemset, node_pointer)
    sample_candidate = compute_corresponding_sample(free_itemset)
    sample_node = compute_corresponding_node(node_pointer)
    count_in_DB(free_itemset, sample_candidate, sample_node)
    if is_frequent(sample) then
        update_results(sample_candidate, node_pointer, results)
    end if
    while number_of_results > threshold do
        create_constraint(XOR_matrix)
        update_free_alphabet_and_free_itemset(XOR_mat, free_itemset)
        prune_results_with_new_constraint(XOR_mat, results)
    end while
end while
```

Algorithm 6 compute_next_itemset

```
if is_leaf(itemset) then
    suffix = leading_bit(prefix)+1
    prefix = prefix - leading_bit(prefix)
    pointer = parent(pointer)
else
    if is_frequent(itemset) then
        prefix = prefix & suffix
        suffix = suffix +1
    else
        prefix = prefix
        suffix = suffix +1
    end if
end if
if sibling(pointer) == prefix & suffix then
    pointer = sibling(pointer)
end if
if child(pointer) == prefix & suffix then
    pointer = child(pointer)
end if
```

Algorithm 7 update_results

```
if pointed_node == candidate then
  pointed_count += candidate_count
else
  if is_parent(pointed_node, candidate) then
    create_first_child(pointed_node, candidate, candidate_count)
  end if
  if is_sibling(pointed_node, candidate) then
    create_first_sibling(pointed_node, candidate, candidate_count)
  end if
end if
```

Algorithm 8 count_in_DB

```
count = 0
for all transaction ∈ Database do
  if itemset ∈ transaction then
    count = count +1
  end if
end for
if sample_node != NULL then
  count = count + node_count
end if
```

the section 4.3.1 of this chapter. The proposed implementation of the function is rather simple compared to the explication detailing all the steps in the example run.

As we make sure the pointer is either pointing at the node containing the candidate itemset, or one link away from its "would be position" if it is absent from the FP-Tree, the node containing the next candidate, or its "would be position", is either the first child, the first sibling, or the sibling of the parent of the current node. For the "sibling of the parent" case, we first update to pointer to the parent (that is necessarily frequent, thus in the FP-Tree), and then all cases can be grouped by checking both the first child and sibling at the same time, the only two positions that could contain the next candidate.

The function *"count_in_DB"* shown in Algorithm 8 too is altered to take in consideration the previous support of the candidate stored in the FP-Tree when checking if a candidate is frequent.

The function *"update_results"* do no longer add the candidate itemset to the end of a chained list. It either updates the support of the node pointed at, or add a node in the FP-Tree as its child or sibling, as it was described previously in the current section 4.3.1. Most of the implementation considerations of the function are to create a node in the tree while keeping the links between the nodes (checking if the node already has a child or sibling for example).

The Algorithm 5 only processes one batch of data. Right before processing a batch, decay will be applied to the samples in the FP-Tree, and infrequent samples are removed. If the number of samples falls under the low threshold, constraints are removed and the FP-Tree is maintained until the threshold is satisfied, as seen in the previous subsection.

4.4 Experiments on Pattern Quality

4.4.1 Synthetic Experiments

We first conduct experiments using synthetic datasets in order to evaluate the capacity of our stream mining algorithm to recovery patterns explicitly planted in the data. These

datasets use specific type of probabilistic densities to measure how well the streaming algorithm can keep track of itemsets in the stream compared to an itemset mining algorithm not tailored for streams. For the datasets, a particular itemset follows the specific probabilistic density, while the other items in the alphabet use a static probability, to behave as background noise that carries no particular information. Synthetic datasets comprise the three following models:

- *Slope model*: Rising probability of itemset, until it reaches a maximum (Figure 4.25).

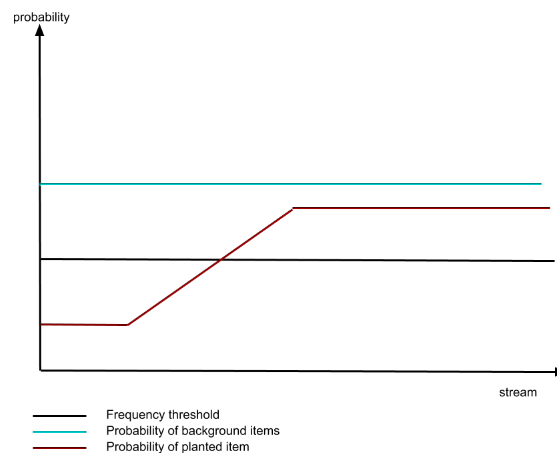

Figure 4.25: Steady rise followed by constant value

- *High probability model*: High probability of an itemset, then lower than threshold (Figure 4.26).
- *Sinusoidal probability model* with decreasing amplitude (Figure 4.27).

Experimental protocol Every dataset of the experiments is a finite stream of batches divided in windows. For a fair comparison, our approach is compared to the non-streaming algorithm with similar budget constraints. Our streaming algorithm is also tested against a ground truth, containing the results of an itemset mining on the entire stream. This ground truth is too costly to compute on real datasets but smaller streams used in this experiments can still be processed. This ground truth provides the upper bound of "patterns quality" for the streaming and non-streaming algorithms. The parameters used are:

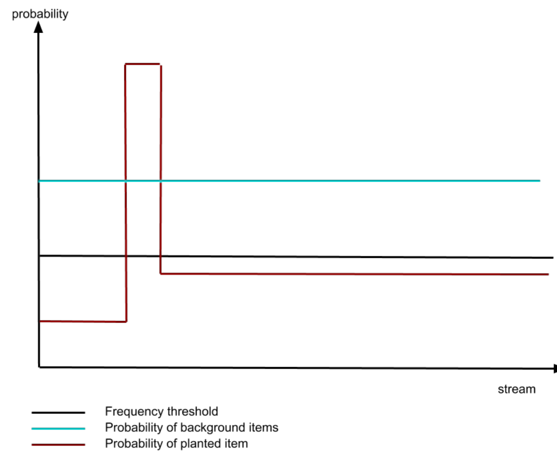

Figure 4.26: Pulse followed by constant value

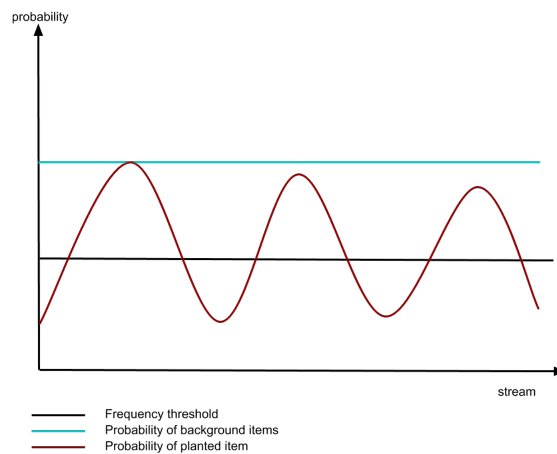

Figure 4.27: Sinusoidal probability

- Size of batches: A batch is a set of new transactions arriving at the same time from the stream. An itemset mining algorithm working on a stream may use incoming batches instead of the whole stream to save computation time. The size of the batches is an important parameter. Very small batches (e.g. 1 transaction per batch) result in calling the stream mining algorithm everytime new data is available, which requires a high throughput. Large batches do not allow to dissociate data at the beginning of a batch from data at the end, thus inducing a loss of temporal precision.

- Decay rate: The decay rate is a parameter proper to streaming algorithms. The

importance of previous data in the stream is reduced everytime a new batch arrives. This method provides results in a damped window. A damped window provides more temporal information on the data (mainly, the older itemsets get their frequencies reduced), but requires less memory and computation than a rigid sliding window.

- mMximum number of samples: To the best of our knowledge, this thesis is the first instance exploring output space sampling of frequent patterns on a streaming database. The maximum number of samples allowed for the output is a parameter that will dictate how aggressive the sampling will be, and potentially a loss in the precision of the results.

Quantitative measures In order to test the quality of our pattern mining algorithm against an algorithm that cannot process streaming data, we use quantitative metrics of *recoverability* and *spuriousness* proposed by [18] for approximate frequent pattern mining algorithms. Each metric compares an approximate mining algorithm against a perfect algorithm. In our case, we measure the performance of two approximate algorithms.

First we evaluate the performance of our streaming algorithm, with regards to a frequent itemset mining algorithm that processes the entire stream at once. Then we evaluate the performance of a frequent itemset mining algorithm that cannot use streaming technique, with regards to a frequent itemset mining algorithm that processes the entire stream at once.

The frequent itemset mining algorithm that processes the entire stream at once serves as the reference perfect itemset mining algorithm in both cases, and is highly impractical, if not impossible, to use in practice, compared to the two approximate algorithms evaluated.

The *recoverability* metric we use measures how well an approximate mining algorithm retrieves the patterns found by the perfect mining algorithm. This metric is similar

to recall. For a base itemset B_i found by the perfect algorithm, its recoverability is

$$R(B_i) = \frac{|B_i \cap F^i_{max}|}{|B_i|},$$

where F^i_{max} is the itemset found with the maximum number of items in common with B_i, and the $|.|$ operator returns the number of items of an itemset.

Spuriousness measures how much an approximate mining algorithm found itemsets that were not found by the perfect mining algorithm. This metric is similar to the error rate, and is linked to precision ($precision = 1 - spuriousness$). In the experiments of this section, the value of this metric will stay at a constant value of 0 because of the setup, and will not be displayed for that reason.

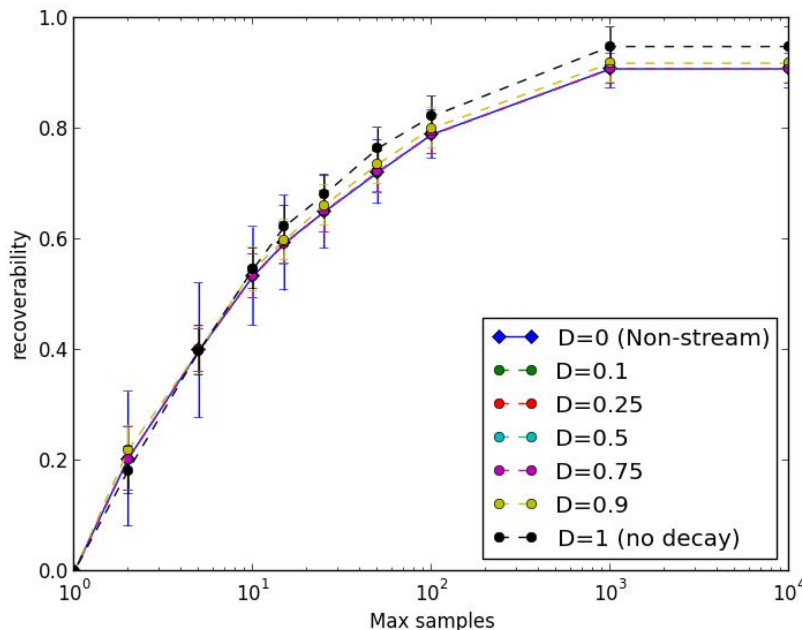

Figure 4.28: Recoverability of stream mining on random data

Figures 4.28, 4.29, 4.30 and 4.31 show the quality of patterns returned by our streaming algorithm with different decay values against the non-streaming algorithm on synthetic, bounded streams. In all these experiments, the itemset algorithms working on a stream return pattern found with a support higher than 1%, as does the ground truth. For this reason, the ground truth lists all patterns with a support of 1%, but the

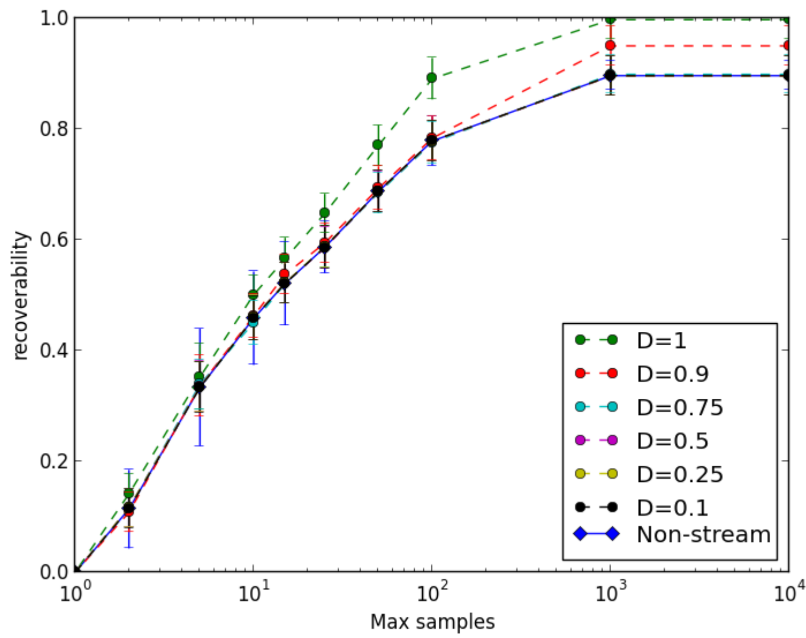

Figure 4.29: Recoverability of stream mining on slope

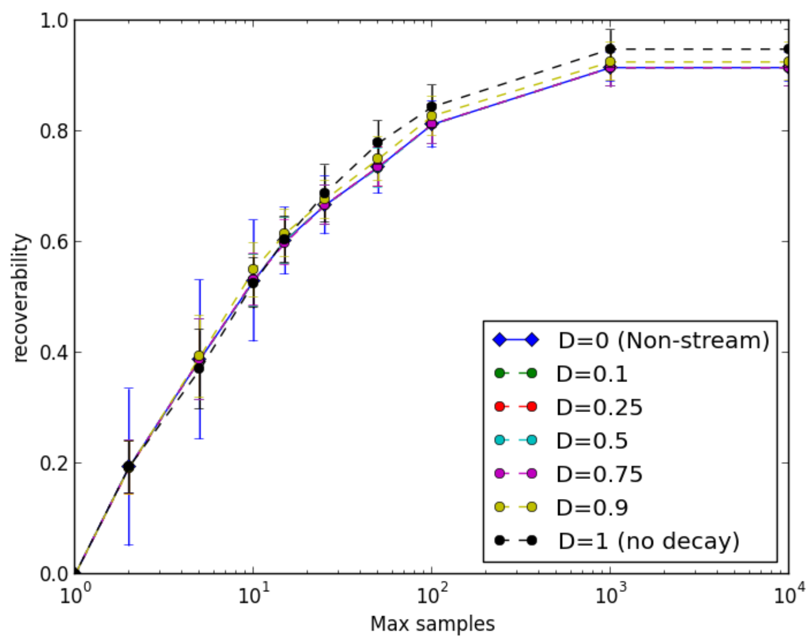

Figure 4.30: Recoverability of stream mining on pulse followed by constant

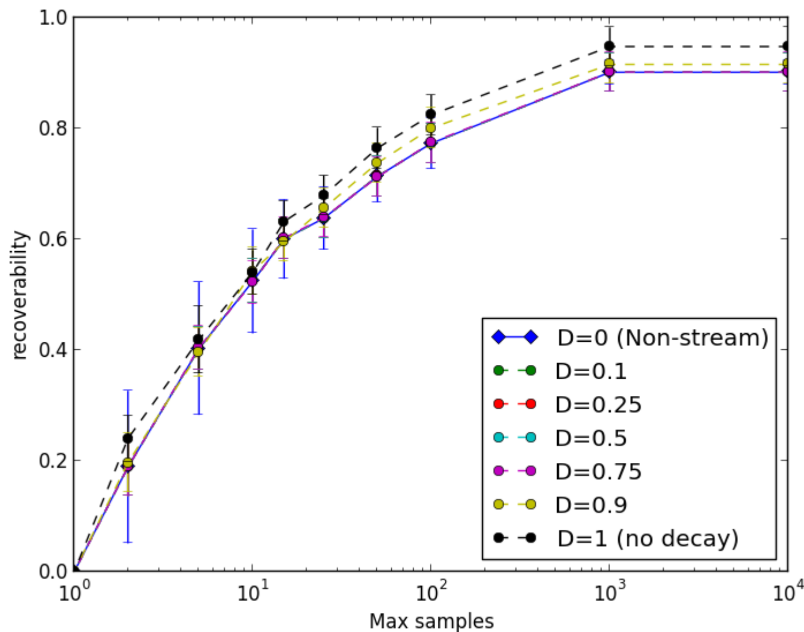

Figure 4.31: Recoverability of stream mining on sine

pattern mining algorithm can only have access to one batch at a time and the patterns they stored in memory. For every run, the pattern mining algorithm will find a subset of the results found by the ground truth.

The Y axis indicates the recoverability of the patterns, and how well they represent the patterns of the ground truth. Note that this metric only indicates if the pattern mining algorithm found patterns present in the ground truth, without considering the support values. The X axis of the figures represents the maximum number of patterns the algorithms can return, in log scale. This parameter is fed to the sampling module of the algorithm that dictates how many XOR constraints must be generated during the run.

The parameter D is linked to the exponential decay used by the stream mining algorithm. After a batch of data is processed, the support of patterns in memory are multiplied by the value of D. The value $D = 1$ means no decay is used and the support of patterns is kept intact between batches. As the non-stream algorithm has no information about data in the stream except for the last batch, it is functionally equivalent to the streaming algorithm with $D = 0$, where patterns from previous batches in memory

are systematically discarded.

As the pattern mining algorithm will always return a subset of the ground truth, it will never return additional patterns that are not present in the ground truth. For this reason, the spuriousness of the experiments is always zero, which is why it is not plotted.

Several observations can be made from these experiments. Firstly, all individual curve are monotonic: allowing to return more pattern results in a higher recoverability, as the patterns returned using $K+1$ constraints are a subset of the patterns returned using K constraints.

On average, increasing the exponential decay (reducing the value of D) decreases the recoverability of the results, and its quality. It is possible for the algorithm to perform better with a lower value of D for a given maximum number of allowed patterns. This is a consequence of the dynamic sampling, where it is possible that retaining only few more patterns in memory can make the algorithm exceed the maximum number of allowed patterns, leading to generating a new constraint, dramatically reducing the number of results, and reducing the recoverability.

4.4.2 Experiments on a Semi-Real Dataset

To test the performance of the stream sampling on more realistic data, we rely on the Instacart [19] dataset. Instacart is a retail dataset available on Kaggle, which provides in total more than 3 million transactions from 200 000 users. For each user, a sequence of its transactions is provided. There is no timestamp so it is not possible to determine when a transaction happened, but the number of days between two consecutive transactions is given. Due to this absence of timestamp, we cannot reconstruct the exact stream of purchases on the Instacart platform. However, because the data contains actual user transactions, it exhibits realistic behaviors and can thus be an interesting setup to test our stream sampling approach.

We thus decided to build a "semi-real" dataset from Instacart, by assigning ourselves timestamps to the first transaction of each user. The timestamp of the next transactions can then be determined thanks to the provided number of day between two consecutive transactions. In practice:

- We consider 632 users among all Instacart users, corresponding to 10 000 orders.
- For the user having its purchases history distributed across the longest period of time, the starting time (timestamp of first transaction) is set at 0. This guarantees to have a stream with a high enough density of transactions from the beginning of the stream.
- For all the other users, the starting time is given a uniform random value to fit inside that period. This avoids to have a too high density of transaction for the first few timestamps followed by a brutal drop in the number of transactions.
- The total 10000 baskets are merged by date, so the purchases of all users made on the same day are in the same transaction. The resulting stream is made of 370 transactions. These transactions are divided in 10 consecutive batches, each batch containing 37 transactions. From a performance experiment point of view this approach allows to have data with a higher density, from a retail point of view one could imagine that we are not interested in individual user purchases, but on the products sold generally by Instacart on a single day.
- There are in average 74.1 items per merged transaction.

This "semi-real" dataset cannot be analyzed to get business insights about Instacart data over time: due to our arbitrary allocation of timestamps, transactions that happen in summer may be found in the same batch as transactions that appear in winter, giving unrealistic frequencies for most itemsets (especially seasonal products like fruits, that are the most important products in Instacart). On the other hand, the distribution of products in the transactions is more realistic than in a purely synthetic dataset.

We will mostly use this semi-real Instacart dataset for performance experiments in the next section, showing that our approach scales with more realistic data. We will also show that how our approach allows to observe how itemsets frequencies can change during time, even if such results have to be taken with a grain of salt w.r.t. their business validity.

Figure 4.32 shows the evolution of the number of frequent itemsets and constraints generated during the stream. For this experiment, we set the threshold to generate new constraints at 100 frequent itemsets. At the beginning of the stream, the number

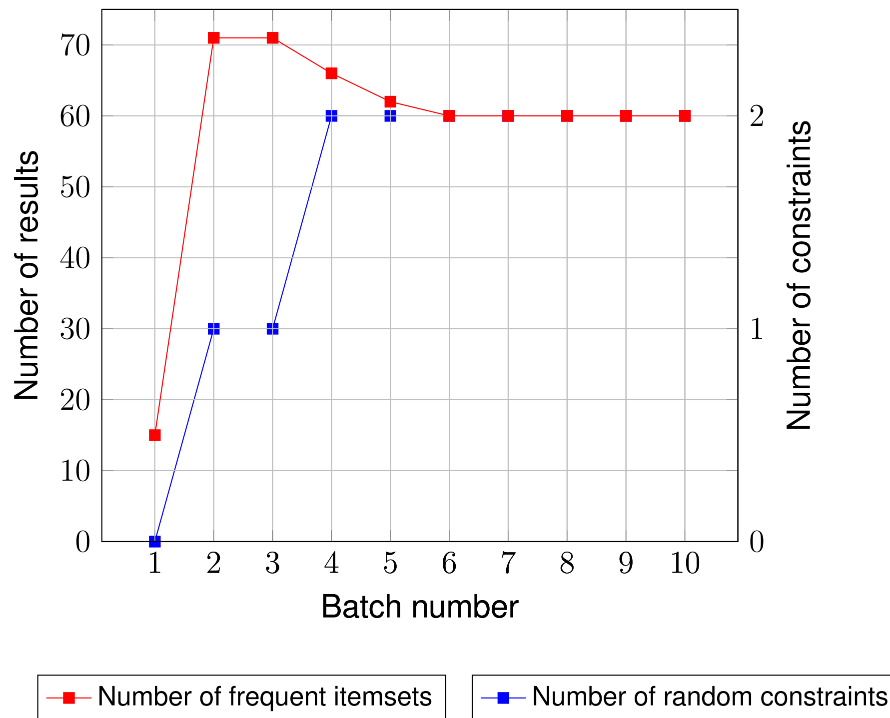

Figure 4.32: Evolution of number of results and constraints

of frequent itemsets is rather low with 15 results found, but rises quickly when new batches arrives. During the second batch, a constraint is generated, meaning a frequent itemset mining algorithm would have found around 140 frequent itemsets when the sampler reduced it to 71. The numbers of itemsets and constraints remain stable for the next batch, but a new constraint is generated during the fourth batch, bringing the number of results to 66. From there, the number of constraints remains at 2 while the number of results reduces slightly during the rest of the stream.

The number of frequent itemsets being so low at the beginning of the stream is likely caused by the way the semi-real dataset was generated. The randomized offset for the user purchases might have grouped the bulk of the user purchases in the middle of the stream, instead of at the beginning with no offset. It is very likely the number of frequent itemset remains stable and does not drop too dramatically because of the decay rate value. With a decay rate value of 0.95%, frequent itemsets from the middle of the stream are likely to be kept in the results while there are less itemsets in the new batches at the end of the stream.

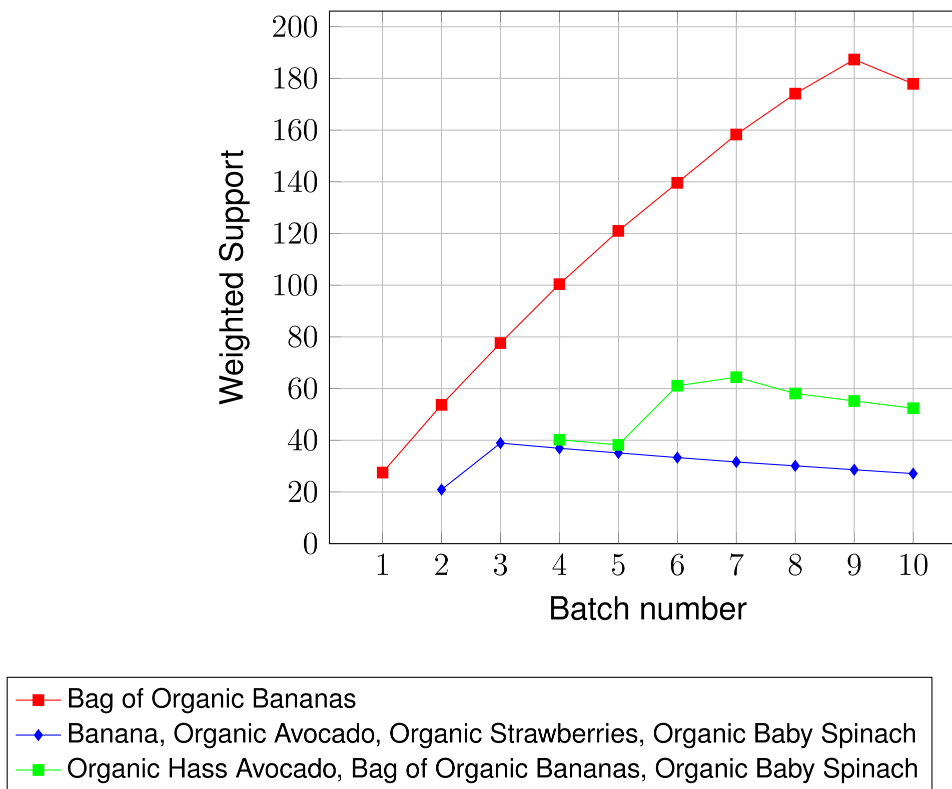

Figure 4.33: Some frequent patterns of the stream

Figure 4.33 shows the evolution of the support for three different itemsets in the stream with a decay rate of 0.95% and support threshold of 20. With this decay rate, the weighted support of itemsets is close to what it would be in the case of a stream mining with a landmark window. The three itemsets we chose to display are "Bag of Organic Bananas", "Banana, Organic Avocado, Organic Strawberries, Organic Baby Spinach", and "Organic Hass Avocado, Bag of Organic Bananas, Organic Baby Spinach", as they show three different expected behaviours.

Because "Bag of Organic Bananas" consists of a single item that is present in a lot of purchases, its weighted support increases with every batch expect for the last one.

The itemset "Banana, Organic Avocado, Organic Strawberries, Organic Baby Spinach" is an itemset with a rather larger number of items, and appeared only in the first few batches of the stream. After these batches, the weighted support only decreases with the exponential decay, but is still considered "frequent" at the end of the experiment.

For the itemset "Organic Hass Avocado, Bag of Organic Bananas, Organic Baby Spinach", we can see it first appeared in the fourth batch, did not appear a significant amount of time in the fifth batch, and reappears in the sixth batch.

For the first two presented itemsets, we can see the issue of a high decay rate value, in one case because the weighted support keeps growing and can dwarf other legitimately frequent itemsets, and in the other case, patterns that no longer appear take a long time to be evicted. Inversely, a low decay rate value would greatly alter the weighted support of the itemsets "Banana, Organic Avocado, Organic Strawberries, Organic Baby Spinach" and "Organic Hass Avocado, Bag of Organic Bananas, Organic Baby Spinach". Because the weighted support of these two itemsets is rather close to the threshold, they could be considered unfrequent before another batch containing the itemsets appears. For example, a decay rate lower than 0.7 would results in a weighted support of $38 * 0.7 * 0.7 = 18.62 < 20$ when the sixth batch arrives.

This illustrates the trade-off that needs to be made when choosing the value of the decay rate, that we currently consider as a user parameter. A high value will create

high retention of previous itemsets, and a low value will prune frequent itemsets that are not frequent enough when they first appear.

In practice, we took measures to not evict unfrequent itemsets from the results until the batch is processed in case the current batch contains enough instances of the itemsets to keep it in memory, to help mitigate the early pruning of new itemsets. This can be seen when looking at the weighted support of "Banana, Organic Avocado, Organic Strawberries, Organic Baby Spinach", starting with a weighted support of 21, its weighted support should be $21 \times 0.95 = 19.95 < 20$ at the beginning of batch 3. The itemset was kept in case it would become frequent again with the information in the current batch.

4.5 Experiments on Hardware Acceleration

Figure 4.34 shows the key differences with the non-stream architecture:

- The data structure holding the results in external memory (DDR) has been overhauled, the linked list has been replaced with the FP-Tree structure.
- The support counting module writes and now also reads the external memory to get information from the FP-Tree for each time it scans the database.
- Instead of having only a pointer at the end of the list of the results like in the non-stream version, we maintain a pointer in the FP-Tree where the currently tested itemset is or would be placed in the FP-Tree at all time. This increases slightly the number of operations to be done at each candidate itemset generation, but can still be realized in O(1).

Hardware cost of the accelerator Figure 4.35 shows the resources required for the accelerator to support Stream Sampling with different alphabet sizes. A quick comparison with Figure 2.11 in previous chapters shows that the accelerator with an alphabet of size 2048 requires an amount of BRAM banks very similar to the versions of the accelerator without support for streaming databases (around 90% of the available memory). The amount of required Flip-Flop and LUT slices significantly increases, due to the

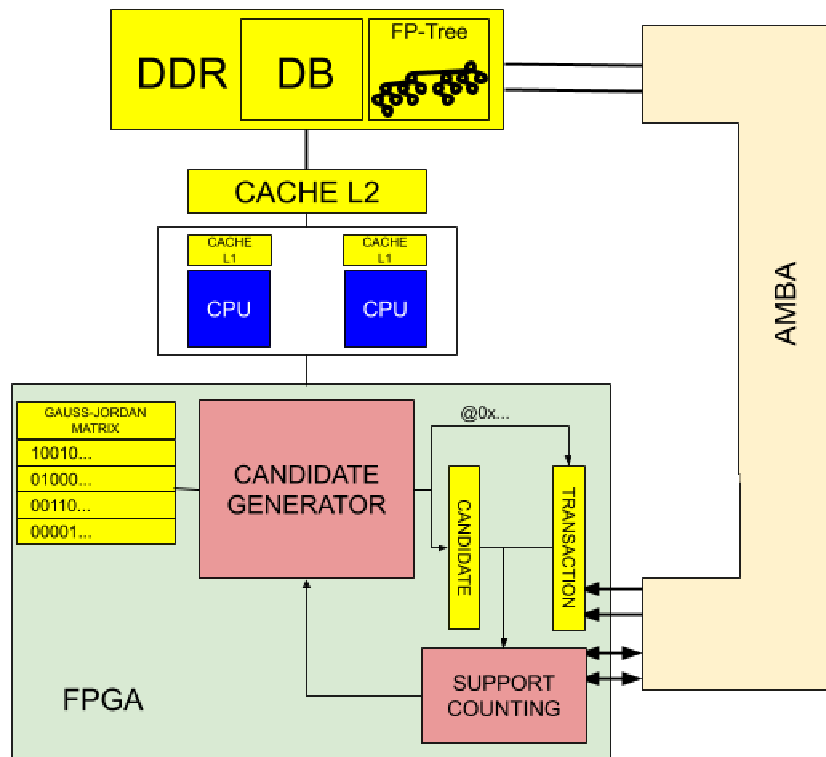

Figure 4.34: Overview of the accelerator architecture with stream support

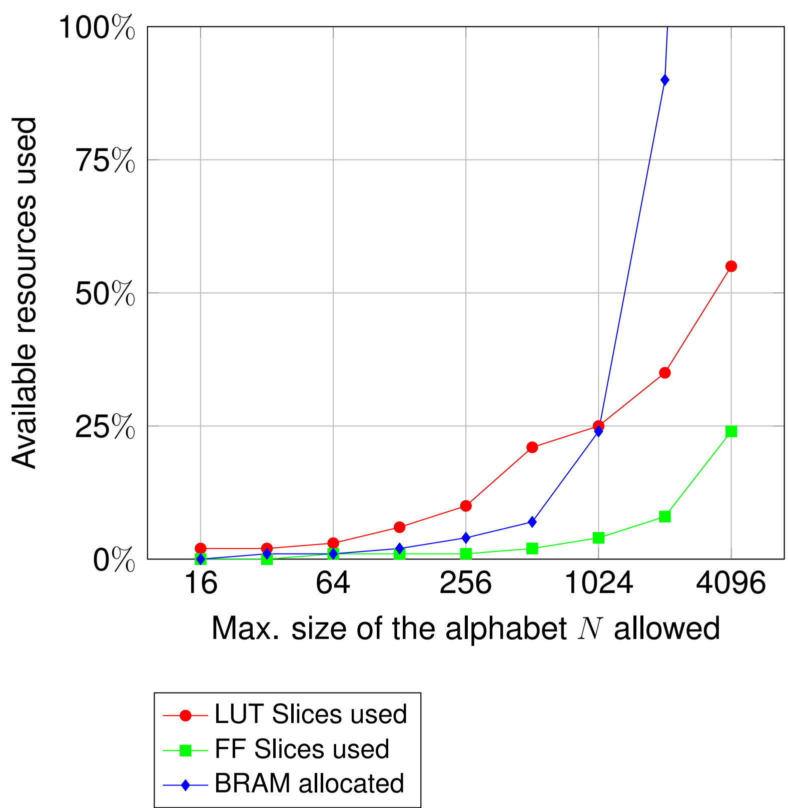

Figure 4.35: Impact of alphabet size on allocated resources

many features such as the FP-Tree management, the batches management, and the fixed point operations on weighted supports. The memory allocation still remains the bottleneck, and there are still more than half of the LUT slices left unused for alphabets of size 2048.

Performance on semi-real dataset The semi-real dataset generated with Instacart [19] we described in the previous section is used to compare the performance of our hardware accelerated stream sampler against its software implementation. The software results were produced using an Intel Xeon CPU running at 1.90GHz. Due to material constraints and a lack of time, the accelerator was not run on a physical FPGA at the time of writing. However using the timing information of the different parts of the accelerator generated by the synthesis tool, we managed to produce an estimation of the number of cycles needed to make the same computations. This number of cycles can be converted to an execution time by assuming the accelerator would run on an FPGA clocked at 100MHz.

We present the execution times in Table 4.3, where we can see the hardware accelerator should be roughly ten times faster than the CPU version given our estimations. These results are promising, and while the software implementation of our stream sampler is not optimised for CPU architectures, we tried to produce a conservative estimation for the hardware timing that would not be over-optimistic. The estimated number of cycle was produced by counting the number of calls of key functions during a run of the software implementation of the algorithm, and with upper bounds when precise counts were not feasible, implying a run using a physical FPGA should provide better execution time. These results could then be compared against a better software counterpart for a fairer comparison, such as a version of EFlexics with adjustments to handle results from previous batches.

Table 4.3: Execution time

	Software version	FPGA accelerator version (estimated)
Instacart dataset	3.48 sec.	0.326 sec.

Conclusion

This thesis primary focused on output space sampling for pattern mining, and its hardware acceleration. We first introduced an FPGA design for regular frequent itemset mining with several performance optimizations. For this design, we proposed a solution that is compatible with output space sampling, and also reduces the memory allocated on the FPGA. For the largest size of frequent itemset mining accelerators that could fit on a ZC706 board, the amount of memory allocated was reduced by 20%, at the cost of more expandable resources available on the FPGA.

With this improved design, we also proposed a simple dataset reduction technique to reduce the amount of data read from the database, we called *correspondence list*. While this particular dataset reduction technique is costly in memory, a rather precious resource for this type of application, it is not exclusive to the design we propose and can be applied to many depth-first pattern mining algorithms. The use of this *correspondence list* yields high speedups when the data being processed has low density. Our experiments showed that very small amount of allocated memory can allow for a speedup of 200%, but quickly reaching a plateau if larger quantities of memory are used.

The design is later built upon to allow new features, most importantly output space sampling. Our work differs from solutions in the literature by adjusting the sampling dynamically, as new itemsets are found, instead of relying on a static estimation. Even with the addition of this feature, it is possible to accelerate it on very small FPGA boards such as a Zybo Z7. When compared to its software state of the art counterpart running on a server class Xeon CPU clocked at 3.2 GHz, our accelerator of frequent itemset sampling can reach performance of one order of magnitude higher using only a small FPGA clocked at 100MHz.

We then proposed to extend the scope of output space sampling to streaming databases. Our contribution is an accelerator architecture tailored and optimised for FPGA, but

would also allow for a software implementation, albeit it might not grant the best performance on a CPU. As more emphasis is given to the most recent data in the stream, the solution we introduced relies on an ad-hoc data structure that can be efficiently traversed to find previous results. Our solution adapts the output space sampling as the density of the data stream might evolve, something absent from other output space samplers in the literature.

Several perspectives are available to improve the work presented in this thesis. A first obvious perspective is to go beyond theoretical works for the parallel implementation of the sampler, by proposing a stable implementation and doing performance experiments. The work stealing method used could also be improved, as it currently tries to steal from processing units with the (estimated) highest amount of work. This is a simple and intuitive solution, but it results in processing units working on itemsets scattered all across the enumeration tree; this behaviour might increase the amount of data transfers when used with tools such as the correspondence list, since the units can access completely different parts of the database.

There are also other perspectives that would require a more in-depth work on the proposed implementation. Firstly, an improvement that the accelerator could greatly benefit from is the use of projected databases. Put simply, this method can allow to process an itemset mining problem with a large alphabet by dividing it on multiple, smaller itemset mining problems, with smaller "projected" alphabets. Since the sampler we proposed can adapt easily to different sizes of alphabet, and the alphabet size is directly linked to the allocation of resources on the FPGA, this could be a great solution to schedule problems with large alphabets on smaller FPGA. At the moment, it is technically possible to use a CPU to schedule the database projection, and call the FPGA to accelerate each data mining problem. There might however be solutions to process this scheduling directly on the FPGA, allowing for the accelerator to process virtually any database on its own by using database projections until the alphabet size is small enough.

The specifics of the proposed implementation relies on a simple representation of the itemsets with presence bit-vectors. This allows for the accelerator to access all information on an itemset at once, but increases the amount of logic needed and the

amount of data transferred when reading the database. While the database projection technique described above can be used to reduce the sizes of alphabets used by the accelerator, the high amount of redundancy in the itemsets fetched from memory and stored in the results (as most itemsets contain a limited number of items) could be reduced by compressing the bit-vectors. This feature could be implemented with a simple interface between the proper accelerator, working on bit-vectors, and the external memory, storing the results and the external database, containing compressed itemsets.

As a more algorithmic perspective, it would be possible to enhance the quality of the output space sampling for streaming databases. The solution we proposed gives asymptotic guarantees after the sampler detects the number of frequent itemsets is too low. In practice, when a constraint is removed, the sampler is only able to estimate parts of the itemsets evicted by its generation. For this reason, the itemsets found by the sampler will not be representative until new itemsets are found. An interesting perspective is thus to improve the constraint removal process to have stricter guarantees on the number of itemsets output at any time, without sacrificing too much of the efficiency of the current approach.

Furthermore, the damped window model we used requires parameter tuning, a task we did not focus too much on. A rather straightforward perspective would be a more in-depth analysis of the quality of patterns with the damped window model against the landmark window and sliding widow models.

A last and higher level perspective is the use of the patterns produced. Our work shows that it is possible to have, for high throughput streams, representative samples of itemsets present in the stream at any point in time. It can also be done relatively cheaply thanks to dedicated hardware. So the question now is to have applications that rely on these samples for some task.

A challenging application would be to analyze on the fly, inside a CPU or an MPSoC, the frequent memory access patterns, in order to relocate tasks or data to optimal positions (e.g., put data closer to tasks that often use them). First proofs of concept in that direction has been provided in the PhD of Sofiane Lagraa [20], but this was done through the offline analysis of execution traces. With our work, such analysis

can now be done online: the next challenge is to use efficiently the extracted itemsets (which here represent memory access patterns) to make relevant data/task relocation decisions.

BIBLIOGRAPHY

[1] Y. Zhang, F. Zhang, Z. Jin, and J. D. Bakos, “An fpga-based accelerator for frequent itemset mining”, *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 6, *1*, 2:1–2:17, May 2013, ISSN: 1936-7406.

[2] V. Dzyuba, M. van Leeuwen, and L. De Raedt, “Flexible constrained sampling with guarantees for pattern mining”, *Data Mining and Knowledge Discovery*, Mar. 2017. (visited on 06/15/2017).

[3] M. J. Zaki *et al.*, “New algorithms for fast discovery of association rules.”, in *Proc. ACM Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, vol. 97, 1997, pp. 283–286.

[4] J. Han, J. Pei, Y. Yin, and R. Mao, “Mining frequent patterns without candidate generation: A frequent-pattern tree approach”, *Data mining and knowledge discovery*, vol. 8, *1*, pp. 53–87, 2004.

[5] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases”, in *Proceedings of 20th International Conference on Very Large Data Bases (VLDB)*, 1994, pp. 487–499.

[6] V. T. Chakaravarthy, V. Pandit, and Y. Sabharwal, “Analysis of sampling techniques for association rule mining”, in *Proceedings of the 12th international conference on database theory*, ACM, 2009, pp. 276–283.

[7] H. Toivonen *et al.*, “Sampling large databases for association rules”, in *VLDB*, vol. 96, 1996, pp. 134–145.

[8] M. Boley, C. Lucchese, D. Paurat, and T. Gärtner, “Direct Local Pattern Sampling by Efficient Two-step Random Procedures”, in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’11, New York, NY, USA: ACM, 2011, pp. 582–590, ISBN: 978-1-4503-0813-7. DOI: 10.1145/2020408.2020500. [Online]. Available: http://doi.acm.org/10.1145/2020408.2020500 (visited on 06/15/2017).

[9] M. Al Hasan and M. J. Zaki, “Output Space Sampling for Graph Patterns”, *Proc. VLDB Endow.*, vol. 2, *1*, pp. 730–741, Aug. 2009.

[10] M. Boley, S. Moens, and T. Gärtner, “Linear Space Direct Pattern Sampling Using Coupling from the Past”, in *Proc. 18th ACM Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2012, pp. 69–77, ISBN: 978-1-4503-1462-6. (visited on 06/15/2017).

[11] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, “Distribution-Aware Sampling and Weighted Model Counting for SAT”, *arXiv:1404.2984*, Apr. 2014.

[12] A. Prost-Boucle, F. Pétrot, V. Leroy, and H. Alemdar, “Efficient and versatile fpga acceleration of support counting for stream mining of sequences and frequent itemsets”, *ACM Trans. on Reconfigurable Technol. and Syst. (TRETS)*, vol. 10, *3*, p. 21, 2017.

[13] S. Shi, Y. Qi, and Q. Wang, “Accelerating intersection computation in frequent itemset mining with fpga”, in *IEEE 10th Int. Conf. on High Perf. Comp. and Comm. (HPCC)*, 2013, pp. 659–665.

[14] J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation”, in *ACM Sigmod Record*, ACM, vol. 29, 2000, pp. 1–12.

[15] D. Dheeru and E. Karra Taniskidou, *UCI machine learning repository*, 2017. [Online]. Available: `http://archive.ics.uci.edu/ml`.

[16] T. Guns, S. Nijssen, and L. De Raedt, “Itemset mining: A constraint programming perspective”, *Artificial Intelligence*, vol. 175, *12-13*, pp. 1951–1983, 2011.

[17] R. Agrawal, R. Srikant, *et al.*, “Fast algorithms for mining association rules”, in *Proceedings of the 20th International Conference on Very Large Data Bases*, vol. 1215, 1994, pp. 487–499.

[18] R. Gupta, G. Fang, B. Field, M. Steinbach, and V. Kumar, “Quantitative evaluation of approximate frequent pattern mining algorithms”, in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2008, pp. 301–309.

[19] Instacart, *Instacart market basket analysis*, [Online; accessed 21-July-2020], 2017. [Online]. Available: `https://www.kaggle.com/c/instacart-market-basket-analysis/data`.

[20] S. Lagraa, F. Pétrot, and A. Termier, “Nouveaux outils de profilage de mpsoc basés sur des techniques de fouille de données”, PhD thesis, University of Grenoble, Jun. 2014.

[21] S. Moens, M. Boley, and B. Goethals, “Providing Concise Database Covers Instantly by Recursive Tile Sampling”, en, in *Discovery Science*, Springer, Cham, Oct. 2014, pp. 216–227. DOI: 10.1007/978-3-319-11812-3_19. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-11812-3_19 (visited on 06/15/2017).

[22] S. Moens and B. Goethals, “Randomly Sampling Maximal Itemsets”, in *Proceedings of the ACM SIGKDD Workshop on Interactive Data Exploration and Analytics*, ser. IDEA ’13, New York, NY, USA: ACM, 2013, pp. 79–86, ISBN: 978-1-4503-2329-1. DOI: 10.1145/2501511.2501523. [Online]. Available: http://doi.acm.org/10.1145/2501511.2501523 (visited on 06/15/2017).

[23] M. Bhuiyan, S. Mukhopadhyay, and M. A. Hasan, “Interactive Pattern Mining on Hidden Data: A Sampling-based Solution”, in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, ser. CIKM ’12, New York, NY, USA: ACM, 2012, pp. 95–104, ISBN: 978-1-4503-1156-4. DOI: 10.1145/2396761.2396777. [Online]. Available: http://doi.acm.org/10.1145/2396761.2396777 (visited on 06/15/2017).

[24] V. Dzyuba and M. v. Leeuwen, “Learning What Matters – Sampling Interesting Patterns”, en, in *Advances in Knowledge Discovery and Data Mining*, Springer, Cham, May 2017, pp. 534–546. DOI: 10.1007/978-3-319-57454-7_42. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-57454-7_42 (visited on 06/15/2017).

[25] S. Jabbour, L. Sais, and Y. Salhi, “Boolean Satisfiability for Sequence Mining”, in *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, ser. CIKM ’13, New York, NY, USA: ACM, 2013, pp. 649–658, ISBN: 978-1-4503-2263-8. DOI: 10.1145/2505515.2505577. [Online]. Available: http://doi.acm.org/10.1145/2505515.2505577 (visited on 07/12/2017).

[26] C. P. Gomes, A. Sabharwal, and B. Selman, “Near-uniform Sampling of Combinatorial Spaces Using XOR Constraints”, in *Proceedings of the 19th International Conference on Neural Information Processing Systems*, ser. NIPS’06,

Cambridge, MA, USA: MIT Press, 2006, pp. 481–488. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2976456.2976517` (visited on 07/17/2017).

[27] *Enhanced Gaussian Elimination in DPLL-based SAT Solvers - PoS10-Soos.pdf*. [Online]. Available: `https://www.msoos.org/wordpress/wp-content/uploads/2010/08/PoS10-Soos.pdf` (visited on 07/20/2017).

[28] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo, *et al.*, "Fast discovery of association rules.", *Advances in knowledge discovery and data mining*, vol. 12, *1*, pp. 307–328, 1996.

[29] T. Uno, M. Kiyomi, and H. Arimura, "Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets", in *Fimi*, vol. 126, 2004.

[30] F. Zhang, Y. Zhang, and J. D. Bakos, "Accelerating frequent itemset mining on graphics processing units", *The Journal of Supercomputing*, vol. 66, *1*, pp. 94–117, 2013.

[31] J. Zhou, K.-M. Yu, and B.-C. Wu, "Parallel frequent patterns mining algorithm on gpu", in *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, IEEE, 2010, pp. 435–440.

[32] C. Silvestri and S. Orlando, "Gpudci: Exploiting gpus in frequent itemset mining", in *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, IEEE, 2012, pp. 416–425.

[33] G. Teodoro, N. Mariano, W. Meira Jr, and R. Ferreira, "Tree projection-based frequent itemset mining on multicore cpus and gpus", in *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, IEEE, 2010, pp. 47–54.

[34] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent itemset mining on graphics processors", in *Proceedings of the fifth international workshop on data management on new hardware*, ACM, 2009, pp. 34–42.

[35] F. Zhang, Y. Zhang, and J. Bakos, "Gpapriori: Gpu-accelerated frequent itemset mining", in *2011 IEEE International Conference on Cluster Computing*, IEEE, 2011, pp. 590–594.

[36] Z. K. Baker and V. K. Prasanna, "Efficient hardware data mining with the apriori algorithm on fpgas", in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, IEEE, 2005, pp. 3–12.

[37] D. W. Thoni and A. Strey, “Novel strategies for hardware acceleration of frequent itemset mining with the apriori algorithm”, in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, IEEE, 2009, pp. 489–492.

[38] Z. K. Baker and V. K. Prasanna, “An architecture for efficient hardware data mining using reconfigurable computing systems”, in *Field-Programmable Custom Computing Machines, 2006. FCCM’06. 14th Annual IEEE Symposium on*, IEEE, 2006, pp. 67–75.

[39] P. Sampath, C. Ramesh, T. Kalaiyarasi, S. S. Banu, and G. A. Selvan, “An efficient weighted rule mining for web logs using systolic tree”, in *Advances in Engineering, Science and Management (ICAESM), 2012 International Conference on*, IEEE, 2012, pp. 432–436.

[40] L. Bustio, R. Cumplido, R. Hernández, J. Bande, and C. Feregrino, “A hardware-based approach for frequent itemset mining in data streams”, in *Proceedings of the 4th workshop on new frontiers in mining complex patterns (nFCPM2015) held in conjunction with PKDD2015*, 2015, pp. 14–26.

[41] S. Sun, M. Steffen, and J. Zambreno, “A reconfigurable platform for frequent pattern mining”, in *Reconfigurable Computing and FPGAs, 2008. ReConFig’08. International Conference on*, IEEE, 2008, pp. 55–60.

[42] G. H. John and P. Langley, “Static versus dynamic sampling for data mining.”, in *KDD*, vol. 96, 1996, pp. 367–370.

[43] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara, “Evaluation of sampling for data mining of association rules”, in *Research Issues in Data Engineering, 1997. Proceedings. Seventh International Workshop on*, IEEE, 1997, pp. 42–50.

[44] S. Moens and M. Boley, “Instant exceptional model mining using weighted controlled pattern sampling”, in *Advances in Intelligent Data Analysis XIII*, H. Blockeel, M. van Leeuwen, and V. Vinciotti, Eds., Cham: Springer International Publishing, 2014, pp. 203–214, ISBN: 978-3-319-12571-8.

[45] M. Boley, T. Gärtner, and H. Grosskreutz, “Formal concept sampling for counting and threshold-free local pattern mining”, in *Proceedings of the 2010 SIAM International Conference on Data Mining*, SIAM, 2010, pp. 177–188.

[46] M. Boley and H. Grosskreutz, “Approximating the number of frequent sets in dense data”, *Knowledge and information systems*, vol. 21, *1*, pp. 65–89, 2009.

[47] M. Deypir, M. H. Sadreddini, and S. Hashemi, “Towards a variable size sliding window model for frequent itemset mining over data streams”, *Computers & industrial engineering*, vol. 63, *1*, pp. 161–172, 2012.

[48] M. Deypir and M. H. Sadreddini, “A dynamic layout of sliding window for frequent itemset mining over data streams”, *Journal of Systems and Software*, vol. 85, *3*, pp. 746–759, 2012.

[49] B. Negrevergne, A. Termier, M.-C. Rousset, and J.-F. Méhaut, “Para miner: A generic pattern mining algorithm for multi-core architectures”, *Data Mining and Knowledge Discovery*, vol. 28, *3*, pp. 593–633, 2014.

[50] J. J. Cameron, A. Cuzzocrea, and C. K. Leung, “Stream mining of frequent sets with limited memory”, in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ACM, 2013, pp. 173–175.

[51] F. Flouvat, “Experimental study of frequent itemsets datasets”, Technical report, LIMOS, France, http://www. isima. fr/flouvat/papers/rr05-ExpStudyDatasets. pdf, Tech. Rep., 2005.

[52] A. Metwally, D. Agrawal, and A. E. Abbadi, “An integrated efficient solution for computing frequent and top-k elements in data streams”, *ACM Transactions on Database Systems (TODS)*, vol. 31, *3*, pp. 1095–1133, 2006.

[53] M. J. Zaki, “Scalable algorithms for association mining”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, *3*, pp. 372–390, 2000.

[54] J. Cheng, Y. Ke, and W. Ng, “A survey on algorithms for mining frequent itemsets over data streams”, *Knowledge and Information Systems*, vol. 16, *1*, pp. 1–27, 2008.

[55] J. H. Chang and W. S. Lee, “Estwin: Online data stream mining of recent frequent itemsets by sliding window method”, *Journal of Information Science*, vol. 31, *2*, pp. 76–90, 2005.

[56] H.-F. Li, S.-Y. Lee, and M.-K. Shan, “An efficient algorithm for mining frequent itemsets over the entire history of data streams”, in *Proc. of First International Workshop on Knowledge Discovery in Data Streams*, vol. 39, 2004.

[57] G. S. Manku and R. Motwani, “Approximate frequency counts over data streams”, in *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB Endowment, 2002, pp. 346–357.

[58] J. H. Chang and W. S. Lee, “A sliding window method for finding recently frequent itemsets over online data streams”, *Journal of Information Science*, vol. 20, pp. 753–762, 2004, [Online; accessed 24-August-2016].

[59] ——, “Finding recent frequent itemsets adaptively over online data streams”, in *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2003, pp. 487–492.

[60] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, “Moment: Maintaining closed frequent itemsets over a stream sliding window”, in *International Conference on Data Mining*, IEEE, 2004, pp. 59–66.

[61] N. Bandi, A. Metwally, D. Agrawal, and A. El Abbadi, “Fast data stream algorithms using associative memories”, in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ACM, 2007, pp. 247–256.

[62] S. Lagraa, A. Termier, and F. Pétrot, “Scalability bottlenecks discovery in mpsoc platforms using data mining on simulation traces”, in *Proceedings of the conference on Design, Automation & Test in Europe*, IEE/ACM, 2014, p. 186.

[63] ——, “Data mining mpsoc simulation traces to identify concurrent memory access patterns”, in *Proceedings of the Conference on Design, Automation and Test in Europe*, EDA Consortium, 2013, pp. 755–760.

[64] O. Iegorov, V. Leroy, A. Termier, J.-F. Méhaut, and M. Santana, “Data mining approach to temporal debugging of embedded streaming applications”, in *Proceedings of the 12th International Conference on Embedded Software*, IEEE Press, 2015, pp. 167–176.

[65] S. H. Mirisaee, “Mining representative items and itemsets with binary matrix factorization and instance selection”, PhD thesis, 2015.

[66] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C. Wu., and V. S. Tseng, “SPMF: a Java Open-Source Pattern Mining Library”, *Journal of Machine Learning Research (JMLR)*, vol. 15, pp. 3389–3393, 2014, Online; accessed 24-August-2016. [Online]. Available: `http://www.philippe-fournier-viger.com/spmf/`.

[67] M. M. Waldrop, “The chips are down for moore’s law”, *Nature News*, vol. 530, *7589*, p. 144, 2016.

[68] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors”, *Communications of the ACM*, vol. 13, *7*, pp. 422–426, 1970.

[69] S. Cohen and Y. Matias, “Spectral bloom filters”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM, 2003, pp. 241–252.

[70] 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, *11*, pp. 1424–1440, 2004.

[71] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The Gem5 Simulator”, *SIGARCH Computer Architecture*, vol. 39, *2*, pp. 1–7, 2011, ISSN: 0163-5964. DOI: `10.1145/2024716.2024718`.

[72] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations”, in *ACM SIGARCH Computer Architecture News*, ACM, vol. 23, 1995, pp. 24–36.

[73] W. Wolf, A. A. Jerraya, and G. Martin, “Multiprocessor system-on-chip (MPSoC) technology”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, *10*, pp. 1701–1713, 2008.

[74] *Catapult c synthesis datasheet*, [Online; accessed 24-August-2016], Calypto Design Systems. [Online]. Available: `http://calypto.agranderdesign.com/catapult_c_synthesis.php`.

[75] Wikipedia, *Instruction pipelining — Wikipedia, the free encyclopedia*, [Online; accessed 24-August-2016], 2014. [Online]. Available: `https://en.wikipedia.org/wiki/Instruction_pipelining`.

Résumé étendu de la thèse de Mael Gueguen

Le domaine de *recherche de motifs fréquents* consiste à trouver tous les motifs récurrents dans une base de données à analyser. De nombreux algorithmes de recherche de motifs ont été proposés dans la littérature scientifique, mais la plupart d'entre eux souffrent du même problème : les résultats sont très volumineux et contiennent beaucoup de redondances, qui rendent les analyses difficiles. Une méthode appelée *échantillonnage de l'espace de sortie* a récemment été introduite. Cette méthode consiste à retourner un échantillon réduit, avec des contraintes statistiques qui permettent d'assurer sa représentativité. Dans un contexte où réagir aux tendances en temps réel est devenu un enjeu important, une analyse sur un échantillon en temps réel peut l'emporter sur une analyse exhaustive hors-ligne. Pour permettre de réaliser en temps réel les calculs coûteux de la recherche de motifs fréquents, cette thèse propose des solutions reposant sur des architectures matérielles dédiées, plus efficaces en temps et énergie que les serveurs classiques.

Les travaux de cette thèse peuvent être regroupés en deux contributions majeures. En premier lieu nous présentons un accélérateur matériel sur FPGA pour l'échantillonnage de motifs fréquents sur bases de données statiques, offrant de meilleures performances que son équivalent logiciel. Nous proposons ensuite une extension de l'accélérateur matériel pour traiter des flux de données.

Il existe trois applications de la technique d'échantillonnage de l'espace de sortie dans le cadre de la recherche de motifs fréquents dans la littérature. [AZ09] se base sur les chaînes de Markov, [Bol+11] propose un échantillonnage direct en deux étapes, et [DLD17] utilise une technique d'échantillonnage de l'espace de sortie originellement conçue pour le domaine des problèmes SAT. Les travaux de cette thèse se basent sur la solution **Flexics** proposée par [DLD17], car très flexible de par son utilisation d'une fonction d'intérêt des échantillon, tout en étant un très bon candidat pour l'accélération matérielle.

En considérant la recherche de motifs fréquents comme un problème de satisfaisabilité booléenne, **Flexics** utilise l'oracle **Weightgen** proposé par [Cha+14]. Cet oracle découpe l'espace de recherche grâce à des contraintes aléatoires pour générer un sous-espace représentatif réduit selon les besoins de l'échantillonnage. L'implémentation de **Flexics** étant basée sur l'algorithme **ECLAT** et des contraintes à base d'opérateurs XOR, les opérations utilisées par l'échantillonneur sont très adaptées pour l'accélération matérielle sur FPGA. En effet l'algorithme **ECLAT** connaît nombres de propositions d'accélération matérielle, notamment l'approche proposée par [Zha+13] sur laquelle nos contributions sont basées.

De nombreux efforts par la communauté de l'accélération matérielle sur FPGA ont été produit pour régulariser les calculs de recherche de motifs fréquents, de nature difficilement prévisibles et très irréguliers. Cependant dans un cadre comme l'échantillonnage de l'espace de sortie, l'altération de l'espace de recherche utilisé introduit de nouvelles irrégularités qui empêchent les accélérateurs proposés dans la littérature de rester efficace comparés à la version logicielle. L'accélérateur matériel proposé dans cette thèse, illustré Figure 1, utilise une série de transformation des contraintes générées pour retrouver une exploration régulière de l'espace de recherche.

Une autre fonctionnalité que nous proposons pour l'accélération d'ECLAT sur FPGA est une méthode de réduction dynamique de la base de données d'entrée, appelée liste de correspondance, et présentée Figure 2. Cette fonctionnalité, optionnelle, permet d'allouer un espace de stockage interne au FPGA pour y stocker l'ensemble des données de la base externe utiles pendant l'exploration de l'espace de recherche. Si cet ensemble est suffisamment petit et entre dans l'espace de stockage, grâce à un théorème appelé **principe d'Apriori**, les prochains calculs pourrons remplacer la lecture de la base de données externe pour se limiter à son sous-ensemble dans la mémoire interne. Bien que cette technique requiert une allocation mémoire supplémentaire à l'accélérateur, elle possède deux avantages : la mémoire à allouer est un paramètre utilisateur, c'est à dire qu'elle peut être fonction du budget disponible pour l'application ; le gain en performance est notable pour de faibles quantités de mémoire allouées, avec un effet de plateau rapidement atteint, comme illustré Figure 3.

Nous montrons dans la Table 1 les performances de notre accélérateur comparées à l'état de l'art sur un serveur de calcul. Excepté pour le jeu de données *kr-vs-kp*, notre accélérateur sur FPGA très modeste surpasse la version logicielle, et atteint une accélération d'un facteur 8 sur le jeu de données

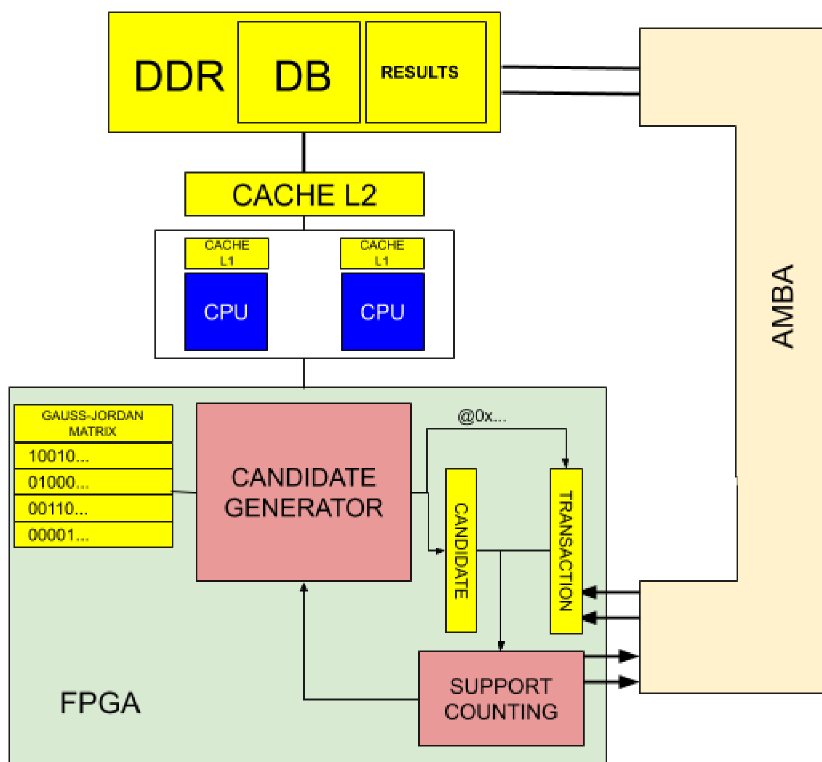

Figure 1: Architecture de l'accélérateur

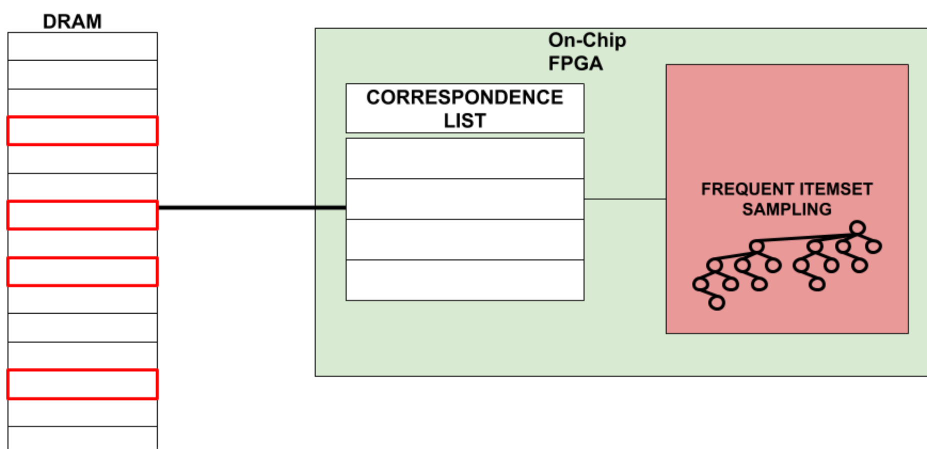

Figure 2: Exemple de fonctionnement de la liste correspondance

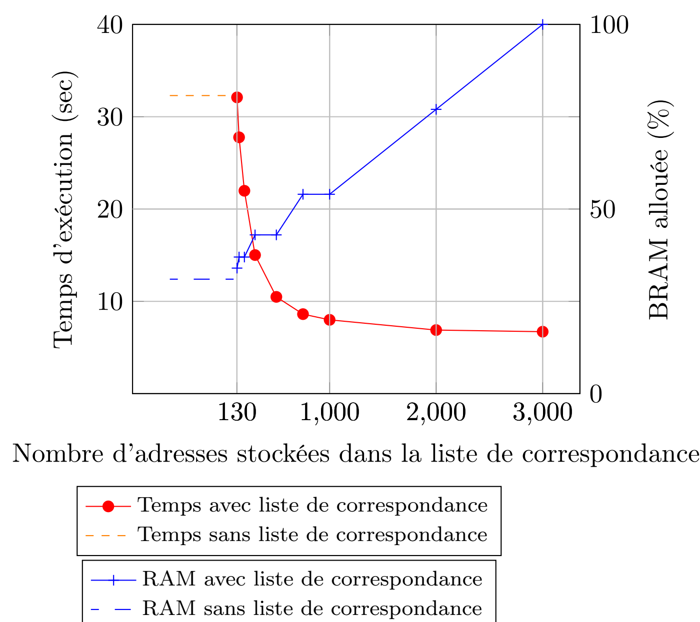

Figure 3: Temps d'exécution en fonction de la taille de la liste

primary. La Figure 4 quand elle illustre la réduction du temps d'exécution exponentielle avec le facteur d'échantillonnage utilisé, montrant que l'accélérateur tire profit au maximum de la réduction de la taille de l'espace de recherche.

Table 1: temps d'exécution (sec.) pour obtenir 1000 motifs

	État de l'art (CPU @ 3.2GHz)	Notre solution (FPGA @ 100MHz)
german	25	11,7
heart	45	11,6
kr-vs-kp	9	12,7
primary	10	1,2
vote	19	5,8

Lorsque les données à traiter arrivent à un débit trop important ou que la mémoire disponible ne permet pas de stocker toutes les données à la fois, un algorithme de recherche de motifs fréquent se retrouve confronté à de nouvelles problématiques. Il est par exemple souvent impossible pour une application de relire une donnée du flux passé une échéance, ce qui signifie que la structure algorithmique doit être fondamentalement revue, en plus de potentielles contraintes de temps-réel qui nécessitent un débit minimal de traitement des données. Pour adapter notre accélérateur aux flux de données, nous proposons une structure de données interne basée sur le **FP-Tree** pour contenir les informations recueillies précédemment dans le flux. L'accélérateur doit donc parcourir l'espace de recherche échantillonné, tout en retrouvant les informations stockées localement dans le **FP-Tree** de manière efficace.

Pour permettre aux résultats d'être actualisé avec les données récentes, plutôt que stagner sur des données trop anciennes qui pourraient persister tout au long de l'exécution, nous avons fait le choix d'utiliser une pondération exponentielle sur les données du flux. A chaque arrivée de nouvelle données, les résultats précédents sont pondérés par un facteur d'affaiblissement $0 < d < 1$. De cette manière, un élément apparu au temps t_1 verra son importance multipliée par $0 < d^{t_2 - t_1} < 1$ au temps t_2. Cette technique introduit une nouvelle problématique quand utilisée conjointement à l'échantillonnage de sortie. Si de nombreux éléments dans les résultats se voient expulsés sans nouveaux éléments fréquents, la taille de l'échantillon des résultats va diminuer, jusqu'à ne plus être représentatif. Pour ce cas d'usage absent des autres échantillonneurs tels que **Flexics**, nous avons détaillé comment il est possible, suite a un réajustement du facteur d'échantillonnage, d'estimer une partie des informations perdues précédemment dans le flux.

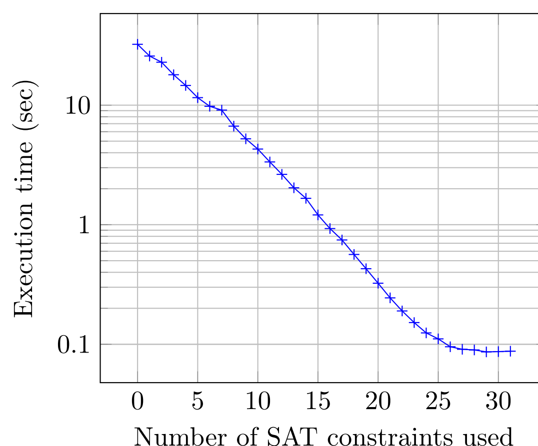

Figure 4: Impact de contraintes aléatoires sur le temps d'exécution

Figure 5 illustre l'impact du facteur d'échantillonnage et du facteur d'affaiblissement sur la *recoverability*, une mesure de précision, quand les échantillons sont comparés à la vérité de terrain. Même si le facteur d'affaiblissement a un effet notable sur la précision, on observe que le facteur d'échantillonnage est beaucoup plus déterminant.

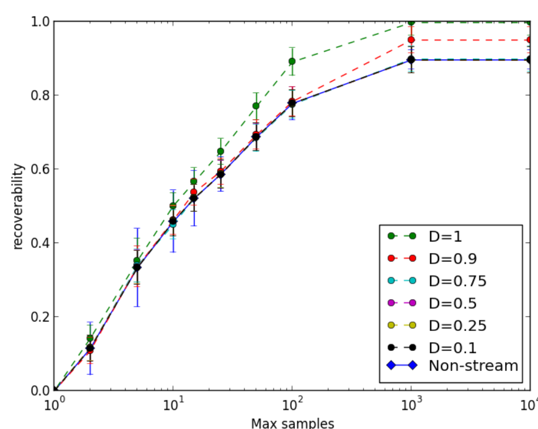

Figure 5: Précision des échantillons sur le flux

References

[AZ09] Mohammad Al Hasan and Mohammed J. Zaki. "Output Space Sampling for Graph Patterns". In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 730–741.

[Bol+11] Mario Boley et al. "Direct Local Pattern Sampling by Efficient Two-step Random Procedures". In: *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '11. New York, NY, USA: ACM, 2011, pp. 582–590. ISBN: 978-1-4503-0813-7. DOI: 10.1145/2020408.2020500. URL: http://doi.acm.org/10.1145/2020408.2020500 (visited on 06/15/2017).

[Zha+13] Yan Zhang et al. "An FPGA-Based Accelerator for Frequent Itemset Mining". In: *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* 6.1 (May 2013), 2:1–2:17. ISSN: 1936-7406.

[Cha+14] Supratik Chakraborty et al. "Distribution-Aware Sampling and Weighted Model Counting for SAT". In: *arXiv:1404.2984* (Apr. 2014).

[DLD17] Vladimir Dzyuba, Matthijs van Leeuwen, and Luc De Raedt. "Flexible constrained sampling with guarantees for pattern mining". In: *Data Mining and Knowledge Discovery* (Mar. 2017). (Visited on 06/15/2017).