J Intell Inf Syst
DOI 10.1007/s10844-017-0461-8

On the design of hardware-software architectures for frequent itemsets mining on data streams

Lázaro Bustio-Martínez[1] · René Cumplido[1] · Raudel Hernández-León[2] · José M. Bande-Serrano[2] · Claudia Feregrino-Uribe[1]

Received: 7 June 2016 / Revised: 30 March 2017 / Accepted: 12 April 2017

Abstract Frequent Itemsets Mining has been applied in many data processing applications with remarkable results. Recently, data streams processing is gaining a lot of attention due to its practical applications. Data in data streams are transmitted at high rates and cannot be stored for offline processing making impractical to use traditional data mining approaches (such as Frequent Itemsets Mining) straightforwardly on data streams. In this paper, two single-pass parallel algorithms based on a tree data structure for Frequent Itemsets Mining on data streams are proposed. The presented algorithms employ Landmark and Sliding Window Models for windows handling. In the presented paper, as in other revised papers, if the number of frequent items on data streams is low then the proposed algorithms perform an exact mining process. On the contrary, if the number of frequent patterns is large the mining process is approximate with no false positives produced. Experiments conducted demonstrate that the presented algorithms outperform the processing time of the hardware architectures reported in the state-of-the-art.

✉ Lázaro Bustio-Martínez
lbustio@inaoep.mx

René Cumplido
rcumplido@inaoep.mx

Raudel Hernández-León
rhernandez@cenatav.co.cu

José M. Bande-Serrano
jbande@cenatav.co.cu

Claudia Feregrino-Uribe
cferegrino@inaoep.mx

[1] Computer Sciences Department, National Institute for Astrophysics, Optics, and Electronics, Luis Enrique Erro ♯ 1, Sta. María Tonantzintla, CP: 72840, Puebla, México

[2] Advanced Technologies Application Center, 7a ♯ 21406, Siboney, Playa, CP: 12200, Havana, Cuba

Keywords Data Mining · Frequent Itemsets Mining · Data streams · Reconfigurable Hardware · Parallel algorithms

1 Introduction

Frequent Itemsets Mining is a basic technique of data mining that achieves outstanding results when it is used to process datasets. In this scenario, Frequent Itemsets Mining can be seen as the initial stage for other important data mining tasks such as Association Rules Mining, automatic summarization and concept drift among others. Since the last decade of the past century, the world is living a real data revolution. Datasets were omnipresent, but recently, data streams are becoming the preferred data source. Data streams are being used in video and audio broadcasting, network traffic analysis and commercial transactions filtering among other applications (Babcock et al. 2002). Data streams can be seen as a particular case of datasets where transactions are received one by one. Also, in data streams, data evolves over time and is transmitted at high speeds making impossible to storage it for offline processing. This situation provokes that traditional approaches for mining datasets cannot be used straightforwardly in data streams. All these issues impose extra challenges to the discovery of frequent itemsets in data streams.

Several algorithms have been proposed for Frequent Itemsets Mining but, when they are used in data streams, they become inefficient or impractical (Manku and Motwani 2002; Metwally et al. 2006): the number of itemsets produced grows exponentially concerning the number of unique items in the data stream. This situation forbids the use of traditional Frequent Itemsets Mining algorithms in data streams. Besides, data in data streams are transmitted at very high speeds, so they must be processed in very short time periods. The available software-based algorithms cannot meet such requirements effectively (Agrawal and Srikant 1994; Zaki 2000; Han et al. 2000). Therefore, finding alternatives to achieve better results in the discovery of frequent itemsets on data streams is an active research topic. One of such alternatives is to propose new hardware-based algorithms capable of handling such immense data volumes in very short time periods exploiting the fine-grained parallelism of hardware based architectures. In such way, some hardware-based approaches for Frequent Itemsets Mining on data streams were proposed to achieve the required performance (Tong and Prasanna 2013; Teubner et al. 2010; Teubner and Müller 2011; Sun et al. 2014) but they were oriented to discover frequent $1-$ itemsets, which is a simplified problem of discovering frequent k-itemsets. Discovering frequent k-itemsets on data streams is an open research problem which is addressed in this paper.

The present paper extends a previous one (Bustio et al. 2015). This extension includes a new algorithm for Sliding Window Model and transactions pre-processing stage. Mining frequent itemsets over high speed, continuous and infinite data streams is a challenging problem due to changing nature of data and limited memory and processing capacities of computing systems. Sliding Window is an interesting model used to solve these issues since it does not need the entire history of received transactions and can handle the concept drift by considering only a limited range of recent transactions. The use of hardware devices as development platforms forces the optimal use of hardware resources and this issue should be specially observed in large data handling algorithms such as Frequent Itemsets Mining on data streams.

The main contribution of this paper is, therefore, two tree-based parallel algorithms for Frequent Itemsets Mining on data streams designed to be implemented and executed in a hardware-software architecture. The software-side on both algorithms perform the data preprocessing and the creation (and maintenance) of the processing window; while the hardware-side accomplishes the mining process in parallel. Frequent Itemsets Mining is the most important process in this research, so it is addressed with more details.

Several experiments were conducted showing that the proposed algorithms outperform the processing times (and throughput) of several well-known hardware-based architectures reported in the state-of-the-art.

The current paper is structured as follows: in Section 2, the theoretical basis that supports this research is presented. A review of the state-of-the-art is addressed in Section 3 while Section 4 introduces the presented algorithms. Results are shown and discussed in Section 5 and finally, conclusions are given in Section 6.

2 Theoretical basis

The general problem of finding frequent itemsets in data streams can be formulated as follows: given a data stream, users are interested in detecting which of the transmitted itemsets were frequents in some period. These frequent itemsets can be used later in other Data Mining task such as automatic summarization, concept drift detection or association rules creation. Formalizing the former description, let $I = \{i_1, i_2, .., i_n\}$ be a set of n different items and T be a transactional data stream. An *itemset* X is a set of items over I such that $X \subseteq I$. A transaction $t \in T$ over I is a couple $t = (tid, X)$. In this definition, tid is the transaction identifier, and X is an itemset. The support of X is the fraction of transactions in T containing X. An itemset is called $frequent$ if its support is no less than a given minimal support threshold.

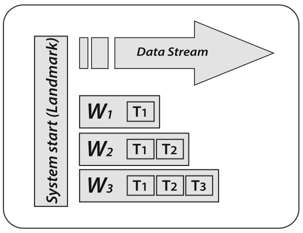
A *data stream* is a continuous, unbounded and not necessarily ordered (the order can be established implicitly by arrival time or explicitly by time stamps) real-time sequence of data items. In such way, a *transactional data stream* is an infinite sequence of transactions in a data stream.

In data streams, three main restrictions are imposed (Babcock et al. 2002; Golab and Özsu 2003):

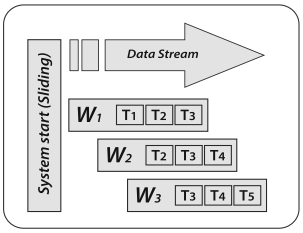
– *Continuity*: Items in streams arrive continuously at high rates.
– *Expiration*: Items can be accessed and processed just once.
– *Infinity*: The total number of data transmitted is unbounded and potentially infinite.

Similar to data sets, in transactional data streams is very interesting to obtain frequent itemsets (*e.g.* as the initial stage for other complex Data Mining techniques such as concept drift detection and automatic summarization among others). In such way, the incoming transactional data stream should be segmented for its proper processing, and this segmentation (named *windows*) allows fulfilling the restrictions of data streams. Segmentation of data streams is also useful to determine inside which boundaries an itemset can be frequent. Therefore, a *window* can be defined as an excerpt of items in a data stream and is constructed using one of the following approaches (Jin and Agrawal 2007):

– *Landmark Window Model*. This model employs some point (called *landmark*) to start recording where a window begins. The landmark usually is referred to the time when the system starts. Moreover, the support value of an itemset is the number of transactions that contain it between the landmark and the current time (see Fig. 1a). This window model cannot be aware of time, and therefore, cannot distinguish between new and old data.



(a) Landmark Window Model for 3 windows.



(b) Sliding Window Model for 3 windows.

Fig. 1 Landmark and Sliding Window Model. W denotes the current *window* and T denotes the *transactions* included in windows

– *Sliding Window Model*. This model uses the latest $|W|$ transactions in the data stream for the mining process. As the newest transactions arrive, the oldest in the sliding windows are excluded. This model can be compared with a FIFO queue. The use of this model imposes a restriction: as some transactions will be excluded from the mining process, methods for finding expired transactions and discounting the frequency counting of the itemsets involved are required (see Fig. 1b). This model is based on the assumption that the number of frequent patterns is not particularly large and, therefore, it is possible to store the transactions in each sliding window.

It is important to notice that the model to use depends on the application and the nature of the received data streams. Landmark Window Model is mainly employed in those situations where transactions have the same meaning regardless the time they occur. An example on this could be situations in which users are interested in all data transmitted (Bai-En et al. 2012; Cheng et al. 2008; Golab and Özsu 2003): the stock market for example. Analyzing the stock market variations can be used to observe the average price of stock in the current month or year. In this example, using other models leads to losing relevant information that would be retained using Landmark Window Model. Nevertheless, using Landmark Window Model, memory and processing time requirements will grow as time passes because more data will be involved in processing. Although such issue can be avoided using some implementation trade-offs, it should be taken into account in resource-limited environments (such as hardware devices).

Sliding Window Model is better suited than Landmark for processing excerpts of the incoming data streams. Also, as it was stated before, using Landmark Window Model all transactions have the same meaning in the mining process conducing to oldest transactions having the same significance respect to newest. As it was proved in Giannella et al. (2003), in some cases it is more useful to mine all recently-arrived transactions than the oldest. Sliding Window Model allows to keep memory almost constant: as it can be seen as FIFO queue, the maximum number of transactions in the processing windows is constant. Thus, Sliding Window Model is more efficient (regarding resources consumption) than Landmark Window Model. If just one window is analyzed (considering the same window size), there is no difference between Landmark and Sliding Window Models concerning the frequent itemsets discovered and their frequency counting. The main difference is in the processing times obtained. The window model to use should be selected based on the data streams nature and the mining problem to deal with. In this paper, both window models were implemented.

2.1 Hardware-accelerated processing

There are two main ways to develop an algorithm (Compton and Hauck 2002): the first one is using an Application-specific Integrated Circuit (ASIC). ASICs are used to perform a well-defined task; and, therefore, they are extremely fast and efficient. However, once ASICs have been built, they cannot be modified.

The second way is using General Purpose Processor (GPP), where the software instructions are decomposed into a set of basic processor-level instructions that can be executed directly by the GPP. So, changing the software instructions implies a change in the algorithms behavior. GPPs provide high flexibility in algorithms execution at the cost of performance degradation.

Just in the middle of both development platforms, Reconfigurable Hardware Computing (RHC) is best suited for building prototypes. The main two RHC platforms are Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU).

FPGAs appeared in 1984 and combined the advantages of ASICs and GPPs trying to fill the gap between hardware and software, increasing the performance concerning of GPPs and maintaining a higher level of flexibility than hardware. The architecture of FPGAs is based on a large number of logic blocks which perform basic logic functions. Because of this, one FPGA can implement from a simple logical gate to a complex mathematical function. FPGAs can be reprogrammed; that is, the circuit can be "erased" and then, a new architecture that implements a new algorithm can be created.

GPU Computing is based on the use of a specialized processing platform known as Graphics Processing Unit (GPU), which was originally designed to accelerate computer graphics processing, thus enhancing the performance of systems based on traditional GPPs. Recently, GPUs have gained the attention of the scientific community since they have been used as hardware accelerators for various non-graphics applications, such as scientific computation, matrices multiplications, and distributed computing projects among others. As it was explained before, data mining on data streams is a challenging task. It is mandatory to develop very efficient algorithms using effective techniques and tools to fulfill the requirements imposes for this tasks. Software-based approaches can deal with massive data streams, but their performance is limited.

GPUs are a relatively easy-to-use hardware, but their parallelism level is limited by their hardware architecture. Using FPGAs allow to develop a fine grain parallelism where designers can create truly custom-made hardware designs that fulfill all design requirements of the algorithms that they are trying to implement. Such level of parallelism is a highly valuable need in data stream processing, and it cannot be obtained using GPUs.

Considering presented facts of FPGAs and GPUs, it was decided to use FPGAs for hardware acceleration in this research.

3 Literature review

Frequent Itemsets Mining is a widely used Data Mining technique (Aggarwal and Han 2014). Recent research efforts have been focused on accelerating the discovering of frequent itemsets over transactional datasets using hardware devices (such as FPGAs) (Baker and Prasanna 2005, 2006; Bustio et al. 2015; Mesa et al. 2010; Shaobo et al. 2013; Song et al. 2008; Song and Zambreno 2008, 2011; Thöni and Strey 2009; Wen et al. 2008; Zhang et al. 2013; Yamamoto et al. 2016). Data mining over data streams is gaining attention and have been tackled from a software perspective with relative success. Nevertheless, all software-based approaches (Cameron et al. 2013; Cheng et al. 2008; Giannella et al. 2003) are based on complex data structures (in memory or disk) where transactions are temporarily stored to be processed. The processing times reported in these papers are prohibited for some applications (such as intrusion detection systems). In general, accelerating Data Mining techniques (e.g. Frequent Itemsets Mining) on data streams is a common need, and it can be achieved using custom hardware architectures. However, the approach followed in software for Frequent Itemsets Mining on data streams is impractical to be used in resource-limited devices, such as FPGAs.

In the reviewed papers, FPGAs have been used as coprocessors or hardware-based accelerators of certain functions in hardware-software implementations of some well-known algorithms, but all of them were targeted to datasets. In this section, only significant advances in Frequent Itemsets Mining using FPGAs are reported regardless used data sources (datasets or data streams). The state-of-the-art of Frequent Itemsets Mining in FPGAs can be summarized as it is shown in Table 1.

Table 1 shows a summary of the characteristics of the architectures described in the state-of-the-art. Column *Paper* references the reviewed papers by the name of the first author and the year of publication. The data source used in each article is shown in column *Data source* (indicating if the described architecture is oriented to datasets or data streams). Column *Design strategy* indicates if the approach was fully implemented in hardware (HW) or if it was implemented in a hybrid approach (hardware-software, HW/SW). The column *Based on* establishes the approach followed by the cited papers.

Revised literature can be divided into three main groups: (1) algorithms based on Apriori (Agrawal and Srikant 1994); (2) algorithms based on FP-Growth (Han et al. 2000) and (3) algorithms based on Eclat (Zaki 2000).

Algorithms that implement Apriori in hardware require loading all the data into the device (Baker and Prasanna 2005, 2006; Thöni and Strey 2009; Wen et al. 2008). This strategy is limited by the capacity of the chosen hardware device: if the number of transactions

Table 1 Summary of main revised papers organized chronologically

Paper	Data source	Design strategy	Based on
Baker and Prasanna (2005)	Dataset	HW	Apriori
Baker and Prasanna (2006)	Dataset	HW	Apriori
Song et al. (2008)	Dataset	HW	FP-Growth
Song and Zambreno (2008)	Dataset	HW	FP-Growth
Wen et al. (2008)	Dataset	HW/SW	Apriori
Thöni and Strey (2009)	Dataset	HW	Apriori
Mesa et al. (2010)	Dataset	HW	FP-Growth
Song and Zambreno (2011)	Dataset	HW	FP-Growth
Shaobo et al. (2013)	Dataset	HW/SW	Eclat
Zhang et al. (2013)	Dataset	HW/SW	Eclat
Bustio et al. (2015)	Streams	HW/SW	Systolic tree
Yamamoto et al. (2016)	Streams	HW	Skip LC-SS

to manage exceeds the hardware capacity, then transactions should be loaded separately in many consecutive times. This procedure degrades the overall performance, and it is forbidden for data streams mining, where data must be processed in a very short period (to fulfill the Continuity and the Expiration constraints).

Hardware implementations of algorithms based on FP-Growth download the mining dataset into the FPGA and, as required by the algorithm, traverse the dataset at least twice (Mesa et al. 2010; Song et al. 2008; Song and Zambreno 2008, 2011). The double traversal is avoided by Mesa et al. in (2010), but they still need to download the complete dataset into the hardware device. The double traversal of FP-Growth-like algorithms is impractical in data streams mining due to the Expiration constraint. Nevertheless, FP-Growth-based algorithms exploit the FP-Tree data structure where data flows from the root node to leaf nodes. Considering this, the FP-Tree data structure can be modified to be efficiently used for frequent itemsets mining on data streams.

Data can also be represented using vertical layout, and the Eclat algorithm was the first one that uses this data representation for Frequent Itemsets Mining. Using the vertical layout, the frequency of an itemset is obtained by intersecting the vectors that compose the itemset. In consequence, hardware-based implementations of Eclat (Shaobo et al. 2013; Zhang et al. 2013) also use the vertical dataset representation to save memory space and processing time. In the vertical dataset representation, the items intersection can be implemented in hardware by using logical *AND* operations. In Shaobo et al. (2013) and Zhang et al. (2013), authors propose a software-hardware architecture where the most time and memory consuming functions were downloaded to hardware while software controls the execution flow and data structures. Although the vertical dataset representation allows saving memory and processing time, it is not compatible with the Expiration constraint because all data involved in the vectors intersection must be known before processing. Also, the pruning strategy in Eclat is inefficient and introduces delays that affect the overall performance of algorithms. These two issues make Eclat impractical to be used as a starting point for new data stream mining algorithms.

In Bustio et al. (2015), a systolic tree is used: in this approach, transactions flow from root to leaf nodes, and it is not necessary a candidates generation stage, neither several iterations over the dataset. This feature is highly valuable in data streams analysis because multiples traverse over data are forbidden. The algorithm described uses Landmark Window Model and two-dimensional search. Experiments conducted to demonstrate that this approach reduces the execution time compared to hardware-based implementations of the baseline algorithms.

In Yamamoto et al. (2016), the Skip LC-SS algorithm, which integrates the Lossy-Counting and Space-Saving algorithms, is implemented in hardware. By identifying the bottleneck in the original algorithm, authors were able to successfully introduce a more efficient replacement process using a batch-replacement concept. The hardware implementation described by Yamamoto et al. uses the Space-Saving algorithm to fix the number of entries to be saved and the processing unit of Lossy-Counting to speed up the calculation. Also, some approximation process (skip) was added. In this algorithm, no information about the window model used was provided. Authors only evaluate their approach using one value of minimum support value (50%), and we consider that is not enough to evaluate the viability of the proposed algorithm: more support threshold values should be evaluated. The algorithm proposed is approximate, no information about this approximation was given. All these issues made this work not eligible for the comparison.

From the review of the state-of-the-art, it is noticed that the frequent itemsets detection problem can be divided into four well-defined subproblems as it is presented in Fig. 2:

1. **Data streams are composed of long transactions and large alphabets.** This is the most complex subproblem of frequent itemsets detection. The total number of itemsets generated is $2^n - 1$, where $n = |I|$. In such cases, hardware solutions cannot hold all generated itemsets (due to resource limitations) and therefore, due to its flexibility, a software-based implementation is a right solution to be adopted.
2. **Data streams are composed of long transactions in small alphabets.** This subproblem can be solved using tree-based approaches where transactions flow inside a tree data structure. The parallelism levels obtained can outperform the processing time compared versus software-based approaches using pipelining techniques.
3. **Data streams are composed of short transactions in small alphabets.** This subproblem can be solved in software, but using hardware approaches will improve the performance of algorithms taking into account the inner parallelism of this devices. Tree-based approaches can also be used to deal with this subproblem.
4. **Data streams are composed of short transactions in large alphabets.** This subproblem cannot be solved using tree-based approaches due to the immense data explosion obtained for a large n. Software approaches would deal with such issue, but the performance can be limited. Therefore, a hardware-based approach that performs in parallel the mining of received transaction based on alternative information, such as the lexicographic order of the arrived itemsets is an appropriate solution.

In general, all reviewed papers were focused on improving technical aspects such as efficient data structures and optimizing data flows, but theoretical foundations of algorithms used as a baseline were intact. This is the major issue detected in the reviewed papers. From the review of the state-of-the-art, and after detecting the main issues in Frequent Itemsets Mining on data streams, it was determined that this paper should address the second subproblem of Frequent Itemsets Mining on data streams.

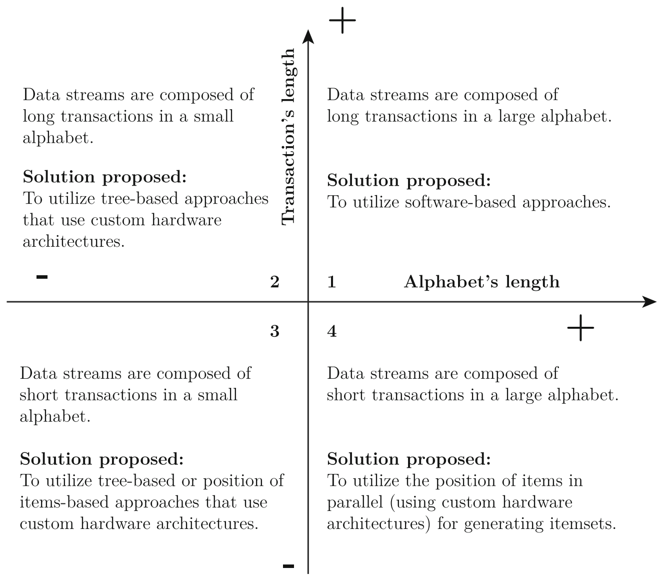

+
Transaction's length
Data streams are composed of long transactions in a small alphabet.
Solution proposed: To utilize tree-based approaches that use custom hardware architectures.
Data streams are composed of long transactions in a large alphabet.
Solution proposed: To utilize software-based approaches.
-
2
1
Alphabet's length
3
4
+
Data streams are composed of short transactions in a small alphabet.
Solution proposed: To utilize tree-based or position of items-based approaches that use custom hardware architectures.
Data streams are composed of short transactions in a large alphabet.
Solution proposed: To utilize the position of items in parallel (using custom hardware architectures) for generating itemsets.
-

Fig. 2 Different subproblems derived from the Frequent Itemsets Mining on data streams

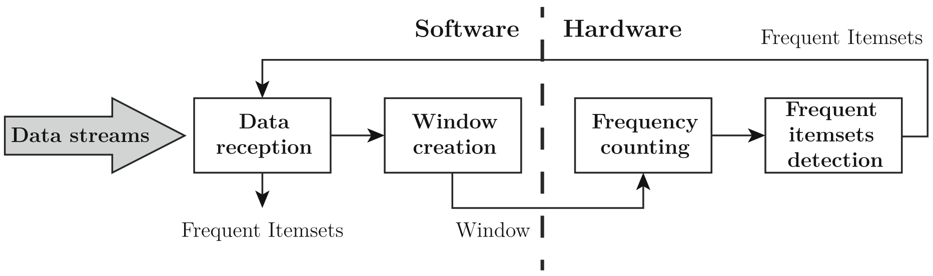


Fig. 3 General flow diagram of the proposed algorithms for Frequent Itemsets Mining on data streams

4 Proposed method

The approach presented in this paper is composed of two algorithms for Frequent Itemsets Mining (one using Landmark Window Model and another using Sliding Window Model) that implement a software-hardware processing scheme. Figure 3 shows a general flow diagram of the proposed algorithms. In these algorithms, the software-side is responsible for window creation and data preprocessing to detect the most frequent 1-itemsets, which is a trivial process. The hardware side (which this paper is focused on) is responsible for executing the mining process in parallel using a tree-based data structure. This tree structure can be seen as a *systolic tree*[1] where each of its nodes has one bottom node (child) and one right node. Figure 4 shows the systolic tree where vertically-arranged nodes represent a prefix path and parent nodes contain the prefix itemset for their children. Let be a random node r, the sub-tree who had the node r as root is formed by all possible combinations of items with the itemset stored in r as their prefix. For instance, the selected sub-tree of Fig. 4 contains all combinations of itemsets starting with the prefix "$\underline{\mathbf{ab}}$" ($\underline{\mathbf{ab}}c$, $\underline{\mathbf{ab}}d$ and $\underline{\mathbf{ab}}cd$). The same idea can be applied recursively to any node of the tree allowing to develop recursive mining strategies.

The main difference between all revised papers and the proposed algorithms is the control structure: in Baker and Prasanna (2005, 2006); Mesa et al. (2010); Shaobo et al. (2013); Song et al. (2008); Song and Zambreno (2008, 2011); Thöni and Strey (2009); Wen et al. (2008); Zhang et al. (2013) a centralized control was employed. In the proposed architecture, each node of the systolic tree is an independent processing unit containing all logic and storage elements required to perform the frequency counting of an itemset. Nodes in the systolic tree have their control block allowing to develop a distributed control structure. Such distributed control removes communications lines and traffic of control signals among nodes. The use of a distributed control also reduces the negative impact of having long and numerous control signals spread all along the architecture.

Following, algorithms proposed are described.

4.1 Landmark Window Model-based approach

The window creation process using Landmark Window Model is executed in the software-side of the proposed architecture. For a window W, the software creates (and maintains) an array (called *processing_window*) of size $w = |W|$ where transactions are stored (duplicated

[1] A systolic tree is an arrangement of pipelined processing elements in a multidimensional tree pattern.

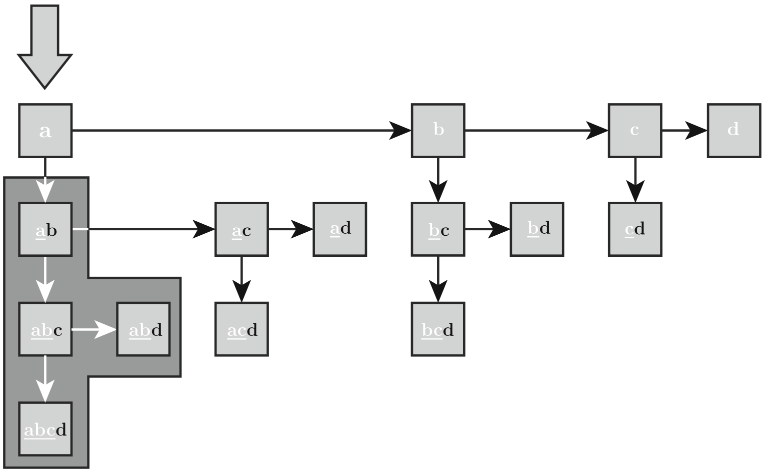


Fig. 4 Systolic tree used in data streams mining. Highlighted section shows a sub-tree with nodes that have the itemset *ab* as root. The highlighted sub-tree is formed by all combinations of items with the itemset *ab* as prefix

items in a transaction are considered as a single item, and therefore, repetitions are removed). Supported by the Infinity constraint of data streams, it is assumed that the data stream has no end and therefore, w will always be filled. When the processing window is created and filled, then the processing window is moved into the hardware to obtain the frequency counting of itemsets. It is valid to notice that Landmark Window Model considers transactions for the mining process from the landmark point until the current time (see Fig. 1a). As the time passes, the processing window grows becoming eventually intractable. To avoid such issue, the frequency counting detected so far is maintained in the hardware, and just the w lastly arrived transactions are moved into the hardware. This adjustment emulates the growing of the processing window in Landmark Window Model and allows maintaining suitable the resources consumption.

When a transaction t in $processing\_window$ arrives at a node, then the following conditions are evaluated:

1. If the itemset X of t arrives at an empty node r, then r will be occupied by the item $\{i\} \subseteq X$ and its frequency will be set to 1.
2. If the itemset X of t arrives at an occupied node r then one of the following decision should be taken:
   (a) If the node r is occupied by an item $\{i\} \subseteq X$ then the frequency counter of r is incremented and the itemset $\{X - \{i\}\}$ is flowed to its child and right nodes.
   (b) If the node r is occupied by an item $\{i\} \nsubseteq X$ then X is flowed to the right node of r.

Using these conditions, a parallel algorithm was developed to be implemented and executed in hardware (Algorithm 1). This algorithm simultaneously uses Depth First Search (DFS) and Breadth First Search (BFS) traversals (from lines 1 to 1) allowing a high grade of parallelism A two-dimensional search is performed concurrently versus a top-down traversal implemented in revised papers. These simultaneous DFS and BFS traversals are also employed in determining which itemsets can be regarded as frequent once the frequency counting of each itemset was computed and stored in nodes of the systolic tree.

Algorithm 1 Parallel algorithm for frequency counting using Landmark Window Model

Require: Processing window P using Landmark Window Model.
Ensure: Systolic tree with the frequency count for each itemset.

```
1:  Procedure MAIN
2:    $r \leftarrow systolic\_tree.RootNode$;    ▷Set $r$ to point to root node of the systolic tree.
3:    for each transaction $t$ in $P$ do
4:      Traverse($t, r$);    ▷Flow received transactions through the root node.
5:    end for each
6:  end procedure

7:  procedure TRAVERSEtransaction $t$, node $r$;
8:    if ($r.IsOccupied == false$) then
9:      $r.Item = t.ItemAt(0)$;    ▷Stores the first item of $t$ in $r$.
10:     $r.IsOccupied = true$;    ▷Marks $r$ with the $occupied$ flag.
11:     FlushInParallel($t, r$);    ▷$t$ is processed in parallel.
12:   else
13:     if($t.Contains(r.Item)$ == true) then    ▷Verifies if $t$ contains the item stored in $r$.
14:        FlushInParallel($t, r$);    ▷In positive case, $t$ is processed in parallel.
15:     else
16:       $\widetilde{r} \leftarrow r.RightNode()$;    ▷$t$ is flowed only to the right node of $r$.
17:       Traverse($t, \widetilde{r}$);    ▷$t$ is processed in parallel in the right node of $r$.
18:     end if
19:   end if
20: end procedure

21: procedure FLUSHINPARALLELtransaction $t$, node $r$;
22:   $r.Counter++$;    ▷The frequency counting of $r$ is increased
23:   $\widetilde{t} = t.Exclude(r.Item)$;    ▷$t$ is reduced.
24:   if ($\widetilde{t}.IsEmpty$ == false) then
25:     StartParallelBlock:    ▷Parallel processing begins.
26:         $\widetilde{r} \longleftarrow r.ChildNode()$;
27:         Traverse($\widetilde{t}, \widetilde{r}$);
28:         $\widetilde{r} \longleftarrow r.RightNode()$;
29:         Traverse($\widetilde{t}, \widetilde{r}$);
30:     EndParallelBlock;    ▷Parallel processing ends.
31:   end if
32: end procedure

33: return $systolic\_tree$;
```

Each node of the systolic tree executes Algorithm 1. Figure 5 presents a diagram of the systolic tree magnifying one processing node. Each node is composed of one memory which stores the received transaction and a finite state machine which implements the control logic for the current node. A hardware-level schematic of the control structure is represented in Fig. 6.

When the *start* signal is received in a node r from its parent, then the transaction t which is received in *data* is used to obtain the item that will be stored in r (line 4). At

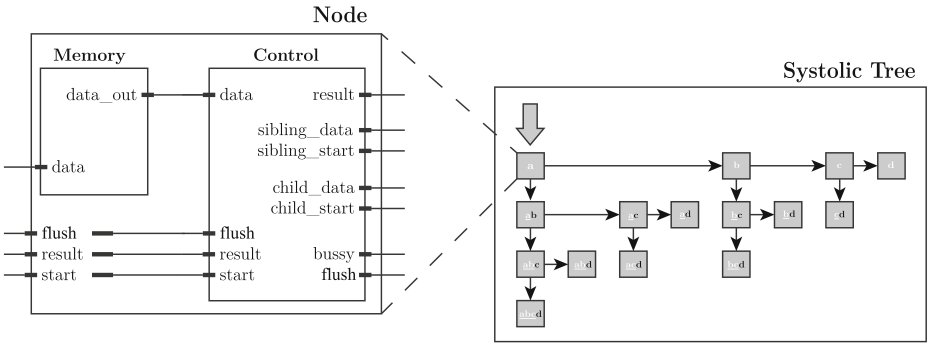


Fig. 5 Diagram of the systolic tree and one processing node

this point, as the node r is busy processing the received transaction, then the *busy* signal is set to 1. As Algorithm 1 states, if r is empty (the *Occupied* register is set to 0) (this condition is evaluated in line 8) then the *Occupied* register is updated to 1 (line 10). Also the *Frequency Counter* register value is increased (line 22) and *Item* register stores the first item of t (line 9) and remove it from t (reducing the received transaction) (in line 23). This reduced transaction $\widetilde{t}$ is flowed to the child and right nodes of r using the *child_data* and *sibling_data* outputs (in lines 25–30). Simultaneously, the *start* signal for those nodes is transmitted using *child_start* and *sibling_start* outputs. Next, the *busy* signal is set to 0, allowing the node r to process another transaction.

On the contrary, if t is received by r and r is occupied (*Occupied* register is set to 1), then it is verified whether t contains the item stored in *Item* register (in line 13). In the affirmative case, the value stored in *Frequency Counter* register is incremented (in line 22) and t is reduced (line 23). The reduced transaction $\widetilde{t}$ is flowed to child and right nodes using the *child_data* and *right_data* outputs (lines 25–30). Simultaneously, the *start* signal for those nodes is transmitted using *child_start* and *sibling_start* outputs. In the negative case (in line 15), t is flowed only to right node of r using the *right_data* output (line 16–17). In this case, the *start* signal is transmitted only to right node of r using *sibling_start* output.

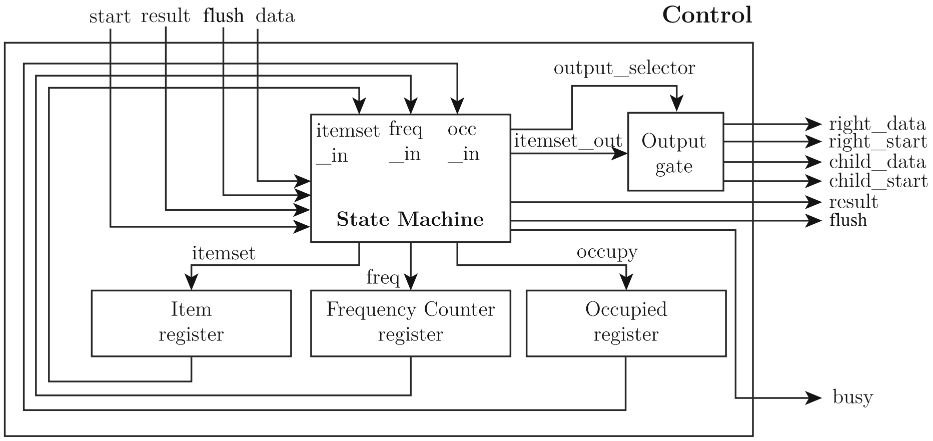


Fig. 6 Hardware-level schematic of the control of a processing node

After the frequency of each itemset is computed, a backtracking strategy using the Downward Closure property (Agrawal and Srikant, 1994) is employed to obtain frequent itemsets from the systolic tree (implemented in Algorithm 2). In this strategy, if a node is declared as frequent (line 8), then its child and right nodes must be processed recursively to determine whether they are frequent or not (line 9). At this point, the current node goes into the *Gateway* mode (line 14) where control signals and data flows through it, and no other processing is performed until the node goes into the *Counting* mode (in line 16). On the contrary, if a node is regarded as infrequent, its descendant's nodes will also be infrequent, and the process can be stopped. This recursive processing (in lines 10–13) optimizes the traverse strategy to obtain the frequent itemsets in the systolic tree. Once again, a simultaneous BFS and DFS strategy is implemented as indicated by lines 10 to 13.

Algorithm 2 Parallel algorithm for finding frequent itemsets in a systolic tree

Require: Flushing flag $flush$, Minimum support value min_sup and $systolic\_tree$.
Ensure: Frequent itemsets and their frequency counting $< itemset, frequency >$.

```
 1: procedure MAIN
 2:    $r \leftarrow systolic\_tree.RootNode$;    ▷ Set $r$ to point to the root.
 3:    if ($flush == true$) then
 4:      FlushMethod($r$, $min\_sup$);    ▷ Starts the flushing procedure recursively.
 5:    end if
 6: end procedure

 7: procedure FLUSHMETHODnode $r$, minimum support threshold $min\_sup$
 8:    if ($r.Counter \geq min\_sup$) then    ▷ $r$ is frequent.
 9:      if(($r.ChildNode != null$) and ($r.RightNode != null$) then
10:        StartParallelBlock:    ▷ Parallel processing begins.
11:             FlushMethod($r.ChildNode$, $min\_sup$);
12:             FlushMethod($r.RightNode$, $min\_sup$);
13:        EndParallelBlock;    ▷ Parallel processing ends.
14:        $r.GatewayMode = true$;    ▷ Puts $r$ to Gateway processing mode.
15:      else
16:        $r.GatewayMode = false$;    ▷ Puts $r$ to Counting processing mode.
17:      end if
18:      flushResult.Add($r$, $r.Counter$);    ▷ Stop recursion and returns the frequency value.
19:    end if
20: end procedure

21: return $flushResult$;
```

4.2 Sliding Window Model-based approach

There are applications where it is more useful to analyze recently received data in data streams instead of analyzing all data transmitted (Bai-En et al. 2012; Cheng et al. 2008; Golab and Özsu 2003). In these cases, Sliding Window Model is more suitable to be used because this model maintains the window size constant and allows to handle the temporal behavior of data streams. Less-occurring itemsets can be excluded from the mining process to maximize the performance of the proposed processing scheme. As a pre-processing stage, the software-side finds the top-*k* frequent 1-itemsets. The basic idea is to modify the initial processing scheme proposed making that the software-side performs the top-*k* frequent 1-itemsets detection and sliding window creation and maintenance. Frequent itemsets are discovered in the hardware-side using a systolic tree capable of handling the sliding processing window. Figure 7 shows the modified processing scheme.

4.2.1 *Frequent 1-itemsets detection*

Data streams are data sources that usually evolve over time. This is caused by the behavior of transmitted items which are not static. Because of this, it is unrealistic and meaningless in several applications to determine all frequent itemsets. Also in such cases, changes in patterns and their trends are more interesting than patterns themselves.

Detecting changes in transmitted patterns in data streams were studied in several ways, and one of these ways is to find the most occurring 1-itemsets (also named *heavy hitters*) (Cormode and Hadjieleftheriou 2009; Metwally et al. 2006, 2005; Thanh and Calders 2010). This problem has also been attacked from the hardware perspective (Lai et al. 2010; Tong and Prasanna 2013). In data streams mining the detection of top-*k* frequent 1-itemsets can be seen as a pre-processing stage where the most representative itemsets in the stream are discovered. Using the top-*k* frequent 1-itemsets as starting point reduces the search space because only participate in mining process those 1-itemsets that were frequent. Also, frequent 1-itemsets detection is useful to verify a concept drift in streams (Jiang and Gruenwald 2006), which is an issue that will be addressed in future stages of this research. Processing data streams using top-*k* frequent 1-itemsets was used in (Lee et al. 2000) and (Baralis et al. 2011). The validity of using top-frequent 1-itemset detection is supported on the Downward Closure property (Agrawal and Srikant 1994). Using this pre-processing, all the items detected as infrequent will be removed avoiding to use hardware resources to process data that will not produce any frequent itemset.

4.2.2 *Window creation and maintenance*

In this Section, the Sliding Window Model creation process is explained. Sliding Window Model (as it was described in Section 2) can be seen as a FIFO queue: older transactions are excluded while new ones arrive. The window creation process is performed in the

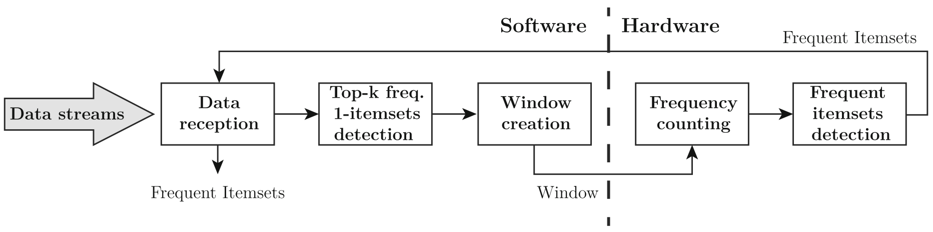


Fig. 7 Modified hardware-software scheme proposed for Frequent Itemsets Mining on data stream using Sliding Window Model

software-side of the proposed architecture: for a window W in a Sliding Window Model with $|W|$ window length and overlapping s, a *Processing Window* P is defined as:

$$P = (W_i \backslash s) \cup (W_{i+1} \backslash s) \tag{1}$$

where

$$s = \left(W_i \cap W_{(i+1)}\right) \tag{2}$$

Taking expression 1 and expression 2 into account, size of P is defined by:

$$|P| = 2 \times (|W| - |s|) \tag{3}$$

i.e. the mining process only needs to know which transactions will be excluded from W (and therefore, frequency counting of itemsets involved in W will be decreased) and which transactions will be included in W (and therefore, frequency counting of itemsets involved in W will be increased). Those transactions that remain in W will not be counted again, and itemsets involved will not be updated. Figure 8 shows the diagram of a processing window P for windows W_i and $W_{(i+1)}$, both with size of 5, and overlapping of size 3.

When the top$-k$ frequent 1$-$itemsets are detected, then the resulting P is transmitted into the hardware. Transactions in P marked with "$-$" will be excluded (and the frequency of involved itemsets will be decreased) and those transactions marked with "+" will be included in the mining process (and the frequency of affected itemsets will be incremented). Therefore, a record of which itemsets must be updated is maintained without using others data structures.

4.2.3 Frequency counting and frequent itemsets detection

To handle Sliding Window Model, the Algorithm 1 was updated and a new algorithm (which its logic is shown as follows) is proposed. For each transaction t in $processing\_window$ of Sliding Window Model:

1. If the itemset X of t arrives to an empty node r, then r will be occupied by the item $\{i\} \subseteq X$ and its frequency will be set to 1.
2. If the itemset X of t arrives to an occupied node r then one of the following decision should be taken:
   (a) If the node r is occupied by an item $\{i\} \subseteq X$ then:
      (i) If t is marked as positive ("+") then the frequency counter of r is incremented and the resulting itemset $\{X - \{i\}\}$ is flowed to its child and right nodes.
      (ii) If t is marked as negative ("$-$") then the frequency counter of r is decremented and the resulting itemset $\{X - \{i\}\}$ is flowed to its child and right nodes.
   (b) If the item $\{i\} \nsubseteq X$ then X is flowed to the right node of r.

Similar to Algorithm 1, Algorithm 3 is executed in each processing node of the systolic tree. After the frequency of each itemset is calculated, those itemsets that can be regarded as frequent are determined using Algorithm 2.

Algorithm 3 Parallel algorithm for frequency counting using Sliding Window Model

Require: Processing window P using Sliding Window Model.
Ensure: Systolic tree with the frequency counting for each itemset.

```
 1: procedure MAIN
 2:     $r \longleftarrow systolic\_tree.RootNode$;   ▷Set $r$ to point to root node of the systolic tree.
 3:     for each transaction $t$ in $P$ do
 4:       Traverse($t, r$);   ▷Flow received transactions through the root node.
 5:     end for each
 6: end procedure

 7: procedure TRAVERSEtransaction $t$, node $r$
 8:     if$r.IsOccupied == false$ then
 9:       $r.Item = t.ItemAt(0)$;   ▷Stores the first item of $t$ in $r$.
10:       $r.IsOccupied = true$;   ▷Marks $r$ with the $occupied$ flag.
11:       FlushInParallel($t, r$);   ▷$t$ is processed in parallel.
12:     else
13:       if $t.Contains(r.Item) == true$ then   ▷Verifies if $t$ contains the item stored in $r$.
14:         FlushInParallel($t, r$);   ▷In positive case, $t$ is processed in parallel.
15:       else
16:         $\widetilde{r} \leftarrow r.RightNode$;   ▷$t$ is flowed only to the right node of $r$.
17:         Traverse($t, \widetilde{r}$);   ▷$t$ is processed in parallel in the right node of $r$.
18:       end if
19:     end if
20: end procedure

21: procedure FlushInParalleltransaction $t$, node $r$
22:     if $t.IsPositive() == true$ then
23:       $r.Counter ++$;   ▷ Frequency counting of $r$ is incremented.
24:     else
25:       $r.Counter --$;   ▷Frequency counting of $r$ is decremented.
26:     End If
27:     $\widetilde{t} = t.Exclude(r.Item)$;   ▷$t$ is reduced.
28:     if$\widetilde{t}.IsEmpty == false$ then
29:         StartParallelBlock:   ▷Parallel processing begins.
30:           $\widetilde{r} \longleftarrow r.ChildNode$;
31:           Traverse($\widetilde{t}, \widetilde{r}$);
32:           $\widetilde{r} \longleftarrow r.RightNode$;
33:           Traverse($\widetilde{t}, \widetilde{r}$);
34:       EndParallelBlock;   ▷Parallel processing ends.
35:     end if
36: end procedure

37: return $systolic\_tree$;
```

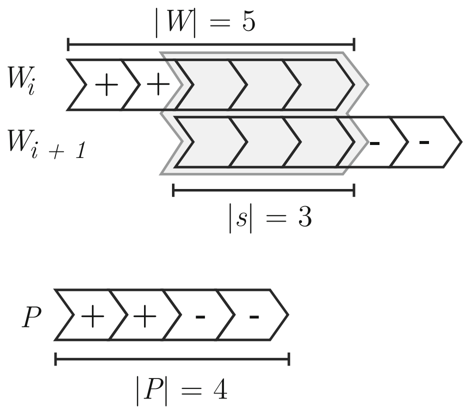


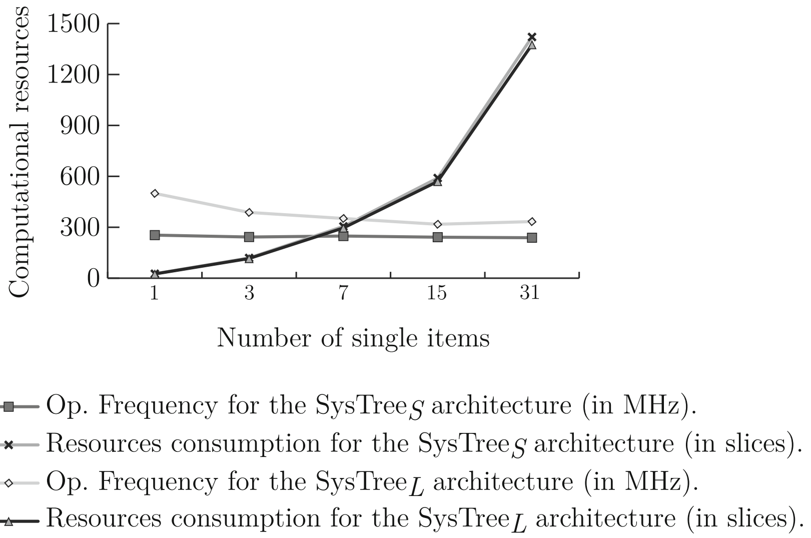
Fig. 8 Diagram of the sliding window creation and maintenance process

5 Results

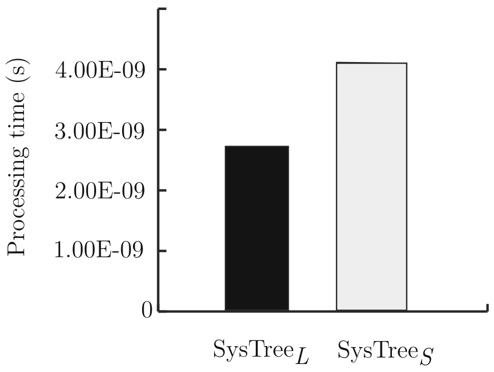
The architectures that implement Algorithms 1, 2 and 3 were described using the VHDL language with the Xilinx ISE Suite 14.2; and targeted for the Virtex 5 XC5VLX330T FPGA device. After the architectures were synthesized and implemented, the *SysTree*$_L$ architecture (which implements the systolic tree for Landmark Window Model) operates at 377.96 MHz, while the *SysTree*$_S$ architecture (which implements the systolic tree for Sliding Window Model) operates at 244.89 MHz. Also, the processing units in the SysTree$_L$ architecture employs 17 slices versus 21 slices occupied in the SysTree$_S$ architecture. This is caused by the conditional path for detecting when a transaction should be removed (those transactions marked with "$-$" symbol) or included (those transactions marked with "$+$" symbol) from/in the processing window. In hardware, the high cost of conditional structures is responsible for having different size of processing units in the SysTree$_L$ and the SysTree$_S$ architectures. For the selected hardware device, using the SysTree$_L$ architecture, a maximum of 8294 processing nodes using the 98.76% of available hardware resources can be placed. In the SysTree$_S$ architecture, the processing units are bigger than the processing units in SysTree$_L$. Therefore, the maximum number of processing nodes that can be placed is 7975, representing the 94.96% of available hardware resources.

Two scenarios were employed to validate the viability (regarding throughput and accepted transmission rate) of the presented architectures. One of such scenario uses real-life data streams and different windows sizes, and the other uses known data sets and synthetic data streams.

Experiments conducted in the first scenario contributes to determining that the *SysTree*$_L$ architecture, working at a maximum operating frequency of 377.96 MHz, can process 2.9×10^7 transactions per second, obtaining a throughput of 1.88 Gbps. The *SysTree*$_S$ architecture, at a maximum operating frequency of 244.89 MHz, can process 2.04×10^7 transactions per second, obtaining a throughput of 1.2 Gbps. Also, several windows sizes were used concluding that the window size does not affect the processing time per item in data streams. The window size influences on detected frequent itemsets because shorter windows contain fewer transactions (and therefore the frequency counting of existing items is affected) compared than larger windows. Nevertheless, it's hard to evaluate the obtained



(a) Computational resources required by the architecture that employs the Landmark Windows Model ($SysTree_L$) and the architecture that employs the Sliding Window Model ($SysTree_S$).



(b) Processing times for the SysTree$_L$ and the SysTree$_S$ architectures without the Top-k 1−frequent items detection preprocessing.

Fig. 9 Synthesis results for the SysTree$_L$ and the SysTree$_S$ architectures

patterns using real-life data streams compared with other reference algorithms and software since no enough implementation details neither source code were given.

The second scenario was used to compare the obtained patterns with other algorithms and software reported in the reviewed literature (Agrawal and Srikant 1994; Zaki 2000; Han et al. 2000). In such way, considering the same windows size and the same preprocessing stage, it was determined that the obtained frequent itemsets using the proposed architectures

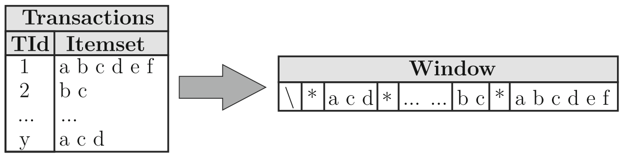


Fig. 10 A dataset can be considered as a data stream after the proper modifications. The symbol "∗" means an end of transactions while the symbol "\" means the end of processing windows

were the same compared with those obtained using the baseline algorithms. In such way, it was decided to use the second scenario to report the obtained results.

Figure 9a shows the resources consumption for the SysTree$_L$ and the SysTree$_S$ architectures. As it was expected, both grow as the number of single items in the incoming stream increase. Although the maximum operating frequency decreases with the increasing of single items, it tends to be stabilized, and the maximum operating frequency can be assumed as constant. Figure 9b shows the processing time needed for each architecture to process one item.

In the revised literature, no FPGA-based architectures for Frequent Itemsets Mining on data streams were reported. Nevertheless, some architectures for Frequent Itemsets Mining in datasets have been reported. Comparing the results reported in this paper versus other software-based approaches for this task is not feasible. Thus, to develop a fair comparison, it was decided to evaluate obtained results versus reported FPGA-based architectures regardless if they were oriented to datasets or data streams. It can be said, without loss of generality, that a dataset can be treated as a data stream. Figure 10 depicts this idea showing a dataset composed by n transactions. Rearranging the transactions and introducing some control symbols to indicate where a transaction and a window ends (symbols * and \respectively), any dataset can be seen as a data stream.

Several experiments were conducted to validate the performance of the proposed architectures. In these experiments, the same datasets used in the reviewed papers were selected. It is valid to say that in the reviewed papers, authors do not offer any implementation details that allow to replicate their experiments and therefore, the comparison was made versus their published results. Three of the selected datasets were taken from UCI repository (Lichman 2013) (*Chess, Accidents* and *Retail*) and other two were created using the Almaden IBM Synthetic Dataset Generator[2] (*T10I4D100K* and *T40I10D100K*). Datasets are described in Table 2. In this table, column *Dataset* lists the used datasets. Column *Size* shows the size of the datasets in megabytes while column *Trans.* shows the number of transactions that composes the datasets. Column *Items* shows the number of items that compose the alphabet, and columns *Min.T.* shows the length of the shortest transaction, *Max.T.* shows the length of the larger transactions and *Ave.T.* shows the average length of transactions.

It can be seen in Fig. 9 that both architectures have the same behavioral pattern, and SysTree$_S$ architecture outperforms SysTree$_L$ although SysTree$_S$ has an operation frequency lower than SysTree$_L$. Such behavior is caused by the fact that the SysTree$_S$ architecture filters the incoming data stream detecting the top-k frequent $1-$itemsets.

Figures 11a, 12 and 13c show the timing results achieved for the proposed architectures with various values of support threshold (expressed in percent and number of items). In these figures, the comparisons were separated by each algorithm and each dataset for the sake of clarity. The processing times for the SysTree$_L$ and the SysTree$_S$ were scaled in figures.

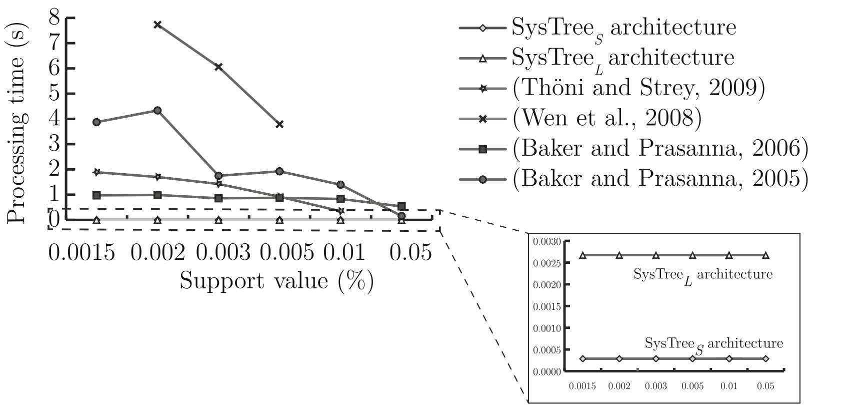
[2]http://www.cs.loyola.edu/cgiannel/assoc_gen.html

Table 2 Description of the selected data sets

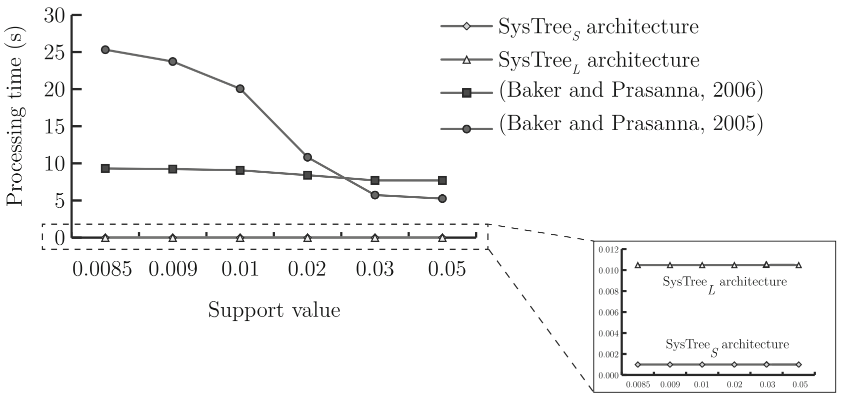
Dataset	Size(MB)	Trans.	Items	Min.T.	Max.T.	Ave.T.
T10I4D100K	4	100 000	870	1	29	11
T40I10D100K	15	100 000	942	4	77	40
Chess	0.34	3 196	75	37	37	37
Retails	3.97	88 162	16 470	1	76	11
Accidents	33.8	340 183	468	18	51	34

5.1 Discussion

Algorithms for frequent itemsets mining described in the state-of-the-art only report memory consumption and processing time. For frequent itemsets mining, it is assumed that the number of frequent itemsets, and therefore, the frequent itemsets detected, are the same no



(a) Processing times obtained using the T10I4D100K dataset.



(b) Processing times obtained using the T40I10D100K dataset.

Fig. 11 Processing times for the SysTree$_L$ and the SysTree$_S$ architectures versus the hardware architectures reported in the state-of-the-art that implement the Apriori algorithm

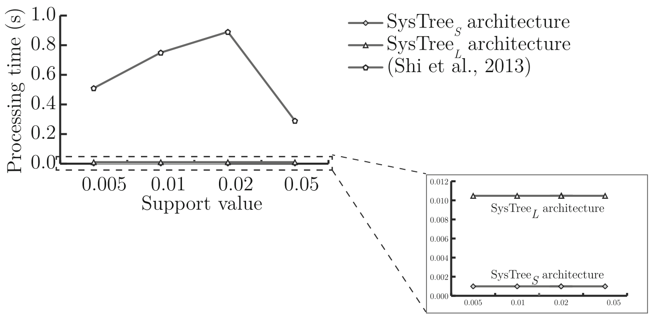


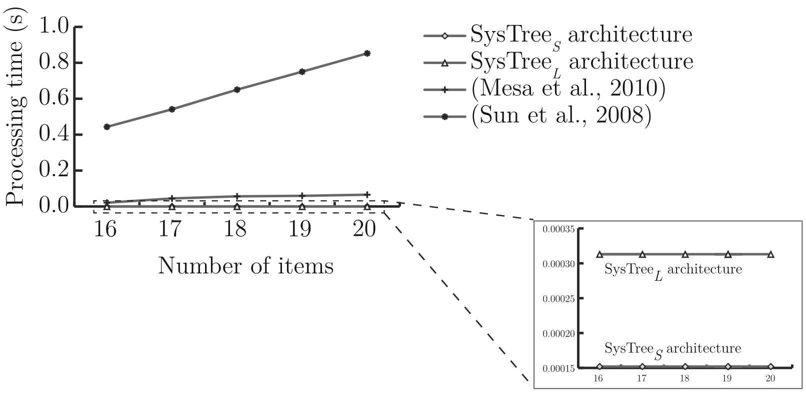
Fig. 12 Processing times for the SysTree$_L$ and the SysTree$_S$ architectures versus hardware architectures reported in the state-of-the-art that implement the ECLAT algorithm using the T40I10D100K dataset

matter which algorithm was used as the baseline. Considering this fact, in this paper, the processing time and hardware resources consumption are used as the evaluation metric. As it is shown in Figs. 11a, 12 and 13c, the processing time of the proposed architectures outperforms all the baseline algorithms. The tree-based data structure adopted allows accessing the data in one pass (fulfilling with the Expiration constraint) at a high rate. As it was demonstrated by the experiments conducted, the processing times achieved meet the Continuity constraint of data streams. Also, no information or any other data related to items (except how often they are found in a processing window) in the transactions flowed is stored in the nodes and therefore, the memory needed to mine the incoming data streams is constant fulfilling the Infinity restriction.

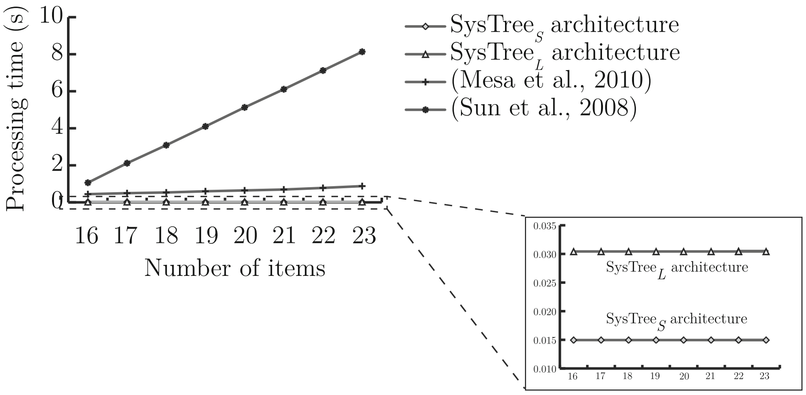
Experiments conducted also demonstrate that the proposed architectures are insensitive to variations in the support threshold value. This is a highly valuable feature of the proposed algorithms since all previous hardware and software implementations of Frequent Itemsets Mining algorithms are deeply tight to the minimum support threshold selected. In the proposed architectures, all hardware resources needed for mining incoming data streams are available and remain invariant regardless of the chosen minimum support threshold value. A high value of the support threshold results in less frequent itemsets while a low support threshold results in more frequent itemsets, however, the processing time remains the same. Here, two main advantages of the proposed hardware architectures can be highlighted: (1) The hardware resources needed to mine some incoming data streams are independent of the support threshold, and (2) The processing time required is independent of the size of the frequent itemsets found in the incoming data stream. Also, the proposed algorithms obtain the same frequent itemsets (and the same frequency counting) as all other algorithms of the state-of-the-art described in Section 3.

The window models establish how a data stream should be segmented to perform the data mining tasks. Those models and the windows sizes are selected concerning the nature of the problems faced. Larger windows conduce to a large number of itemsets while short windows conduce to fewer itemsets (the number of frequent itemsets is related to the windows size). Because of that, no window size is better than other: each of them gives to the mining process a different meaning. In this paper, to get more compatibility with the baseline algorithms used in the comparison, the window size was equal to the dataset size. In such way, frequent itemsets were obtained under equal conditions.

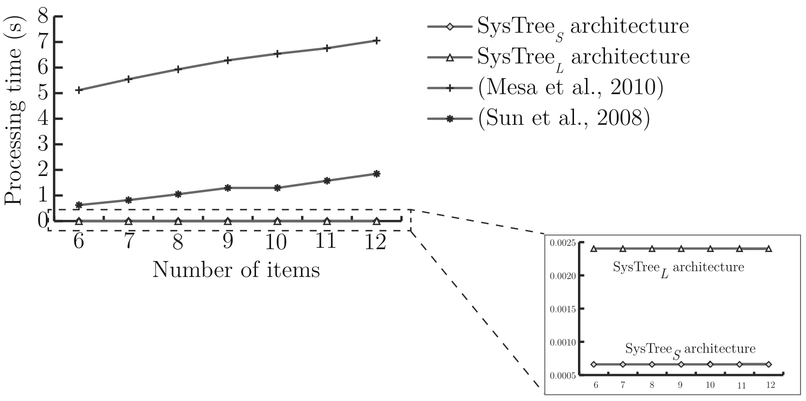
The dimension of the systolic tree is determined by $|I|$, where I is the alphabet of items in the incoming data stream (see Section 2). Since $|I|$ cannot be established a priori (due to the Infinity constraints), the size of the systolic tree will be determined by the capacity of the



(a) Processing times obtained using the Chess dataset.



(b) Processing times obtained using the Accidents dataset.



(c) Processing times obtained using the Retails dataset.

Fig. 13 Processing times for the SysTree$_L$ and the SysTree$_S$ architectures versus the hardware architectures reported in the state-of-the-art that implement the FP-Growth algorithm

development platform. Assuming the development platform contains enough computational resources (ideal case), the size of the systolic tree (in the number of nodes) that can hold any streams formed by items of I is:

$$nodes = 2^{|I|} - 1 \tag{4}$$

In real cases, the available resources in a FPGA are limited. Using a hardware device H and supposing a 100% of device area occupation, let k be the maximum number of processing nodes that can be supported by H. Then the maximum number max_{items} of items in I arrived in the incoming data stream that can be handled by H will be:

$$max_{items} = \log_2(k + 1) \tag{5}$$

For example, if H can hold 1023 nodes in the systolic tree, then the maximum number of items of I that can be handled by H will be 10.

It is important to notice that for a certain number p of items in set I, if $2^p - 1 > k$ (where k is the maximum number of processing nodes available in H) the systolic tree cannot hold all possible combinations of itemsets and therefore some itemsets will not be taken into account during the mining process. In other words, the number of processing nodes needed to handle all possible combinations of itemsets generated from I exceeds the maximum number of processing nodes that can be mapped into the selected FPGA. Here, the mining will be approximate with no false positives produced. In this case, the selection of top$-k$ frequent 1-itemsets allows obtaining the most valuable items to generate frequent itemsets. Collecting some information about the incoming stream introduces some delay (which is negligible) in the mining process, but it implies higher quality in the produced itemsets. If the systolic tree is occupied by items in arrival order as was done in the SysTree$_L$ architecture, it cannot be guaranteed that all of the received items in a transaction will produce frequent itemsets (nodes of the systolic tree will be occupied by items which will not be frequent). In such case, the use of the systolic tree is not optimal. By the contrary, if the systolic tree is occupied by items that had been proved to be frequent (from the preprocessing stage), the use of hardware resources needed to mine a transaction is more efficient. Using this strategy the mining process will also be approximate, and from users, it is more useful to obtain "some" frequent itemsets than to obtain "all" frequent itemsets (this fact is supported in (Metwally et al. 2005)). In this paper, using the top-k frequent 1-itemsets ensures that frequent itemsets obtained are those that better describe the data stream behavior instead of just using the $k-$first (or all) arrived 1-itemsets. On the contrary, if $2^p - 1 \leq k$ then the systolic tree can hold all the possible combinations of itemsets formed by items of I and therefore the mining process will be exact.

6 Conclusions

Frequent itemsets are widely used in data mining applications. Hence, many researchers are currently focused on proposing methods to improve and accelerate the implementations of such methods as the amount of data produced increases every day. Nowadays, data streams have gained more interest by the community due to its applications, and Frequent Itemsets Mining are being used to obtain new knowledge from those continuous data flows. Classic Data Mining methods are ineffective when they deal with data streams. Data streams can be seen as a particular case of datasets except that the data streams processing introduce some restrictions that make them particularly difficult to process .

In this paper, the general problem of finding frequent itemsets was decomposed into four subproblems for better understanding. Considering such separation, new parallel algorithms for Frequent Itemsets Mining on data streams were proposed for solving the second subproblem. The proposed algorithms use Landmark and Sliding Window Models and were designed to be implemented in custom hardware-software architecture. They were based on a systolic tree approach where the control logic is distributed among all processing nodes. Here, the top$-k$ frequent $1-$itemsets selection allows to optimize the use of the systolic tree and to obtain the most valuable frequent itemsets. Experimental results show that the presented algorithms can extract frequent itemsets from data streams with a significant speed up when they are compared against others hardware-based implementations reported in the state-of-the-art. When the proposed algorithms are executed in a device with no resource restrictions or the number of items in $|I|$ can be handled by the selected hardware, then the exact mining process is performed. That is, all itemsets and its frequency counting are obtained in a precise way as if they were calculated with any of the state-of-the-art algorithms. By the contrary, when available hardware resources are restricted, then an approximate mining process with no false positives is performed. That is, some itemsets will be excluded from the analysis, but those that remain will be detected as frequent, and their frequency counting is obtained in an exact way as they were calculated with any of the state-of-the-art algorithms. Also, the proposed architectures are insensitive to changes in the support threshold and incoming data streams sizes, which is highly valuable in Data Mining tasks.

References

Aggarwal, C., & Han, J. (2014). *Frequent pattern mining*. Springer International Publishing.

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th international conference on very large data bases* VLDB ’94 (pp. 487–499). San Francisco.

Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002). Models and issues in Data Stream systems. In *Proceedings of the 21th ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems*, PODS ’02 (pp. 1–16). New York: ACM.

Bai-En, S., Philip, S., & Vincent, S. (2012). Efficient algorithms for mining maximal high utility itemsets from Data Streams with different models. *Expert Systems with Applications*, *39*(17), 12,947–12,960.

Baker, Z., & Prasanna, V. (2005). Efficient hardware Data Mining with the Apriori algorithm on FPGAs. In *Proceedings of the 13th annual IEEE symposium on field-programmable custom computing machines*, FCCM ’05 (pp. 3–12). Washington: IEEE Computer Society.

Baker, Z., & Prasanna, V. (2006). An architecture for efficient hardware data mining using reconfigurable computing systems. In *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2006*. FCCM ’06 (pp. 67–75).

Baralis, E., Cerquitelli, T., Chiusano, S., Grand, A., & Grimaudo, L. (2011). An Efficient Itemset Mining Approach for Data Streams. In Konig, A., Dengel, A., Hinkelmann, K., Kise, K., Howlett, R., & Jain, L. (Eds.) *Knowlege-based and intelligent information and engineering systems, lecture notes in computer science* Vol. 6882 pp 515–523. Berlin: Springer.

Bustio, L., Cumplido, R., Hernández, R., & Bande, J.M. (2015). Feregrino, C.: 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* (pp. 14–26). Portugal: Porto.

Cameron, J., Cuzzocrea, A., Jiang, F., & Leung, C. (2013). Mining Frequent Itemsets from sparse data streams in limited memory environments. In *Web-age information management, lecture notes in computer science* (Vol. 7923 pp. 51–57). Berlin: Springer.

Cheng, J., Ke, Y., & Ng, W. (2008). A survey on algorithms for mining frequent itemsets over data streams. *Knowledge and Information Systems*, *16*(1), 1–27.

Compton, K., & Hauck, S. (2002). Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, *34*(2), 171–210.

Cormode, G., & Hadjieleftheriou, M. (2009). Finding the Frequent Items in streams of data. *Communications of the ACM*, *52*(10), 97–105.

Giannella, C., Han, J., Pei, J., Yan, X., & Yu, P. (2003). Mining frequent patterns in Data Streams at multiple time granularities. *Next Generation Data Mining*, *212*, 191–212.

Golab, L., & Özsu, T. (2003). Data Stream Management Issues–A Survey. Tech. rep., Apr. 2003 https://cs.uwaterloo.ca/tozsu/ddbms/publications/stream/streamsurvey.ps.

Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on management of data, SIGMOD '00* (pp. 1–12). New York: ACM.

Jiang, N., & Gruenwald, L. (2006). Research issues in Data Stream association rule mining. *SIGMOD Record*, *35*(1), 14–19.

Jin, R., & Agrawal, G. (2007). Frequent Pattern Mining in Data Streams. In *Data Streams, advanced in database systems* (Vol. 31 pp. 61–84). Springer.

Lai, Y., Wang, N., Chou, T., Lee, C., Wellem, T., & Nugroho, H. (2010). Implementing on-line sketch-based change detection on a NETFPGA platform. In *1st Asia netFPGA developers workshop*.

Lee, W., Stolfo, S., & Mok, K. (2000). Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, *14*(6), 533–567.

Lichman, M. (2013). UCI Machine learning repository. http://archive.ics.uci.edu/ml. Accessed: 2015-06-20.

Manku, G.S., & Motwani, R. (2002). Approximate frequency counts over Data Streams. In *Proceedings of the 28th international conference on very large data bases, VLDB '02* (pp. 346–357). VLDB endowment.

Mesa, A., Feregrino-Uribe, C., Cumplido, R., & Hernández-Palancar, J. (2010). A Highly Parallel Algorithm for Frequent Itemset Mining. In *Advanced in pattern recognition, lecture notes in computer science*, vol. 6256, (pp. 291–300). Berlin: Springer.

Metwally, A., Agrawal, D., & Abbadi, A. (2006). An integrated efficient solution for computing frequent and top-k elements in Data Streams. *ACM Transactions Database Systems*, *31*(3), 1095–1133.

Metwally, A., Agrawal, D., & Abbadi, A.E. (2005). Efficient computation of frequent and top-k elements in Data Streams. In *Database Theory - ICDT 2005, no. 3363 in lecture notes in computer science*, (pp. 398–412). Berlin: Springer.

Shaobo, S., Yue, Q., & Qin, W. (2013). Accelerating intersection computation in Frequent Itemset Mining with FPGA. In *2013 IEEE 10th International conference on embedded and ubiquitous computing - HPCC-EUC high performance computing and communications 2013* (pp. 659–665).

Song, S., Steffen, M., & Zambreno, J. (2008). A reconfigurable platform for Frequent Pattern Mining. In *International conference on reconfigurable computing and FPGAs, 2008. reconfig '08* (pp. 55–60).

Song, S., & Zambreno, J. (2008). Mining association rules with Systolic Trees. In *International conference on field programmable logic and applications, 2008. FPL 2008* (pp. 143–148).

Song, S., & Zambreno, J. (2011). Design and Analysis of a Reconfigurable Platform for Frequent Pattern Mining. *IEEE Transactions on Parallel and Distributed Systems*, *22*(9), 1497–1505.

Sun, Y., Wang, Z., Huang, S., Wang, L., Wang, Y., Luo, R., & Yang, H. (2014). Accelerating Frequent Item Counting With FPGA. In *Proceedings of the 2014 ACM/SIGDA international symposium on field-programmable gate arrays, FPGA '14* (pp. 109–112). New York: ACM.

Teubner, J., Müller, R., & Alonso, G. (2010). FPGA Acceleration for the Frequent Item Problem. In F. Li, M.M. Moro, S. Ghandeharizadeh, J.R. Haritsa, G. Weikum, M.J. Carey, F. Casati, E.Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, V.J. Tsotras (Eds.) *2010 IEEE 26th International Conference on Data Engineering (ICDE)* (pp. 669–680). IEEE.

Teubner, J., & Müller, R. (2011). Alonso, G.: Frequent Item Computation on a Chip. *IEEE Transactions on Knowledge and Data Engineering*, *23*(8), 1169–1181.

Thanh, L., & Calders, T. (2010). Mining Top-k Frequent Items in a Data Stream with Flexible Sliding Windows. In *Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining KDD '10* (pp. 283–292). New York: ACM.

Thöni, D., & Strey, A. (2009). Novel Strategies for Hardware Acceleration of Frequent Itemset Mining with the Apriori Algorithm. In *FPL 2009. International Conference on Field programmable logic and applications, 2009* (pp. 489–492).

Tong, D., & Prasanna, V. (2013). Online Heavy Hitter Detector on FPGA. In *2013 International conference on reconfigurable computing and FPGAs (reconfig),* (pp. 1–6). IEEE.

Wen, Y., Huang, J., & M.S., C. (2008). Hardware-Enhanced Association Rule Mining with Hashing and Pipelining. *IEEE Transactions on Knowledge and Data Engineering*, *20*(6), 784–795.

Yamamoto, K., Ikebe, M., T., A., & Motomura, M. (2016). FPGA-Based Stream Processing for Frequent Itemset Mining with Incremental Multiple Hashes. *Circuits and System*, *7*(10), 3299–3309.

Zaki, M. (2000). Scalable Algorithms for Association Mining. *IEEE Transactions on Knowledge and Data Engineering*, *12*(3), 372–390.

Zhang, Y., Zhang, F., Jin, Z., & Bakos, J. (2013). An FPGA-Based Accelerator for Frequent Itemset Mining. *ACM Transactions Reconfigurable Technology Systems*, *6*(1), 2:1–2:17.