CICLAD: A Fast and Memory-efficient Closed Itemset Miner for Streams

Tomas Martin, Guy Francoeur, Petko Valtchev
Centre de recherche en intelligence artificielle (CRIA), UQÀM
Montréal, Québec, Canada

Abstract
Mining association rules from data streams is a challenging task due to the (typically) limited resources available vs. the large size of the result. Frequent closed itemsets (FCI) enable an efficient first step, yet current FCI stream miners are not optimal on resource consumption, e.g. they store a large number of extra itemsets at an additional cost. In search for a better storage-efficiency trade-off, we designed Ciclad, an intersection-based sliding-window FCI miner. Leveraging in-depth insights into FCI evolution, it combines minimal storage with quick access. Experimental results indicate Ciclad’s memory imprint is much lower and its performances globally better than competitor methods.

1 Introduction
Association rule (AR) and the component frequent itemset (FI) mining from data streams have found a wide range of practical applications, e.g. in network traffic analysis, click stream mining, online transaction analysis [9, 13, 21].

Both task are challenging due to the specific conditions of the stream environment: dynamic data inflow, potential concept drift and, typically, limited resources [12, 26]. Therefore, the design of efficient stream miners has to reflect concerns such as single access to stream data, compact storage of results, resource limit-awareness, etc. Moreover, various processing models have been investigated to reflect decay of data in the stream, such as landmark, sliding or damped window. In a different vein, while a large number of stream miners target interesting itemsets (frequent [16], rare [10], closed [4], maximal [13], self-sufficient [23], etc.), only few methods would maintain the associations, arguably due to the huge number of rules to process on each window shift.

Our ultimate goal is the design of an efficient AR stream miner, whereby we propose to tackle the inherent complexity of the task by reducing the output to some concise representation of strong rules, e.g. as described in [14]. Such representations typically constructed on top of distinguished itemsets such as closed itemsets (CIs), generators (a.k.a. free itemsets), or maximal Fls [2], and might exploit additional structure on those, e.g. lattice links [29]. A variety of batch methods exist for these representations (e.g. see [2, 14]), yet to the best of our knowledge they have not been studied in stream settings. Stream miners have been designed for (frequent) CIs [4, 11, 15, 24, 25, 27] as well as for frequent generators [6]. The former split into two categories: Methods [4, 15] adapt batch pattern enumeration [28], as opposed to those relying on CI intersection [11, 24, 25, 27]. Intersection-based closure computing is rooted in formal concept analysis (FCA) [5], where a variety of incremental, i.e. landmark, miners target closed itemsets (CIs) (dating back to [7]). Moreover, various derived problems of interest, e.g. mining CIs with links and generators, have been studied [20]. However, resource-awareness is not a prime concern in FCA, and neither is data decay, hence no decremental methods have been designed (to achieve a sliding window mode).

In this paper, we focus on the groundwork task to support the design of fully-blown concise AR stream miner, i.e. the online mining of FCIs. Our analysis of the literature indicates that all existing methods have increased memory consumption as they need to store extra itemsets on top of the target FCIs: The first group requires few infrequent and large number of non closed itemsets whereas the second one maintains all infrequent CIs. However, pattern enumeration miners additionally maintain tidsets, which is costly, especially on large or highly dynamic streams, whereas intersection-based ones skip tids altogether.

We chose to follow an intersection-based approach which, despite the aforementioned overhead due to infrequent CI maintenance, offers distinct advantages such as higher flexibility (e.g. upon support threshold decreases) and versatility (both frequent and rare patterns targeted). Moreover, the CI indexing approach used in both [25, 27] seems particularly appealing, yet both methods suffer on superflo US storage and/or processing.

As a remedy, we propose Ciclad, a two-fold CI stream miner whose incremental part Ciclad⁺ streamlines quick access to CIs and item-wise intersection growth from [25] while skipping non essential storage of itemsets. The decremental Ciclad⁻, in turn, implements some novel insights into CI evolution that allow it to fit the same overall computing schema as Ciclad⁺. The resulting homogeneous compound method achieves both high efficiency and low memory usage. This has been confirmed by a validation study over both real and synthetic datasets (retail, network security, etc.) whose outcome shows that Ciclad outperforms competing methods by a comfortable margin on all but the most dense datasets.

Our paper’s contributions are as follows: (1) formalization of transaction removal, (2) a unified intersection-based sliding-window miner Ciclad, (3) performance study on a variety of CI mining methods for streams (our code and
datasets publicly available\(^1\). Additionally, we provide correctness proofs enhancing \([25]\) (see Appendix).

In what follows, section 2 provides background on CIs and online mining while section 3 summarizes previous work. Our mathematical approach and the algorithmic details of Ciclad are presented in sections 4 and 5, respectively. Section 6 summarizes the performance study. Concluding remarks are given in section 7.

## 2 Background

Below, we recall basics of pattern mining, closed patterns and stream mining.

### 2.1 Frequent (closed) itemsets

Assume a transaction database \(D\) (as in Table 1) defined on top of a set of items \(I = \{a_1, a_2, \ldots, a_n\}\) (here \(\{a, \ldots, h\}\)). A set \(X \subseteq I\) is called an itemset while a transaction is a pair \((tid, \text{itemset})\) where \(tid\) is a transaction identifier. Similarly, a set of tids is called a tidset.

| tidset | itemset | tidset | itemset |
|--------|---------|--------|---------|
| \(1\) | \(\{abce\}\) | \(5\) | \(\{g\}\) |
| \(2\) | \(\{abce\}\) | \(6\) | \(\{eh\}\) |
| \(3\) | \(\{cd\}\) | \(7\) | \(\{abcd\}\) |
| \(4\) | \(\{ef\}\) | \(8\) | \(\{bcgh\}\) |

Table 1. Sample data, further referred to as \(D_{10}\)

Given a tidset \(Y\) from \(D\), \(\iota_{D}(Y) = \bigcap\{Z|\{j, Z \in D, j \in Y\}\}\) denotes the itemsets shared by the respective tids. Conversely, the support set of an itemset \(X\) comprises tids whose itemsets cover \(X\), \(\tau_{D}(X) = \{j|\{j, Z \in D, X \subseteq Z\}\}\). For instance, \(\tau_{D}(ab) = \{1, 2, 7\}\) and \(\tau_{D}(\{2, 7\}) = ab\). For a tid \(j\) from \(D\), \(i(j)\) denotes its itemset: \(i(j) = Z\) iff \((j, Z) \in D\). To shorten notations, we omit the subscript \(D\) (if no confusion is possible) and use \(t_i\) to denote \(i_{\tau}(j)\)’s itemset.

Quality of an itemset \(X\) follows the size of its support set, a.k.a. its support \(\sigma(X) = |\tau(X)|\).

A binary frequency criterion is based on a pre-defined minimum support threshold, or \(\text{min}_\sigma\), denoted \(\varsigma\). The ensuing family of frequent itemsets in \(D\) will be \(\mathcal{F}(D, \varsigma)\). Support sets induce an equivalence relation on \(\phi(I): X \equiv Z\) if \(\tau(X) = \tau(Z)\) (e.g. \(ab \equiv ac\)). The equivalence class of \(X\), \([X]_{D}\) admits a unique maximum, a.k.a. closed itemset (CI) which, following the anti-monotony of \(\sigma\), can be defined as follows:

**Definition 1.** \(X \subseteq I\) is closed if no proper superset thereof has the same support.

In \(D\), \(abc\) is closed while \(b\) is not \(\tau(bc) = \tau(b)\). The CIs of \(D\) will be denoted \(C(D)\) and frequent ones \(\mathcal{F}(C(D, \varsigma))\).

In formal concept analysis (FCA) \([5]\), CIs, termed concept intents, are defined via a closure operator \(\kappa\) induced by \(C(D)\), \(\kappa : X \mapsto \text{max}(\{X\}_{D})\), hence \(X = \kappa(X)\) iff \(X\) is closed. Here, \(\kappa\) is nothing more than the composition of \(\iota\) and \(\tau\):

\[
\kappa(X) = \iota(\tau(X))
\]

Here, \(\kappa(ab) = abc\) (as \(\iota(\tau(ab)) = \iota(\{1, 2, 7\}) = abc\) and \(\kappa(ab) = \text{max}([ab]_{D})\)). Next, CIs are exactly the intersections of arbitrary sets of transactions \([5]\).

**Property 1.** Given a \(X \subseteq I\), \(\kappa(X) = \iota(\tau(X))\).

For example, in \(C(D)\) as given in Table 2, \(ef\) (CI 6), can be generated as \(6 = 2 \cap 6\), whereas \(\{2, 15, 22\}\) yields \(bc\) (CI 16).

As a corollary, \(C(D)\) is closed under \(\cap\) (and so is \(\mathcal{F}(C(D, \varsigma))\)).

### 2.2 Mining FCI\(_{s}\) over a stream

Stream pattern mining amounts to updating the pattern family of a window upon adding or removing a transaction (called increment and decrement, respectively). For instance, assume the transactions in Table 1 are acquired in the order of their tids. Let \(t_n\) be a new transaction not in \(D\) and let \(D^+ = D \cup \{t_n\}\). Simply put, the incremental update results in new CIs being added in \(C(D^+)\) and \(\sigma_{D^+}\) values computed for them as well as for some existing CIs (support sets extended by \(t_n\)). Indeed, following Property 2, no CI from \(D\) can vanish in \(D^+\) as \(C(D^+) = (C(D) \cup \{t_n\})\) entails \(C(D) \subseteq C(D^+)\). As an illustration, assume \(D_{10} = \{1, \ldots, 9\}\) (see Table 1) with its CI family \(C(D_{10})\) as given in Table 3 and \(t_{10} = 10\). Out of the latter two, the incremental method would output \(D_{10}\) (Table 2).

| tidset | itemset | tidset | itemset |
|--------|---------|--------|---------|
| 1      | \(\{abce\}\) | 7      | \(\{gh\}\) |
| 2      | \(\{abce\}\) | 8      | \(\{efgh\}\) |
| 3      | \(cf\)   | 9      | \(\{g\}\) |
| 4      | \(cd\)   | 10     | \(\{eh\}\) |
| 5      | \(f\)    | 11     | \(\{c\}\) |
| 6      | \(ef\)   | 12     | \(\{eh\}\) |
| 7      | \(fg\)   | 13     | \(\{cd\}\) |
| 8      | \(ef\)   | 14     | \(\{abc\}\) |
| 9      | \(g\)    | 15     | \(\{abcd\}\) |

Table 2. The family \(C(D_{10})\) (\(\sigma\) values behind ‘:\)

Algorithm-wise, as shown in \([25]\), all new CIs from \(C(D^+)\) – \(C(D)\) are generated as intersections of \(t_n\) with a CI from \(C(D)\), e.g. CI 20 \((gh)\) arises as \(10 \cap 8\). Let \(\Delta(D, t_n) = \{t_n \cap c | c \in C(D)\}\) denote the intersection set generated by \(t_n\).

Here, \(\Delta(D_{10}, t_{10}) = \{bcgh, egh, bc, gh, e, g, h\}\). Observe that some itemsets correspond to CIs from \(C(D_{10})\), e.g. \(bc\) is the CI 16 in \(D_{10}\). We shall call these CIs promoted, as opposed to new ones, and denote them \(C_{p}(D)\). It is readily shown that promoted CIs are exactly those included in the (itemset of) \(t_n\). Correspondingly, new intersections in \(\Delta(D, t_n)\), denoted

\[\Delta(D, t_n) = \{bcgh, egh, bc, gh, e, g, h\}\]

Footnote 1: https://github.com/guyfrancoeur/ciclad
$C_N(D)$, can only involve CIs that are not included in $t_n$. Observe that a new or a promoted CI, there may be multiple ways to generate an itemset $X \in \Delta(D, t_n)$, e.g. $bc$ is also $10 \cap 2$. In fact, $t_n$ induces an equivalence relation over $C(D)$ in which a class is defined as $[c]_{t_n} = \{ \bar{c} | \bar{c} \cap t_n = c \cap t_n \}$ for any CI $c$ (e.g. $8 \in [7]$). Clearly, each class is associated with some $X$ from $\Delta(D, t_n)$, safe the one gathering CIs that are disjoint with $t_n$ (of no interest here). In Figure 1 (section 4.2), the grey-filled table presents the equivalence classes associated to $\Delta(D_1, s)$, 10.

Crucially, each class $[ ]_{t_n}$ associated to some intersection $X$ has a distinguished member CI that canonically generates $X$ (CI in bold in Figure 1). This is the CI corresponding to $\kappa_D(X)$, the closure of $X$ in $D$, which is, provably, the minimum of the class (see [25]). Now, if $X$ is a promoted CI, then it is closed in $D$ ($X = \kappa_D(X)$), hence it equals the canonical member $X = \min_C \{ X \}$.

Otherwise, $X$ is new CI w.r.t. to $D$, hence non closed and strictly smaller than its closure ($X \subseteq \kappa_D(X)$) that is further called the genitor2 of $X$. In our example, $gh$ is a new CI in $D_1$ whose genitor is 7. The set of all genitors in $D$ will be denoted $C_G(D)$. To sum up, promoted, new and genitor CIs are defined as follows:

- $C_P(D) = \{ c | c \in C(D), c \subseteq t(t_n) \}$,
- $C_N(D) = \{ \bar{c} | \exists \bar{c} \in C(D), \bar{c} = c \cap t_n, \kappa_D(\bar{c}) \neq \bar{c} \}$,
- $C_G(D) = \{ c | c \in C(D) | c = \kappa_D(c \cap t_n), c \nsubset \bar{t}_n \}$.

Finally, the support in $D^+$ for any $X \in \Delta(D, t_n)$ is merely a unit more than the support of the closure in $D$ (increase due to $t_n$). Indeed since $\sigma_D(X) = \sigma_D(\kappa_D(X))$, we have $\sigma_D^+(X) = \sigma_D^+(\kappa_D(X)) + 1$. For instance, in Table 2, the support of the new CI 20 is 4 while that of CI 7, its genitor, is 3.

Dually, let $t_n$ be an obsolete transaction from $D$ and $D^− = D − \{ t_n \}$. The impact of removing $t_n$ is some CIs, called obsolete, vanishing in $D^−$ while others, the demoted, get their support decreased by 1. The corresponding sets of CIs are investigated in section 4.3.

3 Related work

Historically, methods for incrementally listing the closures of a cross-table date back at least to [19]. In the 1990s and 2000s, a variety of intersection-based incremental concept lattice builders were published, starting with [7] which introduced genitors. They compute jointly CIs with respective tidsets and precedence. Later methods, e.g. Galicia-P and Galicia-M [24], reflect FCI mining concerns, thus they forgo precedence and tidsets. However, they are bound to maintain all CIs as some genitors might well be infrequent. CIs are stored compactly, e.g. in prefix trees, and accessed trough inverted lists to avoid spending on empty intersections.

CloStream [11] is a sliding-window CI miner that heavily relies on intersections between CIs (as opposed to CI-to-tidsets ones). Its decrement lists all subsets of $t_n$ and finds their closures as intersections of all encompassing CIs to test in obsolescence; increments are more focused. Overall, genitors are targeted as such yet key properties thereof are ignored.

Galicia-P [25] is a landmark CI miner using inverted lists to selectively access CIs and trie storage. Intersection trie is grown item-wise with each CI pointing to its current prefix. Intersections are split into new/promoted by tracking the minimum generating CI.

CloStream is another intersection-based method, introduced as landmark and later on completed to a sliding-window mode [27]. It uses inverted lists to filter CIs to get intersected with $t_n$. Genitors appear in both add and removal processing yet as two unrelated –and informally defined– notions.

A landmark intersection-based approach is adapted to batch FCI mining in [1]. They use a two-pass scheme and store nearly all CIs (stripped of infrequent items) and no tidsets. The approach reportedly outperformed modern batch FCI miners on specific types of data.

Moment [4] is a sliding-window FCI stream miner adapting pattern enumeration (as in Eclat [28]) supported by a CE-tree. Its increment proceeds by sibling node joins which exploit the non closed promising itemsets plus some infrequent ones. Support is yielded by tidset intersections. Its decrement relies on direct closures computing to spot obsolete CIs. NewMoment [15] enhances Moment with bit vector encoding of tidsets. Yet it forgoes node categories: A node is joined with all its siblings.

GC-Tree [3] adapts the batch FCI miner DCI-Closed [17] to mining CIs from streams. It uses a hybrid scheme: new CIs are generated either by intersection of existing CIs or by the closure climbing [17]. Genitors are absent: Support is computed tidset-wise while the decrement exploits direct closure computing. Unlike Moment, at most one non closed itemset is kept for each class $[ ]_D$.

In summary, all methods incur overhead due to extra itemsets stored and inefficient new closure computing. Moreover, while CloStream and Galicia-P substantially limit the computation effort, they still suffer on sub-optimal memory usage, e.g. redundancy between CI storage and inverted lists.

Our Ciclad method aims at further minimizing both overheads.

4 Overview of the approach

Ciclad combines an incremental part that builds upon basic ideas from Galicia-P with a novel decremental method based on original mathematical results. Both share an overall computing schema, thus achieving high homogeneity and compactness. Below, we outline that schema and illustrate it (in incremental mode), then provide the formal background for decrementing and expand on high-level algorithms.

4.1 Approach summary

The generic online processing comprises three steps: (1) intersection computing; (2) splitting the total set into promoted
(demoted) and new (obsolete) CIs; (3) update of the indexing structures on CIs. Below, we expand on each step while using generic notations, e.g. \( t_x \) to mean \( t_x \) or \( t_o \).

At step (1), intersection itemsets are grown from prefixes, along an iteration over all items in \( t_x \). At an item \( a_k \), each CI comprising \( a_k \): (i) it has its current prefix extended with \( a_k \), and (ii) it has its support checked against the current maximal support of a CI sharing that prefix. Eventually, prefixes grow into complete intersections whereby each intersection \( X \) has netted the minimum generating CI \( \min_C([X]_{t_o}) \). Step (2) categorizes each \( X \) by checking for a genitor CI within \([X]_{t_o}\): In Ciclad\(^*\), the equality \( X = \min_C([X]_{t_o}) \) means \( X \) is promoted, otherwise new. In Ciclad\(^*\), the test is more subtle, as explained in section 4.3. Step (3) updates CI storage and inverted lists for items in \( t_o \).

### 4.2 Key techniques and data structures

On the algorithmic side, Ciclad uses item-wise inverted lists for quick access to CIs and trie-based storage of evolving intersections (allows a unique copy per prefix). Thus, during step (1), a CI only keeps a pointer to the trie node with the last item of its current intersection prefix. At step (2), end nodes of full intersections in the trie are identified by the non-zero count of referring CIs.

Figure 1 is a snapshot of the working memory of Ciclad\(^*\) at the very end of the increment of \( D_{1,9} \) (Table 3) by \( \{10, bgch\} \) (Table 1), i.e. after all four items have been processed. On the left, CI storage structures fields for ID, support, and a reference to an end node (the last field) in the intersection trie. Observe that itemsets are only stored in the inverted lists (on the right of Figure 1), e.g. the ID of 7 (valued \( fgh \)) appears in the lists of \( f \) (not in the figure), \( g \), and \( h \).

The trie, which is built anew on each window shift, has nodes with fields for ID (underlined), the item, a pointer to the minimal CI (\( \min \)) and a counter (\( \text{cpt} \)) for referring CIs. The counters discriminate end nodes of full intersections (shaded in the figure) against the rest: Due to a simple bookkeeping mechanism for shifting last pointers, end nodes have at least one such pointer directed at them (hence a non-zero counter value). For instance, in 16, whose intersection is \( bc \), last points to 3 whereas 10 and 11 point to 7. Conversely, the \( \min \) field of trie node 2 points back to 10, the minimal CI yielding \( h \). Overall, no white node is pointed at by a CI, hence they are ignored at step (2).

The trie grows along an iteration over items in \( t_x \). For each item \( a \), trie nodes with \( a \) are appended to existing paths. To that end, CIs in the list of \( a \) are scanned: For any such \( c \), the node in \( c \).last is replaced with a successor carrying \( a \) (if missing, such a node is created). The \( \text{cpt} \) fields of both former and new last nodes are updated accordingly. Then, \( c \) is tested for minimality w.r.t. the new last. Details of how Ciclad updates the above fields are given in section 5.1.

Finally, categorization tests equality of a node’s intersection to its minimal CI. In our example, equality holds for node 3 vs. CI 16 (16 is thus promoted), but not for 7 vs. 10, hence the intersection \( h \) becomes a new CI (19 in Table 2).

To sum up, the above structures jointly enable rapid computation of intersections and genitors while keeping a low memory footprint w.r.t. competitor methods.

### 4.3 Decrement-related properties

Dually to the incremental case, our decremental method computes \( C(D^-) \) from \( C(D) \) and \( t_o \), an obsolete transaction in \( D \). Simply put, this amounts to yielding Table 3 from Table 2 and 10 (though in a stream, 1 would vanish first). By duality, \( C(D^-) \subseteq C(D) \), i.e. no new closures appear. The relevant CI families are (1) obsolete CIs (to be removed from \( C(D) \)), denoted \( C_o(D) \), and, (2) demoted CIs ‘surviving’ in \( D^- \) but with a decreased support, \( C_{oD}(D) \). The situation is easily illustrated by mirroring our previous example. Thus, in Table 2, 20 becomes an obsolete CI (genitor 7) while 16 is a demoted CI with support decreased from 5 to 4.

Notwithstanding apparent symmetry, an issue with decrements is \( t_o \) does not discriminate obsolete vs. demoted CIs since both are subsets thereof:

**Property 3.** \( \Delta(D, t_o) \) correspond to obsolete and demoted CIs: \( \Delta(D, t_o) = \{ t_o \cap c \mid c \in C(D) \} = C_o(D) \cup C_{oD}(D) \).

The above follows from Property 2 and \( t_o \in C(D) \). Now, while \( C_o(D) \cup C_{oD}(D) \) stands out within \( C(D) \) via inclusion in \( t_o \), further tests are needed to split it into components, e.g. presence of a genitor as a test for obsolescence. Indeed, if \( D \) is seen as the ‘increment’ of \( D^- \) by \( t_o \), then an obsolete \( c_o \) becomes a new CI hence there should be a genitor \( c_g \) in \( D \), s.t. \( c_o = c_g \cap t_o \). Thus, \( c_g \) the closure of \( c_o \) in \( D^- \), i.e. \( c_o = \text{closure}_{D^-}(c_o) \). However, minimality of genitors, \( c_g = \min([c_g]_{t_o}) \), that proved crucial for incrementing, only holds in \( D^- \), but not in \( D \) where the minimum is \( c_o \) (\( c_o = c_o \cap t_o \) hence \( c_o = \text{closure}_{D^-}(c_o) \)).
min(\left|c_o\right|_t_o))$. The genitor only becomes the minimum if non-inclusion in $t_o$ is also required, $c_g = \min(\left|c_o\right|_t_o - \varphi(t_o))$. Albeit appealing (CloStream went this way), extra non-inclusion tests may prove costly. Instead, we leverage the support difference of $c_o$ and $c_g$ in $D$: For potential $c_o$ we look for $c_g \in [c_o]_{t_o}$ s.t. $\sigma_D(c_g) = \sigma_D(c_g) + 1$. Such $c_g$ will nullify Definition 1 for $c_o$ in $D$.

**Property 4.** A CI is obsolete iff it has a genitor, $C_G(D) = \{c \in C(D) : \exists c_g \in C(G) : c = c_g \cap t_o, \sigma_D(c_g) = \sigma_D(c_g) + 1\}$.

The reasoning behind the if part has been exposed in the previous paragraph. For the only if part, given a CI $c \in C(D)$, the above three-fold condition on $c_g$ must be shown to entail $c \notin C(D)$. First, $\sigma_D(c_g) \neq \sigma_D(c)$ means $c_g \neq c$, hence $c_g \not\subseteq t_o$. Consequently, $\sigma_D(c_g) = \sigma_D(c_g)$, as $c_g$ is not impacted by $t_o$, while for $c$ it decreases by one, $\sigma_D(c) = \sigma_D(c) + 1$. Applying 3rd condition, $\sigma_D(c_g) = \sigma_D(c_g)$. This and 2nd condition, $c \subseteq c_g$, imply $c \not\in C(D)$, hence its obsolescence.

To sum up, while a new CI is easy to spot among other intersections since smaller than its genitor CI, here an obsolete CI equals its intersection with $t_o$. Therefore categorization goes support-wise: The genitor of a CI $c \in C_G(D)$ is the CI in $[c]_{t_o}$, with only $t_o$ missing in its support set w.r.t. to $i_D(c)$.

### 4.4 Decrement-specific processing

As explained above, for an intersection, its potential genitor, if any, has support one less than the minimal generating CI. To make genitors emerge at step (1), a gen field is added to trie nodes to store candidate CIs. Thus, whenever the intersection prefix of a CI $c$ is extended, $c$ is confronted to the current minimum of the node in its updated last field. If non minimal, $c$ is then compared to the content of gen. Observe that, due to the way intersections are grown, at some intermediate step more than one CI could satisfy the above criterion. Indeed, since inverted lists are not sorted, support values of CIs may come at arbitrary order. Next, minimal CIs being unique, there can be only one legit candidate in a min field at any point. However, the interplay between computation of gen and min fields requires a set of candidate CIs to be kept in both. Notwithstanding, ultimately a gen field can hold at most one CI satisfying Property 4. The details of the resulting field updates are provided in section 5.3.

The rest of step (1) mirrors the incremental case. At step (2), discrimination should be as follows: The CI in min is demoted if gen field is empty, obsolete otherwise. However, we dropped removals from gen enforcing this condition to gain efficiency and instead check whether CIs are in $[c]_{t_o}$ (see Algorithm 5 for details). Step (3) is the removal of obsolete CIs from the global CI storage, as well as from all inverted lists it appears in.

## 5 CICLAD sliding-window miner

Common parts (superscripted by *) of the overall computing schema are presented below followed by case-specific ones.

Notice that to achieve homogeneity in step (1), we initialize the above global structures with a special CI with id 0, itemset $I$ and support of $0$.

### 5.1 Unified schema

**Ciclad** (Algorithm 1) is the high-level generic method to add/remove a transaction. At step (1) it iterates over $t_o$ to yield its intersections with existing CIs (lines 2 to 6). For an item $a$, the current prefixes of all CIs (in $c$.last) of its inverse list ($a$.list) are extended by appending $a$ (line 6). Prefixes are initialized to the root node (line 5).

#### Algorithm 1: Ciclad$

```
1 trie ← init()
2 foreach $a \in t_o$, items do
3   if $c$.last do
4     if $c$.last - trie.root
5     c.last ← trie.root
6     ExpandPath\(^*(c, a)
7 UpdateClIs\(^*(c)
```

#### 5.2 Increment-specific processing in Ciclad$

**UpdateClIs** (Algorithm 2) is the unit intersection step. From the $c$.last node, it looks up the successor with $a$ (if none, creates it). Relevant fields are updated to reflect the extended prefix, before calling $UpdateGen$ for a case-specific bookkeeping of the top support CI(s). Basically, each a shared by a CI $c$ and $t_x$ pushes $c$.last downwards in the trie by a node, up till the complete intersection is built.

$UpdateClIs$ covers the above steps (2) and (3). Within the final trie, it filters complete intersections and categorizes them. The CI storage update is case-specific (see $UpdateClIs$ and $UpdateClIs$)

#### Algorithm 2: ExpandPath$

```
1 n ← lookup_suc\(c$.last, $a$
2 if $n = null then
3   n ← create_suc\(a$
4 c$.last.\(a = n$.cpt--
5 c$.last ← n
6 UpdateGen\(c$
```

#### 5.2 Increment-specific processing in Ciclad$

Update minimal CIs ($UpdateGen$) is the first step to differentiate both cases. $UpdateGen$ (Algorithm 2, line 6) merely compares current CI $c$ and CI in $n$.min support-wise and updates the latter. We skip it here since straightforward.

$UpdateClIs$ (Algorithm 3) looks up the final trie for intersection end nodes and discriminates them with a cardinality test. A new intersection is recognized by its size (end node's
UpdateGen

last (3) is a mere support increase.

are flushed and c

field): If the support of c

intersection prefix end has been freshly redirected to a trie

above, both store sets of CIs. Given a CI c whose current

pointers of a CI away from the node. Thus, a CI may belong

CI may belong to gen lists of end nodes corresponding to strict prefixes of

its intersection. This is illustrated by part e of Figure 3: the

CI 2 (abcef) remains in the gen set of the end node of abc,

although its last field eventually points to the end of abcef

(see part f of Figure 3). Property 3 (see section 8) guarantees

that only CIs from \( \Delta(D, t_o) \) can be left behind in prefixes’
gen fields in the way described above. To detect them within

gen, we check for last fields pointing to a different trie node.

Conversely, genitor test spots CIs in n.gen whose last

points to n. If positive, the test triggers removeCI (step (3))

that removes the CI in n.min from the inverted lists of all its

items.

As an illustration, consider the removal of transactions \( 1 \) and

2 from the CI family in Table 2. The first one is trivial

as the only obsolete is 1: All CIs being subsets of 1, only

the special CI 0 can be the genitor. Thus, the resulting CI

family is readily derived from Table 2 by decreasing supports.
by 1. Item iterations in the removal of 2, or abcef, (see Algorithm 1) are shown in Figure 3. In the final trie (item f), intersections {abc}, {cf} and {ef} correspond to obsolete CIs. Indeed, each gen field in the respective end node refers to a CI whose last field, in turn, points to that node. For instance, the gen field for 10 ({cf}) contains 4, or {cdfgh}, of support 1 while the min field value is 3 (ef) of support 2. The demoted CIs are {bc} and {c} which have both an empty gen field.

6 Evaluation

We compared experimentally Ciclad to Moment, NewMoment, CloStream and CFI-Stream. The original version of Moment was used as provided by its authors. To level the playing field, we implemented Ciclad in C++ and in a single threaded mode3 as well as NewMoment, CloStream, CFI-Stream4. To investigate their relative efficiency, we put the methods in identical conditions, i.e. we made them compute all CIs from varying datasets and sliding window lengths.

6.1 Datasets

We used seven datasets of varying nature (see metrics in Table 4). Mushroom and Retail, are popular datasets5: Retail is a sparse market basket dataset, while Mushroom, describing mushroom samples, is a dense and correlated dataset made of same-size transactions. Synth and Synth2 are synthetic transactional datasets, generated with SPMF6, of medium and small size, respectively. Three other real-world datasets were used: click streams for BMS-View (from KDD 2000), online purchases for Chainstore (from the Nu-MineBench project), and network logs7 adapted from activity data from the DARPA Transparent Computing program8.

| Dataset  | | D | avg(|t|) | | F | std deviation(|t|) | Density |
|----------|------------------------|--------|------------------------|--------|------------------------|--------|
| Retail   | 88163                  | 10.4   | 16470                  | 55.9   | 0.06%                  |
| Mushroom | 8124                   | 23     | 119                    | 0      | 19.3%                  |
| Synth    | 100000                 | 25.4   | 10000                  | 14.4   | 0.25%                  |
| Synth2   | 100000                 | 25.5   | 100000                 | 14.3   | 2.5%                   |
| BMS-View | 77512                  | 4.6    | 3340                   | 6.1    | 0.13%                  |
| Chainstore | 1112949               | 7.2    | 46086                  | 8.9    | 0.015%                 |
| Net-Log  | 272376                 | 6.1    | 299                    | 3.7    | 2.04%                  |

Table 4. Description of the datasets

6.2 Experimental settings

All experiments were ran on Windows 10 Professional 64 bits with Intel i7-8700 CPU and 32 GB of RAM. We measured the total CPU time over the entire stream and set a time limit of 10K secs: Methods that ran longer on a dataset were aborted, while recording memory usage, and withdrawn from experiments on larger windows over the same dataset. Moreover, we recorded peak memory usage rather than average across all windows.

Noteworthily, finer measures, like the evolution of time/memory values as well the number of CIs along the stream, albeit potentially helpful, could not be provided here for space reasons (e.g. see section 10).

6.3 Results

Results are summarized in Figure 4 whereby w = W indicates window size while the dashed line on the top is the time limit. As a general trend, CFI-Stream exceeded the time limit on the smallest window of every dataset, except Net-Log, followed closely by NewMoment which could process all window sizes only on Mushroom. Next came Moment: It went

---

3Code available at https://github.com/guyfrancoeur/ciclad
4While the original versions would have been preferable, none of these is currently made available by respective authors.
5http://fimi.ua.ac.be/data/
6http://www.philippe-fournier-viger.com/spmf/
7https://gitlab.com/adaptdata/e2
8https://www.darpa.mil/program/transparent-computing
Figure 4. CPU time and memory usage of Ciclad, Moment, NewMoment, CloStream, and CFI-Stream

off-limit on the first window size of Synth and Chainstore, and on the second one for BMS-View, Synth2 and Retail (yet, exceptionally, we let it run on all others and measured memory usage). On Mushroom, it performed slightly better time-wise, yet its memory usage was substantially worse (among successful methods). Finally, CloStream and Ciclad finished within limit overall, whereby Ciclad showed invariably better time and memory figures.

On the dataset side, it is noteworthy how on Retail, with increasing window sizes, Moment gradually approaches the limit of 32 GB of RAM, while Ciclad remains very competitive (39 MB) and CloStream less so. The data in BMS-View is similar to Retail with less items and smaller transactions which enabled larger windows, e.g. of size 20K. It showed similar overall pattern, yet with a smaller gap between Ciclad and CloStream on RAM. On Chainstore the same trend was observed, yet with larger gaps on memory. This is only half a surprise as both represent real-world streams. The Synth and Synth2, complete the picture of the dominance of Ciclad and, to a lesser degree, CloStream.

With Mushroom, a very dense dataset, the trend is different: CPU-wise, there seems to be a tie between Ciclad and Moment, far ahead of CloStream followed by NewMoment. On memory usage, it is less clear, yet Ciclad is somewhat ahead of the rest. This is matched almost perfectly by the pattern on Net-Log, despite the size of the windows being much larger. The only difference here is NewMoment not being among the contenders after its timeout in the second window.

6.4 Discussion

From the above observations, we conclude that on storage intersection-based methods (CFI-Stream excluded) perform invariably better than pattern enumeration ones. We believe this is the impact of storing non closed itemsets in each \(D\) class. This impact deepens with sparse data as classes tend to grow larger due to the ratio FCI/FI.

On sparse data, intersection-based methods are also faster than their competitors. Again, it is the overhead of non closed itemsets: Upon each CI lookup, Moment traverses its \(D\) class from a smallest member up to the largest one (the CI) by walking along a chain of intermediate itemsets. As a result, on sparse datasets (e.g. Synth2) with limited-size windows Ciclad can be up to 40 times faster than Moment.

With dense datasets, pattern enumeration methods are favored as the FCI/FI ratio is higher, hence the smaller \(D\) classes. Conversely, the intersection-based equivalence classes \(J_x\) tend to grow larger which increases the intersection effort per CI in intersection-based methods. However, it is also worth recalling that with such data, the benefit of mining FCIs, as opposed to plain FI, is limited.

Finally, CloStream lags behind Ciclad because of its fully-blown intersection operations and recurrent lookups for
an intersection \( X \) each time it is generated. Ciclad streamlines both operations by its item-wise trie-based intersection growing technique.

7 Conclusion

Our novel sliding-window miner Ciclad implements an efficient intersection-based mining scheme. It exploits the mathematically-grounded notion of genitor, the CI that is the closure of a non closed itemset which changes its status w.r.t. closedness upon window shift. Design pillars in our intersection-based scheme include per-item inverted list storage of CIs, item-wise intersection growth, and support-based genitor detection. The outcome of our experimental study confirmed that Ciclad outperforms its competitors significantly, both on storage and processing.

As a basis for further research, Ciclad lays the groundwork for additional challenges to be taken up. For instance, to tackle the mining of strong AR, or rather condensed representations thereof, over a sliding window, we are designing extensions thereof covering generator itemsets and/or precedence links as proposed in [18]. As a separate track, we investigate mining of rare yet confident AR [22] from the stream. Next, we plan to leverage Ciclad's homogeneity in merging of \( t_m \) and \( t_o \) processing in a single-pass method. Finally, as a way to focus strictly on FCI's, we will look at the evolution of the FI border [8, 13].

Acknowledgments

Thanks go to Y. Chi for the code of MomentFP and S. Benabderrahmane for the pre-formatted DARPA data.

References

[1] C. Borgelt et al. Finding closed frequent item sets by intersecting transactions. In A. 4th EDBT, pages 367–376. ACM, 2011.
[2] T. Calders et al. A Survey on Condensed Representations for Frequent Sets. In Constraint-Based Mining and Inductive Databases, volume 3848 of LNCS, pages 64–80. Springer, 2004.
[3] J. Chen and S. Li. GC-tree: a fast online algorithm for mining frequent closed itemsets. In 11th FAKDD, pages 457–468. Springer, 2007.
[4] Y. Chi et al. Moment: Maintaining closed frequent itemsets over a stream sliding window. In 4th IEEE ICDM, pages 59–66. IEEE, 2004.
[5] B. Gao and R. Wille. Formal concept analysis: mathematical foundations. Springer, 1999.
[6] C. Godin et al. Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices. Computational Intelligence, 11(2):246–267, 1995.
[7] D. Gunopulos et al. Data mining, hypergraph transversals, and machine learning (extended abstract). In 16th ACM SIGACT-SIGMOD-SIGART PODS, pages 209–216. ACM, 1997.
[8] S. Hamadi et al. Compiling packet forwarding rules for switch pipelined architecture. In ACM INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications, pages 1–9, San Francisco, CA, USA, April 2016. IEEE.
[9] T. Jiang and L. Gruenwald. CFI-Stream: mining closed frequent itemsets in data streams. In 12th ACM SIGKDD, pages 592–597. ACM, 2006.
[10] N. Jiang and L. Gruenwald. Research issues in data stream association rule mining. ACM Sigmod Record, 35(1):14–19, 2006.
[11] R. Karim et al. Mining maximal frequent patterns in transactional databases and dynamic data streams: A spark-based approach. Information Sciences, 432:278–300, 2018.
[12] M. Kryszkiewicz. Concise Representations of Association Rules. In ESF Exploratory WS on Pattern Detection and Discovery, pages 92–109, 2002.
[13] H.-F. Li et al. A new algorithm for maintaining closed frequent itemsets in data streams by incremental updates. In ICDM Workshops 2006, pages 672–676. IEEE, 2006.
[14] H.-F. Li and S.-Y. Lee. Mining frequent itemsets over data streams using efficient window sliding techniques. Expert systems with applications, 36(2):1466–1477, 2009.
[15] C. Lucchese et al. DCI Closed: A Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets. In FIMI, 2004.
[16] K. Teh and S. Tan. On Computing the Minimal Generator Family for Concept Lattices and Icebergs. In 3rd Intl. Conf. on Formal Concept Analysis, pages 192–207, Lens (FR), 2005. Springer.
[17] E. M. Norris. An algorithm for computing the maximal rectangles in a binary relation. Revue Roumaine de Math. Pures et Appliquées, 23(2):243–250, 1978.
[18] J. L. Pfaltz. Incremental Transformation of Lattices: A Key to Effective Knowledge Discovery. In Proc. of the 1st ICAT, pages 351–362, 2002.
[19] M. Rashid et al. Mining associated sensor patterns for data stream of wireless sensor networks. In 8th ACM WS on Performance monitoring and measurement of heterogeneous wireless and wired networks, pages 91–98. ACM, 2013.
[20] L. Szathmary et al. Generating rare association rules using the minimal rare itemsets family. Intl. J-t of Software and Informatics, 4(3), 2010.
[21] F. Tang et al. Adaptive self-sufficient itemset miner for transactional data streams. In Pacific Rim Intl. Conf. on Artificial Intelligence, pages 419–430. Springer, 2019.
[22] P. Valtchev et al. Generating Frequent Itemsets Incrementally: Two Novel Approaches Based On Galois Lattice Theory. Journal of Experimental & Theoretical Artificial Intelligence, 14(2–3):115–142, 2002.
[23] P. Valtchev et al. A framework for incremental generation of closed itemsets. Discrete Appl. Math., 156:924–949, March 2008.
[24] Y. Yamamoto et al. Parasol: a hybrid approximation approach for scalable frequent itemset mining in streaming data. Journal of Intelligent Information Systems, pages 1–29, 2019.
[25] S. Yen et al. A fast algorithm for mining frequent closed itemsets over stream sliding window. In IEEE Intl. Conf. on Fuzzy Systems, pages 996–1002. IEEE, 2011.
[26] M. Zaki et al. New Algorithms for Fast Discovery of Association Rules. In 3rd Intl. Conf. KDD, pages 283–286, 1997.
[27] M. Zaki and C.-J Hsiao. Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Transactions on Knowledge and Data Engineering, 17(4):462–478, 2005.
Appendix

Below, we provide additional results about Ciclad that clarify aspects such as its correctness, computational cost, and the way it compares to GC-Tree, a method that was excluded from the final validation study.

8 Correctness results

To show Ciclad is correct, we first examine the decision about where intersections end. Let $T$ denote the final trie and $n$ a node in $T$. Now, $n$ is the end of a full intersection, i.e. the itemset $\text{items}(n)$ made of items along the root-bound path from $n$ is in $\Delta(D, t_n)$, iff its counter $n.cpt$ is strictly positive:

**Property 5.** Given a node $n$ in the trie $T$, $\text{items}(n) \in \Delta(D, t_n)$ iff $n.cpt > 0$.

Consider the evolution of the minimal CIs for a trie node. Understandably, we focus on trie states at the end of a particular iteration, i.e. with the respective item inverted list fully parsed. Assume after the $k$-th iteration, the list of item $a_k$ is processed and yielded a (still partially completed) trie $T_k$. Let $T_k[a_k]$ denote the set of nodes labelled by $a_k$. These nodes represent the increment w.r.t. $T_{k-1}$.

**Property 6.** Given a node $n \in T_k[a_k]$, and a CI $c \in C(D)$, $n.min = c$ iff $c = \text{sup}(\text{items}(n))$.

Noteworthily, Property 6 ensures that in the final trie, all the minimal CIs are correctly positioned. For the decrement case, we need to further show that within any gen field, at most one CI is not a subset of $t_o$.

**Property 7.** Let $n$ be a node in the final trie $T_{[t_o]}$ of an obsolete transaction $t_o$, then $|n.gen - \Delta(D, t_o)| \leq 1$.

This follows from Property 4 and the observation that CIs outside $\Delta(D, t_o)$ keep their supports from $C(D)$ in $C(D^-)$. Indeed, assuming more than a single CI satisfies the conditions, entails there are two different CIs in $C(D^-)$ with support equal to the support of the obsolete/demoted itemset in $n.min$. This, regardless of the exact status of that itemset in $\Delta(D, t_o)$, is a contradiction. To sum up, the $n.gen$ field of a node in the final trie can contain at most one CI outside $\Delta(D, t_o)$ plus a number of CIs from that set. Then, only the former belongs to the class of $n$ (with minimum CI $n.min$).

Finally, the status of a CI $c$ in $n.gen$ depends on its being member of $\Delta(D, t_o)$. To avoid costly tests of inclusion into $t_o$ we rely on the intersection class of $c$: since all CIs from $\Delta(D, t_o)$ are minimal in their own classes, each has a unique value in its last field. Thus, none of the CIs $c$ in a $n.gen$ field that is also in $\Delta(D, t_o)$, could refer to $n$ via its $c.last$ field:

**Property 8.** Let $n$ be a node in the final trie $T_{[t_o]}$, then for $c_g \in n.gen$, $c_g \notin \Delta(D, t_o)$ iff $c_g.last = n$.

9 Complexity analysis

The window shift complexity of Ciclad is $O(k_m \cdot l_m)$ in time and $O(k_m^2 \cdot l_m)$ in space. Here, $k_m$ is the maximal transaction (and CI) size and $l_m$ the maximal number of CIs in a window.

The intersection computing (Ciclad’ up till the end of ExpandPath”) is in $O(k_m \cdot l_m)$. For each item $i$ from $t_o$ and CI $c$ comprising $i$, Ciclad’ pushes the intersection of $c$ down its path in the trie. This involves few operations (half a dozen) all of constant time cost.

Categorizing intersections and creating new CIs (UpdateCIs up till createCI()) has also a cost in $O(k_m \cdot l_m)$. First, detecting end nodes is in $O(k_m \cdot l_m)$ since there are at most $l_m$ intersections, each of size at most $k_m$, hence $O(k_m \cdot l_m)$ nodes in the trie. Next, creating all new CIs is also in $O(k_m \cdot l_m)$: the same number bound $l_m$ multiplied by the unit cost of creation (linear in $k_m$). Inverted lists can be updated in $O(k_m \cdot l_m)$ time as each combination of a new CI and incident item amounts to one list insertion.

Ciclad’ has a memory footprint in $O(k_m^2 \cdot l_m)$. Indeed, the intersection trie will have at most $k_m \cdot l_m$ nodes (see above) whose successor structures might need up to $k_m$ memory cells each. Comparatively, the total footprint of the inverted lists is in $O(k_m \cdot l_m)$. Now, $k_m \cdot l_m$ is, in fact, a gross overestimation of the total number of items in all CIs which is key cost factor in both time and memory: The real figure, especially with sparse data, will be way lower. Noteworthily, the size of the window is not a factor in the above functions: This is the effect of skipping tidsets altogether (yet it has an indirect impact through $l_m$).

Finally, Ciclad is a listing algorithm, hence to be assessed not by total time but rather by the cost per output element, i.e. CI. Thus, assuming the entire stream processing cost is in $O(n_s \cdot k_m \cdot l_m)$, where $n_s$ is total number of transactions, the per-CI cost is, grossly, in $O(n_s \cdot k_m)$, i.e. a polynomial in the size of the dataset. In contrast, the cost of producing a particular new CI $c$ in Moment might go beyond that limit as the number of unpromising nodes traversed while generating $c$ can grow up to exponential in its size.

10 Additional performance tests

10.1 Fine-grained performance analysis

We made Ciclad compete on Moment’s terms, i.e. with higher support thresholds. Thus, we compared both over various min_supp values (1,2,3 and 5), this time using only two datasets, one dense (Mushroom) and one sparse (Synth2), each with two different window sizes. The results are summarized in Figure 5 (CPU time) and in Figure 6 (memory usage).

On Mushroom, variations in min_supp modestly impact the computing time of Moment; this arguably fits the intuition that CIs are more regularly scattered over the pattern space (thus higher values are needed for a palpable drop in the cost). Noteworthily, Ciclad and Moment offer comparable
Figure 5. Runtime comparison varying Moment’s min_supp performances. On Synth2, Moment’s runtime efficiency improves much faster and it outperforms Ciclad for thresholds of 5 and above.

Figure 6. Memory comparison varying Moment’s minsupp

Memory-wise, Ciclad is still somewhat ahead, yet the trend of rapidly decreasing consumption in Moment is visible. Again, for the sparse dataset, with thresholds of 5 and up, Moment catches up with Ciclad, whereas with the dense data the break-even point is still somewhere above.

Figure 7. Evolution of $C(D)$ in Mushr, $w=1k$ and Synth2, $w=1k$

As a possible hint at the reasons behind the observed performances, we track the proportion of new, promoted, demoted and obsolete CIs in windows. Results for Mushroom and Synth2 datasets with windows of size 1K are shown in Figure 7. In summary, the higher ratio of new/obsolete CIs to promoted/demoted ones in sparse data would explain superior performances of Ciclad by the costly tree restructuring in Moment as opposed to inexpensive updates of existing nodes. Conversely, it hints at detecting of promoted/demoted CIs in Ciclad as possible improvement point for speeding up dense data processing.

Figure 8. GC-Tree vs Ciclad vs Moment in landmark mode

An immediate observation is that the hybrid approach, even if appealing, does not perform well with large number of items. The clear gap between GC-Tree and its competitors is, we surmise, due to the number of canonicity tests it needs to perform while extending a closure in order to ensure that the result is indeed the lexicographically smallest among all alternative extensions in its equivalence class $[ ]_D$.

We also examined the storage overhead in Moment, i.e. due to the storage of promising and intermediate itemsets. Table 5 shows the average number of nodes within Moment’s CE-tree (Avg. nodes) against the average number of CIs (Avg. CIs), both taken over the entire stream, for a number of combinations (dataset, window size, min_supp). The wide variation, 3.24 to an extreme 232.21, is intriguing. Yet the trend correlates with the observations on computing time and memory usage, i.e., the higher the value, the less competitive the method vs Ciclad.

10.2 GC-Tree vs Ciclad vs Moment

We studied also GC-Tree in order to assess its hybrid approach. However, its decremental part was impossible to implement due to inconsistencies in the description of the method. Therefore, GC-Tree was compared to Ciclad and Moment, in landmark mode only.

In Figure 8, an extract of the performance tests is given: The figure presents the CPU time on three of the seven datasets in landmark mode. We used a prefix large enough to let a stable trend appear.

### Table 5. Average number of CIs and CET nodes in Moment

| Dataset       | Avg. nodes | Avg. CIs | Ratio |
|---------------|------------|----------|-------|
| Mushr. ($w=1k$, $s=1$) | 677202     | 208952   | 3.24  |
| Mushr. ($w=1k$, $s=5$) | 234949     | 30081    | 7.08  |
| Mushr. ($w=5k$, $s=1$) | 459379     | 16428    | 28.67 |
| Mushr. ($w=5k$, $s=1$) | 855548     | 44835    | 19.50 |
| Synth2 ($w=3k$, $s=1$) | 1655196    | 8913     | 184.78|
| Synth2 ($w=1k$, $s=1$) | 16013671   | 80364    | 199.26|
| Synth2 ($w=2k$, $s=1$) | 62125880   | 267717   | 232.21|

We also examined the storage overhead in Moment, i.e. due to the storage of promising and intermediate itemsets. Table 5 shows the average number of nodes within Moment’s CE-tree (Avg. nodes) against the average number of CIs (Avg. CIs), both taken over the entire stream, for a number of combinations (dataset, window size, min_supp). The wide variation, 3.24 to an extreme 232.21, is intriguing. Yet the trend correlates with the observations on computing time and memory usage, i.e., the higher the value, the less competitive the method vs Ciclad.