An Efficient Hybrid Algorithm for Mining Frequent Closures and Generators

Laszlo Szathmary[1], Petko Valtchev[1], Amedeo Napoli[2], and Robert Godin[1]

[1] Dépt. d'Informatique UQAM, C.P. 8888,
Succ. Centre-Ville, Montréal H3C 3P8, Canada
Szathmary.L@gmail.com, valtchev.petko@uqam.ca, godin.robert@uqam.ca
[2] LORIA UMR 7503, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France
napoli@loria.fr

Abstract. The effective construction of many association rule bases requires the computation of both frequent closed and frequent generator itemsets (FCIs/FGs). However, these two tasks are rarely combined. Most of the existing solutions apply levelwise breadth-first traversal, though depth-first traversal, depending on data characteristics, is often superior. Hence, we address here a hybrid algorithm that combines the two different traversals. The proposed algorithm, *Eclat-Z*, extracts frequent itemsets (FIs) in a depth-first way. Then, the algorithm filters FCIs and FGs among FIs in a levelwise manner, and associates the generators to their closures. In *Eclat-Z* we present a generic technique for extending an arbitrary FI-miner algorithm in order to support the generation of minimal non-redundant association rules too. Experimental results indicate that *Eclat-Z* outperforms pure levelwise methods in most cases.

1 Introduction

The discovery of meaningful associations is a key data mining task [1]. An association miner typically proceeds in two steps: **(i)** extract all frequent patterns X of a database, and **(ii)** break each pattern X into a *premise* Y, and a *conclusion* $X \setminus Y$ parts to form a rule $Y \rightarrow X \setminus Y$. Interestingness measures, such as support and confidence, are applied to prune the set of extracted association rules. However, the number of the remaining rules may still be way too high to be practical. As a remedy, various concise representations of the family of valid association rules have been proposed [2,3,4,5,6]. A good survey can be found in [7].

Here we focus on the computation of frequent closed itemsets (FCIs) and frequent generators (FGs), which underlie the minimal non-redundant association rules ($\mathcal{MNR}$) for instance. Following [2], these are rules with the form $P \rightarrow Q \setminus P$, where $P \subset Q$, P is a *(minimal) generator* (a.k.a. key-sets or free-sets) and Q is a *closed itemset*. In other terms, in such rules the premise is minimal and the conclusion is maximal. As shown in [7], $\mathcal{MNR}$ is a *lossless*, *sound*, and *informative* representation of all valid rules. Moreover, further restrictions can be imposed on the rules in $\mathcal{MNR}$, leading to more compact representations such as the *generic basis* or the *proper basis* (see [7] for a complete list).

From a computational point of view, constructing $\mathcal{MNR}$ or its sub-structures requires the family of frequent closed itemsets (FCIs) and their generators (FGs), and possibly the precedence order between FCIs. A few methods for extracting both FCIs and FGs have been published in the mining literature, e.g. *A-Close* [8] or *Titanic* [9]. Generators have been targeted within the concept analysis field as well [10], e.g. by *Zart* [11]. Well-known FCI/FG-miners exclusively apply levelwise strategies, although the levelwise itemset miners are knowingly outperformed by depth-first methods (e.g. *Eclat* [12], *Charm* [13], *Closet* [14]) on a broad range of dataset profiles, especially on dense ones. Hence the idea of designing a hybrid FCI/FG-miner. The algorithm that we propose called *Eclat-Z* splits the FCI/FG-mining task into three steps. First, it applies the well-known vertical algorithm *Eclat* for extracting the set of FIs. Second, it processes the FIs in a levelwise manner to filter FCIs and FGs. This is why *Eclat-Z* is said to be a hybrid algorithm. Finally, the algorithm associates FGs to their closures (FCIs) to provide the necessary starting point for the production of $\mathcal{MNR}$. Experimental results show that *Eclat-Z* outperforms two other efficient competitors, *A-Close* and *Zart*. During the design of *Eclat-Z* we had to face a challenge. The *Eclat* algorithm, due to its depth-first nature, provides the FIs in a completely unordered way. However, the levelwise post-processing steps require the FIs in ascending order by length. We managed to solve this problem with a special file indexing that proves to be efficient, generic, and gives no memory overhead at all. As we will see, the idea of *Eclat-Z* can be *generalized* and used for arbitrary FI-mining algorithm, either breadth-first or depth-first.

The main contribution of this work is a universal way of extending FI-miners for computing minimal non-redundant association rules too. We present a novel method for storing FIs in the file system if FIs are not provided in ascending order by length. Thanks to our special file indexing technique, which requires no additional memory, FIs can be sorted in a lengthwise manner. Once itemsets are available in this order, we show an original technique for filtering generators, closed itemsets, and associating generators to their closures.

The paper is organized as follows. Section 2 provides the basic concepts and essential definitions. In Section 3, we give an overview of the *Eclat* algorithm. This is followed in Section 4 with the detailed description of the *Eclat-Z* algorithm, where we also give a running example. Next, we provide experimental results in Section 5 for comparing the efficiency of *Eclat-Z* to *A-Close* and *Zart*. Finally, conclusions and future work are discussed in Section 6.

2 Basic Concepts

Consider the following 5×5 sample dataset: $\mathcal{D} = \{(1, ABDE), (2, AC), (3, ABCE), (4, BCE), (5, ABCE)\}$. Throughout the paper, we will refer to this example as **"dataset $\mathcal{D}$"**.

We consider a set of *objects* or *transactions* $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$, a set of *attributes* or *items* $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, and a relation $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$. A set of items is called an *itemset*. Each transaction has a unique identifier (*tid*), and a set of transactions is called a *tidset*.[3] For an itemset X, we denote its corresponding tidset, often called its *image*, as $t(X)$. For instance, in dataset $\mathcal{D}$, the image of AB is 135, i.e. $t(AB) = 135$. Conversely, $i(Y)$ is the itemset corresponding to a tidset Y. The *length* of an itemset is its cardinality, whereas an itemset of length k is called a k-itemset (or a k-long itemset). The *support* of an itemset X, denoted by $supp(X)$, is the size of its image, i.e. $supp(X) = |t(X)|$. An itemset X is called *frequent*, if its support is not less than a given *minimum support* (denoted by min_supp), i.e. $supp(X) \geq min\_supp$. The image function induces an equivalence relation on $\wp(\mathcal{A})$: $X \cong Z$ iff $t(X) = t(Z)$ [15]. Moreover, an equivalence class has a unique maximum w.r.t. set inclusion and possibly several minima, respectively called *closed* itemset (a.k.a. *concept intents* in concept analysis [16]) and *generator* itemsets (a.k.a. *key-sets* in database theory or free-sets). The support-oriented definitions exploiting the monotony of support upon $\subseteq$ in $\wp(\mathcal{A})$ are as follows:

Definition 1 (closed itemset; generator). *An itemset X is* closed *(*generator[4]*) if it has no proper superset (subset) with the same support (respectively).*

The *closure* of an itemset X (denoted by X'' following standard FCA notation) is thus the largest itemset in the equivalence class of X. For instance, in dataset $\mathcal{D}$, the sets AB and AC are generators, and their closures are ABE and AC, respectively (i.e. the equivalence class of AC is a singleton). In our approach, we rely on the following two properties:

Property 1. A closed itemset cannot be the generator of a larger itemset.

Property 2. The closure of a frequent non-closed generator g is the smallest proper superset of g in the set of frequent closed itemsets.

An association rule r: $P_1 \rightarrow P_2$ involves two itemsets $P_1, P_2 \subseteq \mathcal{A}$, s.t. $P_1 \cap P_2 = \emptyset$, and $P_2 \neq \emptyset$. The support of a rule r is $supp(r) = supp(P_1 \cup P_2)$ and its *confidence* $conf(r) = supp(P_1 \cup P_2)/supp(P_1)$. *Frequent* rules are defined in a way similar to frequent itemsets, whereas *confident* rules play an equivalent role for the confidence measure. A *valid* rule is both frequent and confident. Finding all valid rules in a database is the target of a typical association rule mining task.

As their number may grow up to exponential, reduced sub-families of valid rules are defined, which nevertheless convey the same information (*lossless*). Associated expansion mechanisms allow for the entire family to be retrieved from the reduced ones without any non-valid rules to be mixed in (*soundness*). The minimal non-redundant association rule family ($\mathcal{MNR}$) is made of rules $P \rightarrow Q \setminus P$, where $P \subset Q$, P is a *(minimal) generator* and Q is a *closed*

[3] For convenience, we write an itemset $\{A, B, E\}$ as ABE, and a tidset {1,3,5} as 135.
[4] Generators are also called "keys" or "key itemsets".

itemset. A more restricted family arises from the additional constraint of P and Q belonging to the same equivalence class, i.e. $P'' = Q$. It is known as the *generic basis* for exact (100% confidence) association rules [7]. Here the basis refers to the non-redundancy of the family w.r.t. a specific criterion. Inexact rule bases can also be defined by means of generators and closures, e.g. the *informative basis* [7], which further involves the inclusion order between closures.

3 Vertical Frequent Itemset Mining

The frequent itemset mining methods from the literature can be roughly split into breadth-first and depth-first miners. *Apriori*-like [1] levelwise breadth-first algorithms exploit the anti-monotony of frequent itemsets in a straightforward manner: they advance one level at a time, generating candidates for the next level and then computing their support upon the database. Depth-first algorithms, in contrast, organize the search space in a tree. Typically using a sorted representation of the itemsets, they factor out common prefixes and hence limit the computing effort. Typical depth-first FI-miners include *Eclat* [17] and *FP-growth* [18].

3.1 Common Characteristics

Eclat was the first FI-miner using a vertical encoding of the database combined with a depth-first traversal of the search space (organized in a prefix-tree) [17].

Vertical miners rely on a specific layout of the database that presents it in an item-based, instead of a transaction-based, fashion. Thus, an additional effort is required to transpose the global data matrix in a pre-processing step. However, this effort pays back since afterwards the secondary storage does not need to be accessed anymore. Indeed, the support of an itemset can be computed by explicitly constructing its tidset which in turn can be built on top of the tidsets of the individual items. Moreover, in [12], it is shown that the support of any k-itemset can be determined by intersecting the tid-lists of any two of its $(k-1)$-long subsets.

The central data structure in a vertical FI-miner is the IT-tree that represents both the search space and the final result. The IT-tree is an extended prefix-tree whose nodes are $X \times t(X)$ pairs. With respect to a classical prefix-tree or trie, in an IT-tree the itemset X provides the entire prefix from the root to the node labeled by it (and not the difference with the parent node prefix).

Example. Figure 1 presents the IT-tree of our example. Observe that the node $ABC \times 35$ for instance can be computed by combining the nodes $AB \times 135$ and $AC \times 235$. To that end, tidsets are intersected and itemsets are joined. The support of ABC is readily established to 2.

3.2 Eclat

Eclat is a plain FI-miner traversing the IT-tree in a depth-first manner in a pre-order way, from left-to-right [17,12].

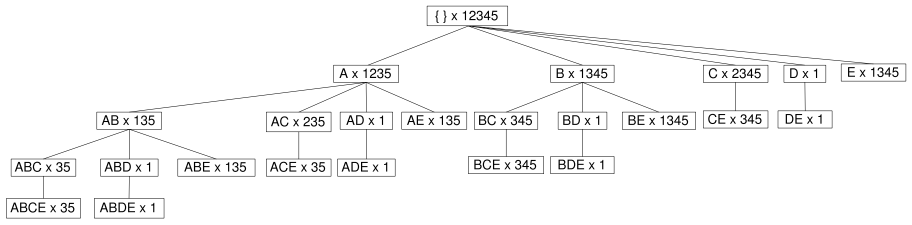


Fig. 1. IT-tree: Itemset-Tidset search tree of dataset $\mathcal{D}$

At the beginning, the IT-tree is reduced to its root (empty itemset). *Eclat* extends the root one level downwards by adding the nodes of all frequent 1-itemsets. Then, each of the new nodes is extended similarly: first, candidate descendant nodes are formed by adding to its itemset the itemset of each right sibling; second, the tidsets are computed by intersection and the supports are established; and third, the frequent itemsets are added as effective descendant nodes of the current node.

Running example. Using Figure 1, we illustrate the execution of *Eclat* on dataset $\mathcal{D}$ with $min\_supp = 1$ (20%). Initially, the IT-tree comprises only the root node whose support is 100%. Frequent items with their tidsets are then added under the root. Each of the new nodes is recursively extended, following a left-to-right order and processing the corresponding sub-trees in a pre-order fashion. For instance, the subtree of A comprises all frequent itemsets starting with A. Thus, at step two, all 2-long supersets of A are formed using the right siblings of A (frequent 1-itemsets). As AB, AC, AD, and AE are all frequent, they are added as descendant nodes under the node of A. The extend procedure is then recursively called on AB and the computation goes one level deeper in the IT-tree. When the algorithm stops, all frequent itemsets are discovered.

4 The Eclat-Z Algorithm

Eclat-Z is a hybrid algorithm that combines the vertical FI-miner *Eclat* with an original levelwise extension. *Eclat* finds all FIs that we save in the file system. Then, this file is processed in a levelwise manner, i.e. itemsets are read in *ascending* order by length, generators and closed itemsets are filtered, and finally generators are associated to their closures. In the following, we present the algorithm in detail.

4.1 Processing Itemsets in Ascending Order by Length

Sorting itemsets in ascending order by length is required for such algorithms that produce FIs in an unordered way. *Eclat*, the algorithm used as itemset mining "engine" here, is a good example of such an algorithm. Levelwise algorithms, like

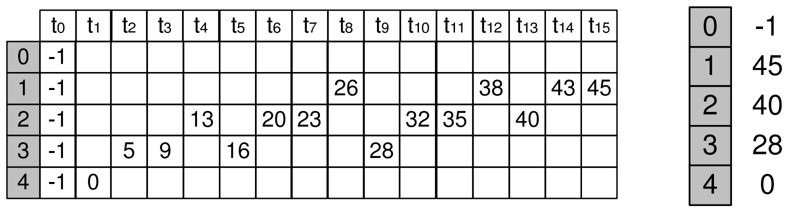
Table 1. Order of frequent itemsets produced by *Eclat*

order	itemset	support
1)	ABCE	2
2)	ABC	2
3)	ABE	3
4)	AB	3
5)	ACE	2
6)	AC	3
7)	AE	3
8)	A	4
9)	BCE	3
10)	BC	3
11)	BE	4
12)	B	4
13)	CE	3
14)	C	4
15)	E	4

Apriori, represent an easier case because they produce FIs in ascending order by length. If someone wants to use such an algorithm, he can continue with the second part in Section 4.2. Here, in the first part, we present an efficient, file-system based approach to process FIs in ascending order by their length. For our example, we use dataset $\mathcal{D}$ with $min\_supp = 2$ (40%). *Eclat* produces FIs in an unordered way, as shown in Table 1.

As in practice it is impossible to keep all FIs in the main memory, we write FIs in a binary file. In main memory we have an index, called *PosIndex*, for storing file positions (Figure 2). *PosIndex* is a simple array of integers. At position k it indicates where the last k-long itemset is written in the binary file. *PosIndex* must always be kept up-to-date. On the left part of Figure 2, it is indicated how *PosIndex* changes in time between t_0 and t_{15}. The right side of the same figure shows the final state of *PosIndex.* Figure 3 shows the contents of the file. For conciseness, support values are omitted. The file structure is explained through the following examples.

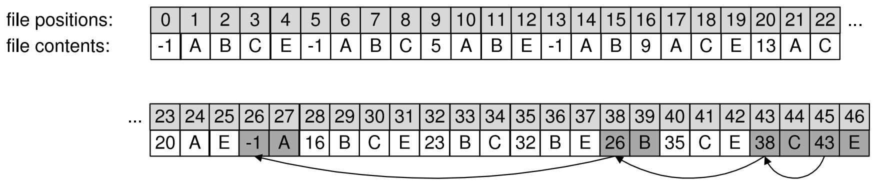
Running example for storing itemsets. In our implementation of *Eclat* an IT-node is processed when we return in recursion. Thus, the first FI found by *Eclat* is $ABCE$ (see Table 1). It is a 4-itemset. The size of the *PosIndex* array is dynamically increased to size $4 + 1$ (+1, because position 0 is not used). The array is initialized: at each of its position we store -1 (time t_0). As the length of the found itemset is 4, we read the value of *PosIndex* at position 4. This value (-1), together with the itemset is written to the binary file (see Figure 3). The value that we read from *PosIndex* is a *backward pointer* that shows the file position of the previous itemset *with the same length.* As the value is -1 here, it simply means that this is the first itemset of this length. After writing $ABCE$ to the file, the 4^{th} position of *PosIndex* is updated to 0 (t_1), because the last 4-long itemset together with its backward pointer was written to position 0 in the file. ABC is written similarly, and *PosIndex* is updated (t_2). When ABE is written to the file, its backward pointer is set to 5. This value is read from *PosIndex* at position 3, since ABE is a 3-itemset. The process continues until all FIs are found. The final state of *PosIndex* is indicated on the right side of Figure 2.



	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15
0	-1															
1	-1								26				38		43	45
2	-1				13		20	23			32	35		40		
3	-1		5	9		16				28						
4	-1	0														

0	-1
1	45
2	40
3	28
4	0

Fig. 2. The *PosIndex* structure. Timeline **(left)** and final state **(right)**



file positions:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	...
file contents:	-1	A	B	C	E	-1	A	B	C	5	A	B	E	-1	A	B	9	A	C	E	13	A	C	

...	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
	20	A	E	-1	A	16	B	C	E	23	B	C	32	B	E	26	B	35	C	E	38	C	43	E

Fig. 3. Contents of the file with the FIs. File positions are also indicated

Running example for reading itemsets. Figure 3 illustrates how to read k-itemsets from the file (here $k = 1$, shown in dark grey). First, we look for the last 1-itemset, which is registered in *PosIndex* (Figure 2) at position 1. The value points at position 45 in the file. Itemset E is read, and we seek to the previous 1-itemset at position 43. C is read, seek to position 38. B is read, seek to position 26. A is read, and -1 indicates that there are no more 1-itemsets. This way FIs can be processed in ascending order by length.

4.2 Finding Generators, Closures, and Associating Them

In the previous subsection, we presented the first part of the algorithm, i.e. how to get frequent itemsets in ascending order by their length, even if they are produced in an unordered way. In this subsection we continue with the second part namely how to associate generators to their closures, once FIs are available in a good order. The main block is shown in Algorithm 1. Two kinds of tables are used, namely F_i for i-long frequent, and Z_i for i-long frequent closed itemsets.

The `readTable` function is in charge of reading frequent itemsets of a given length. If such an algorithm is used that produces FIs in an unordered way, like *Eclat*, then `readTable` reads FIs from the binary file, as explained previously. The function returns FIs in an F_i table. Fields of the table are initialized: itemsets are marked as “keys” and “closed”. Of course, during the post-processing step these values may change. Frequent attributes (frequent 1-itemsets) represent a special case. If they are present in each object of the dataset, then they are not generators, because they have a smaller subset with the same support,

Algorithm 1 (Eclat-Z):

```
 1) $maxItemsetLength \leftarrow$ (size of the largest FI found by the FI-miner);
 2) $\mathcal{FG} \leftarrow \{\}$;   // global list of frequent generators
 3) $F_1 \leftarrow$ readTable(1);   // get frequent 1-itemsets
 4) for ($i \leftarrow 1$; $i < maxItemsetLength$; $i \leftarrow i + 1$) {
 5)       $F_{i+1} \leftarrow$ readTable($i + 1$);   // get frequent $(i + 1)$-itemsets
 6)       findKeysAndClosedItemsets($F_{i+1}$, $F_i$);   // filtering
 7)       $Z_i \leftarrow \{l \in F_i \mid l.\text{closed} = \text{true}\}$;
 8)       Find-Generators($Z_i$);
 9) }
10) $Z_i \leftarrow \{l \in F_i \mid l.\text{closed} = \text{true}\}$;
11) Find-Generators($Z_i$);
12)
13) return $\bigcup_i Z_i$;
```

namely the empty set. In this case the empty set is a useful generator (w.r.t. rule generation).

The `findKeysAndClosedItemsets` procedure is in charge of filtering FCIs and FGs among FIs. The filtering procedure is based on Def. 1.

The `Find-Generators` procedure takes as input a Z_i table. The method is the following. For each frequent closed itemset z in Z_i, it finds its proper subsets in the global list $\mathcal{FG}$, registers them as generators of z, deletes them from $\mathcal{FG}$, and adds non-closed generators from F_i to $\mathcal{FG}$. Properties 1 and 2 guarantee that whenever the subsets of an FCI are looked up in the list $\mathcal{FG}$, only its generators are returned.

Running example. The execution of *Eclat-Z* on dataset $\mathcal{D}$ with $min\_supp = 2$ is illustrated in Table 2. Frequent 1-itemsets are read and stored in F_1. Since their support values are less than the total number of objects in the dataset, all of them are keys (generators). They are also marked as "closed". Then, frequent 2-itemsets are read too and stored in F_2. The algorithm compares F_2 to F_1 in order to filter non-closed and non-generator elements. The itemset BE has two subsets in F_1 with the same support, which means that BE is not a generator, and B and E are not closed (by Def. 1). The remaining closed itemsets A and C are copied from F_1 to Z_1, and their generators are determined. In the global list of frequent generators ($\mathcal{FG}$), which is still empty, they have no subsets, which means that both A and C are generators themselves. Non-closed generators of F_1 (B and E) are added to the $\mathcal{FG}$ list. Comparing F_3 to F_2, it turns out that ABE and BCE are not generators, while AB, AE, BC, and CE are not closed. The remaining closed itemsets AC and BE are copied to Z_2. The generator of AC is itself, and the generators of BE are B and E. These two generators are removed from $\mathcal{FG}$ and AB, AE, BC, and CE are added to $\mathcal{FG}$. The 4-itemset $ABCE$ is the longest FI in the example. Its generators are read from $\mathcal{FG}$. When the algorithm stops, all FCIs *with* their generators are determined (see the union

Table 2. Execution of *Eclat-Z* on dataset $\mathcal{D}$ with $min\_supp = 2$ (40%)

F_1	key?	supp	closed?
{A}	yes	4	yes
{B}	yes	4	~~yes~~
{C}	yes	4	yes
{E}	yes	4	~~yes~~

Z_1	supp	generators
{A}	4	
{C}	4	

$\mathcal{FG}_{before} = \{\}$
$\mathcal{FG}_{after} = \{B, E\}$

F_2	key?	supp	closed?
{AB}	yes	3	~~yes~~
{AC}	yes	3	yes
{AE}	yes	3	~~yes~~
{BC}	yes	3	~~yes~~
{BE}	~~yes~~	4	yes
{CE}	yes	3	~~yes~~

Z_2	supp	generators
{AC}	3	
{BE}	4	{B, E}

$\mathcal{FG}_{before} = \{B, E\}$
$\mathcal{FG}_{after} = \{AB, AE, BC, CE\}$

F_3	key?	supp	closed?
{ABC}	yes	2	~~yes~~
{ABE}	~~yes~~	3	yes
{ACE}	yes	2	~~yes~~
{BCE}	~~yes~~	3	yes

Z_3	supp	generators
{ABE}	3	{AB, AE}
{BCE}	3	{BC, CE}

$\mathcal{FG}_{before} = \{AB, AE, BC, CE\}$
$\mathcal{FG}_{after} = \{ABC, ACE\}$

F_4	key?	supp	closed?
{ABCE}	~~yes~~	2	yes

Z_4	supp	generators
{ABCE}	2	{ABC, ACE}

$\mathcal{FG}_{before} = \{ABC, ACE\}$
$\mathcal{FG}_{after} = \{\}$

of the Z_i tables in Table 2). If *Eclat-Z* leaves the generators of a closed itemset empty, it simply means that the generator is identical to the closed itemset (this is the case for A, C, and AC in the example). Recall that the support of a generator is equivalent to the support of its closure.

5 Experimental Results

We evaluated *Eclat-Z* against *Zart* [11] and *A-Close* [8]. The algorithms were implemented in Java under the Coron data mining platform [19].[5] The experiments were carried out on a bi-processor Intel Quad Core Xeon 2.33 GHz machine running under Ubuntu GNU/Linux with 4 GB RAM. For the experiments we have used the following datasets: T20I6D100K, C20D10K, and Mushrooms. The T20I6D100K[6] is a sparse dataset, constructed according to the properties of market basket data that are typical weakly correlated data. The C20D10K is a census dataset from the PUMS sample file, while the Mushrooms[7] describes mushrooms characteristics. The last two are highly correlated datasets.

[5] http://coron.loria.fr
[6] http://www.almaden.ibm.com/software/quest/Resources/
[7] http://kdd.ics.uci.edu/

Table 3. Response times of *Eclat-Z* and other statistics (response times of *Zart* and *A-Close*, number of FIs, number of FCIs, number of FGs, and the proportion of the number of FGs to the number of FIs)

min_supp	execution time (sec.)			# FIs	# FCIs	# FGs	$\frac{\#FGs}{\#FIs}$
	Eclat-Z	Zart	A-Close				
T20I6D100K							
1%	4.11	6.58	24.06	1,534	1,534	1,534	100.00%
0.75%	3.31	12.39	29.44	4,710	4,710	4,710	100.00%
0.5%	5.82	34.61	72.88	26,836	26,208	26,305	98.02%
0.25%	24.55	121.03	204.69	155,163	149,217	149,447	96.32%
C20D10K							
30%	1.07	6.27	11.27	5,319	951	967	18.18%
20%	1.71	11.32	20.77	20,239	2,519	2,671	13.20%
10%	5.17	23.99	40.70	89,883	8,777	9,331	10.38%
5%	20.24	49.29	62.64	352,611	21,213	23,051	6.54%
MUSHROOMS							
30%	0.82	2.87	5.86	2,587	425	544	21.03%
20%	3.36	7.72	11.68	53,337	1,169	1,704	3.19%
10%	37.46	46.37	29.43	600,817	4,850	7,585	1.26%
5%	368.03	391.97	50.20	4,137,547	12,789	21,391	0.52%

Table 3 contains the experimental evaluation of *Eclat-Z* against *Zart* and *A-Close*. All times reported are real, wall clock times as obtained from the Unix *time* command between input and output. We have chosen *Zart* and *A-Close* because they represent two efficient algorithms that produce exactly the same output as *Eclat-Z*. *Zart* and *A-Close* are both levelwise algorithms. *Zart* is an extension of *Pascal* [15], i.e. first it finds all FIs using pattern-counting inference, then it filters FCIs, and finally the algorithm associates FGs to their closures. *A-Close* reduces the search space to FGs only, then it calculates the closure for each generator. The way *A-Close* computes the closures of generators is quite expensive because of the huge number of intersection operations.

In the sparse dataset T20I6D100K, almost all frequent itemsets are closed and generators at the same time. It means that most equivalence classes are singletons, thus *A-Close* cannot reduce the search space significantly. Since the closure computation of *A-Close* is quite expensive, *Eclat-Z* performs much better. *Zart* and *Eclat-Z* are similar in the sense that first both algorithms extract FIs. While *Zart* is based on *Pascal*, *Eclat-Z* is based upon *Eclat*. The better performance of *Eclat-Z* is due to the better performance of its FI-miner "engine".

In datasets C20D10K and MUSHROOMS, the number of FGs is considerably less than the total number of FIs. In this case, *Zart* can take advantage of its pattern counting inference technique, and *A-Close* can benefit from its search space reduction. Despite these optimizations, *Eclat-Z* still outperforms the two algorithms in most cases. However, if the number of FGs is *much less* than the number of FIs (for instance in MUSHROOMS by $min\_supp = 5\%$), *A-Close* gives better response times.

As a conclusion we can say that *Eclat-Z* clearly outperforms its levelwise competitors on sparse datasets, and it also performs very well on dense, highly correlated datasets if the minimum support threshold is not set too low.

6 Conclusion

In this paper we presented a generic algorithm called *Eclat-Z* that identifies FCIs and their associated generators. From this output numerous concise representations of valid association rules can be readily derived.

Eclat-Z splits the FCI/FG-mining problem into three tasks: **(1)** FI-mining, **(2)** filtering FCIs and FGs, and **(3)** associating FGs to their closures (FCIs). The FI-mining part is solved by a well-known depth-first algorithm, *Eclat*. However, with *Eclat* we had to face a challenge: it produces itemsets in an unordered way. Thanks to a special file indexing technique, we managed to solve this issue in an efficient way, thus steps **(2)** and **(3)** can post-process FIs in a levelwise manner. As seen, the idea of the hybrid algorithm *Eclat-Z* can be generalized and used for *any* FI-mining algorithm, be it breadth-first or depth-first. Experimental results prove that *Eclat-Z* is highly efficient and outperforms its levelwise competitors in most cases.

The study led to a range of exciting questions that are currently investigated. *Eclat-Z* is highly efficient, but first it traverses the whole set of FIs. While in sparse datasets it causes no problem, it can be a drawback in dense datasets with very low minimum support. It would be interesting to combine the search space reduction of *A-Close* with the efficiency of *Eclat-Z*. A further challenge lies in the computation of the FCI precedence order that underlies some of the association rule bases from the literature.

References

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Proc. of the 20th Intl. Conf. on Very Large Data Bases (VLDB '94), San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1994) 487–499
2. Bastide, Y., Taouil, R., Pasquier, N., Stumme, G., Lakhal, L.: Mining Minimal Non-Redundant Association Rules Using Frequent Closed Itemsets. In: Proc. of the Computational Logic (CL '00). Volume 1861 of LNAI., Springer (2000) 972–986
3. Kryszkiewicz, M.: Representative Association Rules. In: Proc. of the 2nd Pacific-Asia Conf. on Research and Development in Knowledge Discovery and Data Mining (PAKDD '98), Melbourne, Australia, Springer-Verlag (1998) 198–209
4. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Closed set based discovery of small covers for association rules. In: Proc. 15emes Journees Bases de Donnees Avancees (BDA). (1999) 361–381
5. Duquenne, V.: Contextual implications between attributes and some representational properties for finite lattices. In: Beitraege zur Begriffsanalyse, B.I. Wissenschaftsverlag, Mannheim (1987) 213–239
6. Luxenburger, M.: Implications partielles dans un contexte. Mathématiques, Informatique et Sciences Humaines **113** (1991) 35–55

7. Kryszkiewicz, M.: Concise Representations of Association Rules. In: Proc. of the ESF Exploratory Workshop on Pattern Detection and Discovery. (2002) 92–109
8. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering Frequent Closed Itemsets for Association Rules. In: Proc. of the 7th Intl. Conf. on Database Theory (ICDT '99), Jerusalem, Israel (1999) 398–416
9. Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., Lakhal, L.: Computing Iceberg Concept Lattices with TITANIC. Data and Knowledge Engineering **42**(2) (2002) 189–222
10. Valtchev, P., Missaoui, R., Godin, R.: Formal Concept Analysis for Knowledge Discovery and Data Mining: The New Challenges. In: Proc. of the 2nd Intl. Conf. on Formal Concept Analysis, Springer Verlag (Feb 2004) 352–371
11. Szathmary, L., Napoli, A., Kuznetsov, S.O.: ZART: A Multifunctional Itemset Mining Algorithm. In: Proc. of the 5th Intl. Conf. on Concept Lattices and Their Applications (CLA '07), Montpellier, France (Oct 2007) 26–37
12. Zaki, M.J.: Scalable Algorithms for Association Mining. IEEE Transactions on Knowledge and Data Engineering **12**(3) (2000) 372–390
13. Zaki, M.J., Hsiao, C.J.: CHARM: An Efficient Algorithm for Closed Itemset Mining. In: SIAM Intl. Conf. on Data Mining (SDM' 02). (Apr 2002) 33–43
14. Pei, J., Han, J., Mao, R.: CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In: ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery. (2000) 21–30
15. Bastide, Y., Taouil, R., Pasquier, N., Stumme, G., Lakhal, L.: Mining frequent patterns with counting inference. SIGKDD Explor. Newsl. **2**(2) (2000) 66–75
16. Ganter, B., Wille, R.: Formal concept analysis: mathematical foundations. Springer, Berlin/Heidelberg (1999)
17. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New Algorithms for Fast Discovery of Association Rules. In: Proc. of the 3rd Intl. Conf. on Knowledge Discovery in Databases. (August 1997) 283–286
18. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD '00), ACM Press (2000) 1–12
19. Szathmary, L.: Symbolic Data Mining Methods with the Coron Platform. PhD Thesis in Computer Science, Univ. Henri Poincaré – Nancy 1, France (Nov 2006)