MEIT: Memory Efficient Itemset Tree for Targeted Association Rule Mining

Philippe Fournier-Viger[1], Espérance Mwamikazi[1], Ted Gueniche[1] and Usef Faghihi[2]

[1]Department of Computer Science, University of Moncton, Canada
[2]Department of Computer Science, Sull Ross State University, TX, USA

philippe.fournier-viger@umoncton.ca,{eem7706, etg8697}@umoncton.ca, ufaghihi@sulross.edu

Abstract. The Itemset Tree is an efficient data structure for performing targeted queries for itemset mining and association rule mining. It is incrementally updatable by inserting new transactions and it provides efficient querying and updating algorithms. However, an important limitation of the IT structure, concerning scalability, is that it consumes a large amount of memory. In this paper, we address this limitation by proposing an improved data structure named MEIT (*M*emory *E*fficient *I*temset *T*ree). It offers an efficient node compression mechanism for reducing IT node size. It also performs on-the-fly node decompression for restoring compressed information when needed. An experimental study with datasets commonly used in the data mining literature representing various types of data shows that MEIT are up to 60 % smaller than IT (43% on average).

Keywords: frequent pattern mining, association rule mining, itemset mining, itemset tree, memory constraint, targeted queries

1 Introduction

Association rule mining [1] is a fundamental data mining task with wide applications [2]. It consists of discovering associations in a transaction database. To mine association rules in a database, a user has to provide two thresholds, namely the minimum confidence and minimum support thresholds [1]. Several algorithms have been proposed to discover association rules such as Apriori, FPGrowth, HMine, Eclat and TopKRules [1, 2, 3, 4, 10, 11]. However, those algorithms are batch algorithms, i.e. if new data is added to the input database, users need to run the algorithms again to get updated results. This is inefficient when new data is regularly added to databases. To address this problem, incremental versions of batch algorithms were proposed [5, 6]. Nevertheless, these algorithms still suffer from an important limitation. That is, they are designed to discover (and update) all association rules in a database (meeting the user-defined thresholds mentioned above) rather than allowing the user to perform targeted queries. Targeted queries are useful for applications where the user wants to discover association rules involving a subset of the items contained in a database,

instead of all items [7, 8]. To process targeted queries for association rule mining efficiently in the context of static or incremental databases, the *I*temset *T*ree (IT) data structure was proposed [7, 8]. The IT is a tree structure, which can be incrementally updated and efficiently queried. The IT structure allows performing a vast array of important targeted queries such as (1) calculating the frequency of a given set of items, (2) discovering all valid association rules given a set of items as antecedent and (3) finding all frequent itemsets subsuming a set of items and their support [7, 8]. The IT structure has various applications such as predicting missing items in shopping carts in real-time [9]. However, ITs are inefficient when it comes to memory efficiency. Thus, to use ITs for large and/or incremental databases, we need to improve their memory efficiency. Given this limitation, an important research question is: Could we design a more memory efficient structure for targeted association rule mining? In this paper, we answer this question positively by proposing an improved IT structure that we name the *M*emory *E*fficient *I*temset *Tree* (MEIT).

The contributions of this work are twofold. First, we propose the MEIT structure. It incorporates effective tree node compression and decompression mechanims to reduce the information stored in IT nodes and restore it when needed. Second, we perform an extensive experimental study on six datasets commonly used in the data mining literature to compare the MEIT and IT data structures. Results show that MEIT is up to 60 % smaller than an IT (43% on average).

The remainder of this paper is organized as follows. Section 2 reviews the problem of association rule mining and the definition of IT. Section 3 describes the MEIT. Section 4 presents the experimental study. Finally, Section 5 draws a conclusion and discusses future work.

2 Related Work

Association rule mining is a fundamental data mining problem [2]. It is stated as follows [1]. Let $I = \{a_1, a_2, \ldots a_n\}$ be a finite set of items. A transaction database is a set of transactions $T=\{t_1,t_2\ldots t_m\}$ where each *transaction* $t_j \subseteq I$ $(1 \leq j \leq m)$ represents a set of items purchased by a customer at a given time. An *itemset* is an unordered set of distinct items $X \subseteq I$. The *support count* of an itemset X is denoted as $sup(X)$ and is defined as the number of transactions that contain X. An *association rule* $X \rightarrow Y$ is a relationship between two itemsets X, Y such that $X, Y \subseteq I$ and $X \cap Y = \emptyset$. The *support of a rule* $X \rightarrow Y$ is defined as $sup(X \rightarrow Y) = sup(X \cup Y) / |T|$. The *confidence of a rule* $X \rightarrow Y$ is defined as $conf(X \rightarrow Y) = sup(X \cup Y) / sup(X)$. The *problem of mining association rules* [1] is to find all association rules in a database having a support no less than a user-defined threshold *minsup* and a confidence no less than a user-defined threshold *minconf*. For instance, Figure 1 shows a transaction database (left) and the corresponding association rules (right) for *minsup* = 0.5 and *minconf* = 0.5. For example, rule $\{1\} \rightarrow \{4\}$ has a support of 0.5 because it appears in two transactions out of 6 (*t1* and *t4*). Furthermore, it has a confidence of 0.75 because {1, 4} appears in two transactions while {1} appears in 3 transactions. Mining associations is generally done in two steps [1]. Step 1 is to discover all frequent itemsets in the database (itemsets appearing in at least *minsup* × |T| transactions). Step 2 is to generate association rules

using the frequent itemsets found in Step 1. For each frequent itemset X, pairs of frequent itemsets P and Q = X – P are selected to generate rules of the form P→Q. For each such rule P→Q, if *sup*(P→Q) ≥ *minsup and conf*(P→Q) ≥ *minconf*, the rule is output. The most popular algorithms for association rule mining such as FPGrowth, HMine, Eclat and Apriori [1, 2, 3, 4, 10] are batch algorithms.

TID	Transactions
t1	{1, 4}
t2	{2, 5}
t3	{1, 2, 3}
t4	{1, 2, 4}
t5	{2,5}
t6	{2,4}

ID	Rule	Supp.	Conf.
r1	{2}→{5}	0.33	0.4
r2	{5}→{2}	0.33	1.0
r3	{1}→{4}	0.33	0.66
r4	{4}→{1}	0.33	1.0
r5	{2}→{4}	0.33	0.4
r6	{4}→{2}	0.33	0.66

Fig. 1. A transaction database (left) and some association rules found (right)

As an alternative to batch algorithms, the *Itemset-Tree* data structure was proposed. It is a structure designed for efficiently processing targeted queries on a transaction database [7, 8]. An IT is built by recursively inserting transactions from a transaction database, or any other sources, into the tree. It can be incrementally updated by inserting new transactions after the initial tree construction. An IT is formally defined as a tree where each IT node *k* stores (1) an itemset *i(k),* (2) the support count *s*(*k*) of the itemset and (3) pointers to children nodes when the node is not a leaf. The itemset associated to an IT node represents a transaction or the intersection of some transactions [7]. The root of an itemset tree is always the empty set ∅.

Figure 2 shows the algorithm for inserting a transaction in an IT. For instance, Figure 3 shows the six steps for the construction of an IT for the database depicted in the left part of Figure 1. In Step A, the transaction {1, 4} is inserted as a child of the root with a support of 1. In Step B, the transaction {2, 5} is inserted as a child of the root with a support of 1. In Step C, the transaction {1, 2, 3} is inserted into the tree. Because {1, 4} and {1, 2, 3} share the same leading item {1} according to the lexical ordering of items, a new node is created for the itemset {1} with a support of 2, such that {1, 2, 3} and {1, 4} are its children. In Step D, the transaction {1, 2, 4} is inserted into the tree. Given that {1, 2, 3} and {1, 2 4} share the same first leading items according to the lexical ordering, a node {1, 2} is created with a support of 2 with nodes {1, 2, 3} and {1, 2, 4} as its children. In Step E, the transaction {2, 5} is inserted into the tree. Since the transaction is already in the tree, its support count is incremented by 1 and no node is created. Finally, in Step F, the transaction {2, 4} is inserted. Since this transaction shares the itemset {2} with {2, 5}, a node {2} with a support of 3 is created with {2, 4} and {2, 5} as its children. Note that when the support of a node is increased in an IT, the support of all its ancestors is also increased. The expected cost of transaction insertion in an IT is ≈ O(1) [7]. For a proof that the transaction insertion algorithm is correct, the readers are referred to the paper proposing the IT structure [7].

An IT allows performing efficient queries for itemset mining and association rule mining such as (1) calculating the frequency of a set of items, (2) discovering all valid association rules containing a set of items as antecedent and (3) finding all frequent

itemsets subsuming a set of items and their frequency. Because of space limitation, we here only briefly explain the query-processing algorithm for counting the support of an itemset. The other query processing algorithms work in a similar way and the readers are referred to [7, 8] for more details. The pseudo-code of the query processing algorithm for support counting is shown in Figure 4. Consider the case of calculating the support of itemset {1, 2}. The algorithm starts from the root. Since the query itemset {1, 2} is not contained in and is smaller than the root itemset, the algorithm will visits the root's child nodes, which are {1, 2} and {2, 5}. Then given that the query itemset is equal to the itemset of the node {1, 2}, the support of 2 attached to the node {1, 2} will be kept and the subtree of that node will not be explored. The subtree {2, 5} will not be explored because the last item of {2, 5} is larger than the last item of {1, 2}. The algorithm will terminate and return 2 as the support count of {1, 2}. The expected cost of calculating the frequency of an itemset with this algorithm is $\approx O(n)$, where n is the number of distinct items in it. Improved querying algorithms for IT have recently been proposed [8]. Our proposal in this paper is compatible with both the old and the new querying algorithms.

INSERT(a transaction to be inserted *s*, an itemset-tree *T*)
1. $r = root(T)$; $s(r) := s(r) + 1$;
2. **IF** $i(s) = i(r)$ **THEN exit**;
3. Choose $T_s = subtree(r)$ such that $i(root(T_s))$ is comparable with s
4. **If** T_s does not exist **THEN**
5. create a new son x for r, $items(x) = s$ and $f(x) = 1$;
6. **ELSE IF** $i(root(T_s)) \subset s$ **THEN** call Construct(s, T_s)
7. **ELSE IF** $s \subset i(root(T_s))$ **THEN** create a new node x as a son of r
8. and a father of $root(T_s)$; $i(x) := s$; $s(x) := s(root(T_s))+1$;
9. **ELSE** create two nodes x and y, x as the father of $root(T_s)$,
10. such that $i(x) = s \cap root(T_s)$, $s(x) = s(root(T_s))+1$,
11. and y as a son of x, such that $i(y) = s$, $s(y) = 1$.

Fig. 2. The algorithm for transaction insertion in an Itemset-tree

Lastly, note that IT should not be confused with *trie-based* structures used in pattern mining algorithms such as the FP-Tree structure used by FPGrowth [3]. In an FP-Tree, each node contains a single item and each branch represents a transaction. In an IT, each node represents a transaction or the intersection of some transactions [7, 8]. The trie-based structures and the IT are designed for a different purpose.

3 The Memory-Efficient Itemset-Tree

We now describe MEIT, our improved IT data structure for targeted association rule mining. MEIT is an enhanced form of IT that uses an efficient and effective node compression scheme to reduce the memory usage of IT. We have designed the MEIT based on three observations that are formalized by the following three properties.

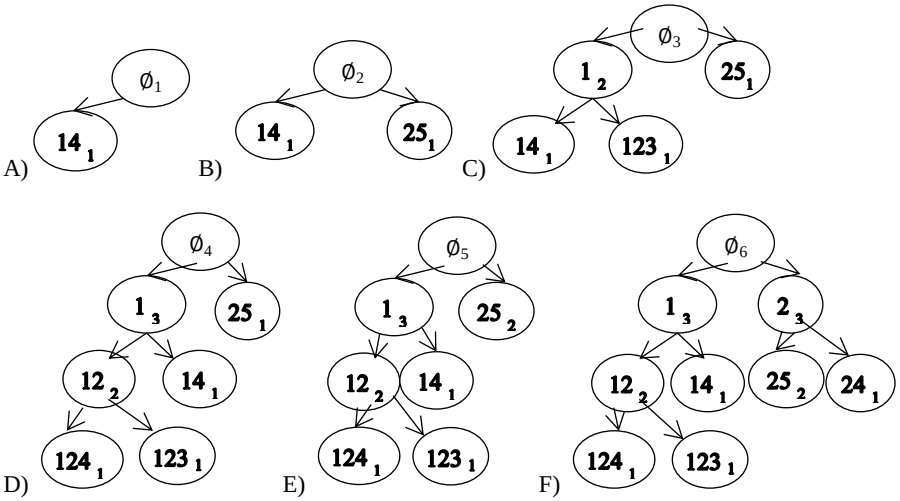

Fig. 3. An example of itemset-tree construction

COUNT(an itemset *s*, an itemset-tree *T*)
12. *r* = *root*(*T*);
13. **IF** *s* ⊆ *r* **THEN** *s*(*s*) := *s*(*s*)+ *s*(*r*);
14. **IF** *r* < *s* according to lexical order **AND** *last-item*(*r*) < *last-item*(*s*) **THEN**
15. **FOR EACH** subtree *T* of *r* **DO**
16. *s*(*s*) := *s*(*s*) + **COUNT**(*s*, *T*);
17. **RETURN** *s*(*s*);

Fig. 4. The algorithm for processing a support count query on an IT

Property 1. In an IT, transactions are inteserted by traversing branches in a top-to-bottom manner. **Rationale**. The insertion algorithm is given in Figure 3 and detailed in [7]. As it can be seen from the pseudo-code of this algorithm, the tree is always traversed in a top-to-bottom order to find the appropriate location for inserting a transaction (either by creating new node(s) or by raising the support of an existing node).

Property 2. Queries on an IT are always processed by traversing tree branches in a top-to-bottom manner. **Rationale**. In Figure 4, we have presented the pseudo-code of the algorithm for processing a support count query. As it can be seen from the pseudo code, the branches from the tree are traversed from top to bottom rather than from bottom to top. Other querying algorithms are described in [7, 8] and they also respect this property.

Property 3. Let *k* be an IT node and *parent*(*k*) be its parent. The relationship *i*(*parent*(*k*)) ⊂ *i*(*k*) holds between *k* and its parents. More generally, this property is transitive. Therefore, it can be said that for any ancestor *x* of *k*, *i*(*x*) ⊂ *i*(*k*).

Example 1. Consider the itemset {1, 2, 4} of the leftmost leaf node of Figure 3(F). The itemset of this node contains the itemset {1, 2} of its parent node. The itemset of this latter node contains the itemset {1} of its parent node. This latter contains the itemset ∅ of its parent.

Based on the aforementioned properties, we propose an efficient scheme for compressing node information in IT to improve its memory efficiency. It comprises two mechanisms, which are node compression and node decompression.

Definition 1. Node compression. Consider a node k of an itemset tree having an uncompressed itemset $i(k)$ such that $i(k) \neq \emptyset$ (i.e. k is not the root). Suppose k has n ancestor nodes denoted as $ancestor_1(k)$, $ancestor_2(k)$, … $ancestor_n(k)$. Compressing node k consists of setting $i(k)$ to $i(k) / \bigcup_{m=1}^{n} i(ancestor_m(k))$.

Example 2. For instance, Figure 5 shows the construction of an IT where the node compression scheme is applied on each new node during the IT construction, for the dataset of Figure 1 (left). During Step A and Step B, the transaction {1, 4} and {2, 5} are inserted into the IT with no compression because their parent is the empty set. In Step C, the transaction {1, 2, 3} is inserted. A new node {1} is created as a common ancestor of {1, 2, 3} and {1, 4}. The nodes {1, 2, 3} and {1, 4} are thus compressed respectively as {2, 3} and {4}. In Step D, the transaction {1, 2, 4} is inserted. A new node {2} is created as a common ancestor of {1, 2, 3} and {1, 2, 4} (note that {1, 2, 4} is represented as {4} in the tree because it is compressed). The nodes {1, 2, 3} and {1, 2, 4} are compressed respectively as {3} and {4}. In Step E, the transaction {2, 5} is inserted. Because this transaction already appears in the tree, the support of the corresponding node and its ancestors is increased. Finally, in Step F, the transaction {2, 4} is inserted. A new node {2} is created as a common ancestor of {2, 4} and {2, 5}. The nodes {2, 4} and {2, 5} are compressed respectively as {4} and {5}. By comparing the compressed trees of Figure 5 with the corresponding itemset tree of Figure 3(F), we can see that the total number of items stored in nodes is greatly reduced by compression. The tree of Figure 5 contains 8 items, while the tree of Figure 3(F) contains 16 items.

Having described the node compression scheme, we next describe how decompression is performed to restore the original information.

Definition 2. Node decompression. Consider a node k of an itemset tree having a compressed itemset $i(k)$ such that $i(k) \neq \emptyset$ (i.e. k is not the root). Suppose that k has n ancestor nodes denoted as $ancestor_1(k)$, $ancestor_2(k)$, … $ancestor_n(k)$. Node decompression consists of calculating $i(k) \cup [\bigcup_{m=1}^{n} i(ancestor_m(k))]$ to obtain the uncompressed representation of $i(k)$.

Example 3. Consider the leftmost leaf node of Figure 5(F). It contains the itemset {4}, which is the compressed representation of the itemset {1, 2, 4} (cf. Figure 3(F)).

To restore the uncompressed representation, the union of the itemsets of the ancestor nodes with the itemset {4} is performed. The result is $\emptyset \cup \{1\} \cup \{2\} \cup \{4\} = \{1, 2, 4\}$. Note that the root can be ignored when performing the union of ancestor itemsets. This is because the root node always contains the empty set, and thus never changes the result of node decompression.

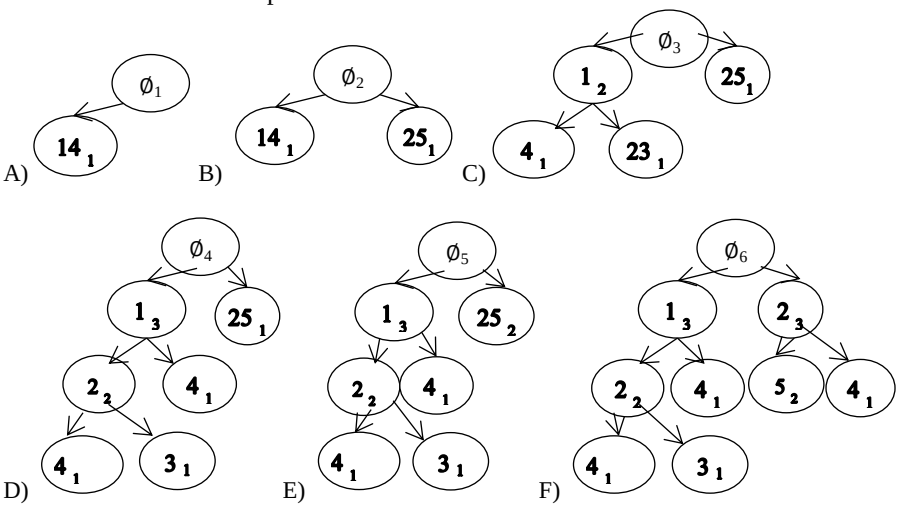

Fig. 5. An example of memory efficient itemset-tree construction

Definition 3. ***M*emory *E*fficient *I*temset *T*ree (MEIT)**. A MEIT is an IT where (1) node compression is applied during tree construction so that each node content is compressed and (2) where node decompression is performed on-the-fly during tree construction and query processing to restore node information when required.

We next show that node compression/decompression can be applied during tree construction and query processing, and that applying compression/decompression does not affect the result of queries.

Property 4. Node compression is always applicable during transaction insertion. To demonstrate that node compression can always be applied to compress new nodes that are created by the transaction insertion algorithm, we need to examine the conditions that have to be met to perform node compression. The first condition that has to be met to apply node compression to a given node is that the node has at least a parent node. The only node that does not have a parent in an IT is the root node and the root is never compressed by definition (cf. Definition 1). Second, to perform compression of a node *k*, it is necessary to access ancestor nodes' itemsets to perform their union. This condition is always met during tree construction because branches are always traversed from top to bottom rather than from bottom to top (Property 1). Therefore, the itemsets of ancestor nodes can be collected while traversing each branch so that the information required for compression is always available to the transaction inser-

tion algorithm. Therefore, any node that is visited by the transaction insertion algorithm can be compressed when it is created.

Property 5. Node decompression is always applicable for transaction insertion. To demonstrate that node decompression can always be applied by the transaction insertion algorithm, we use a similar reasoning as for Property 4. As mentioned, the transaction insertion algorithm traverses branches from top to bottom starting from the root (Property 1). This traversal order makes it possible to collect the itemsets of nodes visited while traversing a branch so that ancestor nodes' itemsets are always available for performing decompression of a node. Therefore, any node that is visited by the transaction insertion algorithm can be decompressed.

Property 6. Node decompression is always applicable during query processing. For query processing, only decompression is used to answer queries. It must be noted that by definition queries are not allowed to modify an IT (transaction insertion is not viewed as a query in IT terminology [7]). Similar to transaction insertion, queries are always processed by traversing IT branches from top to bottom rather than from bottom to top (Property 2), as it can be seen for example in the pseudo-code of Figure 4. Because of this traversal order, the information required for decompressing each node can be collected as the branches are traversed by keeping itemsets from the same branch into memory. Therefore, any node that is visited by one of the query processing algorithm can be decompressed.

Property 7. Performing node compression/decompression does not affect query results. Compressing an itemset and decompressing it does not result in a loss of information, by the definition of compression and decompression. Itemsets are compressed during transaction insertion and are decompressed on the fly when needed during transaction insertion and query processing. Because of this, the process of compression/decompression is completely transparent to the operations of the transaction insertion and query processing. Thus, it does not affect query results.

Implementing node compression efficiently. In the following, we explain why the complexity of node compression is linear. When a new node k is inserted into an MEIT, the cost of compression consists of performing the union of the itemset to be inserted m with the itemsets of the n ancestors of k, to calculate $i(k) = m / \bigcup_{p=1}^{n} i(ancestor_p(k))$. To perform the union of the ancestor itemsets efficiently, one can notice that when a node is inserted, all ancestor nodes already in the tree have been compressed. Given that compressed nodes do not share items with their ancestors, the union of the ancestor itemsets can be performed simply by a concatenation in linear time rather than by an expensive union operation. Now let's consider how to perform the set subtraction of the itemset $\bigcup_{p=1}^{n} i(ancestor_p(k))$ from m. To perform this operation efficiently, itemsets in the tree should always be sorted in lexicographical order. If itemsets are in lexicographical order (or any other total order), set subtraction can be performed efficiently by the means of a two-way comparison, which

requires scanning m and the itemset $\bigcup_{p=1}^{n} i(ancestor_p(k))$ at most one time. Thus the complexity of set subtraction is O(m+k) and the complexity of concatenation is linear.

Implementing node decompression efficiently. The complexity of node decompression is also linear. Let k be a node having n ancestors $ancestor_1(k)$, $ancestor_2(k)$, … $ancestor_n(k)$ when the tree is traversed from top to bottom from the root to k. To restore the itemset compressed in node k, it is necessary to perform the union of all itemsets stored in its ancestor nodes $ancestor_1(k)$, $ancestor_2(k)$, … $ancestor_n(k)$ with $i(k)$, as previously mentioned. Performing the union of several sets can be costly if implemented naïvely. To implement the union efficiently, we suggest using the following strategy. All items stored in each tree node should be sorted according to the lexical ordering (or any other total order). Then, while recursively traversing the tree from top to bottom to reach k, the union can be efficiently performed by simply concatening the itemsets $i(ancestor_1(k))$, $i(ancestor_2(k))$, … $i(ancestor_n(k))$ with $i(k)$, in that order. Thus the cost of node decompression is O(m), where m is the cardinality of the decompressed itemset. In practice, the cost of decompression is even smaller because intermediate concatenation results can be kept into memory when traversing a branch from top to bottom. For instance, consider a node k and its parent $parent(k)$. The concatenation of $i(k)$ and $parent(i(k))$ needs only to be performed once for all descendant nodes of k. This implementation strategy can greatly improve efficiency.

4 Experimental Study

We have implemented MEIT and IT in Java. The IT and MEIT source codes as well as all the datasets used in the experiments can be downloaded from http://goo.gl/hDtdt as part of the open-source SPMF data mining software. The following experiments are performed on a computer with a Core i5 processor running Windows 7 and 1 GB of free RAM. All memory measurements were performed using the core Java API. Experiments were carried on real-life and synthetic datasets commonly used in the association rule mining literature, namely *Accidents*, *C73D10K*, *Chess*, *Connect*, *Mushrooms*, *Pumsb* and *Retail*. Table 1 summarizes their characteristics.

Experiment 1. Memory Usage Comparison. We first compared the size of IT and MEIT for all datasets. To assess the efficiency of node compression, we measured the total number of items (including duplicates) stored in IT and MEIT nodes for each dataset. Results are shown in Table 1. As it can be seen in the third column of Table 1, the compression of nodes achieved by MEIT varies from 26.4 % to 88.7 % with an average of 58.7 %. The largest compression is achieved for dense datasets (e.g. *Mushrooms*), while less compression is achieved for sparse datasets (e.g. *Retail*). This is because transactions in dense datasets share more items. Thus, the intersection of transactions is larger than in sparse datasets. Thus, each IT node generally contains more items than for a sparse dataset. This gives more potential for compression in the MEIT.

Because MEIT only compress itemsets inside nodes and nodes contain other information such as pointers to child nodes, it is also important to compare the total

memory usage of MEIT and IT. To do that, we measured the total memory usage of IT and MEIT. Results are shown in Table 2. The third column shows that a compression of 13 % to 60 % is achieved, with an average of 43 %. We can conclude that the compression of itemset information has a considerable impact on the total memory usage.

Table 1. Datasets' Characteristics

Dataset	Transaction count	Distinct items count	Average transaction size
Accidents	340,183	468	22
C73D10K	10,000	1,592	73
Chess	3,196	75	37
Connect	67,557	129	43
Mushrooms	8,416	128	23
Pumsb	49,046	7,116	74
Retail	88,162	16,470	172

Table 1. Memory Usage of IT and MEIT (total items stored)

Dataset	IT size (items)	MEIT size (items)	Size reduction (%)
Accidents	16057697	5262555	67.2%
C73D10K	882196	508878	42.3%
Chess	196579	39550	79.9%
Connect	4446791	1092208	75.4%
Mushrooms	308976	35003	88.7%
Pumsb	4046316	2773440	31.5%
Retail	938568	690889	26.4%

Table 2. Total Memory Usage of IT and MEIT (MB)

Dataset	IT size (MB)	MEIT size (MB)	Size reduction (%)
Accidents	84.1	43.0	49%
C73D10K	4.0	2.6	36%
Chess	1.0	0.4	60%
Connect	21.8	9.0	59%
Mushrooms	1.7	0.7	60%
Pumsb	18.3	13.5	26%
Retail	7.3	6.4	13%

Experiment 2. Compression Overhead. We next compared the construction time of MEIT and IT for each dataset to assess the overhead in terms of execution time incurred by node compression and decompression. Results are shown in Table 3. As it can be seen, the overhead during tree construction is generally more or less the same for each dataset, averaging 45 %. We expected such an overhead because additional operations have to be performed to compress and decompress node information on-

the-fly. We also compared the query processing time of MEIT and IT for 10,000 random queries for each dataset to assess the overhead of on-the-fly decompression in terms of execution time for query processing. Results are shown in Table 4. As it can be seen, the overhead for query processing is generally more or less the same for each dataset, averaging 44 %. Again, we expected such an overhead because of extra operations performed for on-the-fly node decompression in MEIT. For real applications, we believe that this overhead is an excellent trade-off given that it allows building itemset-trees that can contains up to twice more information into memory (cf. Experiment 1). Moreover, as it can be seen in this experiment, the overhead in terms of execution time is predictable. It is more or less the same for each dataset no matter the size of the dataset or the type of data stored. In future work, we will assess the possibility of using caching algorithms to store the uncompressed form of frequently accessed nodes to reduce the overhead for popular queries.

Table 3. Tree construction time and 10K query processing time for IT/MEIT

Dataset	Tree construction time (s)		Time for processing 10K queries (s)	
	IT	MEIT	IT	MEIT
Accidents	3.71	5.82	880.4	1696.8
C73D10K	0.25	0.38	23.8	39.9
Chess	0.22	0.30	2.6	5.0
Connect	0.56	0.82	153.5	231.1
Mushrooms	0.21	0.23	1.0	2.1
Pumsb	0.63	0.95	134.2	202.1
Retail	4.33	7.75	84.8	193.7

5 Conclusion

An efficient data structure for performing targeted queries for itemset mining and association rule mining is the Itemset Tree. However, a major drawback of the IT structure is that it consumes a large amount of memory. In this paper, we addressed this drawback by proposing an improved data structure named the Memory Efficient Itemset Tree. During transaction insertion, it employs an effective node compression mechanism for reducing the size of tree nodes. Moreover, during transaction insertion or query processing, it relies on an on-the-fly node decompression mechanism for restoring node content.

Our experimental study with several datasets that are commonly used in the data mining literature shows that MEIT are up to 60 % smaller than IT, with an average of 43 %. In terms of execution time, results show that the overhead for on-the-fly decompression is predictable. The amount of overhead is more or less the same for each dataset no matter the amount of data or the type of data stored. We believe that the overhead cost is an excellent trade-off between execution time and memory given that it allows building itemset-trees that can store up to twice the amount of information for the same amount of memory. For future work, we will explore other possibilities

for compressing IT such as exploiting the links between nodes. We also plan to develop a caching mechanism, which would store the decompressed form of the most frequently visited nodes to improve efficiency for popular queries. We also plan to develop new querying algorithms for targeted top-k association rule mining [11, 12].

Source code of IT and MEIT as well as all the datasets used in the experiments can be downloaded from http://goo.gl/hDtdt as part of the open-source SPMF data mining software.

Acknowledgment. This work has been financed by an NSERC Discovery grant from the Government of Canada.

References

1. Agrawal, R., Imielminski, T., Swami, A.: Mining Association Rules Between Sets of Items in Large Databases. In: Proc. ACM Intern. Conf. on Management of Data, pp. 207-216, ACM Press (1993)
2. Han, J., Kamber, M.: Data Mining: Concepts and Techniques, 2nd ed. Morgan Kaufmann, San Francisco (2006)
3. Han, J., Pei, J., Yin, Y., Mao, R.: Mining Frequent Patterns without Candidate Generation. Data Mining and Knowledge Discovery 8, 53-87 (2004)
4. Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., Yang, D.: H-Mine: Fast and space-preserving frequent pattern mining in large databases. IIE Transactions 39 (6), 593-605 (2007)
5. Cheung, D. W., Han, J., Ng, V. T., Wong, C. Y.: Maintenance of discovered association rules in large databases: An incremental updating technique. In: Proceedings of the Twelfth International Conference on Data Engineering, pp. 106-114. IEEE Press (1996)
6. Ezeife, C. I., Su, Y.: Mining incremental association rules with generalized FP-tree. In: Proc. of the 15th Canadian Conf. on Artificial Intelligence (AI 2002). LNCS, vol. 2338, pp. 147-160. Springer, Heidelberg. (2002)
7. Kubat, M., Hafez, A., Raghavan, V. V., Lekkala, J. R., Chen, W. K.: Itemset trees for targeted association querying. IEEE Transactions on Knowledge and Data Engineering 15(6), 1522-1534 (2003)
8. Lavergne, J., Benton, R., Raghavan, V. V. : Min-Max itemset trees for dense and categorical datasets. In: Proc. 20th International Symposium on Methodologies for Intelligent Systems (ISMIS 2012). LNCS, vol. 7661, pp. 51-60. Springer, Heidelberg (2012)
9. Wickramaratna, K., Kubat, M., & Premaratne, K.: Predicting missing items in shopping carts. IEEE Transactions on Knowledge and Data Engineering 21(7), 985-998 (2009)
10. Zaki, M. J., Gouda, K.: Fast vertical mining using diffsets. In: Proc. of the ninth ACM SIGKDD international conference on Knowledge Discovery and Data mining, pp. 326-335. ACM Press (2003)
11. Fournier-Viger, P., Wu, C.-W., Tseng, V. S.: Mining Top-K Association Rules. In Proc. of the 25th Canadian Conf. on Artificial Intelligence (AI 2012). LNAI, vol. 7310, pp. 61-73. Springer, Heidelberg (2012)
12. Fournier-Viger, P., Tseng, V.S.: Mining Top-K Non-Redundant Association Rules. Proc. 20th International Symposium on Methodologies for Intelligent Systems (ISMIS 2012), LNCS vol. 7661, pp. 31- 40, Springer, Heidelberg (2012)