

# DRFP-tree: disk-resident frequent pattern tree

Muhaimenul Adnan · Reda Alhadj

Published online: 19 October 2007  
© Springer Science+Business Media, LLC 2007

**Abstract** Frequent itemset mining methods basically address time scalability and greatly rely on available physical memory. However, the size of real-world databases to be mined is exponentially increasing, and hence main memory size is a serious bottleneck of the existing methods. So, it is necessary to develop new methods that do not fully rely on physical memory; new methods that utilize the secondary storage in the mining process should be the target. This motivates the work described in this paper; we mainly propose (**Disk Resident Frequent Pattern**) DRFP-Growth as a disk based approach similar to FP-Growth. DRFP-growth uses DRFP-tree, which is treated exactly as FP-tree when constructed in main memory and gets into a modified structure when it turns into disk resident to overcome the main memory bottleneck. This way, we are able to mine for frequent itemsets from databases of arbitrary sizes without being restricted by the available physical memory. In other words, we initially try to mine the database using the original FP-growth; we expand into the secondary memory only if we run out of physical memory. So, DRFP-growth is very comparable to FP-growth for small databases and high support threshold values. On the other hand, using DRFP-growth, we are still able to mine huge databases for low support threshold values (the only limitation is the available secondary storage rather than physical memory). The reported test results demonstrate how the proposed approach succeeds for cases where main memory based approaches fail.

**Keywords** Frequent pattern · Association rules · Data mining · FP-growth · FP-tree · DRFP-growth · DRFP-tree

## 1 Introduction

Since it was identified in 1993 [3], the problem of frequent itemset mining (FIM) or association rule mining (ARM) has received great attention within the research community in the data mining arena. A formal definition of the problem, as given in [3, 10], can be stated as follows. Let  $I = \{i_1, i_2, \dots, i_m\}$  be a set of literals, called items. Let  $D$  be a set of transactions, where each transaction  $T$  is a set of items, such that  $T \subseteq I$ . Each transaction in  $D$  is assigned a unique identifier, denoted  $TID$ . A transaction  $T$  is said to contain an itemset  $X$  if  $X \subseteq T$ . The support of an itemset  $X$  in  $D$ , denoted  $\sigma_D(X)$ , is the fraction of the total number of transactions in  $D$  that contain  $X$ . Let  $\sigma$  ( $0 < \sigma < 1$ ) be a constant called minimum support, mostly user-specified. An itemset  $X$  is said to be frequent in  $D$  if  $\sigma_D(X) \geq \sigma$ . The set of all frequent itemsets, denoted  $L(D, \sigma)$ , is defined formally as,  $L(D, \sigma) = \{X : X \subseteq I, \sigma_D(X) \geq \sigma\}$ . An association rule is a correlation of the form  $X \Rightarrow Y$ , where  $X \subseteq I, Y \subseteq I$ , and  $X \cap Y = \phi$ . The rule  $X \Rightarrow Y$  has support  $\sigma_r$  in  $D$ , if  $\sigma_r\%$  of the transactions in  $D$  contain  $X \cup Y$ . The rule  $X \Rightarrow Y$  holds in  $D$  with confidence  $c_r$ , if  $c_r\%$  of the transactions in  $D$  that contain  $X$  also contain  $Y$ . Given  $D$ , the problem of association rules mining is to generate all association rules that have support and confidence greater than the prespecified minimum support ( $\sigma$ ) and minimum confidence ( $c$ ), respectively. The association rule mining task can be broken into two steps. The first step identifies all frequent  $k$ -itemsets from  $D$ , where  $k$  is the cardinality of the itemset; and the second step generates rules from these large itemsets in a relatively straightforward way. Consequently, re-

---

M. Adnan · R. Alhadj (✉)  
Department of Computer Science, University of Calgary, Calgary,  
AB, Canada  
e-mail: [alhadj@cpsc.ucalgary.ca](mailto:alhadj@cpsc.ucalgary.ca)

R. Alhadj  
Department of Computer Science, Global University, Beirut,  
Lebanon

searchers concentrate on effective and efficient methods for identifying frequent itemsets.

Most of the approaches developed to handle the frequent itemset mining problem use some kind of special memory resident data structures to compute the set of frequent itemsets from transactional databases. Some of the well-known data structures are Trie-structure [5], FP-tree [12], item-covers [15], COFI-tree [8], CATS-tree [7], CanTree [13], among others. These approaches are basically physical memory dependent. In other words, they address the time scalability issue of the problem and greatly rely on the available physical memory. On the other hand, huge real-world databases are very common as the size of the datasets to be mined is increasing exponentially; and hence the size of the physical memory is a bottleneck for existing mining approaches. If the size of the database is very large and/or the support threshold is low, it is very likely that these data structures would not fit into the available physical memory. This argument is supported by the works of Goethals [11] and Vaarandi [14], who investigated the memory requirement issues of some prominent association rule mining methods on some real world datasets. It is evident from their work that even the standard physical memory available in modern CPUs is not sufficient enough to mine for association rules from the tested large real world datasets. Therefore, it is necessary to tackle space scalability by either developing new secondary storage (disk) based approaches, or expanding existing approaches to facilitate using the secondary storage when the database size increases beyond the limit that could be handled using the available main memory. We argue that approaches capable of using the secondary storage are of great demand. Unfortunately, this area has not received enough attention in the community yet. The approach presented in this paper articulates our effort in this direction.

In this paper, we present DRFP-growth as a novel disk-based approach for mining frequent itemsets from large databases; and hence we eliminate the main memory size bottleneck. DRFP-growth is based on one of the most prominent and efficient (in terms of both time and space utilization) frequent itemset mining approaches described in the literature, namely FP-Growth proposed by Han et al. [12]. Further, we expand the FP-tree into DRFP-tree, which has the additional advantage that it turns into disk resident when necessary. In other words, we construct a memory resident FP-tree as long as the database size is manageable using the available physical memory; and we propose a new data structure to facilitate storing the tree on secondary storage as the tree size grows beyond the size that could be handled using main memory. Hence, DRFP-growth would be very much comparable to the traditional FP-growth approach. On the other hand, if the size of the database is very large and/or the support threshold is very low to the extent that we are unable to accommodate the tree in main memory, the proposed approach turns into disk-resident to overcome the space bot-

tleneck and to facilitate for mining frequent itemsets from huge databases.

The basic idea of the proposed approach is to sort the items within each transaction based on the global frequency of each item (as in the original FP-tree) and then considering the sequence of items in each transactions as a string to sort all the transactions in the dataset. The latter sorting is accomplished using the sort-merge strategy because huge databases do not fit in the main memory at once. Furthermore, the latter sorting helps in constructing the tree in depth-first order, which facilitates swapping branches from the tree to the disk as the tree grows and we run out of main memory. The detailed description and discussion in the sequel will highlight more the effectiveness of this process.

To sum up, the main contribution of the work described in this paper is solving the main memory bottleneck by developing a novel approach capable of utilizing the secondary storage when it is necessary to store the FP-tree as its size grows beyond the available main memory. Hence, our approach takes over and becomes vital when the traditional FP-growth and other memory intensive approaches fail because of physical memory limitation. This is supported by the test results reported in this paper.

The rest of the paper is organized as follows. Relevant related works are covered in Sect. 2. The different aspects of the proposed disk-based approach are presented in Sect. 3. Experimental results are reported in Sect. 4. Section 5 is conclusions and future work.

## 2 Related work

Since it was introduced in 1993, Apriori [3] has received great attention from researchers in the data mining arena. This algorithm generates candidate itemsets  $C_k$  for level  $k$  from frequent itemsets  $L_k$  at level  $k - 1$ . So, each iteration of the algorithm requires a complete database scan to get the counts for the candidate itemsets. This algorithm relies on the so-called monotonicity property that any subset of a frequent itemset must also be frequent. This monotonicity property helps to reduce the search space for the itemsets. But even with this property, in order to find a frequent itemset of length  $k$ , this technique must generate all the  $2^k$  frequent subsets. So, when the support threshold for mining is low and the size of the maximal frequent itemset is large, this approach is not efficient with respect to both space and time because of the sheer large number of candidate itemsets and the multiple database scans. There are many other techniques which are basically Apriori-like or level-wise in nature, and hence suffer from the same kind of problems.

Most of the association rule mining processes differ in how the transaction subsets are projected. While Apriori-like algorithms do not materialize the projected transactions, other research efforts like Amir et al. [5], Cheung et al. [7],

and Han et al. [12] rely on projecting the transactions or transactions’ subsets in one way or another. For instance, the work of Amir et al. [5] involves projecting all the transactions in the database in a trie structure. It takes transactions one by one, extracts the powerset of the transaction, and projects each member of the powerset into the trie structure to update the support count. Once built, the trie structure can be traversed in a depth first manner to find all the frequent patterns. If the support threshold is low and the database size is large, it is very likely that the trie would become closer to the powerset  $2^{|I|}$  of the set of items  $I$  present in the database  $D$ , and hence would suffer from the same main memory bottleneck problem. Han et al. [12] proposed a compact tree like structure, namely the FP-tree to project the transactions of the database. All nodes in the tree correspond to items in the database and nodes in the tree paths are sorted in descending ordering according to the global frequency of the represented items. Once the tree is built, the frequent patterns can be mined by the FP-growth approach [12], or alternatively by the COFI approach [8]. Eclat [15] is also depth-first approach like FP-growth and uses databases in a vertical format. Eclat keeps covers of items in main memory, i.e., for each item, Eclat keeps the list of transactions in which the item appears. Keeping the covers of itemsets this way facilitate the computation of support count for any itemset. Unfortunately, as suggested by Goethals [11] and Vaarandi [14], both Eclat and FP-Growth are highly memory intensive approaches and are not suitable for large transactional databases, especially when the specified minimum support threshold is low.

In their seminal work of the FP-growth approach, Han et al. [12] suggested that the FP-tree can be saved in the secondary memory if it can not be accommodated in main memory. Their proposal is a disk resident version of the FP-tree that would use the  $B^+$ -tree, which is a very popular structure. They proposed that the top level nodes of the  $B^+$ -tree can be split based on the roots of item prefix sub-trees, and the second level based on the common prefix paths, and so on. But their work remains as suggested future work and never came to fruition. Another research group who realized the need for disk-resident trees is Cheung et al. [6]. They proposed an approach which can construct the suffix tree on disk using a bounded amount of physical memory. The suffix tree is a very important data structure for sequence matching, and plays very important role, especially for DNA sequence matching. Their proposed algorithm, called *DynaCluster*, is an improvement over an existing approach *Prepar*. It uses a dynamic partitioning technique to cluster the DNA sequences. They construct small suffix trees for each of these dynamic clusters, which can be saved on disk. Using dynamic clusters, they are able to improve the locality of the DNA sequences, which in turn reduces the disk I/O cost and makes the approach more efficient compared to the traditional *Prepar* approach.

### 3 DRFP-growth and DRFP-tree

In this section, we describe the proposed disk based approach, which is very efficient and effective, realistic and practical way to mine frequent patterns from very large databases without being bounded by the size of the physical memory. The proposed approach is divided into various steps. The first step is dedicated to the preprocessing of the database required to prepare the transactional database for the whole process. The second step involves a novel method to materialize the database on disk in a way that simultaneously provides both the horizontal and vertical views of the database. Finally, the actual mining is performed by invoking DRFP-growth that utilizes DRFP-tree; these are variants of the FP-growth and FP-tree, respectively.

#### 3.1 Data preprocessing

The target of the data preprocessing step is sorting the items within each transaction, and then sorting the transactions in the database. We order items within each transaction by their support frequency, and all transactions are ordered based on the sequence of items in each transaction treated as one string. To do this, let the items in the database constitute the alphabet. When need to be sorted, item  $x$  precedes item  $y$  in a sequence if and only if the frequency of item  $x$  is greater than or equal to the frequency of item  $y$ . Further, strings of items (transactions) are sorted by the same way as words in the dictionary; but for the preprocessing step, transactions are sorted in descending order rather than ascending. In other words, transactions are ordered within the database such that for any two transactions  $T_i$  and  $T_j$ , we say  $T_i > T_j$ , i.e.,  $T_i$  precedes  $T_j$  in the final ordering of the transactions if and only if  $T_i$  and  $T_j$  agree on the first  $k \geq 0$  items, and the frequency of the  $(k + 1)^{th}$  item in  $T_i$  is greater than the frequency of the  $(k + 1)^{th}$  item in  $T_j$ , if any.

The preprocessing step is accomplished by invoking Algorithm 1, which uses the following functions. Consider a database  $\mathbf{D} = \{T_1, T_2, \dots, T_i, \dots, T_n\}$  of  $n$  transactions, where the  $i^{th}$  transaction,  $T_i = \{i_{i_1}, i_{i_2}, \dots, i_{i_n}\}$ , contains  $i_n$  items, and  $i_{i_k}$  is the  $k^{th}$  item in transaction  $T_i$ .  $\mathbf{Freq}(i)$  is defined as the fraction of the total number of transactions that belong to  $\mathbf{D}$  and include item  $i$ ;  $\mathbf{Sort}(T_i)$  is defined as a mapping from  $T_i : \{i_{i_1}, i_{i_2}, \dots, i_{i_n}\} \rightarrow T_i^S : \{i_{i_1}^S, i_{i_2}^S, \dots, i_{i_k}^S, \dots, i_{i_n}^S\}$ , where  $\mathbf{Freq}(i_{i_k}^S) \geq \mathbf{Freq}(i_{i_{k+1}}^S)$ , for all  $k, 1 \leq k < (i_n - 1)$ , and each item in  $T_i$  also exists in  $T_i^S$ ; and  $\mathbf{Order}(T_i, T_j)$  is defined as:

$$\mathbf{Order}(T_i, T_j) = \begin{cases} 1 & \text{if } \mathbf{Freq}(i_{i_k}) > \mathbf{Freq}(i_{j_k}) \\ & \text{and } \mathbf{Freq}(i_{i_h}) = \mathbf{Freq}(i_{j_h}) \\ & \forall h, 0 \leq h < k, \\ 0 & \text{if } \mathbf{Freq}(i_{i_h}) = \mathbf{Freq}(i_{j_h}) \\ & \forall h, 0 \leq h \leq j_n \text{ and } i_n \geq j_n, \\ -1 & \text{otherwise.} \end{cases}$$

**Algorithm 1** Preprocessing the database**procedure** PREPROCESS(**D**)

The original database **D** is preprocessed to get the sorted database **D<sup>S</sup>**.

1. Scan **D** to collect the frequency of each item that appears in the database.
2. Sort the items in each transaction of **D**, to produce the new database **D<sup>ˆ</sup>**, such that  $\text{Sort}(T_i) = \hat{T}_i$ , where  $T_i$  is the  $i^{\text{th}}$  transaction of **D** and  $\hat{T}_i$  is the  $i^{\text{th}}$  transaction of **D<sup>ˆ</sup>**.
3. Apply the sort-merge algorithm [9] to sort the transactions of **D<sup>ˆ</sup>** to produce a new Database **D<sup>S</sup>** =  $\{T_1^S, T_2^S, \dots, T_j^S, \dots, T_n^S\}$ , such that  $\text{Order}(T_j^S, T_{j+1}^S) \geq 0, \forall j, 1 \leq j < n$ .
4. **return** **D<sup>S</sup>**.

**end procedure****Table 1** Example transactional database **D**

TID	ItemsBought
T001	A, B, D, E, F, G, K, L
T002	A, C, D, G
T003	E, F, H, I, O
T004	A, B, C, D, E
T005	A, B, E, F, G, M
T006	C, E, F, S

**Table 2** Reorganized **D<sup>S</sup>** produced from Table 1

TID	ItemsBought
T001	E, A, F, B, D, G, K, L
T005	E, A, F, B, G, M
T004	E, A, B, C, D
T006	E, F, C, S
T003	E, F, H, I, O
T002	A, C, D, G

The objective of the preprocessing step is to reorganize the database **D** into **D<sup>S</sup>** =  $\{T_1^S, T_2^S, \dots, T_j^S, \dots, T_n^S\}$ , such that  $\text{Order}(T_j^S, T_{j+1}^S) \geq 0, \forall j (1 \leq j < n)$ ; and for each transaction,  $T_j^S (1 \leq j \leq n)$  in **D**, there is a corresponding transaction,  $T_k^S (1 \leq k \leq n)$  in **D<sup>S</sup>** such that  $T_k^S = \text{Sort}(T_j)$ .

The sort-merge algorithm [9] involves two phases. The first phase splits the file into chunks, each called run and can fit in the main memory; then each run is sorted individually. During the second phase, runs are recursively merged until we get a single sorted run that includes all the transactions from the original database. Actually, Steps 2 & 3 of Algorithm 1 are pipelined such that while sorting individual transactions we can split the file into several runs and sort each run before it is stored back to the disk.

To illustrate the preprocessing step, Table 1 shows a sample transactional database which consists of 6 transactions; and Table 2 presents the corresponding reorganized database after invoking Algorithm 1.

## 3.2 Materializing the database

After the preprocessing step, the database is materialized into DRFP-tree, which simultaneously gives both the horizontal and vertical views of the database. The horizontal view provides information about which items appear in a particular transaction, and the vertical view gives information about the transactions that contain a particular item. DRFP-tree extends the FP-tree into disk resident structure. The proposed approach has the additional advantage that the process is no longer limited by the size of the physical memory. Rather, the space scalability problem is handled in a realistic, practical and applicable way; the upper limit is the size of the free secondary storage.

The basic idea in constructing the DRFP-tree is altering the structure of the FP-tree such that instead of one single item, each node now corresponds to a skewed sub-path from the tree, i.e., consecutive sequence of items. The latter sequence has the unique characteristics: (1) the first node is either the root or a child of a node that has at least two children, (2) each intermediate node has a single child, and (3) the last node is either a leaf or a node that has at least two children. Items in a skewed sub-path are ordered according to their global frequency in descending order, and hence maintain the key property of the FP-tree. Constructing the DRFP-tree in this way has the additional advantage that it is more compressed than the regular FP-tree.

Figure 1(a) presents the traditional memory based FP-tree for the transactional database in Table 2; and Fig. 1(b) shows the same FP-tree with all the *subpaths* identified with boundaries. Recall that each node in a *subpath* has a single child except the last node, which is either a leaf node or a node with at least two children. For instance, the *subpath* marked as  $S_3$  in Fig. 1(b) has the node sequence  $\{F, B\}$ , where node  $F$  has only node  $B$  as child, and  $B$  has two children  $D$  and  $G$ . In a step to produce the DRFP-tree, each *subpath* of the FP-tree is assigned a nondecreasing number in a depth-first (**DFS**) manner, where the assigned number is the unique identifier of the *subpath* in the DRFP-tree. The

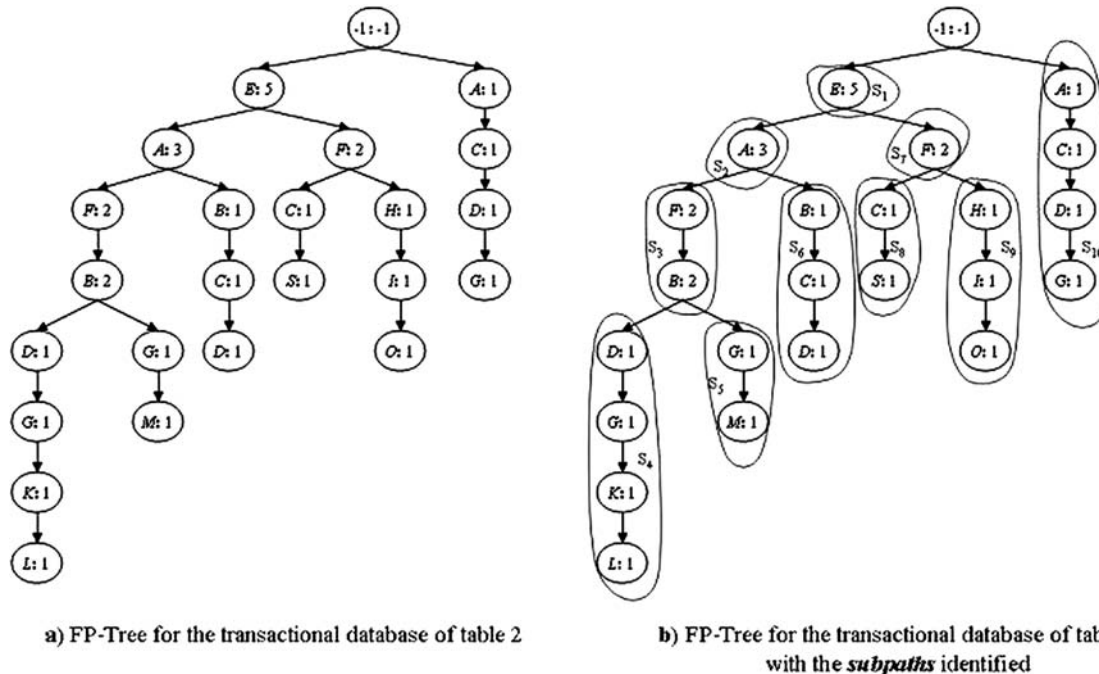


Fig. 1 FP-Tree for the transactional database in Table 2

Table 3 The *subpath* descriptor table

Id	parentId	length	itemArray	countArray
S <sub>1</sub>	-1	1	E	5
S <sub>2</sub>	S <sub>1</sub>	1	A	3
S <sub>3</sub>	S <sub>2</sub>	2	F, B	2, 2
S <sub>4</sub>	S <sub>3</sub>	4	D, G, K, L	1, 1, 1, 1
S <sub>5</sub>	S <sub>3</sub>	2	G, M	1, 1
S <sub>6</sub>	S <sub>2</sub>	3	B, C, D	1, 1, 1
S <sub>7</sub>	S <sub>1</sub>	1	F	2
S <sub>8</sub>	S <sub>7</sub>	2	C, S	1, 1
S <sub>9</sub>	S <sub>7</sub>	3	H, I, O	1, 1, 1
S <sub>10</sub>	-1	4	A, C, D, G	1, 1, 1, 1

Table 4 The *subpath* list

Id	FilePtr
S <sub>1</sub>	filePos(S <sub>1</sub> )
S <sub>2</sub>	filePos(S <sub>2</sub> )
S <sub>3</sub>	filePos(S <sub>3</sub> )
S <sub>4</sub>	filePos(S <sub>4</sub> )
S <sub>5</sub>	filePos(S <sub>5</sub> )
S <sub>6</sub>	filePos(S <sub>6</sub> )
S <sub>7</sub>	filePos(S <sub>7</sub> )
S <sub>8</sub>	filePos(S <sub>8</sub> )
S <sub>9</sub>	filePos(S <sub>9</sub> )
S <sub>10</sub>	filePos(S <sub>10</sub> )

motivation for numbering the *subpaths* in DFS manner will become evident in the forthcoming subsection.

Tables 3 and 4 list the tabular representation of the DRFP-tree for the transactional database in Table 2. We require two files to represent the DRFP-tree. The first file keeps the *subpaths*; we call this file the *subpath* descriptor table. The record structure for this file has the following fields: Id, parentId, length, itemArray and countArray, where Id is the unique identifier of the *subpath*, parentId is parent *subpath*'s identifier, length is the sequence length, itemArray is an array for the actual sequence of items, and each entry in the integer array countArray is the count of the corresponding item in itemArray. As each *subpath* may have item-sequence of different length, the actual size of each

record of the *subpath* descriptor table may vary based on the item-sequence length. So, the second file, called the *subpath* list, keeps track of the starting file pointers for each *subpath* in the *subpath* descriptor table.

The motivation for defining the DRFP-tree structure this way may be justified as follows. If we observe the construction and mining process of the traditional FP-tree carefully, we realize that the approach is designed in such a way that we need to traverse the tree top-down when we construct the tree in main memory. On the other hand, we need to traverse the tree bottom-up to collect the conditional pattern bases for constructing the conditional FP-trees during the mining phase. So, we never need to traverse the tree both top-down and bottom-up simultaneously. This observation enables us to design the DRFP-tree data structure in a way such that,



**Table 5** The header table for item  $B$

subpathId
$S_3$
$S_6$

**Table 6** The header table for item  $D$

subpathId
$S_4$
$S_6$
$S_{10}$

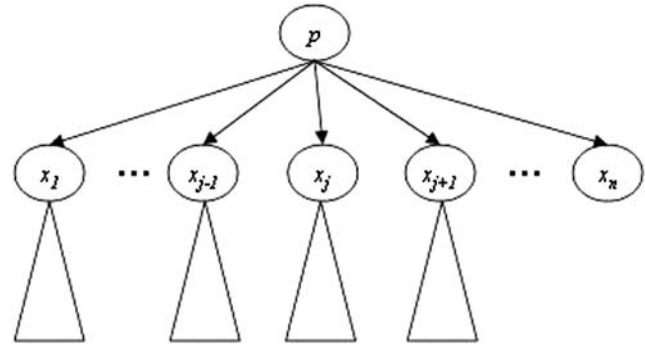
once saved, we traverse the tree only in a bottom-up fashion. This facilitates keeping reduced number of fields for the *subpath* data structure because after saving the DRFP-tree using the proposed data structure, we need to traverse the tree only during the mining phase where the bottom-up traversal is sufficient. The reader will have a better perspective on this aspect once we describe the DRFP-tree construction algorithm next in Sect. 3.2.1. Finally, we use separate file to save the *header* table for each item of the DRFP-tree. For instance, Tables 5 and 6, respectively, present the *header* tables for items  $B$  and  $D$  as given in the DRFP-tree represented in Tables 3 and 4. As item  $D$  appears in the three *subpaths* marked as  $S_4$ ,  $S_6$ , and  $S_{10}$ , the header table for item  $D$  contains 3 entries, corresponding to each of these three *subpaths*.

### 3.2.1 The algorithms

In this section, we describe the tree building and mining algorithms for the proposed disk based frequent pattern mining approach. The proposed approach is organized such that we continue to mine for frequent patterns from the transactional database using only the physical memory as long as there is enough physical memory to fit all the necessary data structures. However, we switch into using the secondary storage only when we run out of main memory.

Algorithm 2 is dedicated for the DRFP-tree building process. Initially, we start by reading the transactions from the database  $D^S$  one by one, and then projecting the transactions onto a traditional memory based FP-Tree called  $Tree_{DS}$ . If after adding some transactions into the tree structure, we find out that it is no longer possible to grow the tree to accommodate it into the physical memory, we reduce the tree in size by saving the *stale* subtrees of the tree  $Tree_{DS}$  on the secondary storage.

A subtree of  $Tree_{DS}$  is said to be *stale* if it is guaranteed that it will not be used further in the construction of the tree, i.e., while inserting the remaining transactions into the tree. This is possible because of the preprocessing step,



**Fig. 2** Chronological ordering of nodes

which sorts transactions in descending order using the sort-merge algorithm. The immediate gain of such ordering is realized in the *insertTransactionToTree* procedure, where any new branch generated during the insertion process becomes the right most child of its parent. So, for any node  $x_i$  in the memory based FP-Tree  $Tree_{DS}$ , if a sibling node  $x_j$  appears to the left of  $x_i$ , then  $x_j$  would be *older* than  $x_i$ . Similarly, any sibling node  $x_k$  that appears to the right of  $x_i$  would be *younger* in age. For example, the parent node  $p$  of the tree shown in Fig. 2 has  $n$  child nodes, namely  $x_1, x_2, \dots, x_j, \dots, x_n$ . Here nodes  $x_1, x_2, \dots, x_{j-1}$  are *older* than node  $x_j$  as these nodes appear to the left of  $x_j$ ; and nodes  $x_{j+1}, x_{j+2}, \dots, x_n$  are younger than node  $x_j$  as they appear to the right of node  $x_j$ . For a particular node, the left sibling subtrees will always be *stale* because transactions are ordered before they are added to the tree, and a new branch is always created as the right most child of its parent. In other words, we never need to consult/access a left sibling subtree while adding the remaining transactions of the sorted database  $D^S$ . For instance, after node  $x_n$  is added as a child of node  $p$  in Fig. 2, all the left siblings of  $x_n$ , namely  $x_1, x_2, \dots, x_{n-1}$ , become *stale*.

If after adding all the transactions to the memory based FP-tree we get the situation where part of the tree is on disk and part of the tree is in main memory (we call this state of the tree as *dirty*), then it is required to flush the rest of the tree from the main memory to the disk in DFS manner, and the result is the actual DRFP-tree. On the other hand, if we find out that it is possible to fit the whole tree into the main memory, we need to determine somehow whether there is enough main memory to keep all the future conditional FP-trees in the main memory; and hence invoke FP-growth for the mining process. However, if it is not possible to fit everything in main memory and the memory based FP-tree turns into DRFP-tree, then we invoke DRFP-growth, given in Sect. 3.2.3, for the mining process.

The steps of building the DRFP-tree are demonstrated in Fig. 3. Figure 3(a) shows the state of the tree after adding (in order) transactions  $T001, T005, T004, T006$ , and  $T003$  of the database  $D^S$ . Assume that there is enough main mem-

**Algorithm 2** Build the DRFP-Tree

---

```

procedure BUILDTREE( $D^S$ )
  repeat
     $tran \leftarrow$  ReadNextTransaction( $D^S$ )
    InsertTransactionToTree( $tran, Tree_{D^S}$ )
    /*  $Tree_{D^S}$  is the current memory based FP-Tree */
    if ( $Size(Tree_{D^S}) > MAX\_MEM$ ) then
      SaveStaleSubTreesToDisk( $Tree_{D^S}$ )
    /* save stale sub-trees of  $Tree_{D^S}$  to disk; stale sub-trees of  $Tree_{D^S}$  are chosen on the basis that they would no longer be accessed in future during the construction process*/
     $Tree_{D^S}.dirty \leftarrow 1$ 
    end if
  until eof( $D^S$ )
  if ( $Tree_{D^S}.dirty = 1$  Or  $ProjectedMemReq(Tree_{D^S}) > MAX\_MEM$ ) then
    SaveRestOfTreeToDisk( $Tree_{D^S}$ )
  end if
end procedure

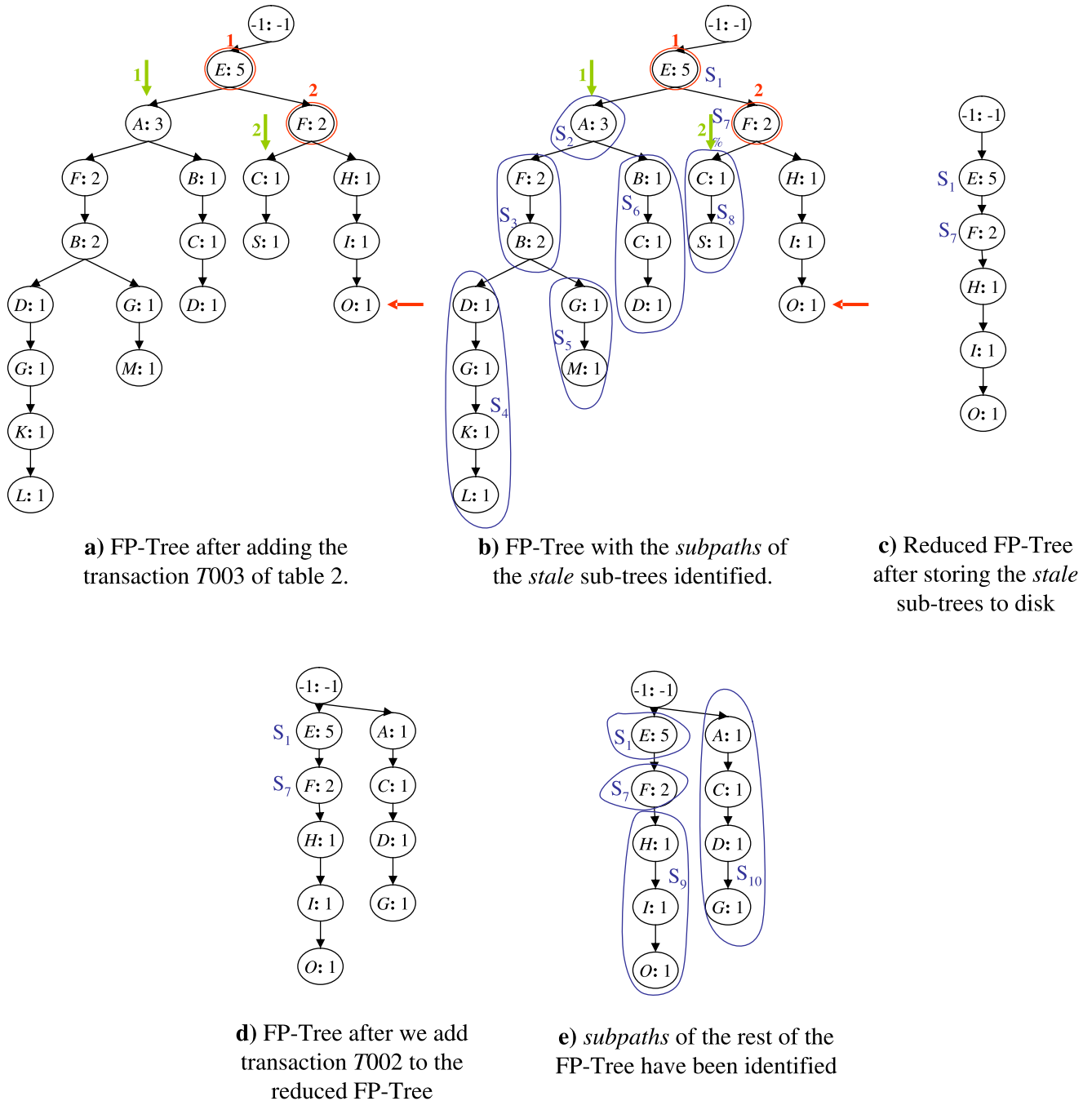
procedure SAVESTALESUBTREESTODISK( $Tree_{D^S}$ )
   $S \leftarrow \phi$ 
  /*  $S$  is a stack of nodes initially empty */
   $n \leftarrow$  FindLastLeafNode( $Tree_{D^S}$ )
  repeat
     $p \leftarrow$  Parent( $n$ )
    if  $p$  has older children than  $n$ 
      Push( $S, \langle p, n \rangle$ )
    end if
  until  $p \neq root$ 
     $\langle p, n \rangle \leftarrow$  Pop( $S$ )
  while  $p$  is not null do
    SaveSubTreeToDisk( $p, n$ )
     $\langle p, n \rangle \leftarrow$  Pop( $S$ )
  end while
end procedure

```

---

ory just to keep only 20 nodes in main memory, and we run out of main memory after adding node  $O$  of transaction  $T003$ . So, it is necessary to find the *stale* subtrees of the current memory based FP-tree. To do this, we first need to find the most recent node added to the tree. For the FP-tree in Fig. 3(a), this node is  $O$ . So, we traverse upwards from node  $O$  using the parent pointers to find candidate parent nodes, which may have some subtrees that would be *stale*. For the FP-tree in Fig. 3(a), the candidate parent nodes are  $B$  and  $F$ , which are marked with red circles. For these parent nodes, the subtrees rooted at nodes  $A$  and  $C$  are *stale*. These *stale* nodes are marked with green arrows. The next step is to save these *stale* subtrees to disk in order to reduce the size of the tree in main memory, and hence enable the tree to grow to accommodate new transactions.

Note that having all subtrees saved in **DFS** manner helps while reading the tree from the disk to get the conditional pattern bases during the mining phase; we will read all conditional pattern bases in a sorted order. This eliminates the need for one more sorting phase (to sort the conditional pattern bases) before we create the conditional FP-Trees from the conditional pattern bases. For the same reason, it should be noted that the *stale* subtree rooted at node  $A$  should be saved before the *stale* subtree rooted at node  $C$ . Saving the subtrees in **DFS** manner has the additional advantage that it would preserve the maximum locality among the *subpaths* of a particular transaction so as to minimize the disk bound I/O. Figure 3(c) shows the reduced tree after the *stale* subtrees are saved to disk. We can then resume the transaction insertion part, and Fig. 3(d) shows the tree after adding the last transaction  $T002$  to the memory based FP-tree. Lastly,



**Fig. 3** Steps of building the DRFP-tree

Fig. 3(e) shows the rest of the tree that remain in main memory, with the subpaths identified; these subpaths must be saved on disk in order to complete the DRFP-tree building process.

### 3.2.2 Determining memory requirements

Recall that after initially building the tree in main memory, we need to determine whether the tree is *dirty* (i.e., part of

the tree is in main memory and part of the tree is on disk), or whether we can accommodate all the future conditional FP-trees in main memory. Determining whether the tree is *dirty* or not is fairly straightforward, by just maintaining a flag. However, we need to make some clever assumptions for getting a close picture of how much main memory we need to accommodate all future conditional FP-trees in main memory. This is required because FP-growth is a progressive pattern growth approach, and we need to recursively mine the



conditional FP-trees in order to grow the frequent patterns. To increase the efficiency of the pattern mining approach, it would be great if it is possible to keep all future conditional FP-trees in main memory because this reduces disk bound I/O.

Let the number of different items in the current FP-tree be  $u$ , and let the current FP-tree has  $v$  nodes. The conditional FP-tree at the next level can have at most  $v - 1$  nodes. So, a very crude assumption on the amount of main memory required to accommodate all the future conditional FP-trees in main memory would be  $v + (v - 1) + (v - 2) + \dots + 2 + 1 = \frac{v(v+1)}{2}$ . But we know that the value of  $v$  can be very large, and we can have at most  $|u|$  conditional FP-trees in main memory at a time because there are only  $|u|$  different items in the initial FP-tree to begin with. So, a closer assumption of main memory requirement would be  $v + (v - 1) + (v - 2) + \dots + v - |u| + 1 = \frac{u(2v - |u| + 1)}{2}$ , if  $v > |u|$ . We can further improve on the assumption of the main memory requirement by keeping some additional information along with the *header* table of each item. We can keep the summation of path lengths,  $L_i$  of a particular item  $i$  along with the *header* table of item  $i$ . So, the revised assumption of the main memory requirement would be  $v + L_i + (L_i - 1) + (L_i - 2) + \dots + (L_i - |u| + 2) = \frac{(2L_i - |u| + 2)(u - 1)}{2}$ . Finally, we can write the closest assumption of the required main memory to accommodate all the future conditional FP-trees as

$$\max_i \left\{ \min \left\{ v + \frac{(2L_i - |u| + 2)(|u| - 1)}{2}, \frac{v(v + 1)}{2} \right\} \right\}.$$

### 3.2.3 DRFP-growth: disk based mining of frequent patterns

The proposed disk based version of the frequent pattern mining method is designed in such way that it can mine the frequent patterns using only physical memory if it can fit all the data structures required to store the tree and afterwards the data structures for the conditional FP-trees in physical memory. If either we can not accommodate the tree itself or we predict that we would be unable to accommodate in main memory the future conditional FP-trees created during the mining phase, we need to turn into DRFP-tree. As a result, if we do not need to save the FP-tree on disk, we invoke the traditional memory based FP-growth method to mine for frequent patterns from the memory based FP-tree. Otherwise, we invoke the DRFP-growth method to mine for frequent patterns from the DRFP-tree.

Algorithm 3 is roughly a disk based version of the FP-growth method. First, we check whether the DRFP-tree contains only single path. If it does, then we just generate all the combinations of itemsets that take part in that single path, and concatenate the itemsets with  $\alpha$  to grow frequent pattern  $\alpha$ . Otherwise, we search the disk based *header* table of

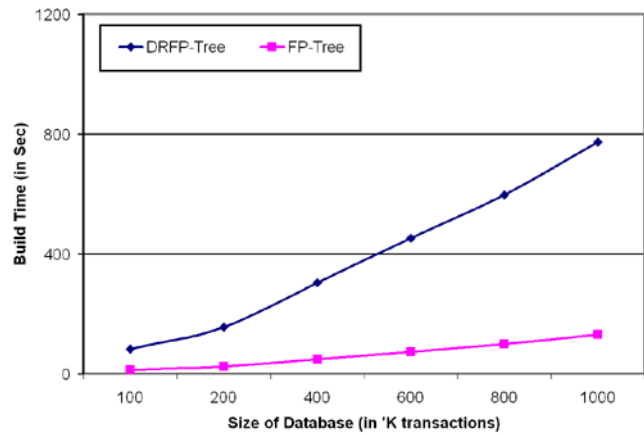


Fig. 4 Run time of BuildTree on subsets of various sizes from T1014

each item that appears in  $Tre_{DISK}$  and whose support meets the required threshold level in order to find the conditional pattern bases of those items. Note that the conditional pattern bases are already sorted. Once we find the conditional pattern bases, we first try to construct the conditional FP-tree in main memory. If we find out that we run out of main memory while constructing the tree, or we find out that we would not be able to fit all future conditional FP-trees in the physical memory, we store this conditional FP-tree on disk; we call the DRFP-Growth algorithm to mine this conditional FP-tree and to progressively grow the size of the frequent pattern. On the other hand, if we find out that we can accommodate the current conditional FP-tree and all future FP-trees in the physical memory, we do not save the conditional FP-trees on disk, and we initiate the mining process using the traditional memory based FP-growth method from the memory based conditional FP-trees.

## 4 Experimental results

In this section, we study the performance of the DRFP-growth approach on different datasets, as compared to the traditional memory based FP-Growth approach. The algorithm has been implemented in Java, and all the experiments have been conducted on an IBM Pentium IV machine with 2.0 GHz CPU and 512 MB main memory; running Windows-XP. All the datasets are synthetic and are generated using the method described in [4].

Figures 4 and 5 show the computational performance of the DRFP-tree construction method versus the traditional memory based FP-tree construction method. For these tree building methods, we restrict our approach to use only 256 MB of main memory. For the disk based approach, we first try to build the FP-tree using the available physical memory. If we run out of physical memory during the tree construction process, or we find out that we can not fit into main

**Algorithm 3** DRFP-Growth

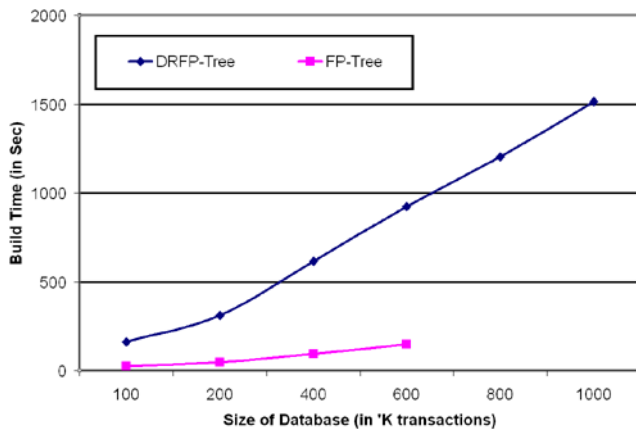
---

```

procedure DISKBASEDFPGRGROWTH( $Tree_{DISK}, \alpha$ )
  if checkForSinglePath( $Tree_{DISK}, P$ ) then
     $S \leftarrow$  generateSubSets( $P$ )
    /* generate the power set of the single path  $P$  */
    for each element  $\beta$  of  $S$  do
       $newPattern \leftarrow \beta \cup \alpha$ 
       $newPattern.support \leftarrow$  minSupport( $\beta$ )
    endFor
  else
    for each item in the header table of  $Tree_{DISK}$ 
       $\beta \leftarrow item \cup \alpha$ 
      /*  $\beta$  is the new pattern */
       $\beta.support \leftarrow$  FindSupport(item)
       $cpb \leftarrow$  ReadConditionalPatternBase( $Tree_{DISK}, \beta$ )
      /* read  $\beta$ 's conditional pattern base */
       $Tree_{\beta} \leftarrow$  BuildTree( $cpb$ )
      if  $Tree_{\beta} \neq \phi$  then
        if ( $Tree_{\beta}.dirty \neq 1$  and  $ProjectedMemReq(Tree_{\beta}) \leq MAX\_MEM$ ) then
          MemBasedFPGrowth( $Tree_{\beta}, \beta$ )
        else
           $Tree_{DISK\beta} \leftarrow$  DiskEquTree( $Tree_{\beta}$ )
          DiskBasedFPGrowth( $Tree_{DISK\beta}, \beta$ )
        end if
      end if
    end for
  end if
end procedure

```

---



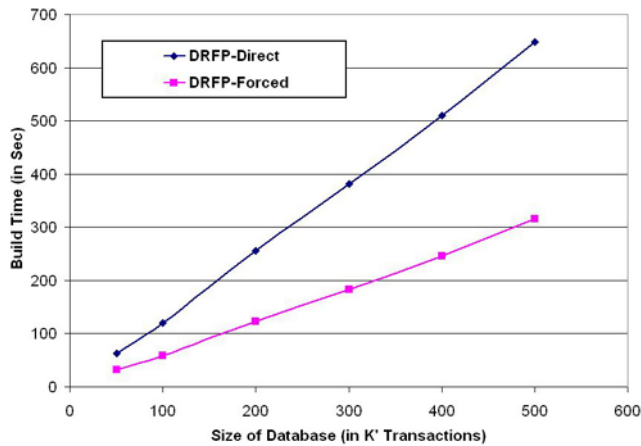
**Fig. 5** Run time of BuildTree on subsets of various sizes from T20I10

memory all the conditional FP-trees required for the mining process—the tree is then saved on the secondary storage. However, for the results reported in Figs. 4 and 5, we force the disk-based approach to save the trees on disk after the tree construction step, regardless of the available main

memory; this is done in order to show the worst case time requirements of the disk based approach on some particular databases.

Figure 4 reports the timing for the two tree construction methods when we vary the size of the databases from 100 K to 1 M transactions. For each of these databases within this size range, the average transaction length is 10 items, the average maximal potentially frequent itemset size is 4 items, and the number of different items in the databases is 1 K. Figure 5 reports the time requirements for databases with a similar size range and same number of distinct items; but, average transaction length is 20 items, and average maximal potentially frequent itemset size is 10 items. The support threshold to construct the trees in both cases is 0.1%.

Figures 4 and 5 show that the timing for both the disk based and memory based approaches increase with respect to the database size. And from the figures, it is quite evident that the disk based tree construction approach would require more time to build and save the tree on disk than the traditional memory based tree construction approach. This is true because the disk based approach performs I/O on sec-

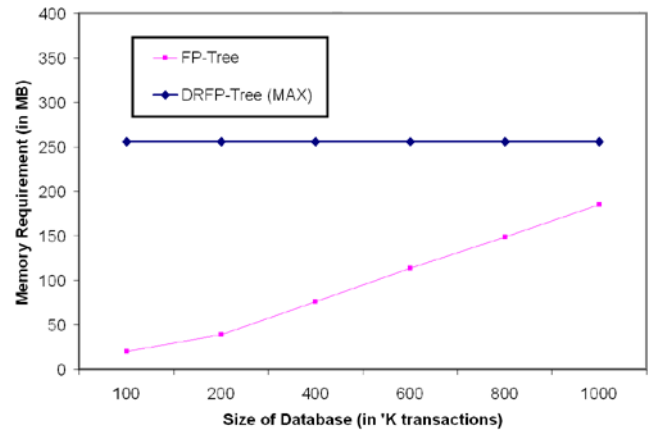


**Fig. 6** Run time of BuildTree on subsets of various sizes from T1015

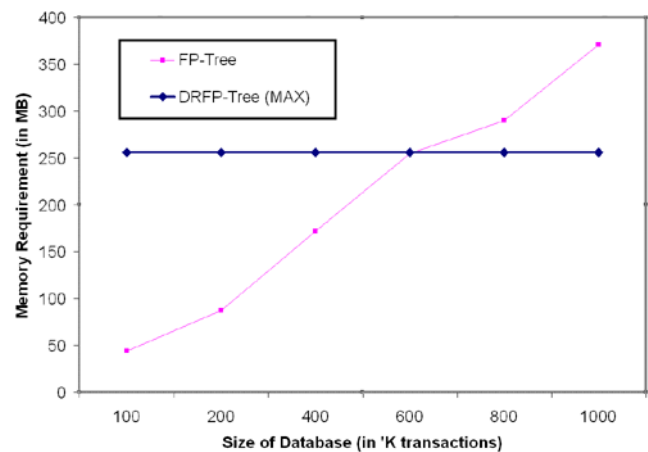
ondary storage, which is not the case for the memory based approach. But as Fig. 5 confirms, we would fail to keep the trees in main memory if the database is large and the support threshold is low. So, the traditional tree construction method fails in the case of Fig. 5 when the database size is over 600K. This scenario, however, does not happen for the database of Fig. 4 because here the average size of the transactions is low, and hence the tree does not grow fast as compared to the database of Fig. 5.

In Fig. 6, we present the time requirements of building the tree directly on the disk with using the minimum physical memory for various sizes of database T1015. We compare this result with the time that is required to build the trees using only 64 MB of physical memory and then forcing the trees to be saved on disk. In both cases, the support threshold was 0.1%. The direct construction curve depicts the worst case time requirements to construct the trees. Please note that, it is possible to have a direct construction of the trees on disk because of the nature of growth of the trees. We have already mentioned that, due to the preprocessing step the trees grow from top to bottom and from left to right; because of this the left subtrees are never accessed again during the construction process. So the minimum amount of memory that we require for the initial tree is an amount which can store two tree paths of maximum length.

Figures 7 and 8 report the memory requirement of the disk based tree construction method versus the memory requirement of the traditional memory based tree construction method for the same datasets used in the tests reported in Figs. 4 and 5, respectively. Figure 8 shows that when the database size is large, the memory required to keep the FP-tree goes beyond the restricted memory boundary. Hence, for large databases with low support threshold, it is not possible to keep the trees in the main memory, and it is necessary to expand into a disk resident structure as described in this paper. To demonstrate the power of the proposed disk-based approach, we run some tests by restricting the use of

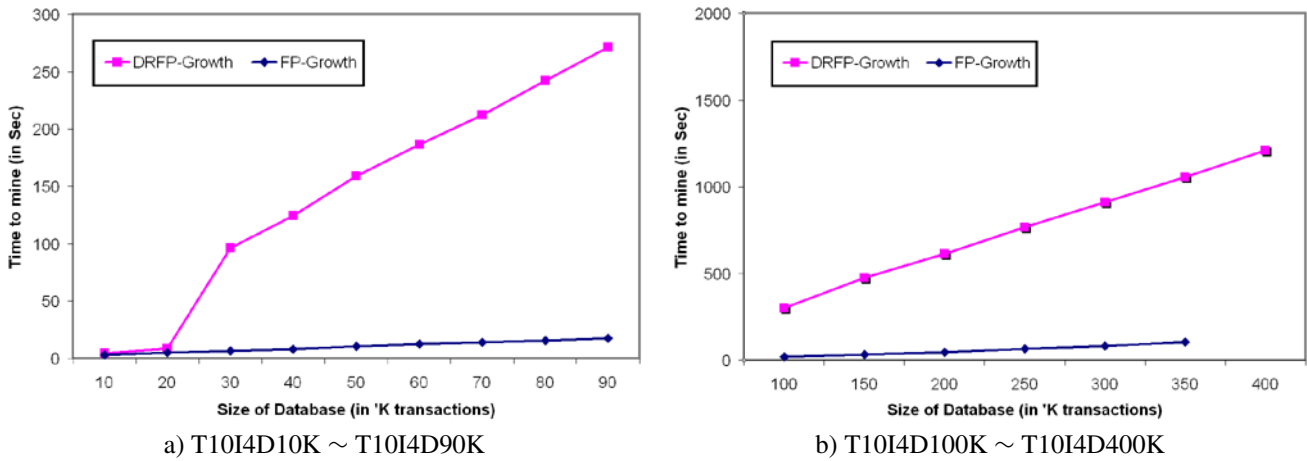


**Fig. 7** Memory requirement of BuildTree on subsets of various sizes from T1014

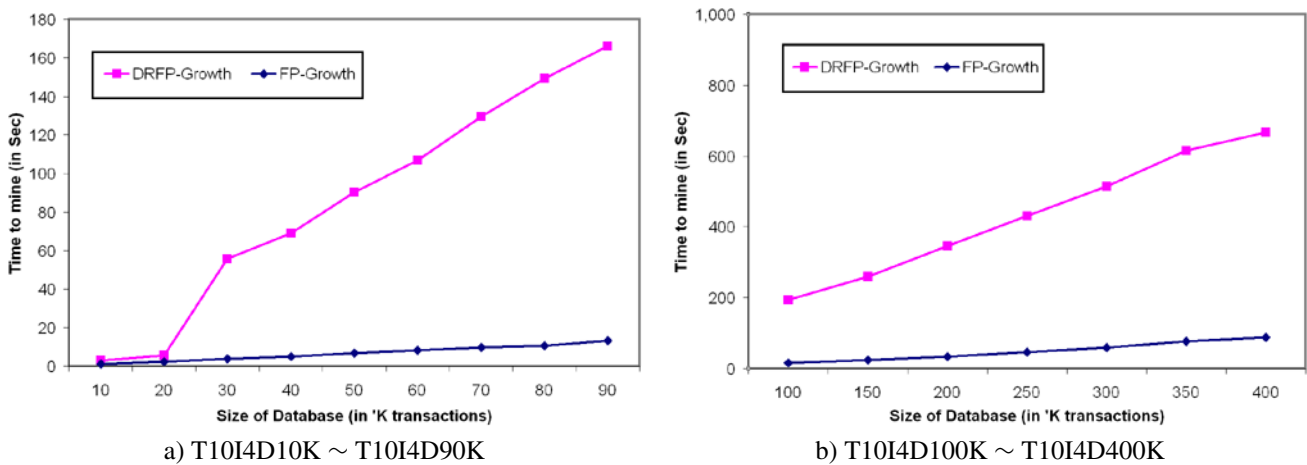


**Fig. 8** Memory requirement of BuildTree on subsets of various sizes from T20110

the physical memory to only 64 MB. Figures 9, 10, 11 report the time comparison of DRFP-growth with the memory based FP-growth. The three figures report results from the experiments conducted using the same database used in the tests of Fig. 4. We show the time comparison for three different support threshold values and we vary the size of the subsets of the database used in the experiment from 10K–400K transactions, except for the last case where we vary the size in the range 10K–100K. The corresponding support threshold values for Figs. 9, 10, and 11 are 0.1%, 1%, and 2%, respectively. We observe from the figures that as the size of the databases increases, the time required for the DRFP-growth increases rapidly because we force it to use secondary storage, which is a time consuming task. The time required to mine for lower support threshold is more than that required for the higher support threshold, which is the expected behavior. Finally, there are couple of important phenomena to notice between the two approaches.



**Fig. 9** Run time of DRFP-growth on subsets of various sizes (Support Threshold: 0.1%)



**Fig. 10** Run time of DRFP-growth on subsets of various sizes (Support Threshold: 1%)

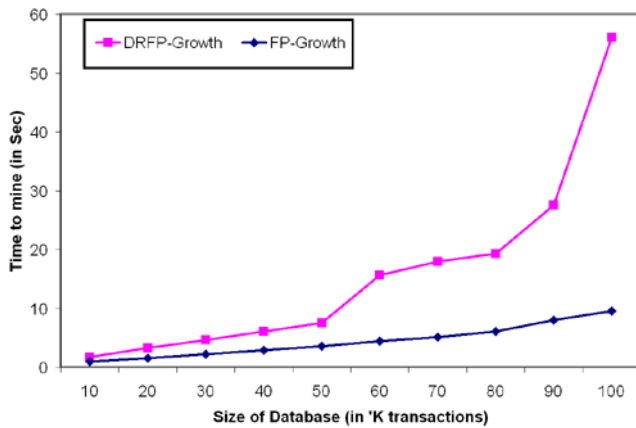
1. There is a gap between the timing of the disk based approach and the memory based approach;
2. There is a sharp change in the timing of the disk based approach at the early stage when the database size is small, and this sharp change moves rightwards when the support threshold increases;
3. There is a point in the memory based FP-growth timing plotted in Fig. 9 after which it ceases to continue;
4. The gap between the timing of the disk based approach and the memory based approach decreases when the support threshold increases.

These phenomena can be explained (in order) as follows:

1. The first point is mainly because the disk based approach saves the tree on disk when it is not possible to keep it in the main memory. And afterwards, the tree is read back from disk into memory during the mining phase. As secondary storage I/O is costly compared to the main memory I/O, we have to sacrifice time for the disk based approach. But, not to forget that the disk based approach

provides a way to mine large databases and for arbitrary parameters, which the current memory based approaches fail to do.

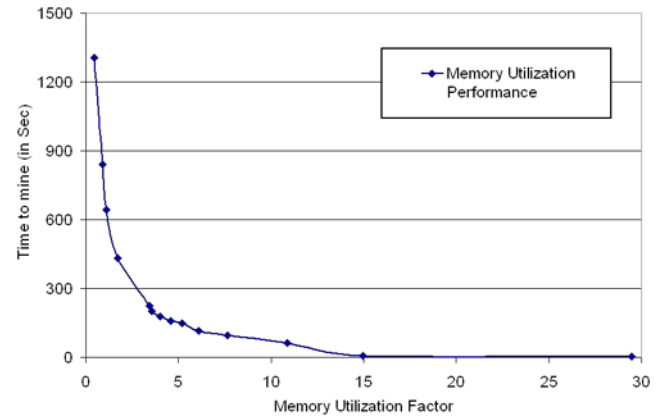
2. The second point happens when the disk based approach starts to use the disk once it predicts that it can not keep all the conditional trees in main memory. For Figs. 9 and 10, this happens when the size of the database is 30K or more; and for Fig. 11, this starts at 60K of the database. Since for low support threshold values the size of the FP-tree is larger than the size of the FP-tree when the support threshold is high, the disk-based approach starts using the disk early when the support is low. However, our prediction on the amount of memory required for future conditional FP-trees is still loose. Hence, the disk based approach starts saving and retrieving the trees, to and from secondary memory, too early and somehow suffers with respect to the computational time, even though it can still mine using the main memory. As FP-growth is recursive, it is not possible to predict the exact amount of future memory requirement.



**Fig. 11** Run time of DRFP-growth on subsets of various sizes from T10I4 (Support Threshold: 2%)

- For the third point, as reported in Fig. 9, the memory based FP-growth fails to continue to mine using only 64 MB of memory when the database size is 400K. But using the disk based approach, we are still able to mine the database using only 64 MB of memory. However, this does not happen to the databases of Figs. 10 and 11, because here the size of the tree is small enough for the mining process to continue using the main memory.
- For the fourth point, the gap between the timing curves of the disk based approach and the memory based approach decreases when the support threshold value increases. This happens because when the support threshold becomes higher, the size of the tree in turn becomes smaller. And if the size of the tree is small, we do not need to save the tree to disk to be able to continue mining. So, when the support is high, we start to use the secondary memory for comparatively larger databases. Moreover, when the tree is small, the time requirement to save the tree on disk is low and the performance of the disk based approach goes closer to the performance of the memory based approach.

Figure 12 shows the performance of the DRFP-Growth approach against memory utilization factor. Memory utilization factor is defined as the inverse of the fraction of memory taken by the FP-tree against the available physical memory size. Hence, Memory utilization factor = available physical memory/memory taken by the actual FP-trees. For this experiment, we fixed the available physical memory to 64 MB and tested the performance (time taken) of the mining algorithm for database, T10I5 for a support threshold of 0.1%. The performance curve shows that, as the memory taken by the actual trees becomes larger than the available physical memory the performance of mining deteriorates. It is also noticeable from the curve that, even though the actual tree size is smaller the available physical memory the performance is not as per expectation. This is again due to the



**Fig. 12** Performance against memory utilization factor for T10I5

loose prediction of the memory requirement for the future conditional FP-trees; this forces the trees to be saved on disk even though it is not required.

## 5 Conclusion and future work

Mining frequent patterns is an essential problem that has received considerable attention. This problem has two main dimensions to consider, namely improving the timing of the mining process and providing for the mining of huge databases by bypassing the main memory bottleneck. The first dimension has been investigated by several researchers within the realm of incremental mining. However, the approach developed in this paper is one of the first proposals to deal with the space issue. This is achieved by expanding the FP-tree and FP-growth into DRFP-tree and DRFP-growth. We empirically demonstrated how DRFP-growth can mine for frequent itemsets from databases of arbitrary sizes using only a bounded amount of physical memory. The proposed approach first tries to mine the database using the traditional FP-growth and FP-tree, and switches into using the secondary memory only if it runs out of physical memory. Experimental results demonstrate that the performance of the DRFP-growth is very much comparable to the memory based counterpart for small databases or when the support threshold is high. On the other hand, when the size of the database is huge and/or the support threshold is low, DRFP-growth can still mine the database; a case for which traditional FP-growth fails. Although the proposed approach is highly effective as the size of the database to be mined grows, we have identified several issues that are still to be investigated. First, we need to develop a more solid analysis of the memory requirement for keeping the conditional FP-trees. Second, we need to adapt an incremental update process to the DRFP-tree. Here, we will benefit from our findings described in [1, 2]. Empowered with these improvements, the disk-based approach proposed in this paper is ex-



pected to turn into attractive approach for mining huge temporal datasets. Finally, we are also working on adapting the same structures used for the disk based approach described in this paper to other tree structures, including the suffix-tree.

## References

1. Adnan M, Alhaji R, Barker K (2006) Constructing complete fp-tree for incremental mining of frequent patterns in dynamic databases. In: Proceedings of the international conference on industrial & engineering applications of artificial intelligence & expert systems, Annecy, France, June 2006. Springer, Berlin
2. Adnan M, Alhaji R, Barker K (2006) Alternative method for incrementally constructing the fp-tree. In: Proceedings of IEEE international conference on intelligent systems, UK, September 2006
3. Agrawal R, Imielinski T, Swami A (1993) Mining association rules between sets of items in large databases. In: Proceedings of ACM SIGMOD. ACM, New York, pp 207–216
4. Agrawal R, Srikant R (1998) Fast algorithms for mining association rules, pp 580–592
5. Amir A, Feldman R, Kashi R (1997) A new and versatile method for association generation. *Inf Sys* 22(6-7):333–347
6. Cheung C-F, Yu JX, Lu H (2005) Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Trans Knowl Data Eng* 17(1):90–105
7. Cheung W, Zaiane OR (2003) Incremental mining of frequent patterns without candidate generation or support constraint. In: Proceedings of IEEE IDEAS, p 111
8. El-Hajj M, Zaiane OR (2004) Cofi approach for mining frequent itemsets revisited. In: Proceedings of ACM SIGMOD workshop on research issues in data mining and knowledge discovery. ACM, New York, pp 70–75
9. Elmasri R, Navathe SB (2000) Fundamentals of database systems. Addison-Wesley, Reading
10. Ganti V, Gehrke J, Ramakrishnan R (2001) Demon: mining and monitoring evolving data. *IEEE Trans Knowl Data Eng* 13(1):50–63
11. Goethals B (2004) Memory issues in frequent itemset mining. In: Proceedings of ACM symposium on applied computing. ACM, New York, pp 530–534
12. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of ACM SIGMOD, Dallas, TX, pp 1–12
13. Leung CK-S, Khan QI, Hoque T (2005) Cantree: a tree structure for efficient incremental mining of frequent patterns. In: Proceedings of IEEE ICDM, pp 274–281
14. Vaarandi R (2004) A breadth-first algorithm for mining frequent patterns from event logs. In: Proceedings of INTELLCOMM, pp 293–308
15. Zaki MJ (2000) Scalable algorithms for association mining. *IEEE Trans Knowl Data Eng* 12(3):372–390