Reducing the Main Memory Consumptions of FPmax* and FPclose

Gösta Grahne and Jianfei Zhu
Concordia University
Montreal, Canada
{grahne, j_zhu}@cs.concordia.ca

Abstract

In [4], we gave FPgrowth, FPmax* and FPclose for mining all, maximal and closed frequent itemsets, respectively. In this short paper, we describe two approaches for improving the main memory consumptions of FPmax* and FPclose. Experimental results show that the two approaches successfully reduce the main memory requirements of the two algorithms, and that in particular one of the approaches does not incur any practically significant extra running time.*

1. Introduction

In FIMI'03 [2], many implementations of algorithms for mining all, maximal and closed frequent itemsets were submitted and tested independently by the organizers. The experimental results in [3] showed that our algorithms FPgrowth*, FPmax* and FPclose [4] have great performance on most datasets, and that FPmax* and FPclose were among the fastest implementations. Our experimental results in [7] also showed that the three algorithms are among the algorithms that consume the least amount of main memory when running them on dense datasets.

However, we also found in [7] that FPmax* and FPclose require much more main memory than other algorithms in [2] especially when the datasets are sparse. This is because the FP-trees constructed from the sparse datasets sometimes are fairly big, and they are stored in main memory for the entire execution of the algorithms FPmax* and FPclose. The sizes of some auxiliary data structures for storing maximal and closed frequent itemsets, the MFI-trees and the CFI-trees also always increase even though many nodes in the trees become useless.

In this short paper, we describe two approaches for reducing the main memory usages of FPmax* and FPclose. We also give the experimental results which show that FPmax* with either of the approaches needs less main memory for running on both synthetic dataset and real dataset.

2. Improving the Main Memory Requirements

We first give a detailed introduction to the main memory requirements of the two algorithm implementations in [4]. Then two approaches for improving the main memory consumption are introduced. Since FPmax* and FPclose have similar main memory requirements, here we only consider main memory improvements in the implementation of FPmax*.

The Basic Case

In [4], when implementing FPgrowth*, FPmax* and FPclose, each node P of an FP-tree, an MFI-tree or a CFI-tree has 4 pointers that point to its parent node, left-child node, right-sibling node and the next node that corresponds to the same itemname as P. The left-child node and right-sibling node pointers are set for the tree construction. The parent node pointer and next node pointer in an FP-tree are used for finding the conditional pattern base of an item. In MFI-trees and CFI-trees, they are used for maximality checking and closedness testing.

In all algorithms, the FP-tree $T_\emptyset$ constructed from the original database $\mathcal{D}$ is always stored in main memory during the execution of the algorithms. For FPmax*, during the recursive calls, many small FP-trees and MFI-trees will be constructed. The biggest MFI-tree is $M_\emptyset$ whose size increases slowly. At the end of the call of FPmax*($T_\emptyset, M_\emptyset$), $M_\emptyset$ stores all maximal frequent itemsets mined from $\mathcal{D}$.

We can see that the main memory requirement of FPmax* in the basic case is at least the size of $T_\emptyset$ plus the size of $M_\emptyset$ which contains all maximal frequent itemsets in $\mathcal{D}$.

Approach 1: Trimming the FP-trees and MFI-trees Continuously

To see if we can reduce the main memory requirement of FPmax*, let's analyze FPmax* first.

Suppose during the execution of FPmax*, an FP-tree T and its corresponding MFI-tree M are constructed. The items in T and M are $i_1, i_2, \ldots, i_n$ in decreasing order of their frequency. Note that the header tables of T and M have the same items and item order. Starting from the least frequent item i_n, FPmax* mines maximal frequent itemsets from T. A candidate frequent itemset X is compared with the maximal frequent itemsets in M. If X is maximal, X is inserted into M. When processing the item i_k, FPmax* needs the frequency information that contains only items $i_1, i_2, \ldots, i_{k-1}$, and the frequency information of $i_{k+1}, \ldots, i_n$ will not be used any more. In other words, in T, only the nodes that correspond to $i_1, i_2, \ldots, i_k$ are useful, and the nodes corresponding to $i_{k+1}, \ldots, i_n$ can be deleted from T. If a candidate maximal frequent itemset X is found, X must be a subset of $i_1, i_2, \ldots, i_k$. Thus in M, only the nodes corresponding to $i_1, i_2, \ldots, i_k$ are used for maximality checking, and the nodes corresponding to $i_{k+1}, \ldots, i_n$ will never be used, and therefore can be deleted.

Based on the above analysis, we can reduce the main memory requirement of FPmax* by continuously trimming the FP-trees and MFI-trees. After processing an item i_k, all i_k-nodes in T and M are deleted. This can be done by following the head of the link list from i_k in the header tables $T.header$ and $M.header$. Remember that the children of a node are organized by a right-sibling linked list. To speed up the deletions we make this list doubly linked, i.e. each node has pointers both to its right and left siblings.

Before calling FPmax*, $T_\emptyset$ has to be stored in the main memory. By deleting all nodes that will not be used any more, the sizes of FP-trees, especially the size of $T_\emptyset$, become smaller and smaller. The sizes of the MFI-trees still increase because new nodes for new maximal frequent itemsets are inserted, however, since obsolete nodes are also deleted, the MFI-trees will grow more slowly. At the end of the call of FPmax*, the sizes of $T_\emptyset$ and $M_\emptyset$ are all zero. We assume that the sizes of the recursively constructed FP-trees and MFI-trees are always far smaller than the size of the top-level trees $T_\emptyset$ and $M_\emptyset$, and that the main memory consumption of these trees can be neglected. Besides $T_\emptyset$, the main memory also stores $M_\emptyset$. At the initial call of FPmax*, the size of $M_\emptyset$ is zero. Then $M_\emptyset$ never reaches its full size because of the trimming. We estimate that the average main memory requirement of FPmax* with approach 1 is the size of $T_\emptyset$ plus half of the size of $M_\emptyset$.

In [4], we mentioned that we can allocate a chunk of main memory for an FP-tree, and delete all nodes in the FP-tree at a time by deleting the chunk. Time is saved by avoiding deleting the nodes in the FP-tree one by one. Obviously, this technique can not be used parallel with approach 1. Therefore, FPmax* with approach 1 will be slower than the basic FPmax*, but its peak main memory requirement will be smaller than that of the basic FPmax*.

Approach 2: Trimming the FP-trees and MFI-trees Once

In approach 2, we use the main memory management technique by trimming the FP-trees and MFI-trees only once. We still assume that main memory consumption of the recursively constructed FP-trees and MFI-trees can be neglected, and only the FP-tree $T_\emptyset$ and the MFI-tree $M_\emptyset$ are trimmed.

Suppose the items in $T_\emptyset$ and $M_\emptyset$ are $i_1, i_2, \ldots, i_n$. In our implementation, we allocate a chunk of main memory for those nodes in $T_\emptyset$ and $M_\emptyset$ that correspond to $i_{\lfloor n/2 \rfloor}, \ldots, i_n$. The size of the chunk is changeable. During the execution of FPmax*, $T_\emptyset$ and $M_\emptyset$ are not trimmed until item $i_{\lfloor n/2 \rfloor}$ in $T_\emptyset.header$ is processed. The main memory of the chunk is freed and all notes in the chunk are deleted at that time.

In this approach, before processing $i_{\lfloor n/2 \rfloor}$ and freeing the chunk, $T_\emptyset$ and a partial $M_\emptyset$ are stored in the main memory. On the average, the size of $M_\emptyset$ is half of the size of the full $M_\emptyset$. After freeing the chunk, new nodes for new maximal frequent itemsets are inserted and they are never trimmed. However, considering the fact that MFI-tree structure is a compact data structure, the new nodes are for the $\lfloor n/2 \rfloor$ most frequent items, and $M_\emptyset$ already has many branches for those nodes before trimming, we can expect that the size of $M_\emptyset$ will be a little bit more than half of the size of the complete $M_\emptyset$. Therefore the peak main memory consumption is a little bit more than the size of $T_\emptyset$ plus half of the size of $M_\emptyset$. Compared with approach 1, the FPmax* with approach 2 is faster but consumes somewhat more main memory.

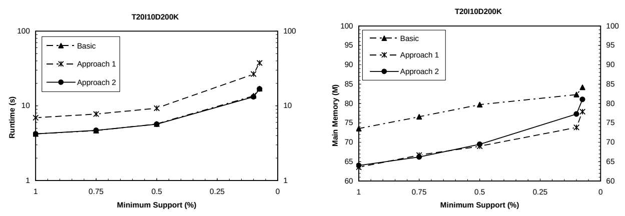
3. Experimental Evaluation

We now present a comparison of the runtime and main memory consumptions of the basic case and the two approaches. We ran the three implementations of FPmax* on many synthetic and real datasets. The synthetic datasets are sparse datasets, and the real datasets are all dense. Due to the lack of space, only the results for one synthetic dataset and one real dataset are shown here.

The synthetic dataset *T20I10D200K* was generated from the application on the website [1]. It contains 200,000 transactions and 1000 items. The real dataset *pumsb** was downloaded from the FIMI'03 website [2]. It was produced from census data of Public Use Microdata Sample (PUMS).

All experiments were performed on a 1GHz Pentium III with 512 MB of memory running RedHat Linux 7.3.

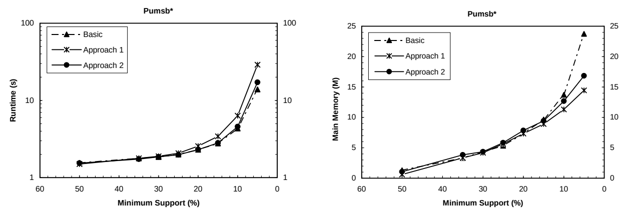
Figure 1 shows the runtime and the main memory usage of running FPmax* with the implementations of the basic case and the two approaches on the dataset *T20I10D200K*. As expected, in the runtime graph, FPmax* with approach 1 took the longest time. Its runtime is almost twice the runtime of the basic case and approach 2. However, approach 1



Runtime Main Memory Consumption

Figure 1. *T20I10*

consumes the least amount of main memory. The peak main memory of approach 1 is always less than the basic case for about 10 megabytes, or about 15%. The speed of approach 2 is similar to that of the basic case, since approach 2 only trims the FP-tree $T_\emptyset$ and the MFI-tree $M_\emptyset$ once. The main memory consumption of approach 2 is similar to that of approach 1, which means the approach 2 successfully saves main memory.



Runtime Main Memory Consumption

Figure 2. *pumsb**

The runtime and main memory usage of running FPmax* on real dataset *pumsb** are shown in Figure 2. The results are similar to those results on synthetic dataset. Dataset *pumsb** is a very dense dataset, its FP-trees and MFI-trees have very good compactness, and there are not many nodes in the trees. Therefore, in the two graphs in Figure 2, the differences of the runtime and main memory consumptions for the basic case and the two approaches are not very big.

4. Conclusions

We have analyzed the main memory requirements of the FPmax* and FPclose implementation in [4]. Two approaches for reducing the main memory requirements of FPmax* and FPclose are introduced. Experimental results show that both approach 1 and approach 2 successfully decrease the main memory requirement of FPmax*. While the continuous trimming of the trees in approach 1 slows down the algorithm, the "one-time-trimming" used in approach 2 shows speed similar to the original method.

We also noticed that the PatriciaMine in [6] using Patricia trie structure to implement the FP-growth method [5] shows great speed and less main memory requirement. We are currently considering implementing FPmax* and FPclose using a Patrica trie.

References

[1] `http://www.almaden.ibm.com/software/quest/Resources/index.shtml`.

[2] `http://fimi.cs.helsinki.fi`.

[3] B. Goethals and M. J. Zaki (Eds.). *Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '03)*. CEUR Workshop Proceedings, Vol 80 `http://CEUR-WS.org/Vol-90`.

[4] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceeding of Special Interest Group on Management of Data* , pages 1-12, Dallas, TX, May 2000.

[6] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using Patricia tries. In *Proceedings of the 1st Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov. 2003.

[7] J. Zhu. Efficiently mining frequent itemsets from very large databases. Ph.D. thesis, Sept. 2004.