A Theoretical and Practical Comparison of Depth First and FP-growth Implementations of Apriori

Jeannette M. de Graaf [a] Walter A. Kosters [a] Wim Pijls [b]
Viara Popova [b]

[a] Leiden Institute of Advanced Computer Science
Universiteit Leiden
P.O. Box 9512, 2300 RA Leiden, The Netherlands
{graaf, kosters}@liacs.nl

[b] Department of Computer Science
Erasmus University
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands
{pijls, popova}@few.eur.nl

Abstract

We examine the complexity of Depth First and FP-growth implementations of Apriori, two of the fastest known data mining algorithms to find frequent itemsets in large databases. We describe the algorithms in a similar style, derive theoretical formulas, and provide experiments on both synthetic and real life data to illustrate the theory.

1 Introduction

We examine the theoretical and practical complexity of Depth First ($\mathcal{DF}$, see [5]) and FP-growth ($\mathcal{FP}$, see [3]) implementations of Apriori (see [1]), two of the fastest known data mining algorithms to find frequent itemsets in large databases. There exist many implementations of Apriori (see, e.g., [4, 6]). We would like to focus on algorithms that assume that the whole database fits in main memory, this often being the state of affairs; among these, $\mathcal{DF}$ and $\mathcal{FP}$ are the fastest. In most papers so far little attention has been given to theoretical complexity.

Apriori is an algorithm that finds all frequent itemsets in a given database of transactions; frequent sets are the necessary building blocks for association rules, i.e., if-then-rules of the form "if a customer buys products X and Y, he or she also buys product Z". So the situation is the following: we are given products and customers, where every customer buys a set of products, the so-called transaction. More general, an itemset is an arbitrary set of products; a k-itemset has k elements. The *support* of an itemset is the number of customers that buy all products from this itemset — and maybe more. An itemset is frequent if and

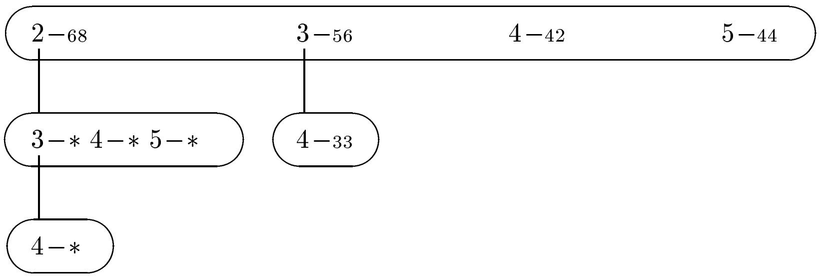

Figure 1: Example trie

only if its support is larger than or equal to *min_sup*, a certain threshold given in advance. The APRIORI algorithm successively finds all frequent 1-itemsets, all frequent 2-itemsets, all frequent 3-itemsets, and so on. The frequent k-itemsets are used to generate candidate $(k+1)$-itemsets, where the candidates are only known to have two frequent subsets with k elements. (In fact, the candidates are uniquely constructed out of these two subsets.) A pruning step discards those candidates for which not all subsets are frequent. This relies on the property that all subsets of frequent itemsets are frequent too. Finally the supports of the remaining candidates are computed (by means of a run through the database) in order to determine the frequent ones.

The crux of any implementation is the chosen representation for the candidates and the way to check their support. $\mathcal{DF}$ builds a *trie* of all candidate sets, and then pushes all customers through this trie — one at a time. Eventually a trie containing all frequent itemsets remains. In the example trie from Figure 1 we have 5 items 1, 2, 3, 4 and 5, and $min\_sup = 25$. The small numbers indicate the supports of the itemsets, for instance the support of the itemset $\{3,4\}$ is 33. The trie has just been augmented with 2 and its candidate subtrie. A $*$ indicates that the corresponding support is not known yet. From the picture we infer that the only frequent itemsets so far are $\{3\}$, $\{4\}$, $\{5\}$ and $\{3,4\}$, and $\{2\}$ itself; note that every path in the trie from the root node downwards identifies a unique itemset. For instance $\{3,4,5\}$ and $\{4,5\}$ are not frequent here, since the corresponding paths do not occur in the trie. In the current step all customers are pushed through the part of the trie rooted at 2, in the meantime updating all counters.

$\mathcal{FP}$ proceeds differently. It makes use of a so-called FP-tree (see Section 4; FP denotes Frequent Pattern) to maintain databases of parts of transactions that matter in the future. Instead of a trie it internally has a recursion tree in which the candidates are gathered. Informally speaking, in the example trie from Figure 1 (this being an implementation of the recursion tree), the node $2-_{68}$ would also contain all 68 transactions that have 2, restricted to the products 3, 4 and 5. An FP-tree is built that efficiently contains all these transactions, and subdatabases are derived for the recursive steps. More details can be either found in the original papers [5, 3] or in the following sections.

We will give a theoretical basis for the analysis of the two algorithms. We have chosen for a somewhat informal presentation of the algorithms, intertwined with our analysis. We will present practical results, and we mention several difficulties. Indeed, $\mathcal{FP}$ is a complicated algorithm, involving recursion and serious garbage collection; the algorithm also allows several refinements that are hard to analyze. In this paper we first describe the algorithms in parallel with a theoretical analysis, we then present practical results, and finally provide conclusions and issues for further research.

2 Definitions

Let m denote the number of transactions (also called customers), and let n denote the number of products (also called items). Usually m is much larger than n. For a non-empty itemset $A \subseteq \{1, 2, \ldots, n\}$ we define:

- $\mathit{supp}(A)$ is the *support* of A: the number of customers that buy all products from A (and possibly more);
- $\mathit{sm}(A)$ is the smallest number in A;
- $\mathit{la}(A)$ is the largest number in A;
- $|A|$ is the number of elements of A.

In line with this definition we let $\mathit{supp}(\emptyset) = m$. A set $A \subseteq \{1, 2, \ldots, n\}$ is called *frequent* if $\mathit{supp}(A) \geq \mathit{min\_sup}$, where the so-called *support threshold* $\mathit{min\_sup}$ is a fixed number given in advance.

We assume every 1-itemset to be frequent; this can be effected by the first step of the algorithms we are looking at, which might be considered as preprocessing.

3 The Depth First Algorithm — Complexity

In the Depth First algorithm a trie of frequent itemsets is built. Suppose that all frequent itemsets in $\{k+1, \ldots, n\}$ have been gathered in a trie $\mathcal{T}$. The root node is now augmented on the left side with k. Then $\mathcal{T}$ is placed in the trie under k, providing new candidate itemsets. In the example trie from Figure 1 we have $k = 2$. Then the necessary supports (for this new subtrie) are calculated by "pushing" all customers through the trie: for every customer the paths starting at k are examined in a depth first fashion and checked against the customer, meanwhile updating the node counters denoting the supports so far. Infrequent sets are removed once all customers have been queried. The complexity of this algorithm largely depends on these queries, and we shall therefore pay special attention to them. Omitting the printing of the results, the algorithm can be described as follows:

```
procedure DF (database D)
    for k := n − 1 downto 1 do
        augment the trie in the root with k
        attach a copy of the old trie underneath this k
        for all transactions in D
            update counters in the subtrie starting from this k
```

A "database query" is defined as a question of the form "Does customer C buy product P?" (or "Does transaction T has item I?"), posed to the original database. Note that we have mn database queries in the "preprocessing" phase in which the supports of the 1-itemsets are computed and ordered: every field of the database is inspected once. (By the way, the sorting takes $O(n \log n)$ time.) The number of database queries for $\mathcal{DF}$ equals:

$$m(n-1) + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{j=1}^{sm(A)-1} supp(\{j\} \cup A \setminus \{la(A)\}) \quad . \tag{1}$$

The proof is as follows. We wonder how often a particular number in a trie node is "questioned". Suppose it is not at the root level. Then (a) the path from the root down to the node, except for the root, must be frequent (otherwise the path would not have existed in the trie as a "candidate" in the first place; it comes from a frequent itemset in the part of the trie that has already been fully examined), and (b) the number of questions is the support of the path from the root down to the node, this time except for the node itself! This leads to the summation in Formula (1). Stated otherwise, the frequent itemset A is placed under all nodes j to the left of $sm(A)$, giving rise to $supp(\{j\} \cup A \setminus \{la(A)\})$ database queries concerning the question whether a customer bought $la(A)$ or not, having already bought the rest of $\{j\} \cup A$. Finally the root level node gives rise to $m(n-1)$ database queries: the last node is not questioned anymore. This completes the proof.

It makes sense to look at the number of nodes of the trie, which is connected to the effort of maintaining and using the datastructures. Counting each trie-node with the number of items it contains, the total is computed to be:

$$n + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{j=1}^{sm(A)-1} 1 \quad . \tag{2}$$

If we let $sm(\emptyset) = n+1$ we could put the term n into the big summation by omitting the condition $A \neq \emptyset$.

Notice that the complexity heavily depends on the sorting order of the items at the top level. It turns out that an increasing order of items is beneficial here.

4 The FP-growth Algorithm — Complexity

The FP-growth algorithm is a clever implementation of Apriori. The algorithm proceeds recursively. Suppose that a frequent itemset B has been found. The algorithm has saved all transactions that support B, restricted to items i with $i > la(B)$ and such that $B \cup \{i\}$ is frequent, in a "local database" D (in the form of an FP-tree; we shall return to this later). Now for these i's a recursive call (with $A = B \cup \{i\}$ as frequent itemset; note that $i = la(A)$) is made — unless for basic cases. Omitting the printing of the results, the algorithm can be described as follows:

```
procedure FP (itemset B, database D)
    for all i > la(B) with B ∪ {i} frequent do
    (i.e., for all items i occurring in D)
        set D' = the subset of transactions in D that support i
        count support for all items k from D' with k > i (%)
        remove infrequent items and items k with k ≤ i from D'
        build new FP-tree for D'
        if there are items left (#) call FP(B ∪ {i}, D')
```

Although the title of [3] suggests otherwise, the algorithm also generates candidate itemsets; in fact, this is the recursion tree that operates underneath the surface. In this case a node gets as its candidate children all its right siblings — but only if they are frequent. Because the child node is only a real candidate if the parent node (or rather: the entire path from the root down to the node) is frequent, we see that candidates (with t items) are now known to have at least two frequent subsets with $t-1$ items. The local databases are handed down the recursion tree, where customers not buying a node are omitted.

Let us define a "database query" in this case as a question of the form "Does customer C buy product P?" posed to a local database D, necessary for the counting in step (%). The number of database queries (except for the "preprocessing" phase) for $\mathcal{FP}$ equals:

$$\sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \; \sum_{\substack{j=la(A)+1 \\ \{j\} \cup A \setminus \{la(A)\} \text{ frequent}}}^{n} supp(A) \quad . \tag{3}$$

The proof relies on the following. If we are in a node of the recursion tree, we in fact have a frequent itemset A (being the path from the root down to the node, including both ends). The current tree-node, being the "last" node of A, is $la(A)$. All its right siblings that correspond to frequent itemsets are placed under this node as candidate itemsets. This easily implies the formula.

A consequence of Formula (1) and Formula (3) is that the contribution of the 1-itemsets is exactly the same for both algorithms. In fact, this number equals

$$\sum_{i=1}^{n} (n-i) \; supp(\{i\}) \quad , \tag{4}$$

which is minimal exactly if the 1-itemsets are sorted increasingly. The contribution for longer itemsets is more complex — and different for the two algorithms.

An FP-tree is an efficient datastructure that stores the whole local database of customers that buy (at least) some itemset A. For the current discussion it is not necessary to provide full detail on this subject, but it is good to know that every path in the FP-tree keeps track of an itemset along with its support. However, contrary to the situation in the tries above, itemsets are not uniquely represented, but may occur as subset of several itemsets. In Figure 2 the node $2-_8$ tells us that there are 8 customers (out of 17 buying item 1) that buy the itemset $\{1,2\}$; however, there are $5+1+3=9$ customers buying the scattered itemset $\{1,4\}$.

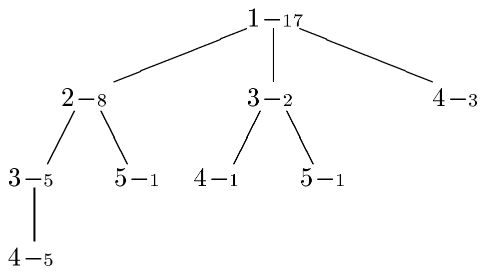

Figure 2: Example FP-tree

Again we try to compute the number of nodes involved, this number being related to the effort necessary to build the FP-trees. In this case the situation is much more complex, since the recursive algorithm builds and destroys nodes on the fly, and in the FP-tree transactions sharing the same prefix are joined into one path, thereby diminishing the number of nodes. Note that the nodes we count are no trie (or recursion tree) nodes, but the nodes necessary to maintain the FP-trees. The number of nodes is *at most* equal to the summation in Formula (3), since we have to turn the local database of customers who buy A into an FP-tree. This database has $supp(A)$ rows, each column dealing with one product.

Note that $\mathcal{FP}$ is a real implementation of Apriori (except for the pruning step), since every candidate itemset is known to have two frequent subsets. In case of $\mathcal{DF}$ only one subset is certainly frequent!

5 Practical Complexity

Using the well-known IBM-Almaden synthetic database (see [2]) and a real retail database (with 4,698 items and 62,797 transactions) we shall examine the "real" complexity of the two algorithms. This should be compared to the theoretical complexity from the previous sections. The parameters for generating a synthetic database are the number of transactions D (in thousands), the average transaction size T and the average length I of so-called maximal potentially large itemsets. The number of maximal frequent itemsets was set to $L = 2{,}000$ and the number of items was set to $N = 1{,}000$, following the design in [2]. The experiments were conducted at a Pentium-III machine with 256 MB memory at 733 MHz, running Windows NT. The programs were developed under the Borland C++ 5.02 environment, but are also usable with the Gnu C++ compiler.

Tabulated in Figure 3 are the number of frequent 1-itemsets (this number plays the role of n, since in previous sections we assumed all 1-itemsets to be frequent), the total number of frequent itemsets, the execution time in seconds for both algorithms (for $\mathcal{FP}$ averaged over three runs), and the number of database queries according to Formula (1) and Formula (3), respectively.

We see that for the sparse retail database $\mathcal{FP}$ performs better, probably due to $\mathcal{DF}$'s generation of too many non-frequent candidates. In the case of more dense

D100T20I6				
min_sup	0.5%	1.0%	1.5%	2.0%
# fr. 1-itemsets	730	561	435	357
# fr. itemsets	25,040	1,651	928	717
$\mathcal{DF}$ exec. time	31	13	9	7
$\mathcal{FP}$ exec. time	35	20	17	16
# queries $\mathcal{DF}$	804,898,976	427,097,723	301,111,212	234,051,516
# queries $\mathcal{FP}$	442,562,012	329,761,608	252,044,086	198,043,844
D100T20I4				
min_sup	0.5%	1.0%	1.5%	2.0%
# fr. 1-itemsets	688	561	435	357
# fr. itemsets	9,438	2,605	987	751
$\mathcal{DF}$ exec. time	25	14	9	7
$\mathcal{FP}$ exec. time	37	26	17	15
# queries $\mathcal{DF}$	731,815,487	446,136,987	315,448,527	251,949,116
# queries $\mathcal{FP}$	417,996,012	338,850,828	262,917,783	213,788,927
D100T20I2				
min_sup	0.5%	1.0%	1.5%	2.0%
# fr. 1-itemsets	635	527	445	357
# fr. itemsets	5453	1942	1153	798
$\mathcal{DF}$ exec. time	23	13	10	7
$\mathcal{FP}$ exec. time	33	23	18	16
# queries $\mathcal{DF}$	757,631,634	432,277,091	331,223,803	249,973,785
# queries $\mathcal{FP}$	396,861,892	321,545,022	271,150,238	210,231,042
Retail data				
min_sup	0.25%	0.50%	0.75%	1.00%
# fr. 1-itemsets	2,206	981	487	320
# fr. itemsets	3,284	1,170	528	330
$\mathcal{DF}$ exec. time	79	14	5	2
$\mathcal{FP}$ exec. time	10	7	5	4
# queries $\mathcal{DF}$	778,171,180	281,647,840	114,469,871	66,731,996
# queries $\mathcal{FP}$	605,209.096	213,361.480	82,454,945	46,351,860

Figure 3: Results from experiments

databases $\mathcal{DF}$ outperforms $\mathcal{FP}$ despite the fact that it does more queries; this might be explained by the complex data handling that is needed for the FP-trees.

6 Conclusions and Further Research

We gave a theoretical analysis of Depth First and FP-growth implementations $\mathcal{DF}$ and $\mathcal{FP}$ of Apriori. Formulas (1) and (3) capture the number of database queries for the two algorithms. $\mathcal{FP}$, as used in the experiments in Section 5, has three refinements that improve its efficiency:

1. Identical transactions are identified in the FP-tree. From then on, the algorithm treats them as single ones, keeping track of the correct supports by means of an extra counter. This did not occur too often in our experiments.
2. If the database "collapses", i.e., in situation (#) all transactions turn out to be prefixes of one of them, the recursion terminates. This happens if the FP-tree is a list. Instead of giving all subsets of the current itemset I, the algorithm states that the $2^{|I|}-1$ non-empty subsets of I are frequent. During experiments this hardly ever occurred: at most 100 times during one run.
3. At every step, and not only in the first one, the children are again sorted with respect to their supports.

We would like to examine the theoretical and practical behavior with respect to these refinements. Also, it can be interesting to use hash-tables instead of FP-trees.

The runtimes of both algorithms are in the same order of magnitude. We suggest that in more dense databases (like the synthetic ones) one could choose $\mathcal{DF}$, while in more sparse databases (like the retail database) $\mathcal{FP}$ might be better.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J.B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 1994.

[3] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, 2000.

[4] J. Hipp, U. Günther, and G. Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analyzing today's aproaches. In D.A. Zighed, J. Komorowski, and J. Żytkov, editors, *Principles of Data Mining and Knowledge Discovery, Proceedings of the 4th European Conference (PKDD 2000)*, Springer Lecture Notes in Computer Science 1910, pages 159–168. Springer Verlag, 2000.

[5] W. Pijls and J.C. Bioch. Mining frequent itemsets in memory-resident databases. In E. Postma and M. Gyssens, editors, *Proceedings of the Eleventh Belgium-Netherlands Conference on Artificial Intelligence (BNAIC1999)*, pages 75–82, 1999.

[6] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In F. Provost and R. Srikant, editors, *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2001)*, pages 401–406, 2001.