Complexity Analysis of Depth First and FP-Growth Implementations of APRIORI

Walter A. Kosters[1], Wim Pijls[2], and Viara Popova[2]

[1] Leiden Institute of Advanced Computer Science
Universiteit Leiden
P.O. Box 9512, 2300 RA Leiden, The Netherlands
kosters@liacs.nl
[2] 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 [6]) and FP-growth ($\mathcal{FP}$, see [4]) 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., [5,7]). 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. This paper is a continuation of [3].

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 only if its support is larger than or equal to *minsup*, 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

P. Perner and A. Rosenfeld (Eds.): MLDM 2003, LNAI 2734, pp. 284–292, 2003.

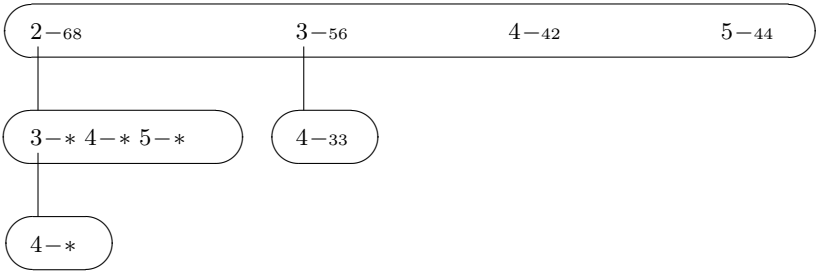

Fig. 1. Example trie

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 main application of frequent itemsets is in the area of association rules. These are often used in the analysis of retail data and in medical environments. Moreover, frequent itemsets are relevant for other topics in data analysis, such as clustering and pattern recognition. The techniques can be applied in more general contexts: instead of binary data one can use data with continuous or categorical values; it is also possible to introduce hierarchies.

The crux of any implementation of Apriori 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 Fig. 1 we have 5 items 1, 2, 3, 4 and 5, and $minsup = 25$. 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 downward 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. Since $\{3, 5\}$ is not frequent, $\{2, 3, 5\}$ is not a candidate. 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 Sect. 4; FP means 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 Fig. 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