1-1-2006

Implementation and analysis of apriori algorithm for data mining

Pavankumar Bondugula

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/1927

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
IMPLEMENTATION AND ANALYSIS OF APRIORI ALGORITHM FOR DATA MINING

by

Pavankumar Bondugula

Bachelor of Science
Jawaharlal Nehru Technological University, India
2004

A thesis submitted in partial fulfillment of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2006
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform 1436740
Copyright 2006 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Thesis prepared by

PAVANKUMAR PONDUGULA

Entitled

IMPLEMENTATION AND ANALYSIS OF APRIORI ALGORITHM FOR DATA MINING.

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
ABSTRACT

Implementation and Analysis of Apriori Algorithm for Data Mining

by

Pavankumar Bondugula

Dr. Kazem Taghva, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Data mining represents the process of extracting interesting and previously unknown knowledge from data. In this thesis we address the important data mining problem of discovering association rules. An association rule expresses the dependence of a set of attribute-value pairs, also called items, upon another set of items.

We also report on various implementation techniques for the well-known Apriori Algorithm and their time complexity.
# TABLE OF CONTENTS

ABSTRACT ........................................................................................................ iii

LIST OF TABLES ...................................................................................................... vi

LIST OF FIGURES ...................................................................................................... vii

ACKNOWLEDGEMENTS .............................................................................................. viii

CHAPTER 1  INTRODUCTION ................................................................................. 1

  1.1 Thesis organization ...................................................................................... 3

CHAPTER 2  BACKGROUND ................................................................................... 4

CHAPTER 3  PROBLEM DESCRIPTION .................................................................. 7

  3.1 Definitions ...................................................................................................... 7

  3.2 Problem ......................................................................................................... 10

  3.2.1 Mining Frequent Itemsets .......................................................................... 12

  3.2.2 Mining Association Rules .......................................................................... 20

CHAPTER 4  THE APRIORI ALGORITHM .......................................................... 21

  4.1 Apriori Principle ........................................................................................... 21

  4.2 Frequent Itemset Mining in Apriori Algorithm ................................................. 24

  4.3 Association Rule Mining ................................................................................ 26

  4.4 Data Structures ............................................................................................. 29

    4.4.1 Hash Tree ................................................................................................... 29

    4.4.2 Trie ............................................................................................................ 34

CHAPTER 5  OPTIMIZATION .................................................................................. 37

  5.1 FP-growth .................................................................................................... 38

CHAPTER 6  EXPERIMENTAL EVALUATION .................................................... 44

  6.1 Experimental Datasets ................................................................................. 44

  6.2 Experiments .................................................................................................. 45

CHAPTER 7  CONCLUSION ..................................................................................... 56
LIST OF TABLES

Table 1  An Example Transaction Database ........................................................... 9
Table 2  An example of preprocessed transaction database .............................. 39
Table 3  Data Set Characteristics ............................................................................ 45
Table 4  Running times for Chess data set with confidence 1 ......................... 46
Table 5  Running times for Mushroom data set with confidence 1 ................. 47
Table 6  Running times for Mushroom data set with confidence 0.5 ............ 48
Table 7  Running times for T10I4D100K data set with confidence 0.5 .......... 49
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Lattice of itemsets</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Maximal frequent itemset</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Breadth-first and depth-first traversals</td>
<td>16</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Generating candidate itemsets using the depth-first approach</td>
<td>17</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Top-down and Bottom-up traversal</td>
<td>19</td>
</tr>
<tr>
<td>Figure 6</td>
<td>An illustration of the Apriori principle</td>
<td>22</td>
</tr>
<tr>
<td>Figure 7</td>
<td>An illustration of support-based pruning</td>
<td>23</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Frequent itemsets mining in Apriori Algorithm</td>
<td>25</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Apriori Association rule mining</td>
<td>28</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Counting the support of itemsets using hash structure</td>
<td>31</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Hashing a transaction at the root node of the hash tree</td>
<td>33</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Subset operation on the left most sub tree of the root of the candidate hash tree</td>
<td>34</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Trie</td>
<td>36</td>
</tr>
<tr>
<td>Figure 14</td>
<td>An example of an FP-tree</td>
<td>40</td>
</tr>
<tr>
<td>Figure 12</td>
<td>FP-growth Algorithm</td>
<td>41</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Running time of the algorithms for the Chess Dataset with confidence 1.</td>
<td>50</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Running time of the algorithms for the Mushroom Dataset with confidence 1.</td>
<td>51</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Running time of the algorithms for the Mushroom Dataset with confidence 0.5</td>
<td>53</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Running time of the algorithms for the T10I4D100K Dataset with confidence 0.5</td>
<td>54</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

The work presented in this thesis has been done in the Department of Computer Science of the University of Nevada, Las Vegas. My thanks go to my advisor Dr. Kazem Taghva, for teaching me to express problems formally and for his helpful suggestions and discussions; to Dr. Coombs and Julie for their feedback and suggestions regarding my thesis; and to my friend Darshana Gala, for her support and for being my collaborator in the work on which Chapter 4 is based. I am also very grateful to Dr. Ajoy Datta, Dr. Tom Nartker and Dr. Shahram Latifi for their participation in my committee.

I would like to thank my brother and parents a lot for trusting me, helping me to continue my education in top schools, and for their invaluable support that has leaded me to be here now.
CHAPTER 1

INTRODUCTION

In recent years the sizes of databases has increased rapidly. This has led to a growing interest in the development of tools capable in the automatic extraction of knowledge from data. The term Data Mining, or Knowledge Discovery in Databases, has been adopted for a field of research dealing with the automatic discovery of implicit information or knowledge within databases [4].

The implicit information within databases, and mainly the interesting association relationships among sets of objects, that lead to association rules, may disclose useful patterns for decision support, financial forecast, marketing policies, even medical diagnosis and many other applications. This fact attracted a lot of attention in recent data mining research [4]. As shown in [1], mining association rules may require iterative scanning of large databases, which is costly in processing. Many researchers have focused their work on efficient mining of association rules in databases ([1], [4], [8], [11]).

This chapter introduces a data mining-based approach known as association analysis, which is used for extracting interesting relationships hidden in large transaction data sets. The extracted relationships are represented in the form of association rules that can be used to predict the presence of certain
items in a transaction based on the presence of other items. For example, the following rule suggests that many customers who buy cereal also tend to buy milk.

\[ \text{Cereal} \rightarrow \text{Milk} \]

The association rules may be useful in many applications such as supermarket transactions analysis, store layout and promotions on the items, university course enrollment analysis, customer behavior analysis in retailing, catalog design, word occurrence in text documents, user's visits to world wide web pages, stock transactions, tumor detection in digital mammography[7], building statistical thesauri from text databases [10, 6] and discovering associated images from huge sized image databases [12, 6].

A very influential association rule mining algorithm, Apriori [1], has been developed for rule mining in large transaction databases. Many other algorithms developed are derivative and/or extensions of this algorithm. A major step forward in improving the performances of these algorithms was made by the introduction of a novel, compact data structure, referred to as frequent pattern tree, or FP-tree [2], and the associated mining algorithm, FP-growth.

The key challenges of association analysis are two-fold: (1) to design an efficient algorithm for mining association rules from databases, and (2) to develop an effective strategy for distinguishing interesting rules from spurious or obvious ones.
1.1 Thesis Organization

This thesis includes seven chapters except the first introduction chapter. Each chapter gives a summary on each topic mentioned and then makes a detailed explanation of the topic. The thesis is organized as follows:

Chapter Two explains the previous work done in the field of data mining for the generation of association rules. AIS and SETM algorithms are explained briefly in this chapter. Chapter Three explains the basic fundamental definitions and problem description of algorithms we use in this thesis.

Chapter Four describes the most common algorithm Apriori and also the data structures used. Chapter Five describes the optimization and FP-growth algorithm in brief. Chapter Six explains the implementation of algorithms and experimentation results are presented. The algorithms are compared in this chapter. Chapter Seven concludes the thesis.
CHAPTER 2

BACKGROUND

The problem of discovering association rules was first explored in AIS [13] on supermarket basket data, which is the set of transactions that includes the items purchased by the customers. An algorithm called Set Oriented Mining (SETM [16]) was proposed to solve this problem using relational operations. In this pioneering work the data was considered to be in binary, i.e. an item exists in the transaction or not, and the quantity of the item in the transaction is irrelevant.

In AIS [13], mining of association rules was decomposed into two subproblems,

• Discovering all frequent patterns (represented by frequent itemsets defined below) and

• Generating the association rules from those frequent itemsets.

The second subproblem is straightforward, and can be done efficiently in a reasonable time. However, the first subproblem is very tedious and computationally expensive for very large databases and this is the case for many real life applications. In large retailing data, the number of transactions is generally in the order of millions, and number of items (attributes) is generally in the order of thousands. When the data contains $N$ items, then the number of
possibly frequent itemsets is $2^n$. However the frequent itemsets existing in the
database are much smaller than $2^n$. Thus, brute force search techniques, which
require exponential time, waste too much effort to obtain the set of frequent
itemsets. To reduce the number of possibly frequent itemsets, many efficient
algorithms have been proposed. Those algorithms generally use clever data
structures (such as hash tables, hash trees, lattices, multi-hyper graphs, etc.) in
order to reduce the size of possibly frequent itemsets and speedup the search
process.

Most of the association rule algorithms make multiple passes over the
data. A counter is associated with each itemset that is used to keep its number of
occurrences in the database. In the first pass over the database, the set of
frequent itemsets of length 1 (one item actually) are determined by counting each
item in the database. Each subsequent pass aims to find the frequent itemsets of
a certain length in the increasing order, i.e. second pass finds the frequent
itemsets of length two and so on. Each pass starts with a seed set consisting of
the frequent itemsets found in the previous pass, and tries to generate a set of
possibly frequent itemsets for that pass (candidate itemsets), and minimize the
cardinality of that set. Then, by scanning the database, the actual support for
each candidate itemset is computed and those that are large are qualified to the
set of the seed set of next pass. This process goes on until no new frequent
itemsets are found in a pass.

Generally the efficiency of an association rule algorithm depends on the
size of the candidate set (while generating and counting), and the number of
scans over the database. SETM [16] uses SQL commands to mine association rules. The number of scans over the database is equal to the length of maximal itemset. As suggested in [14,15], most of the association rule algorithms concentrate on the generation of the candidates, counting of the support of a candidate itemset, number of scans over the database and the data structure employed to extract frequent itemsets efficiently.
CHAPTER 3

PROBLEM DESCRIPTION

3.1 Definitions

Let \( I = \{i_1, i_2, \ldots, i_m\} \) 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 \( T \subseteq I \) and each transaction is associated with a unique identifier called \( TID \). We define the following basic terms as given in [17].

Definition 3.1.1 An itemset \( X \) is a set of items in \( I \). An itemset \( X \) is called a \( k \)-itemset if it contains \( k \) items from \( I \).

Definition 3.1.2 A transaction \( T \) satisfies an itemset \( X \) if \( X \subseteq T \). The support of an itemset \( X \) in \( D \), \( \text{support}(X) \), is the number of transactions in \( D \) that satisfy \( X \).

Definition 3.1.3 An itemset is called a frequent itemset if the support of \( X \) in \( D \) exceeds a minimum support threshold explicitly by the user, and an infrequent itemset otherwise.

Definition 3.1.4 The negative border or frequent itemset border of the set of frequent itemsets is the set of itemsets that are generated as a candidate but fail to qualify into the set of frequent itemsets.

Definition 3.1.5 An association rule is an implication of the form \( X \rightarrow Y \), where \( X \subseteq I \), \( Y \subseteq I \), and \( X \cap Y = \emptyset \). \( X \) is called the antecedent of the rule, and \( Y \) is
called the consequent of the rule. The rule $X \rightarrow Y$ holds in $D$ with confidence $c$ where $c = \frac{\text{Support}(X \cup Y)}{\text{Support}(X)}$. The rule $X \rightarrow Y$ has support $s$ in $D$ if the fraction of the transactions in $D$ contains $X \cup Y$.

Note that the support of an itemset or rule, and the confidence of a rule, takes values in the interval $[0, 1]$. In most algorithms, itemsets are actually implemented as lists where the items appear in lexicographic order, which allows them to be processed easily and efficiently.

Example 3.1.1

Consider the example transaction database of Table 1. There are 5 transactions in the database with TIDs 100, 200, 300, 400 and 500. The set of items $I = \{A, B, C, D, E\}$. There are totally $2^5 - 1 = 32$ non-empty itemsets (each non-empty subset of $I$ is an itemset). \{A\} is a 1-itemset and \{AB\} is 2-itemset, and so on. $\text{support} (A) = 4$ since 4 transactions include A in it. Let us assume that the minimum support (minsup) is taken as 40%. Then, \{A, B, C, D, AB, AC, AD, BD, ABD\} are the set of frequent itemsets since

\[ \text{Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.} \]
Table 1 An Example Transaction Database

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>A,B,C</td>
</tr>
<tr>
<td>200</td>
<td>B,D</td>
</tr>
<tr>
<td>300</td>
<td>A,C,D</td>
</tr>
<tr>
<td>400</td>
<td>A,B,D</td>
</tr>
<tr>
<td>500</td>
<td>A,B,D,E</td>
</tr>
</tbody>
</table>

their support is greater than or equal to 2 (40%*5), and the remaining ones are infrequent itemsets. Let us assume that the minimum confidence (minconf) is set to 60%. Then, $A \rightarrow D$ is an association rule with respect to the specified minsup and minconf (its support is 3 and its confidence is)

$$\frac{\text{Support}(AD)}{\text{Support}(A)} \times 100 = \frac{3}{4} \times 100 = 75\%.$$

On the other hand $A \rightarrow C$ is not a valid association rule since its confidence is 50%.
3.2 Problem

Given a set of transactions \( D \), the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified \( \text{minsup} \) and \( \text{minconf} \), respectively. Formally, the problem is generating all association rules \( X \rightarrow Y \), where

\[
\text{Support}(X \cup Y) \geq \text{minsup} \ast |D| \quad \text{and} \quad \frac{\text{Support}(X \cup Y)}{\text{Support}(X)} \geq \text{minconf}.
\]

The problem of finding association rules can be decomposed into two subproblems [9, 17]

- Generate all combinations of items with fractional transactions support (i.e. \( \frac{\text{Support}(X)}{|D|} \)) above a certain threshold, called \( \text{minsup} \).

- Use the frequent itemsets to generate association rules. For every frequent itemset \( I \), find all non empty subsets of \( I \). For every such subset \( a \), output a rule of the form \( a \rightarrow (l-a) \) if the ratio of the support(\( I \)) to \( \text{Support}(a) \) is at least \( \text{minconf} \). If an itemset is found to be frequent in the first step, the support of that itemset should be maintained in order to compute the confidence of the rule in the second step.

Finding all frequent itemsets is the more computationally intensive of the two subproblems, and research efforts have focused on finding more efficient algorithms. Some methods search for only a subset of all frequent itemsets that has the property of summarizing or of allowing to infer the information on all frequent itemsets.
The rule generation stage is usually much more efficient than the first stage, but has the drawback of possibly producing a very large number of rules, more than a human analyst could handle. It is not unusual to obtain tens of thousands of rules. To address this problem, researchers have focused either on defining a measure for the interestingness or degree of surprise of the rule, or on determining subsets of rules from which all other rules can be inferred using a set of interference rules.

Note that both these stages have exponential complexity in their worst case scenarios. In the case of finding the frequent itemsets, the complexity is in terms of the number of items. Indeed, given n items, we have $2^n$ itemsets, all of which could be frequent. In the case of generating the association rules, the complexity is determined by the number of frequent itemsets and by their size, because for a k-itemset we can generate $2^k - 2$ association rules. This value was obtained by considering the number of possible antecedents of the rule and by excluding the case of the empty set and that of the maximal set, which would imply an empty antecedent and, respectively, an empty consequent. Although both subproblems have exponential complexity, in practice the first one is more costly because for the computation of the support of an itemset we need to access the transactions of T and verify the inclusion of the itemset in each transaction.
3.2.1 Mining Frequent Itemsets

A maximal frequent itemset is defined as a frequent itemset for which none of its immediate supersets are frequent. [20]

To illustrate this concept, consider the itemset lattice shown in Figure 2. The itemsets in the lattice are divided into two groups: those that are frequent
and those that are infrequent. A frequent itemset border, which is represented by a solid line, is also illustrated in the figure. Every itemset located above the border is frequent, while those located below the border, (shaded nodes) are infrequent. Among the itemsets residing near the border, \{a, b\}, \{b, c\} and \{c, d\} are considered to be maximal frequent itemsets because their immediate super sets are infrequent. An itemset such as \{b, c\} is maximal frequent because all its immediate super sets, \{a, b, c\} and \{b, c, d\} are infrequent. In contrast, \{a\} is non-maximal because one of its immediate super sets \{a, b\} is frequent.

Maximal frequent itemsets effectively provide a compact representation of itemsets. In other words, they form the smallest set of itemsets from which all frequent itemsets can be derived. For example the frequent itemsets shown in
Figure 2 can be divided into two groups:

- Frequent itemsets that begin with item a and that may contain items c or d. This group includes itemsets such as \{a\} and \{a, b\}.
- Frequent itemsets that begin with items b, c or d. This group includes itemsets such as \{b\}, \{b, c\}, \{c\} and \{c, d\}

Frequent itemsets that belong to the first group are subsets of \{a, b\}, while those that belong in the second group are subsets of \{b, c\} or \{c, d\}. Hence, the maximal frequent itemsets \{a, b\}, \{b, c\} and \{c, d\} provide a compact representation of the frequent itemset shown in Figure 2.

Maximal frequent itemsets provide a valuable representation for data sets that can produce very long, frequent itemsets, as there are exponentially many frequent itemsets in such data. Nevertheless, this approach is practical only if an efficient algorithm exists to explicitly find the maximal frequent itemsets without having to enumerate all their subsets.

Despite providing a compact representation, maximal frequent itemsets do not contain the support information of their subsets. For example, the support of the maximal frequent itemsets \{a, b\}, \{b, c\} and \{c, d\} do not provide any hint about the support of their subsets. An additional pass over the data set is therefore needed to determine the support counts of the non-maximal frequent itemsets. In some cases, it might be desirable to have a minimal representation of frequent itemsets that preserves the support information.
To discover frequent itemsets, algorithms have to explore the lattice.

Based on how the traversal is made, algorithms for mining frequent itemsets can be classified as [9]:

1. Breadth-first algorithms, which explore the lattice level by level.
2. Depth-first algorithms, which explore the lattice by moving from node to a next level node if possible or otherwise to the next node at the current level.

Based on the point from which they start the lattice traversal, algorithms can also be categorized as:

1. Bottom-up algorithms, which start their traversal at the bottom of the lattice.
2. Top-down algorithms, which start their traversal at the top of the lattice.
The breadth-first algorithm traverses the lattice as in the Figure 3(a) [20]. It first discovers all the frequent 1-itemset, followed by the frequent 2-itemsets and so on, until no new frequent itemsets are generated. The itemset lattice can also be traversed in a depth-first manner as shown in Figure 3(b) [20] and 4 [20]. The algorithm can start from, say node a in Figure 4, and count its support to determine whether it is frequent. If so, the algorithm progressively expands the next level of nodes, i.e. ab, abc and so on, until an infrequent node is reached, say abcd. It then backtracks to another branch, say abce, and continues the search from there.
The depth-first approach is often used by algorithms designed to find maximal frequent itemsets. This approach allows the frequent itemset border to be detected more quickly than using a breadth-first approach. Once a maximal frequent itemset is found, substantial pruning can be performed on its subsets. For example, if the node bode shown in Figure 4 [20] is maximal frequent, then the algorithm does not have to visit the sub trees rooted at bd, be, c, d and e because they will not contain any maximal frequent itemsets. However, if abc is maximal frequent, only the nodes such as ac and bc are not maximal frequent (but the sub trees of ac and bc may still contain maximal frequent itemsets). The depth-first approach also allows a different kind of pruning based on the support.
of itemsets. For example, suppose the support for \( \{a, b, c\} \) is identical to the support for \( \{a, b\} \). The sub trees rooted at abd and abe can be skipped because they are guaranteed not to have any maximal frequent itemsets.

### Top-Down versus Bottom-Up

In the top-down traversal strategy, pairs of frequent \((k-1)\) itemsets are merged to obtain candidate \(k\)-itemsets. This top-down strategy is effective provided the maximal length of a frequent itemset is not too long. The configuration of frequent itemsets that work best with this strategy is shown in Figure 5(a) [20], where the darker nodes represent infrequent itemsets.

Alternatively, a bottom-up traversal strategy looks for more specific frequent itemsets first, before finding the more general frequent itemsets. This strategy is useful to discover maximal frequent itemsets in dense transactions, where the frequent itemset border is located near the bottom of the lattice, as shown in Figure 5(b) [20].
A trivial algorithm for mining frequent itemsets would just compute the support of all itemsets from the lattice. The performance of this algorithm would then be $\Theta(2^k)$. Practical algorithms attempt to minimize the number of nodes examined in their traversal of the lattice of itemsets, so as to compute their support. Because it is impossible to know in advance which itemsets are frequent, algorithms will use various techniques to minimize the number of itemsets that they examine, and to ensure that no frequent itemsets are overlooked.
3.2.2 Mining Association Rules

After the frequent itemsets are computed, we have to generate the association rules. A very simple algorithm could work as follows: for each frequent itemset \( I \), and for each of its non empty subsets \( I_a \), we generate the rule \( I_a \rightarrow I - I_a \) if its confidence is equal to or greater than minconf. This algorithm would generate all possible association rules and improved version of it is presented in next chapter.

As suggested in [14, 15], most of the association rule generation algorithms concentrate on the following aspects to extract frequent itemsets efficiently:

- Reducing I/O time by reducing the number of scans over the database
- Minimizing the set of candidate itemsets and
- Counting the support of candidate itemsets over the database in less time

In this sense, association rule algorithms generally differ on

- The generation of the candidates
- Counting of the support of a candidate itemset
- Number of scans over the database and
- The data structures employed
CHAPTER 4

THE APRIORI ALGORITHM

In [1], Agrawal and Srikanth introduced the classic algorithm Apriori. The algorithm has been presented in [19]. In this section we describe Apriori and the ideas behind it.

4.1 Apriori Principle

"If an itemset is frequent, then all of its subsets must also be frequent. Conversely if an itemset is infrequent then all of its supersets must be infrequent too."

To illustrate the idea behind the Apriori principle, consider the itemset lattice shown in Figure 6 [20]. Suppose \{c, d, e\} is a frequent itemset. Clearly, any transaction that contains \{c, d, e\} must also contain its subsets, \{c, d\}, \{c, e\}, \{d, e\}, \{c\}, \{d\} and \{e\}. As a result, if \{c, d, e\} is frequent, then all subsets of \{c, d, e\} (i.e. the shaded itemsets in this figure) must also be frequent.
Figure 6 An illustration of the Apriori principle. If \{c, d, e\} is frequent, then all subsets of this itemset are frequent.

Conversely, if an itemset such as \{a, b\} is infrequent, then all of its supersets must be infrequent too. As illustrated in Figure 6 [20], the entire subgraph containing the supersets of \{a, b\} can be pruned immediately once \{a, b\} is found to be infrequent. This strategy of trimming the exponential search space based on the support measure is known as support-based pruning. Such a pruning strategy is made possible by a key property of the support measure, namely, that the support for an itemset never exceeds the support for its subsets. This property is also known as the anti-monotone property of the support measure.
Figure 7 An illustration of support-based pruning. If \{a, b\} is infrequent, then all supersets of \{a, b\} are infrequent.

Monotonicity Property

Let \( I \) be a set of items, and \( J = 2^I \) be the power set of \( I \). A measure \( f \) is monotone if

\[
\forall X, Y \in J : (X \subseteq Y) \rightarrow f(X) \leq f(Y),
\]

which means that if \( X \) is a subset of \( Y \), then \( f(X) \) must not exceed \( f(Y) \). On the other hand, \( f \) is anti-monotone if

\[
\forall X, Y \in J \text{ then } X \subseteq Y \rightarrow f(X) \geq f(Y)
\]

which means that if \( X \) is a subset of \( Y \), then \( f(Y) \) must not exceed \( f(X) \).
Any measure that possesses an anti-monotone property can be incorporated directly into the mining algorithm to efficiently prune the exponential search space of candidate itemsets.

4.2 Frequent Itemset Mining in Apriori Algorithm

Apriori mines the frequent itemsets in a bottom-up, breadth-first fashion and its pseudo code is given by Figure 8, Algorithm [21]. The algorithm works iteratively, generates the candidate itemsets to be counted in a pass by using only the itemsets found large in the previous pass, without considering the transactions in the database. At iteration k, Apriori starts with a collection of possible k-frequent itemsets called candidate itemsets, scans the data to determine which candidates are frequent, and then generates candidates for the next iteration. The basic intuition is that any subset of a large itemset must be large. Therefore, the candidate itemsets having k items can be generated by joining frequent itemsets having k-1 items, and deleting those that contain any subset that is not large. This procedure results in generation of a much smaller number of candidate itemsets. The algorithm starts initially with a collection of candidate itemsets consisting of all 1-itemsets.

We use the notation $X[j]$, to represent the $i^{th}$ item in $X$. The $k$-prefix of an itemset $X$ is the $k$-itemset $\{X[1], \ldots, X[k]\}$ and $\sigma$ to represent support.
Algorithm 1: Apriori - Itemset Mining

Input: \( D, \sigma \) // \( D \) is the set of transactions.
Output: \( F(D, \sigma) \)

1: \( C_1 := \{\{i\} | i \in I\} \)
2: \( K := 1 \)
3: while \( C_k \neq \{\} \) do
4: // Compute the supports of all candidate itemsets
5: for all transactions \((t_d, l) \in D\) do
6: for all candidate itemsets \( X \in C_k \) do
7: if \( X \subseteq l \) then
8: \( X\).support++
9: end if
10: end for
11: end for
12: // Extract all frequent itemsets
13: \( F_k := \{X \mid X\).support \geq \sigma\} \)
14: // Generate new candidate itemsets
15: for all \( X, Y \in F_k, X[i] = Y[i] \) for \( 1 \leq i \leq k-1 \), and \( X[k] < Y[k] \) do
16: \( I = X \cap \{Y[k]\} \)
17: if \( \forall J \subseteq I, |J| = k : J \in F_k \) then
18: \( C_{k+1} := C_{k+1} \cap I \)
19: end if
20: end for
21: \( k++ \)
22: end while

Figure 8: Frequent itemsets mining in Apriori Algorithm

The algorithm performs a breadth-first search through the search space of all itemsets by iteratively generating candidate itemsets \( C_{k+1} \) of size \( k+1 \), starting with \( k = 0 \) (line 1). An itemset is a candidate if all of its subsets are known to be frequent. More specifically, \( C_1 \) consists of all items in \( I \), and at a certain level \( k \), all itemsets of size \( k + 1 \) are generated. This is done in two steps. First, in the join step, \( F_k \) is joined with itself. The union \( X \cup Y \) of itemsets \( X, Y \in F_k \) is generated if
they have the same $k - 1$ prefix (lines 20–21). In the prune step, $X \cup Y$ is only inserted into $C_{k+1}$ if all of its $k$-subsets occur in $F_k$ (lines 22–24).

To count the supports of all candidate $k$-itemsets, the database, which retains on secondary storage in the horizontal database layout, is scanned one transaction at a time, and the supports of all candidate itemsets that are included in that transaction are incremented (lines 6–12). All itemsets that turn out to be frequent are inserted into $F_k$ (lines 14–18).

If the number of candidate $k +1$ itemsets is too large to retain into main memory, the candidate generation procedure stops and the supports of all generated candidates is computed as if nothing happened. But then, in the next iteration, instead of generating candidate itemsets of size $k +2$, the remainder of all candidate $k+1$ itemsets is generated and counted repeatedly until all frequent itemsets of size $k + 1$ are generated.

4.3 Association Mining Rule

Given all frequent itemsets, we can now generate all frequent and confident association rules. The algorithm is very similar to the frequent itemset mining algorithm and is given in Figure 8, Algorithm 4.3.1 [21].

First, all frequent itemsets are generated using Algorithm 4.2.1. Then, every frequent itemset $I$ is divided into a candidate head $Y$ and a body $X = \{I - Y\}$. This process starts with $Y = \emptyset$, resulting in the rule $I \rightarrow \emptyset$, which always holds with 100% confidence (line 4). After that, the algorithm iteratively generates candidate heads $C_{k+1}$ of size $k + 1$, starting with $k = 0$ (line 5). A head is a candidate if all of
its subsets are known to represent confident rules. This candidate head generation process is exactly like the candidate itemset generation in Algorithm 4.2.1 (lines 11–16). To compute the confidence of a candidate head $Y$, the support of $I$ and $X$ is retrieved from $F$. All heads that result in confident rules are inserted into $H_k$ (line 9). In the end, all confident rules are inserted into $R$ (line 20).

It can be seen that this algorithm does not fully exploit the monotonicity of confidence. Given an itemset $I$ and a candidate head $Y$, representing the rule $\{I - Y\} \rightarrow Y$, the algorithm checks for all $Y' \subset Y$ whether the rule $\{I - Y'\} \rightarrow Y'$ is confident, but not whether the rule is $\{I - Y\} \rightarrow Y'$ confident. Nevertheless, this is perfectly possible if all rules are generated from an itemset $I$, only if all rules are already generated for all itemsets $I' \subset I$.

We use the notation $\sigma$ to represent support and $\gamma$ to represent confidence in the following algorithm.
ALGORITHM APRIORI – ASSOCIATION RULE MINING

INPUT: \( D, \sigma, \gamma \)  \hspace{1em} // D is the set of transactions
OUTPUT: \( R(D, \sigma, \gamma) \)

1: Compute \( F(D, \sigma) \)

2: \( R := \emptyset \)

3: for all \( I \in F \) do

4: \( R := R \cap I \Rightarrow \emptyset \)

5: \( C_i := \{ \{i\} | i \in I \} \)

6: \( K := 1 \)

7: \hspace{1em} while \( C_k \neq \emptyset \) do

8: \hspace{2em} // Extract all heads of confident association rules

9: \hspace{2em} \( H_k := \{ X \in C_k | \text{confidence} (I - X) \Rightarrow X, D) \geq \gamma \} \)

10: \hspace{2em} // Generate new candidate heads

11: \hspace{3em} for all \( X, Y \in H_k \), \( X[i] = Y[j] \) for \( 1 \leq i \leq k - 1 \), and \( X[k] < Y[k] \) do

12: \hspace{4em} \( l = X \cap \{Y[k]\} \)

13: \hspace{4em} if \( \forall J \subseteq l, |J| = k : J \in H_k \) then

14: \hspace{5em} \( C_{k+1} := C_{k+1} \cap l \)

15: \hspace{4em} end if

16: \hspace{3em} end for

17: \hspace{2em} \( k++ \)

18: \hspace{1em} end while

19: \hspace{1em} // Cumulate all association rules

20: \( R := R \cap \{ I - X \Rightarrow X | X \in H_1 \cap \cdots \cap H_k \} \)

21: end for

Figure 9 Apriori Association rule mining

However, exploiting monotonicity as much as possible is not always the best solution. Since computing the confidence of a rule only requires the lookup of the support of at most 2 itemsets, it might even be better not to exploit the confidence monotonicity at all and simply remove the prune step from the candidate generation process, i.e., remove lines 13 and 15. Of course, this depends on the efficiency of finding the support of an itemset or a head in the used data structures.
Luckily, if the number of frequent and confident association rules is not too large, then the time needed to find all such rules consists mainly of the time that was needed to find all frequent sets.

Since the proposal of this algorithm for the association rule generation phase, no significant optimizations have been proposed anymore and almost all research has been focused on the frequent itemset generation phase.

4.4 Data Structures

The candidate generation and the support counting processes require an efficient data structure in which all candidate itemsets are stored since it is important to efficiently find the itemsets that are contained in a transaction or in another itemset.

4.4.1 Hash Tree

In order to efficiently find all $k$-subsets of a potential candidate itemset, all frequent itemsets in $F_k$ are stored in a hash table.

Candidate itemsets are stored in a hash-tree [19]. A node of the hash-tree either contains a list of itemsets (a leaf node) or a hash table (an interior node). In an interior node, each bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth $d$ points to nodes at depth $d + 1$. Itemsets are stored in leaves.

When we add a $k$-itemset $X$ during the candidate generation process, we start from the root and go down the tree until we reach a leaf. At an interior node
at depth $d$, we decide which branch to follow by applying a hash function to the $X[d]$ item of the itemset, and following the pointer in the corresponding bucket. All nodes are initially created as leaf nodes. When the number of itemsets in a leaf node at depth $d$ exceeds a specified threshold, the leaf node is converted into an interior node, only if $k > d$.

In order to find the candidate-itemsets that are contained in a transaction $T$, we start from the root node. If we are at a leaf, we find which of the itemsets in the leaf are contained in $T$ and increment their support. If we are at an interior node and we have reached it by hashing the item $i$, we hash on each item that comes after $i$ in $T$ and recursively apply this procedure to the node in the corresponding bucket. For the root node, we hash on every item in $T$.

In the Apriori algorithm, candidate itemsets are partitioned into different buckets and stored in a hash tree. During support counting, itemsets contained in each transaction are also hashed into their appropriate buckets. That way, instead of comparing each itemset in the transaction with every candidate itemset, it is matched only against itemsets that belong to the same bucket, as shown in Figure 10 [20]
Figure 11 [20] shows an example of a hash structure. Each internal node of the tree uses the following hash function, $h(p) = p \mod 3$, to determine which branch of the current node should be followed next. For example, items 1, 4 and 7 are hashed to the same branch (i.e. the leftmost branch) because they have the same remainder after dividing the number by 3. All candidate itemsets are stored at the leaf nodes of the hash tree. The hash tree shown in Figure 10 contains 15 candidate 3-itemsets, distributed across 9 leaf nodes.

Consider a transaction, $t = \{1, 2, 3, 4, 5\}$. To update the support counts of the candidate itemsets, the hash tree must be traversed in such a way that all the leaf nodes containing candidate 3-itemsets belonging to $t$ must be visited at least once. At the root node of the hash tree, the items 1, 2 and 3 of the transaction
are hashed separately. Item 1 is hashed to the left child of the root node, item 2 is hashed to the middle child and item 3 is hashed to the right child. After hashing on item 1 at the root node, items 2, 3 and 5 of the transaction are hashed. Items 2 and 5 are hashed to the middle child while item 3 is hashed to the right child, as shown in Figure 12 [20]. This process continues until the leaf nodes of the hash tree are reached. The candidate itemsets stored at the visited leaf nodes are compared against the transaction. If a candidate is a subset of the transaction, its support count is incremented. In this example, 5 out of the 9 leaf nodes are visited and 9 out of the 15 itemsets are compared against the transaction.
Figure 11 Hashing a transaction at the root node of the hash tree
4.4.2 Trie

Another data structure that is commonly used is a trie (or prefix-tree) [22, 23, 24, 25]. In a trie, every k-itemset has a node associated with it, as does its k − 1 prefix. The empty itemset is the root node. All the 1-itemsets are attached to the root node, and their branches are labeled by the item they represent. Every other k-itemset is attached to its k − 1 prefix. Every node stores the last item in the itemset it represents, its support, and its branches. The branches of a node
can be implemented using several data structures such as a hash table, a binary
search tree or a vector.

At a certain iteration k, all candidate k-itemsets are stored at depth k in the
trie. In order to find the candidate-itemsets that are contained in a transaction T,
we start at the root node. To process a transaction for a node of the trie, (1)
follow the branch corresponding to the first item in the transaction and process
the remainder of the transaction recursively for that branch, and (2) discard the
first item of the transaction and process it recursively for the node itself. This
procedure can still be optimized, as is described in [23].

Also the join step of the candidate generation procedure becomes very
simple using a trie, since all itemsets of size k with the same k − 1 prefix are
represented by the branches of the same node (that node represents the k − 1
prefix). Indeed, to generate all candidate itemsets with k − 1 prefix X, we simply
copy all siblings of the node that represents X as branches of that node.
Moreover, we can try to minimize the number of such siblings by reordering the
items in the database in support ascending order [23, 24, 25].

Using this heuristic, we reduce the number of itemsets that is generated
during the join step, and hence, we implicitly reduce the number of times the
prune step needs to be performed. Also, to find the node representing a specific
k-itemset in the trie, we have to perform k searches within a set of branches.
Obviously, the performance of such a search can be improved when these sets
are kept as small as possible.
Figure 13 presents a trie (without the counters) that stores the itemsets \{A\}, \{C\}, \{E\}, \{F\}, \{A, C\}, \{A, E\}, \{A, F\}, \{E, F\}, \{A, E, F\}.
CHAPTER 5

OPTIMIZATION

A lot of other algorithms proposed after the introduction of Apriori retain the same general structure, adding several techniques to optimize certain steps within the algorithm. Since the performance of the Apriori algorithm is almost completely dictated by its support counting procedure, most research has focused on that aspect of the Apriori algorithm. As already mentioned before, the performance of this procedure is mainly dependent on the number of candidate itemsets that occur in each transaction.

The first algorithm proposed to generate all frequent itemsets in a depth-first manner is the Eclat algorithm by Zaki [26, 27]. Later, several other depth-first algorithms have been proposed [28, 29, 30] of which the FP-growth algorithm by Han et al. [30, 2] is the most well known. In this chapter, we explain the FP-growth algorithm.

Given a transaction database D and a minimal support threshold $\sigma$, denote the set of all frequent k-itemsets with the same k - 1 prefix $i \subseteq I$ by $F[I](D, \sigma)$. (Note that $F[\{}(D, \sigma) = F(D, \sigma).) FP$-growth recursively generate for every item $i \in I$ the set $F[\{i\}](D, \sigma)$.
For the sake of simplicity and presentation, we assume that all items that occur in the transaction database are frequent. In practice, all frequent items can be computed during an initial scan over the database, after which all infrequent items will be ignored.

5.1 FP-growth

In order to count the supports of all generated itemsets, FP-growth uses a combination of the vertical and horizontal database layout to store the database in main memory. Instead of storing the cover for every item the database, it stores the actual transactions from the database in a trie structure and every item has a linked list going through all transactions that contain that item. This new data structure is denoted by FP-tree (Frequent Pattern tree) and is created as follows [20]. Again, we order the items in the database in support ascending order for the same reasons as before. First, create the root node of the tree, labeled with "null". For each transaction in the database, the items are processed in reverse order (hence, support descending) and a branch is created for each transaction. Every node in the FP-tree additionally stores a counter which keeps track of the number of transactions that share that node. Specifically, when considering the branch to be added for a transaction, the count of each node along the common prefix is incremented by 1, and nodes for the items in the transaction following the prefix are created and linked accordingly. Additionally, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links. Each item in this header table also stores its support.
The reason to store transactions in the FP-tree in support descending order is that in this way, it is hoped that the FP-tree representation of the database is kept as small as possible since the more frequently occurring items are arranged closer to the root of the FP-tree and thus are more likely to be shared.

Example 5.1.1 Assume we are given a transaction database and a minimal support threshold of 2. First, the supports of all items is computed, all infrequent items are removed from the database and all transactions are reordered according to the support descending order resulting in the example transaction database in Table 2. The FP-tree for this database is shown in Figure 14.

<table>
<thead>
<tr>
<th>Tid</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>{a, b, c, d, e, f}</td>
</tr>
<tr>
<td>200</td>
<td>{a, b, c, d, e}</td>
</tr>
<tr>
<td>300</td>
<td>{a, d}</td>
</tr>
<tr>
<td>400</td>
<td>{b, d, f}</td>
</tr>
<tr>
<td>500</td>
<td>{a, b, c, e, f}</td>
</tr>
</tbody>
</table>

Given such an FP-tree, the supports of all frequent items can be found in the header table. Obviously, the FP-tree is just like the vertical and horizontal database layouts a lossless representation of the complete transaction database for the generation of frequent itemsets. Indeed, every linked list starting from an
item in the header table actually represents a compressed form of the cover of that item. On the other hand, every branch starting from the root node represents a compressed form of a set of transactions.

![FP-tree diagram](image)

Figure 14 An example of an FP-tree

FP-growth algorithm uses some additional steps to maintain the FP-tree structure during the recursion steps. More specifically, in order to generate for every $i \in I$ all frequent itemsets in $F[[i]](D, \sigma)$, FP-growth creates the so called $i$-projected database of $D$. The FP-growth algorithm is given Figure 15.
ALGORITHM: FP-GROWTH

Input: \( D, \sigma, I \subseteq I \)
Output: \( F[I|D, \sigma] \)

1: \( F[I] = \{ \} \)
2: for all \( i \in I \) occurring in \( D \) do
3: \( F[I] = F[I] \cup \{ i \} \)
4: // Create \( D' \)
5: \( D' = \{ \} \)
6: \( H = \{ \} \)
7: for all \( j \in I \) occurring in \( D \) such that \( j > i \) do
8: \( \text{if } \text{sup}(\bigcup \{ i, j \}) \geq \sigma \text{ then} 
9: \( H = H \cup \{ j \} \)
10: \( \text{end if} \)
11: \( \text{end for} \)
12: for all \( (\text{tid}, X) \in D \) with \( i \in X \) do
13: \( D' := D' \cup \{ (\text{tid}, X \cap H) \} \)
14: \( \text{end for} \)
15: // Depth-first recursion
16: Compute \( F[I \cup \{ i \}|D', \sigma] \)
17: \( F[I] := F[I] \cup F[I \cup \{ i \}] \)
18: end for

Figure 15 FP-growth Algorithm

First, FP-growth computes all frequent items for \( D' \) at lines 6–10, which is of course different in every recursion step. This can be efficiently done by simply following the linked list starting from the entry of \( i \) in the header table. Then at every node in the FP-tree it follows its path up to the root node and increments the support of each item it passes by its count. Then, at lines 11–13, the FP-tree for the \( i \)-projected database is built for those transactions in which \( i \) occurs, intersected with the set of all frequent items in \( D \) greater than \( i \). These transactions can be efficiently found by following the node links starting from the
entry of item i in the header table and following the path from every such node up to the root of the FP-tree and ignoring all items that are not in H. If this node has count n, then the transaction is added n times. Of course, this is implemented by simply incrementing the counters, on the path of this transaction in the new i-projected FP-tree, by n. However, this technique does require that every node in the FP-tree also stores a link to its parent. Additionally, we can also use the technique that generates only those candidate itemsets that occur at least once in the database. Indeed, we can dynamically add a counter initialized to 1 for every item that occurs on each path in the FP-tree that is traversed.

These steps can be further optimized as follows. Suppose that the FP-tree consists of a single path. Then, we can stop the recursion and simply enumerate every combination of the items occurring on that path with the support set to the minimum of the supports of the items in that combination. Essentially, this technique is similar to the technique used by all other algorithms when the support of an itemset is equal to the support of any of its subsets.

As can be seen, at every recursion step, an item j occurring in D' actually represents the itemset I ∪ {i, j}. In other words, for every frequent item i occurring in D, the algorithm recursively finds all frequent 1-itemsets in the i-projected database D'.

Although the authors of the FP-growth algorithm claim that their algorithm [2, 30] does not generate any candidate itemsets, we have shown that the algorithm actually generates a lot more candidate itemsets since it essentially
uses the same candidate generation technique as is used in Apriori but without its prune step.

In FP-growth, the cover of an item is compressed using the linked list starting from its node-link in the header table, but, every node in this linked list needs to store its label, a counter, a pointer to the next node, a pointer to its branches and a pointer to its parent.
CHAPTER 6

EXPERIMENTAL EVALUATION

6.1 Experimental Datasets

We choose real and synthetic datasets for testing the performance of the algorithms. All datasets are taken from the source LARIM [35]. Table 6.1 shows the characteristics of the real and synthetic datasets used in our experiments. It shows the number of items, average transaction length, number of transactions and size for each dataset.

For all experiments, we used three datasets with different characteristics. Thus, the advantages and disadvantages of the algorithms can be observed. We used two real datasets which are Chess and Mushroom. We used one synthetic dataset which is T10I4D100K. [33]

Chess dataset is derived from the chess game. It contains real entries. There are 3196 transactions in it, while each transaction has average 37 items. Chess is very dense dataset. It produces many long frequent itemsets even for with high values of support.

Mushroom dataset contains characteristics of various species of mushrooms. It contains real entries. There are 8124 transactions in it, while each
transaction has average 23 items. It produces many long frequent itemsets even for with high values of support.

T10I4D100K is a synthetic datasets, which have been used as benchmarks for testing association mining algorithms. Dataset T10I4D100K contains an average 20 transaction size, an average size of the maximal potentially frequent itemsets of 4. There are 100000 transactions in it. It produces many long frequent itemsets even for with high values of support.

Table 3 Data Set Characteristics

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Items</th>
<th>Records</th>
<th>Average Transaction Length</th>
<th>DB size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess</td>
<td>76</td>
<td>3196</td>
<td>37</td>
<td>335kb</td>
</tr>
<tr>
<td>Mushroom</td>
<td>120</td>
<td>8124</td>
<td>23</td>
<td>558kb</td>
</tr>
<tr>
<td>T10I4D100K</td>
<td>1000</td>
<td>100000</td>
<td>20</td>
<td>3928kb</td>
</tr>
</tbody>
</table>

6.2 Experiments

In this section, we report our performance study of the three algorithms for mining frequent itemsets and finding association rules: Apriori using hash data structure, Apriori using trie data structure and FP-growth using trie data structure.
All the experiments are performed on a 3 Ghz. Pentium 4 PC with 1 Gb. RAM, running on Red Hat Linux. We tested the three algorithms on various datasets which are described in section 5.3.

All the algorithms produced almost the same frequent itemsets and association rules with differences in execution times. The performance measure was the execution time of the algorithms on the datasets with the different minimum support values.

Table 4 through Table 7 shows the results we obtained from experimentation for the datasets Chess, Mushroom and T10I4D100K.

Table 4 Running times for Chess data set with confidence 1

<table>
<thead>
<tr>
<th>Support</th>
<th>Apriori with hash</th>
<th>Apriori with trie</th>
<th>FP-growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5(1598)</td>
<td>64724</td>
<td>36149</td>
<td>22804</td>
</tr>
<tr>
<td>0.65(2077)</td>
<td>16510</td>
<td>1678</td>
<td>854</td>
</tr>
<tr>
<td>0.75(2397)</td>
<td>388</td>
<td>313</td>
<td>101</td>
</tr>
<tr>
<td>0.85(2717)</td>
<td>8</td>
<td>34</td>
<td>7</td>
</tr>
<tr>
<td>0.9(2876)</td>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 5 Running times for Mushroom data set with confidence 1

<table>
<thead>
<tr>
<th>Support</th>
<th>Apriori with hash</th>
<th>Apriori with trie</th>
<th>FP-growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25(2031)</td>
<td>268</td>
<td>300</td>
<td>243</td>
</tr>
<tr>
<td>0.35(2843)</td>
<td>7</td>
<td>36</td>
<td>23</td>
</tr>
<tr>
<td>0.45(3656)</td>
<td>2</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>0.5(4062)</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>0.55(4468)</td>
<td>0.5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0.65(5281)</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0.75(6093)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.85(6905)</td>
<td>0.5</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>0.9(7312)</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Table 6 Running times for Mushroom data set with confidence 0.5

<table>
<thead>
<tr>
<th>Support</th>
<th>Apriori with hash</th>
<th>Apriori with trie</th>
<th>FP-growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25(2031)</td>
<td>17432</td>
<td>2422</td>
<td>2393</td>
</tr>
<tr>
<td>0.35(2843)</td>
<td>35</td>
<td>183</td>
<td>171</td>
</tr>
<tr>
<td>0.45(3656)</td>
<td>3</td>
<td>36</td>
<td>32</td>
</tr>
<tr>
<td>0.5(4062)</td>
<td>1</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>0.55(4468)</td>
<td>1</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>0.65(5281)</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0.75(6093)</td>
<td>0.5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0.85(6905)</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.9(7312)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 7 Running times for T10l4D100K data set with confidence 0.5

<table>
<thead>
<tr>
<th>Support</th>
<th>Time Apriori with Hash</th>
<th>Time Apriori with Trie</th>
<th>Time FP-growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.002(200)</td>
<td>5067</td>
<td>1944</td>
<td>1882</td>
</tr>
<tr>
<td>0.003(300)</td>
<td>437</td>
<td>268</td>
<td>217</td>
</tr>
<tr>
<td>0.004(400)</td>
<td>293</td>
<td>79</td>
<td>55</td>
</tr>
<tr>
<td>0.005(500)</td>
<td>231</td>
<td>32</td>
<td>17</td>
</tr>
<tr>
<td>0.006(600)</td>
<td>196</td>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td>0.007(700)</td>
<td>174</td>
<td>17</td>
<td>8</td>
</tr>
<tr>
<td>0.008(800)</td>
<td>137</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>0.009(900)</td>
<td>115</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>0.01(1000)</td>
<td>106</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

The x-axis of the result figures shows the support factor and y-axis shows the process time in seconds.

Figure 16 shows the results we obtained from experimentation for the Chess dataset with confidence 1.
The Chess data set emerged 1272932 frequent itemsets and 3285832 association rules with minimum support 0.5 and confidence 1.

From the graph, Apriori with hash and FP-growth performed almost the same and faster than Apriori with trie until the minimum support was reduced to 0.8. For lower values of minimum support from 0.8 to 0.5 FP-growth performed very well and Apriori with hash performed badly.

FP-growth > Apriori with trie > Apriori with hash for minimum supports 0.5 to 0.8

FP-growth = Apriori with hash > Apriori with trie for minimum supports 0.8 to 1
Figure 17 shows the results we obtained from experimentation for the Mushroom dataset with confidence 1.

The Mushroom data set emerged 5545 frequent itemsets and 24033 association rules with minimum support 0.25 and confidence 1.

From the graph, Apriori with hash performed well over the FP-growth and Apriori with trie for minimum support values from 1 to 0.35 and then FP-growth performed little good over other two for minimum support value for 0.25.
Apriori with hash > FP-growth > Apriori with trie for minimum supports from 0.35 to 1

FP-growth > Apriori with hash > Apriori with trie for minimum supports from 0.35 to 0.25

Figure 18 shows the results we obtained from experimentation for the mushroom dataset with confidence 0.5.
The mushroom data set emerged 5545 frequent itemsets and 234007 association rules with minimum support 0.25 and confidence 0.5.

From the graph, Apriori with hash performed well over the FP-growth and Apriori with trie for minimum support values from 1 to 0.35 and then FP-growth performed little good over other two for minimum support value for 0.25. There is no performance difference when the confidence has changed from 1 to 0.5

Apriori with hash > FP-growth > Apriori with trie for minimum supports from 0.35 to 1.

Figure 18 Running time of the algorithms for the Mushroom Dataset with confidence 0.5.
FP-growth > Apriori with hash > Apriori with trie for minimum supports from 0.35 to 0.25

Figure 19 shows the results we obtained from experimentation for the T10I4D100K dataset.

![Graph showing running time vs support for T10I4D100K dataset]

Figure 19 Running time of the algorithms for the T10I4D100K Dataset with confidence 0.5.

The T10I4D100K data set emerged 13255 frequent itemsets and 20775 association rules with minimum support 0.2 and confidence 0.5. When the confidence was set to 1 there were no association rules.
From the graph, FP-growth performed well over the other two algorithms for all values of minimum support. Apriori with trie performed little good over the Apriori with hash. Apriori with hash performed badly for all values of minimum support.

FP-growth > Apriori with trie > Apriori with hash

Frequent itemsets algorithms generate very large number of itemsets and association rules. Figures 6.2.1 through 6.2.4 shows performance curves for the algorithms that generate frequent itemsets.

For all of the two real - datasets, the results show that for higher minimum support values all the three implementations perform almost the same since less number of frequent itemsets are generated and does not take much processing time. As we can observe from the above results for lower values of minimum support large numbers of frequent itemsets are generated and hence large numbers of association rules are also generated and it takes more processing time. At this level FP-growth performs better than other two implementations and Apriori with hash is much slower.

On the other hand for synthetic dataset T10I4D100k FP-growth performs faster than other two and Apriori with hash performs slower for all the values of minimum support.
CHAPTER 7

CONCLUSION

Association rule mining explores for interesting relationships among items in a given data set. An objective of association rule mining is to develop a systematic method using the given data set and finds relationships between the different items. Goal of association rule is finding associations among items from a set of transactions which contain a set of items.

There are many new and efficient algorithms have been developed in association mining. Additionally, new proposals offered for algorithms that improve run time for generating association rules or frequent itemset. The efficiency of a mining algorithm is a very important point for data mining.

Our work, described in this thesis, has focused on explaining the fundamentals of association mining and analyzes implementations of the well known association rule algorithms. Study focuses on algorithms Apriori and FP-Growth. Common principles and differences between the algorithms have been shown. Pros and cons of these algorithms are thoroughly investigated. All the algorithms produced almost the same frequent itemsets and association rules with differences in execution times. The performance measure was the
execution time of the algorithms on the datasets with the different minimum support values.

The results show that Apriori cannot be run very effective than FP-growth. In small size datasets and for higher values of minimum support both the algorithms perform well. For large datasets FP-growth algorithm is faster than Apriori algorithm.

In this thesis we only consider the presence or absence of an item in a transaction but not the number of occurrences of a particular item in a database transaction. Taking the weight of occurrence of item into consideration would be interesting and provide more opportunities for research in data mining.
BIBLIOGRAPHY


[2] J. Han, J. Pei, Y. Yin. "Mining Frequent Patterns without Candidate Generation". Proc. of ACM-SIGMOD, 2000.


[14] Ming-Syan Chen, Jiawei Han and Philip S Yu, “Data Mining: A overview from database perspective”. IEEE Transactions on Knowledge and Data Engineering, 8(6) pages 866-883, 1996.


[35] Laboratoire de Recherche sur l'Information Multimédia, référence http://larim.uog.ca


VITA

Graduate College
University of Nevada, Las Vegas

Pavankumar Bondugula

Home Address:

4235 Cottage Circle
Apt #3
Las Vegas, Nevada 89119

Degrees:

Bachelor of Science, Computer Science and Information Technology, 2004
Jawaharlal Nehru Technological University

Master of Science, Computer Science, 2006
University of Nevada, Las Vegas

Special Awards and Honors:
Young Scientist award, 1995

Publications:

Kazem Taghva, Pavankumar Bondugula, Darshana Gala, Jeffery Coombs, Untitled, ITCC-2006 (International Conference on Information Technology: Coding and Computing), Las Vegas, USA (to submit)


Pavankumar Bondugula, “Food Reaper”, National Technical Symposium of ISTE (Indian Society of Technical Education), Oct 2003, Hyderabad, India

Thesis Title: Implementation and Analysis of Apriori Algorithm for Data Mining

Thesis Examination Committee:
- Chairperson, Dr. Kazem Taghva, Ph. D.
- Committee Member, Dr. Ajoy Datta, Ph. D.
- Committee Member, Dr. Thomas Nartker, Ph. D.
- Graduate Faculty Representative, Dr. Shahram Latifi, Ph. D.