UPDATING LARGE ITEMSETS WITH EARLY PRUNING

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION SCIENCE AND THE INSTITUTE OF ENGINEERING AND SCIENCE OF BILKENT UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

By
Necip Fazıl Ayan

July, 1999

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Erol Arkun(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Uğur Güdükbay

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet Baray, Director of Institute of Engineering and Science

ABSTRACT

UPDATING LARGE ITEMSETS
WITH EARLY PRUNING

Necip Fazıl Ayan
M.S. in Computer Engineering and Information Science
Supervisor: Prof. Dr. Erol Arkun
July, 1999

With the computerization of many business and government transactions, huge amounts of data have been stored in computers. The existing database systems do not provide the users with the necessary tools and functionalities to capture all stored information easily. Therefore, automatic knowledge discovery techniques have been developed to capture and use the voluminous information hidden in large databases. Discovery of association rules is an important class of data mining, which is the process of extracting interesting and frequent patterns from the data. Association rules aim to capture the co-occurrences of items, and have wide applicability in many areas. Discovering association rules is based on the computation of large itemsets (set of items that occur frequently in the database) efficiently, and is a computationally expensive operation in large databases. Thus, maintenance of them in large dynamic databases is an important issue. In this thesis, we propose an efficient algorithm, to update large itemsets by considering the set of previously discovered itemsets. The main idea is to prune an itemset as soon as it is understood to be small in the updated database, and to keep the set of candidate large itemsets as small as possible. The proposed algorithm outperforms the existing update algorithms in terms of the number of scans over the databases, and the number of candidate large itemsets generated and counted. Moreover, it can be applied to other data mining tasks that are based on large itemset framework easily.

Key words: Data mining, association rules, large itemsets, update of large itemsets, early pruning.

ÖZET

ERKEN ELİMİNASYON İLE
YOĞUN NESNE KÜMELERİNİN GÜNCELLENMESİ

Necip Fazıl Ayan
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Erol Arkun
Temmuz, 1999

Bilişim uygulamalarının yaygınlaşması ile, bilgisayarlarda büyük miktarlarda veri depolanmasına başlanmıştır. Günümüz veri tabanı sistemleri, kullanıcıya depolanan bütün bilgilere kolayca ulaşabileceği araçları ve fonksiyonları sunmamaktadır. Büyük veri tabanlarında saklı olan bu bilgilere ulaşmak ve bu bilgileri kullanmak üzere, otomatik bilgi keşfetmeye yarayan teknikler geliştirilmektedir. Bu tekniklerden biri olan bağıntı kuralları bulma, depolanan verilerden, ilginç ve sıklıkla rastlanan şemaları tanıma işlevinin, yani veri araştırmasının çok önemli bir dalıdır. Bağıntı kuralları, nesnelerin bir arada olma durumlarını belirlemeyi amaçlar ve bir çok alanda geniş kullanılabilirliğe sahiptir. Bağıntı kuralları bulma, yoğun nesne kümelerinin (verilerde sıkça bir arada görülen nesnelerin) hesaplanması esasına dayanır ve büyük veri tabanlarında hesaplanması oldukça pahalı bir işlemdir. Bu yüzden, daha önce belirlenmiş bağıntı kurallarının korunması oldukça önemli bir konudur. Bu tezde, daha önceden bulunmuş olan nesne kümelerini göz önüne alarak, yoğun nesne kümelerini güncellemekte kullanılan hızlı bir algoritma sunulmaktadır. Algoritmanın temel fikri, herhangi bir nesne kümesini güncellenen veri tabanında yoğun olmadığı anlaşılır anlaşılmaz elemek ve böylece yoğun olması muhtemel nesne kümelerinin sayısını olabildiğince küçük tutmaktır. Sunulan algoritma, veri tabanı üzerindeki tarama sayısı ile üretilen ve sayılan nesne kümelerinin sayısı bakımından daha önce önerilen bütün güncelleme algoritmalarından daha iyidir. Ayrıca, sunulan algoritma yoğun nesne kümelerinin hesaplanması esasına dayanan diğer veri araştırması işlerine de kolayca uyarlanabilir.

Anahtar kelimeler: Veri araştırması, bağıntı kuralları, yoğun nesne kümeleri, yoğun nesne kümelerinin güncellenmesi, erken eliminasyon.

To my only love Burcu

ACKNOWLEDGMENTS

I am very grateful to my supervisor, Prof. Dr. Erol Arkun, for saving time for me among lots of administrative duties and guiding me in this thesis. I owe special thanks to Assoc. Prof. Abdullah Uz Tansel for his lots of contributions to this thesis and forcing me to publish two papers, and Assoc. Prof. Pierre Flener for improving my writing skills, teaching me to study in a more organized way, and pursuading me to remain in academic life. I would also like to thank to Assoc. Prof. Özgür Ulusoy and Asst. Prof. Uğur Güdükbay for accepting to read this thesis and for their valuable comments on it.

I would like to thank my 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.

The last, but the most, thanks are to my wife Burcu for her contributions to this thesis by discussing my ideas and reading the earlier drafts of this thesis, supporting me under any circumstances and being next to me everytime I need. And, as being more important than the others, thanks a lot for being my best friend, my darling, and the most valuable thing in my life, and allowing me to be my own.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Overview of the Thesis 4

2 A Survey in Association Rules **5**

2.1 Knowledge Discovery and Data Mining 5

2.2 Association Rules 8

2.3 Formal Problem Description 10

2.3.1 Definitions 10

2.3.2 Problem 12

2.4 Apriori and Partition Algorithms 14

2.5 Analysis of Algorithms 19

2.6 Variations of Association Rules 24

2.6.1 Association Rules with Hierarchy 24

2.6.2 Constrained Association Rules 24

2.6.3 Quantitative Association Rules 25

2.6.4 Sequential Patterns 26
2.6.5 Periodical Rules 26
2.6.6 Weighted Association Rules 27
2.6.7 Negative Association Rules 27
2.6.8 Ratio Rules 28
2.7 A Criticism on Large Itemset Framework 28
2.8 A Discussion on Association Rules 29

3 Updating Large Itemsets 31

3.1 Formal Problem Description 32
3.2 Previous Algorithms 33
3.3 Update with Early Pruning ($UWEP$) 35
3.4 Data Structures Employed 42
3.5 An Example Execution of the Algorithm 44
3.5.1 Comparison with the Existing Algorithms 47
3.6 Completeness and Efficiency of the Algorithm 49
3.7 Experimental Results 51
3.8 Theoretical Discussion of the Update Algorithms 54
3.8.1 Number of Candidates 54
3.8.2 Time Complexity 55

4 Case of Deleted Transactions 58

4.1 Existing Approaches 60

4.2 Challenges in Update for Deletion Case 61

5 Conclusion 64

5.1 Future Work on *UWEP* 65

List of Figures

2.1 Rule Generation Algorithm 13

2.2 Candidate Generation Algorithm 14

2.3 Apriori Algorithm 15

3.1 Update of Frequent Itemsets 36

3.2 Initial Pruning Algorithm 37

3.3 Candidate Generation Algorithm in *UWEP* 42

3.4 Speedup by *UWEP* over *Partition* Algorithm 52

3.5 Execution Times of *UWEP* and *Partition* Algorithms 53

4.1 FUP_2 Algorithm: Deletion of Transactions 61

List of Tables

2.1 An Example Transaction Database 12

2.2 An Overview of Association Rule Algorithms 23

3.1 Notation Used in Algorithm $UWEP$ 33

3.2 Possible Cases in Addition of Transactions 41

3.3 Set of Transactions DB and db 44

3.4 Number of Candidates Generated and Counted in the Example Database 48

3.5 Number of Candidates Generated and Counted on Synthetic Data 54

4.1 Possible Cases in Deletion of Transactions 60

Chapter 1

Introduction

With the storage of huge amounts of data in every field of life, it has become a difficult and time consuming task to examine and properly interpret the stored information. The human beings have become incapable of managing all the information stored in various forms of databases. The automatic **knowledge discovery tools** have emerged in order to overcome this difficulty, and have taken great attention of the researchers in the database literature. Knowledge discovery process includes all pre-processing steps on the data stored, discovering interesting patterns on the data, and the post-processing of the results found on the data. Pre-processing of the data includes the cleaning of data and preparing data to the discovery of frequent interesting patterns. **Data mining** refers to the discovery of interesting and frequent patterns from the data in the knowledge discovery process. These interesting patterns may be in the form of associations, deviations, regularities, etc. Post-processing step is the pruning of the discovered patterns and the presentation of them in an understandable and easy-to-handle manner to end-users.

Association rules are just one of the patterns that can be extracted from data by means of data mining techniques. Specifically, an association rule, $X \Rightarrow Y$, is a statement of the form "for a specified fraction of the total transactions, a particular value of the attribute set X determines the value of an attributes set Y with a certain confidence". In this sense, association rules aim to explain the presence of some attributes according to the presence or

absence of some other attributes. The problem was studied first by Agrawal et al. [AIS93] in 1993 on a supermarket basket data, and has been widely explored to date. On a supermarket basket data, an example association rule is "In 10% of the transactions, 85% of the people buying milk also buy yoghurt in that transaction". Here, the support of the rule is 10%, and the confidence of the rule is 85%.

Because of the applicability and usefulness of association rules in many fields such as supermarket transactions analysis, telecommunications, university course enrollment analysis, word occurrence in text documents, user's visit to WWW pages, etc., many researchers have proposed efficient algorithms to discover association rules. The problem of discovering co-occurrences of items in a small data is a very simple task. However, the large volume of data makes this problem difficult and efficient algorithms are needed.

In [AIS93], the problem of discovering association rules is decomposed into two parts: Discovering all frequent patterns (represented by *large itemsets*) in the database, and generating the association rules from those frequent itemsets. The second subproblem is a straightforward problem, and can be managed in polynomial time. On the other hand, the first task is difficult especially for large databases. The *Apriori* [AS94] is the first efficient algorithm on this issue, and many of the forthcoming algorithms are based on this algorithm. We leave the analysis of the major algorithms for extracting association rules to Chapter 2.

1.1 Motivation

Since the discovery of large itemsets in a large database is a computationally expensive process, their maintenance is also an important issue in dynamic databases. When the existing database is updated by adding new transactions or deleting existing ones, the computation of large itemsets in the updated database again is very costly, because it repeats much of the work done in the previous computations. There are two possibilities when the database is updated: (1) Some of the old large itemsets are no longer large in the updated

database, and (2) some new itemsets that were not large previously may become large in the updated database. The straightforward solution is to re-run an association algorithm on the updated database. However, as we noted previously, this discards all the rules discovered previously, and repeats all the work done. The maintenance of large itemsets has been an important issue, and a few algorithms were proposed to efficiently update large itemsets by taking the set of previously discovered rules into account. Instead of finding all large itemsets again, they generally use some heuristics to remove some of the old large itemsets, and to add new ones without doing much work. Especially, when the size of the added transactions is large, these algorithms perform much better than re-running an association rule algorithm over the updated database.

The efficiency of an update algorithm strongly depends on the size of the set of candidate itemsets (possibly large itemsets). The smaller the set of candidate itemsets is, the more efficient the update algorithm would be. In this thesis, we propose an efficient algorithm called **U***pdate* **W***ith* **E***arly* **P***runing (UWEP)* which updates large itemsets when new transactions are added to the existing database. It works iteratively on the new set of transactions, like most of the update algorithms. The major advantages of $UWEP$ are:

1. It scans the old database of transactions at most once and new database exactly once.
2. It generates and counts the minimum number of candidates in order to determine the set of new large itemsets.

The first advantage is achieved by converting the databases into inverted files, and counting itemsets over these inverted structures instead of scanning databases. $UWEP$ takes its power from reducing the set of candidate itemsets to a minimum. This is achieved by pruning an itemset that will become small from the set of generated candidate set as early as possible by means of a look-ahead pruning. In other words, it does not wait for the kth iteration for pruning a small k-itemset as the other algorithms do, but removes it from consideration as soon as it is determined to be small. Moreover, $UWEP$ promotes an itemset to the set of candidate itemsets if and only if it is large both in the new transactions and in the updated database. This feature yields a much smaller

candidate set when some of the old large itemsets are eliminated due to their absence in the new set of transactions. $UWEP$ is proposed as the best update algorithm in terms of the number of scans over the database, and the number of candidates generated and counted.

1.2 Overview of the Thesis

This thesis is organized as follows. Chapter 2 gives a broad survey on data mining, and association rules. The analysis of the algorithms to discover the association rules and the challenges faced are explained in this chapter in detail. Chapter 3 presents the algorithm $UWEP$, which is an efficient algorithm to update large itemsets. The completeness and optimality of $UWEP$, and the experimental and theoretical comparison with the existing algorithms are discussed in this chapter. In Chapter 4, the case of deleted transactions is examined in detail, and the challenges in update of large itemsets for the case of deletion are discussed. Finally, the thesis concludes with some future work in Chapter 5.

Chapter 2

A Survey in Association Rules

2.1 Knowledge Discovery and Data Mining

With the recent developments in computer storage technology, many organizations have collected and stored massive amounts of data. Even though very useful information is buried within this data, this information is not readily available for the users. Obviously, there is a need for developing techniques and tools that assist users to analyze and automatically extract hidden knowledge. *Knowledge discovery in databases* (*KDD*) includes techniques and tools to address this need.

Fayyad et al. [FPSS96a] defines knowledge discovery in databases as follows:

> "*KDD* is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in the data."

KDD, in fact, aims at discovering unexpected, useful and simple patterns, and it is an inter-disciplinary research area. It is of interest to researchers in machine learning, pattern recognition, databases, statistics, artificial intelligence, expert systems, graph theory, and data visualization. *KDD* systems generally use methods, algorithms, and techniques from all of these fields.

KDD process is an interactive and iterative multi-step process which uses

data mining techniques to extract interesting knowledge according to some specific measures and thresholds. Fayyad et al. [FPSS96a, FPSS96b] and Mannila [Man96, Man97] describe the steps of knowledge discovery as follows:

1. Understanding the domain, the prior knowledge and the goals of end-user,
2. creating a target data set,
3. pre-processing the data set (selection of data resources, cleaning the data from errors and noise, handling unknown values, reduction and projection of data, etc.),
4. choosing the data mining task and algorithm,
5. searching for interesting and frequent patterns (data mining),
6. post-processing the discovered patterns (further selection, elimination or ordering of patterns, visualization of the results), and
7. putting the results into use.

Note that data mining is a step of KDD and aims at discovering frequent and interesting patterns in data. These patterns can be of the form of regularities, exceptions, co-occurrences, etc. Data mining is an application dependent issue and different applications may require different data mining techniques. Fayyad et al. [FPSS96a, Fay98] classify the primary data mining techniques into 5 categories as *predictive modeling, clustering, summarization, dependency modeling*, and *deviation detection*. Classification and regression are examples of predictive modeling, association rules are examples of summarizing, functional dependencies are examples of dependency modeling, and sequential patterns are examples of deviation detection.

Chen et al. [CHY96] classify data mining methods according to three criteria:

1. What kind of databases to work on (relational, attribute–oriented, etc.)

2. What kind of knowledge to be mined (association rules, classification rules, characteristics rules, discriminating rules, sequential patterns, deviations, similarity, clustering, regression, etc.)

3. What kind of techniques to be utilized (data–driven miner, query–driven miner, interactive miner, etc.)

The easiest application areas for KDD seem to be the ones where human experts can be found in that area but the data is continuously changing. Another appropriate application area involves the fields that are difficult for the human beings to handle. In general, data mining techniques are useful in decision making, information management, query processing, and process control. The major areas in which data mining methods have been applied are database marketing, financial applications, weather forecasting, astronomy, molecular biology, health care data, and scientific data. For a good overview of application areas, refer to [FPSS96a].

The data mining task is a difficult problem. As pinpointed in [Fay98], the most important challenge in data mining is that the data mining problems are *ill–posed problems*. Many solutions exist for a given problem, but there is no absolute answer for the quality of the results. This is fundamentally different from the difficulties faced in well-defined problems like sorting data or matching a query to records. In most of the data mining applications, the size of the database is very large and moreover a large volume of data should be collected in order to reach stable and valid results. Generally, the results of the data mining activity is very large and post-processing of the results is inevitable for understanding them. Data mining is a discovery-driven process, i.e., end-users generally do not know what to discover in advance. The major challenges faced in knowledge discovery in databases are summarized in [FPSS96a] as follows:

- Large databases,
- high dimensionality of databases,
- over fitting,
- different types of data,

- changing data and knowledge,
- missing and noisy data,
- complex relationships between attributes,
- usefulness, certainty and expressiveness of results,
- understandability of results,
- interactive mining at multiple abstraction levels,
- user interaction and usage of prior knowledge,
- integration with other systems,
- mining from multiple sources of data, and
- protection of privacy and security.

2.2 Association Rules

Association rules are one of the promising aspects of data mining as a knowledge discovery tool, and have been widely explored to date. They allow to capture all possible rules that explain the presence of some attributes according to the presence of other attributes. An association rule, $X \Rightarrow Y$, is a statement of the form "for a specified fraction of transactions, a particular value of an attribute set X determines the value of attribute set Y as another particular value under a certain confidence". Thus, association rules aim at discovering the patterns of co-occurrences of attributes in a database. For instance, an association rule in a supermarket basket data may be "In 10% of transactions, 85% of the people buying milk also buy yoghurt in that transaction." The association rules may be useful in many applications such as supermarket transactions analysis, store layout and promotions on the items, telecommunications alarm correlation, university course enrollment analysis, customer behavior analysis in retailing, catalog design, word occurrence in text documents, user's visits to WWW pages, and stock transactions.

The problem of discovering association rules was first explored in [AIS93] on supermarket basket data, that is the set of transactions that include items purchased by the customers. In this pioneering work, the data was considered to be binary, i. e. an item exists in a transaction or not, and the quantity of the item in the transaction is irrelevant.

In [AIS93], mining of association rules was decomposed into two subproblems: discovering all frequent patterns (represented by large 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 are generally in the order of millions, and number of items (attributes) are generally in the order of thousands. When the data contains N items, then the number of possibly large itemsets is 2^N. However, the large 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 large itemsets. To reduce the number of possibly large itemsets, many efficient algorithms have been proposed. These algorithms generally use clever data structures (such as hash tables, hash trees, lattices, multi-hypergraphs, etc.) in order to reduce the size of possibly large 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 large itemsets of length 1 (one item actually) are determined by counting each item in the database. Each subsequent pass aims to find the large itemsets of a certain length in increasing order, i.e., second pass finds the large itemsets of length two, and so on. Each pass starts with a seed set consisting of the large itemsets found in the previous pass, and tries to generate a set of possibly large 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 large 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. As suggested in [AY98a, CHY96], most of the association rule algorithms concentrate on the following aspects to extract large itemsets efficiently:

1. Reducing I/O time by reducing the number of scans over the database,
2. minimizing the set of candidate itemsets,
3. counting the supports of candidate itemsets over the database in less time, and
4. parallelizing the itemset generation.

In this sense, association rule algorithms generally differ on

1. the generation of the candidates,
2. counting of the support of a candidate itemset,
3. number of scans over the database, and
4. the data structures employed.

Readers are referred to [ZO98] for a theoretical discussion of the association rule discovery process.

2.3 Formal Problem Description

2.3.1 Definitions

Agrawal et al. define the problem of discovering association rules in databases in [AIS93, AS94].

Let $I = \{I_1, \ldots, I_m\}$ 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.

Definition 2.1 *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 2.2 *A transaction T satisfies an itemset X if $X \subseteq T$. The* **support** *of an itemset X in D, $support_D(X)$, is the number of transactions in D that satisfy X.*

Definition 2.3 *An itemset X is called a* **large itemset** *if the support of X in D exceeds a minimum support threshold explicitly declared by the user, and a* **small itemset** *otherwise.*

Definition 2.4 *The* **negative border** *of a set $S \subset P(R)$, closed with respect to the set inclusion relation, is the set of minimal itemsets $X \subset R$ not in S. The negative border of the set of large itemsets is the set of itemsets that are generated as a candidate but fail to qualify into the set of large itemsets.*

Definition 2.5 *An* **association rule** *is an implication of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, 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 $c = \frac{support_D(X \cup Y)}{support_D(X)}$. The rule $X \Rightarrow Y$ has* **support** *s in D if the fraction s of the transactions in D contain $X \cup Y$.*

Example 2.1 *Consider the example transaction database ETDB in Table 2.1. 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 a 2-itemset, and so on. $support_{ETDB}(A) = 4$ since 4 transactions include A in it. Let's assume that the minimum support (minsup) is taken as 40%. Then, $\{A, B, C, D, AB, AC, AD, BD, ABD\}$ are the set of large itemsets since*

TID	**Items**
100	A,B,C
200	B,D
300	A,C,D
400	A,B,D
500	A,B,D,E

Table 2.1: An Example Transaction Database

their support is greater than or equal to 2 ($40\% \times 5$)*, and the remaining ones are small itemsets. Let's 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* $\frac{support_{ETDB}(AD)}{support_{ETDB}(A)} \times 100 = \frac{3}{4} \times 100 = 75\%$*). On the other hand* $A \Rightarrow C$ *is not a valid association rule since its confidence is 50%.*

2.3.2 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 *minsup* and *minconf*, respectively. Formally, the problem is generating all association rules $X \Rightarrow Y$, where $support_D(X \cup Y) \geq minsup \times |D|$ and $\frac{support_D(X \cup Y)}{support_D(X)} \geq minconf$.

The problem of finding association rules can be decomposed into two parts [AIS93, AS94]:

Step 1: Generate all combinations of items with fractional transaction support (i.e., $\frac{support_D(X)}{|D|}$) above a certain threshold, called *minsup*.

Step 2: Use the large itemsets to generate association rules. For every large itemset l, find all non-empty subsets of l. For every such subset a, output a rule of the form $a \Rightarrow (l - a)$ if the ratio of $support_D(l)$ to $support_D(a)$ is at least *minconf*. If an itemset is found to be large 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.

```
    generate_rules(L);
1   for all large k-itemsets l_k, k ≥ 2, in L do
2   begin
3       H_1 = { consequents of rules from l_k with one item
4               in the consequent}
5       ap_genrules(l_k, H_1)
6   end

7   ap_genrules(l_k, H_m);
8   if k > m + 1 then
9   begin
10      H_{m+1} = apriori_gen(H_m)
11      for all h_{m+1} ∈ H_{m+1} do
12      begin
13          conf = support_D(l_k)/support_D(l_k − h_{m+1})
14          if conf ≥ minconf then
15              add (l_k − h_{m+1}) ⇒ h_{m+1} to the rule set
16          else
17              delete h_{m+1} from H_{m+1}
18      end
19      ap_genrules(l_k, H_{m+1})
20  end
```

Figure 2.1: Rule Generation Algorithm

The second subproblem is straightforward, and an efficient algorithm for extracting association rules from the set of large itemsets is presented in [AMS+96]. The algorithm uses some heuristics as follows:

1. If $a \Rightarrow (l - a)$ does not satisfy the minimum confidence condition, then for all non-empty subsets b of a, the rule $b \Rightarrow (l - b)$ does not satisfy the minimum confidence, either. Because, the support of a is less than or equal to the support of any subset b of a.

2. If $(l - a) \Rightarrow a$ satisfies the minimum confidence, then all rules of the form of $(l - b) \Rightarrow b$ must have confidence above the minimum confidence.

The rule generation algorithm is given in Figure 2.1. Firstly, for each large itemset l, all rules with one item in the consequent are generated. Then, the

```
    apriori_gen(L_{k-1});
1   C_k = ∅
2   for all itemsets X ∈ L_{k-1} and Y ∈ L_{k-1} do
3      if X_1 = Y_1 ∧ ··· ∧ X_{k-2} = Y_{k-2} ∧ X_{k-1} < Y_{k-1} then begin
4          C = X_1 X_2 ... X_{k-1} Y_{k-1}
5          add C to C_k
6      end
7   delete candidate itemsets in C_k whose any subset is not in L_{k-1}
```

Figure 2.2: Candidate Generation Algorithm

consequents of these rules are used to generate all possible rules with two items in the consequent, etc. The *apriori_gen* function in Figure 2.2 is used for this purpose.

On the other hand, discovering large itemsets is a non-trivial issue. The efficiency of an algorithm strongly depends on the size of the candidate set. The smaller the number of candidate itemsets is, the faster the algorithm will be. As the minimum support threshold decreases, the execution times of these algorithms increase because the algorithm needs to examine a larger number of candidates and larger number of itemsets.

2.4 Apriori and Partition Algorithms

In this section, we would like to present two association rule algorithms, namely *Apriori* [AS94, AMS$^+$96] and *Partition* [SON95]. The *Apriori* algorithm is a state of the art algorithm and most of the association rule algorithms are somehow variations of this algorithm. Thus, it is necessary to mention *Apriori* in detail for an introduction to association rule algorithms.

The *Apriori* algorithm works iteratively. It first finds the set of large 1-itemsets, and then set of 2-itemsets, and so on. The number of scans over the transaction database is as many as the length of the maximal itemset. *Apriori* is based on the following fact:

```
    Apriori()
1   L_1 = { large 1-itemsets}
2   k = 2
3   while L_{k-1} ≠ ∅ do
4   begin
5       C_k = apriori_gen(L_{k-1})            %See figure 2.2
6       for all transactions t in D do
7       begin
8           C^t = subset(C_k, t)
9           for all candidates c ∈ C^t do
10              c.count = c.count + 1
11      end
12      L_k = {c ∈ C_k | c.count ≥ minsup}
13      k = k + 1
14  end
```

Figure 2.3: Apriori Algorithm

"All subsets of a large itemset are also large."

This simple but powerful observation leads to the generation of a smaller candidate set using the set of large itemsets found in the previous iteration.

The *Apriori* algorithm presented in [AMS+96] is given in Figure 2.3. *Apriori* first scans the transaction database D in order to count the support of each item i in I, and determines the set of large 1-itemsets. Then, one iteration is performed for each of the computation of the set of 2-itemsets, 3-itemsets, and so on. The kth iteration consists of two steps:

1. Generate the candidate set C_k from the set of large $(k-1)$-itemsets, L_{k-1}.
2. Scan the database in order to compute the support of each candidate itemset in C_k

The candidate generation procedure computes the set of potentially large k-itemsets from the set of large $(k-1)$-itemsets. A new candidate k-itemset is generated from two large $(k-1)$-itemsets if their first $(k-2)$ items are the same (The new itemset contains the items in those two large itemsets in order).

In fact, the candidate set C_k is a superset of large k-itemsets. The candidate set is guaranteed to include all possible large k-itemsets because of the fact that all subsets of a large itemset are also large. Since all large itemsets in L_{k-1} are checked for contribution to a candidate itemset, the candidate set C_k is certainly a superset of large k-itemsets. The pruning step in *apriori_gen* function is necessary to reduce the size of the candidate set. For example, if L_{k-1} includes AB, AC, then a candidate ABC is generated in the join step of *apriori_gen*. However, it can not be a large itemset if L_{k-1} does not include BC, so it can be pruned from the candidate set. For efficiently finding whether a subset of a large itemset is small or not, a hash table is used for storing the large itemsets.

After the candidates are generated, their counts must be computed in order to determine which of them are large. The counting step of an association rule algorithm is very crucial in the efficiency of the algorithm, because the set of candidate itemsets may be possibly huge. *Apriori* handles this problem by employing a hash tree for storing the candidates. The *subset* function in *apriori_gen* is used to find the candidate itemsets contained in a transaction using this hash tree structure. For each transaction t in the transaction database D, the candidates contained in t are found using the hash tree, and then their counts are incremented. After examining all transactions in D, the set of candidate itemsets are checked to eliminate the small itemsets, and the ones that are large are inserted into L_k.

Example 2.2 *Consider again the transaction database given in Table 2.1. Suppose that the minimum support is set to 40%, i.e., 2 transactions. In the first pass,* $L_1 = \{A, B, C, D\}$*. The apriori_gen function computes* $C_2 = \{AB, AC, AD, BC, BD, CD\}$*. The database is scanned to find which of them are large, and it is found that* $L_2 = \{AB, AC, AD, BD\}$*. This set is used to compute* C_3*. In the join step* ABC*,* ABD*, and* ACD *are inserted into* C_3*. However,* ABC *can not be large because* BC *is not an element of* L_2*. Similarly,* ACD *can not be large because* CD *is not an element of* L_2*. Thus,* ABC *and* ACD *are pruned from the set of candidate itemsets. The database is scanned and it is found that* $L_3 = \{ABD\}$*.* C_4 *is found to be empty, and the algorithm terminates.*

The major drawback of the *Apriori* is the number of scans over the database. Especially for the huge databases, the I/O overhead incurred reduces the performance of the algorithm. In [AMS+96], two variations of *Apriori* were also presented to overcome this I/O cost. The *Apriori_TID* algorithm constructs an encoding of the candidate itemsets and uses this structure to count the support of itemsets instead of scanning the database. This encoded structure consists of elements of the form $< TID, \{X_k\} >$ where each X_k is a large k-itemset. In other words, the original database is converted into a new table where each row is formed of a transaction identifier and the large itemsets contained in that transaction. The counting step is over this structure instead of the database. After identifying new large K-itemsets, a new encoded structure is constructed. In subsequent passes, the size of each entry decreases with respect to the original transactions and the size of the total database decreases with respect to the original database. *Apriori_TID* is very efficient in the later iterations but the new encoded structure may require more space than the original database in the first two iterations.

To increase the performance of *Apriori_TID*, a new algorithm, namely *Apriori_Hybrid*, was proposed in [AMS+96]. This algorithm uses *Apriori* in the initial passes, and then switches to *Apriori_TID* when the size of the encoded structure fits into main memory. In this sense, it takes benefits of both *Apriori* and *Apriori_TID* to efficiently mine association rules.

The three algorithms mentioned above scale linearly with the number of transactions and the average transaction size.

The *UWEP* algorithm is based on the framework of *Partition* algorithm [SON95]. Thus, we would like to describe this algorithm in detail. The major advantage of *Partition* algorithm is scanning the database exactly twice to compute the large itemsets by means of constructing a transaction list for each large itemset. Initially, the database is partitioned into n overlapping partitions, such that each partition fits into main memory. By scanning the database once, all locally large itemsets are found in each partition, i.e., itemsets that are large in that partition. Before the second scan, all locally large itemsets are combined to form a global candidate set. In the second scan of the database, each global candidate itemset is counted in each partition and the

global support (support in the whole database) of each candidate is computed. Those that are found to be large are inserted into the set of large itemsets.

The correctness of the *Partition* algorithm is based on the following fact:

"A large itemset must be large in at least one of the partitions."

The same argument is applied when updating the large itemsets, and a formal proof can be found in [SON95].

Two scans over the database are sufficient in *Partition*. This is due to the creation of *tidlist* structures while determining large 1-itemsets. A *tidlist* for an item X is an array of transaction identifiers in which the item is present. For each item, a *tidlist* is constructed in the first iteration of the algorithm, and the support of an itemset is simply the length of its *tidlist*. The support of longer itemsets are computed by intersecting the *tidlists* of the items contained in the itemset. Moreover, the support of a candidate k-itemset can be obtained by intersecting the *tidlists* of the large $(k-1)$-itemsets that were used to generate that candidate itemset. Since the transactions are assumed to be sorted, and the database is scanned sequentially, the intersection operation may be performed efficiently by a sort-merge join algorithm.

For higher minimum supports, *Apriori* performs better than *Partition* because of the extra cost of creating *tidlists*. On the other hand, when the minimum support is set to low values and the number of candidate and large itemsets tend to be huge, *Partition* performs much better than *Apriori*. This is due to the techniques in counting the support of itemsets and fewer number of scans over the database. One final remark is that the performance of the *Partition* algorithm strongly depends on the size of partitions, and the distribution of transactions in each partition. If the set of global candidate itemsets tends to be very huge, the performance may degrade.

2.5 Analysis of Algorithms

AIS [AIS93] is the first study on the association rules. It works iteratively and computes large k-itemsets in the kth iteration. Thus, it makes as many passes as the length of maximal itemset over the database. The candidates are generated and counted at the same time. Once the set of large k-itemsets is determined, the database is scanned to identify large $(k+1)$-itemsets. By processing each transaction sequentially, the large itemsets contained in that transaction are extended with the other items in the transaction, and the support of the new candidate is incremented. In this sense, *AIS* generates too many candidates which turn out to be small in the database, causing it to waste too much effort.

Apriori [AS94, AMS+96] also works iteratively and it makes as many scans as the length of maximal itemset over the database. The candidate k-itemsets are generated from the set of large $(k-1)$-itemsets by means of join and pruning operations. Then the itemsets in the candidate set are counted by scanning the database. *Apriori* forms the foundation of the later algorithms on association rules.

Apriori_TID and *Apriori_Hybrid* [AS94, AMS+96] have the similar ideas in *Apriori*. The former uses an encoded structure which stores the itemsets that exist in each transaction. In other words, the items in the transaction are converted to an itemset representation. The candidates are generated as in *Apriori* but they are counted over the constructed encoding. The latter algorithm tries to get benefits of both *Apriori* and *Apriori_TID* by using *Apriori* in the initial passes and switching to the other in later iterations. Both algorithms make as many passes as the length of maximal itemset.

Offline Candidate Determination (OCD) [MTV94] is very similar to *Apriori*. It also makes as many passes as the length of the maximal itemset. It differs from *Apriori* in the candidate generation algorithm. Both generate the candidates from the set of L_{k-1} but *OCD* generates a new candidate from two large $(k-1)$-itemsets if they have $k-2$ items in common while *Apriori* generates it if $k-2$ items of two large $(k-1)$-itemsets are same. The candidates are counted after generating the candidates and by scanning the database.

Set Oriented Mining (SETM) [HS95] uses SQL commands to mine association rules. The number of scans over the database is equal to the length of maximal itemset. The candidate set C_k is generated by the natural join of L_{k-1} with L_1 in the attribute TID, and it is implemented by a merge-sort join. The candidates are counted using SQL commands. $SETM$ generates too many candidates with respect to *Apriori* and is less efficient.

Readers are referred to [HP96] for the evaluation of the algorithms above, and their cost of computation. Lower and upper bounds for their computational complexity are provided in this paper.

The motivation behind *Dynamic Hashing and Pruning (DHP)* [PCY95a] is the attempt to reduce the size of candidate 2-itemsets. Park et al. realized that the dominant factor in an association rule algorithm is the generation and counting of candidate 2-itemsets. It first finds the set of large 1-itemsets and creates a hash table for the candidate 2-itemsets. In the later iterations, it generates the candidates from the set of L_{k-1} by incorporating the knowledge in the hash table to the algorithm. An itemset is put into the candidate set if and only if its subsets are in L_{k-1} and it is hashed into a hash entry whose value is at least the minimum support. It counts the supports of candidate itemsets by scanning the database. It also creates a hash table for the candidate $(k+1)$-itemsets in this scan. DHP constantly performs well for low level minimum supports and executes better in the later iterations, especially in the second iteration. In [PCY97], sampling techniques are incorporated into the framework of DHP. With the advantage of controlled sampling, the proposed algorithms produce rules with high accuracy.

As we pointed out in Section 2.4, *Partition* [SON95] is the best algorithm in terms of scans over the database. It makes at most two scans over the database by means of partitioning the database into n partitions, finding large itemsets in each partition, and determining which of them are large in the whole database. It executes iteratively while finding large itemsets in a particular partition, but the number of scans is limited to one by using a *tidlist* structure we mentioned previously. It counts the supports of the candidates over the created *tidlists* instead of the database. The major advantages of *Partition* are the reduction in I/O cost, and usage of main memory while computing large itemsets.

MONET System [HKMT95] discovers association rules by using only a general-purpose database management system and the operations of relational algebra, *union* and *intersection* operations. The database is stored as a set of items (columns), where TIDs of the transactions that contain the item are enumerated in this column. The candidates are generated by the method employed in *Apriori*. It does not scan the whole database to count the supports of itemsets, but intersects the columns of items contained in the itemset and finds its length instead. This approach is in fact the same as the *tidlist* structure employed in *Partition*. The performance of the system strongly depends on the implementation of the *union* and *intersection* operations.

In [Toi96], Toivonen uses sampling to discover the association rules. The algorithm picks a random sample and computes the large itemsets with a lower minimum support (in order not to miss any large itemset). Then, it verifies this set of large itemsets and its negative border against the entire database. If no itemset in the negative border is large in the entire database, this approach finds the set of large itemsets in one pass over the database. Otherwise, it requires an additional scan over the database. The candidate are generated and counted as in *Apriori*. The reduced I/O cost is the major advantage of the algorithm. Zaki et al. [ZPLO97] also analyze the effects of sampling on the discovery of association rules, and propose efficient and optimal strategies for choosing a sample size.

Dynamic Itemset Counting (DIC) [BMUT97] attempts to reduce the number of scans over the database. As soon as it suspects that a k-itemset may be large, it begins to count its support without waiting the kth iteration. Thus, the number of scans is generally smaller than the length of the maximal itemset. The database is logically partitioned into sets of size of M, and database is processed sequentially by reading chunks of size of M. A new candidate is added to the candidate set when all its subsets are large at that point. In other words, it does not wait for the kth iteration to generate candidates, but does that in every M transactions read. The candidates up to that point are counted while reading M transactions. The experiments yielded that DIC generally makes two passes if the data is homogeneously distributed in the database and M is suitably chosen.

Four algorithms in [ZPOL97a], *Eclat*, *MaxEclat*, *Clique*, *MaxClique*, make only one pass over the database. They use one of the itemset clustering schemes (*equivalence classes* or *maximal hypergraphs*) to generate potential maximal large itemsets (maximal candidates). Each cluster induces a sub-lattice and this lattice is traversed bottom-up or hybrid top-down/bottom-up to generate all frequent itemsets and all maximal frequent itemsets, respectively. Clusters are processed one by one. The *tidlist* structure in *Partition* is employed in these algorithms, and the supports of candidate itemsets are computed by a simple *intersection* operation. They have low memory utilization since only frequent k-itemsets in the processed cluster must be kept in main memory at that time.

Max–Miner [Bay98] attempts to *look ahead* in order to quickly identify longer itemsets, and prune their subsets as soon as possible. It scales linearly on the number of frequent patterns and the size of the database irrespective of the length of longest pattern. The candidate generation and counting processes are similar to *Apriori*, and it requires at most N passes where N is the length of maximal itemset. *Max–Miner* especially performs well when the size of large itemsets increases, but the number of scans is a drawback.

Carma [Hid99] is a recently proposed algorithm for computing association rules online, which requires exactly two passes over the database. In the first scan of the database, a lattice of potentially large itemsets with respect to the scanned transactions is constructed. The user is free to change the support threshold in the first scan. In a second scan, the algorithm determines the support of each itemset in the lattice, and removes the itemsets that are small with respect to the whole set of transactions. While the lattice is constructed, a new candidate is inserted or removed according to the upper and lower bound values associated with each itemset. The counting process takes place in the second scan.

Aggarwal et al. [AY98c] uses the preprocess-once-query-many paradigm of OLAP in order to generate the rules quickly, again by using a lattice structure to pre-store itemsets. The algorithm is proportional to the size of the rule set.

Table 2.2 summarizes the sequential association rule algorithms in terms of

Algorithm	**Number of scans**	**Candidate Generation**	**Candidate Counting**
AIS	N	Extend L_{k-1} with items in each transaction	Scan database
Apriori	N	Join L_{k-1} with L_{k-1}	Scan database
Apriori_TID	N	Join L_{k-1} with L_{k-1}	Scan the encoded itemset representation
OCD	N	Join L_{k-1} with L_{k-1}	Scan the database
SETM	N	Join L_{k-1} with L_1	*SQL* commands
DHP	N	Join L_{k-1} with L_{k-1} and check its hash entry	Scan database
Partition	2	Join L_{k-1} with L_{k-1}	Intersect *tidlists*
MONET	N	Join L_{k-1} with L_{k-1}	Intersect columns
Sampling	≤ 2	Join L_{k-1} with L_{k-1}	Scan database
DIC	$\leq N$ (generally 2)	Check all its subsets whether they are large	Scan database
MaxClique	1	Examine maximal frequent itemsets	Intersect *tidlists*
Max–Miner	N	Join L_{k-1} with L_{k-1}	Scan database
Carma	2	According to upper and lower bounds	Scan database

Table 2.2: An Overview of Association Rule Algorithms

number of scans over the database, methods used to generate and count the candidates. N refers to the length of maximal itemset in the column *Number of Scans*.

As well as the sequential algorithms above, a number of parallel and distributed algorithms for discovering large itemsets were presented. *Candidate Distribution, Data Distribution*, and *Count Distribution* [AS96] are the parallelized versions of *Apriori*, and *Count Distribution* was shown to be superior to the others. *DMA* [CNFF96] attempted to parallelize the *Partition* algorithm, and *PDM* [PCY95b] is a parallelization of *DHP*. Finally, *Par–Eclat*, *Par–MaxEclat*, *Par–Clique*, and *Par–MaxClique* [ZPOL97b] are the parallel versions of the four algorithms in [ZPOL97a].

2.6 Variations of Association Rules

As we pointed out in Section 2.2, the association rule algorithms rely on the existence or absence of items in a transaction. They do not take the other properties of attributes, such as quantity, weight, hierarchical information, into account. In this section, we will briefly mention some variations of association rules, which are also based on the generation of itemsets.

2.6.1 Association Rules with Hierarchy

In most cases, taxonomies (is–a hierarchies) over the items are available. Such a taxonomy, for instance, "jackets and ski pants are outer wear which is a type of cloth, and shoes and hiking boots are footwear". *Generalized (multiple–level) association rules* [SA95] aim to find association rules between items in different levels of a taxonomy as well as the rules between items in the same level. An example of a generalized association rules states that "jackets $\Rightarrow$ footwear". A straightforward but not efficient solution is to generate a new column for the levels of hierarchy that are not in the original database of transactions (generally the levels except the bottom level). Efficient algorithms, which incorporate hierarchical information into the algorithm, were proposed in [SA95, HF95]. An object-oriented approach is proposed in [FL96] and SQL queries are used to find multiple–level association rules in [TS98]. Finally, flexible multiple–level association rules are discussed in [SS98b].

2.6.2 Constrained Association Rules

In real life, end-users are generally interested in a small subset of the association rules extracted from a database. For instance, a user may want to see the associations only between some items. In [SVA97], *constrained association rules*, which handles the constraints that are boolean expressions over the presence or absence of some items, were proposed. One example of a constraints that can be handled is $(Jacket \wedge Shoes) \vee (descendants(Clothes) \wedge \neg ancestors(Hikingboots))$, which expresses the constraint on the rules that

either (a) contain both jackets and shoes, or (b) contain descendants of clothes and do not contain ancestors of hiking boots. Instead of discovering all rules and pruning some of them with respect to the given constraints, they incorporate the constraints into the association rule algorithm. In [NLHP98, LNHP99], *constrained association queries* are introduced to handle more complicated constraints in association rule discovery. One example of a constrained association query is $\{(S_1, S_2)|S_1.Type \subseteq \{Snacks\} \wedge S_2.Type \subseteq \{beers\} \wedge max(S_1, Price) \leq min(S_2, Price)\}$, which finds pairs of sets of cheaper snack items and sets of more expensive beer items. In a recent study, Bayardo et al. [BAG99] push constraints on the minimum support, minimum confidence and a new constraint that guarantees every rule has a predictive advantage over its simplifications.

2.6.3 Quantitative Association Rules

The original association rule problem handles only the case for boolean attributes, i.e., an item exists or not. For handling the quantity of numerical attributes and categorical attributes that can take more than two values, *quantitative association rules* were proposed in [SA96a]. An example quantitative association rule is $\langle Age : 30..39\rangle \wedge \langle Married : Yes\rangle \Rightarrow \langle NumCars : 2\rangle(40\%, 90\%)$, which means "In 40% of the total transactions, 90% of the people whose age is between 30 and 40 and who are married have two cars". In [SA96a], an efficient algorithm, which attempts to divide the values of quantitative and categorical attributes into ranges which maximize the strength of the association rules, was proposed.

The important point in computation of quantitative association rules is how to partition the values of a quantitative attribute into non-overlapping partitions optimally. Fukuda et al. [FMMT96] introduced *optimized association rules*, which tries to find the partitioning of values of numerical attributes to maximize the support or confidence. The same concept was also investigated in [RS98] for numerical and categorical attributes. Wang et al. [WTL98] proposed an interestingness–based interval merger for combining different intervals

to one in order to maximize the interestingness of a rule. In another study related to numerical attributes [KFW98], *fuzzy association rules* were proposed. A fuzzy association rule is of the form of "If X is A, then Y is B" where X, Y are sets of attributes and A, B are fuzzy sets which describe X and Y respectively. It is assumed that the fuzzy sets for each attribute are provided as input. In [FWS+98], a clustering schema is employed to extract those fuzzy sets.

2.6.4 Sequential Patterns

The association rules aim at discovering co-occurrences at a certain time. With the storage of data over a long time period and development of temporal databases, the discovery of *sequential patterns* became an important issue. An example of a sequential pattern is "Customers typically rent "Star Wars" then "Empire Strikes Back" and then "Return of the Jedi"." [AS95]. The items in a sequential pattern need not be consecutive but only in that order. Three algorithms were proposed in [AS95] to extract sequential patterns in a transaction database. This work was extended to handle sliding windows and hierarchical information in [SA96b]. Mannila et al. [MTV95, MT96] discovers the frequent *episodes* (a collection of events in a certain pattern), and generalized episodes (episodes that satisfy certain conditions) in a sequence of data. Guralnik [GWS98] and Das et al. [DLM+98] also propose efficient algorithms for discovering frequent episodes.

2.6.5 Periodical Rules

In a sequence of data, association rules may reveal periodical properties. Moreover, some of the rules may have enough support in a smaller time period even it does not have enough support in the global database. Ozden et al. [ORS98] introduce *cyclic association rules*, which are the rules that have the specified confidence and support in regular time intervals. One such rule states that "People buy newspapers along with milk every Sunday". Instead of finding the rules at each time point, and then attempting to generate periodical rules

from those set of rules, two algorithms that incorporate some heuristics to the algorithm were proposed in [ORS98]. These algorithms handle only the case where the rules are repeated in every t time points. In [RMS98], this study is extended to find the *calendric association rules* which follow the patterns in a user-specified calendar. Moreover, the algorithms for extracting rules in any calendar were proposed. These studies are based on the full periodicity, i.e., the rule must be valid in every time point in the pattern. Han et al. [HDY99] drop this restriction and attempt to find *partial periodic patterns*, which is a looser kind of periodicity.

2.6.6 Weighted Association Rules

All items in the data are treated with the same importance in previous association rule algorithms. Cai et al. [CFCK98] generalized this to the case where items are assigned weights to reflect their importance. The weights may correspond to special promotions on some products or their price. They define *weighted support* of an itemset and association rule. The previous methods are not applicable by changing only the computation of support because the bottom-up property of itemsets (all subsets of a large itemset are also large) is not valid. Thus, they propose a new algorithm in [CFCK98].

2.6.7 Negative Association Rules

Savasere et al. [SON98] investigate the *negative association rules* instead of positive associations between items. One such rule is "Most of the people buy frozen food do not buy vegetables". The straightforward solution is to set minimum support and confidence as low as possible. However, this solution yields many and uninteresting negative association rules. The idea is to extract the combinations of items where a high degree of positive association is expected but the actual support is significantly smaller than what is expected. An efficient algorithm was presented in [SON95].

2.6.8 Ratio Rules

While association rules try to discover the co-occurrences of items, *ratio rules* introduced in [KLKF98] try to find correlations between the quantities or prices of different items. An example of a ratio rule is "Customers typically spend 1:2:5 dollars on bread: milk: butter". A one–pass algorithm is presented to find ratio rules in [KLKF98]. The proposed method attempts to determine how good the derived rules are by introducing *guessing error*. Ratio rules can be used for estimating the missing values, even if multiple values are missing simultaneously.

2.7 A Criticism on Large Itemset Framework

Most of the association rule algorithms work in a bottom-up fashion, i.e., first the set of large 1-itemsets is found, then set of large 2-itemsets is generated and this process is repeated until a set of large itemsets of a certain length is empty. The general heuristic is the fact that "All subsets of a large itemset are also large". Thus, before an itemset is found large, all of its subsets must be verified against the database. When the size of large itemsets existing in the database tends to increase, this process may yield a bottleneck. For instance in USA census data, the size of the large itemsets may increase up to 40, and the computation of those itemsets require all of its 2^{40} subsets must be firstly validated. Therefore, most of the algorithms, except *Max–Miner* [Bay98], perform poorly when the cardinality of maximal large itemset is large (when greater than 10).

Aggarwal et al. [AY98a, AY98b] pinpoint to some of the drawbacks of itemset generation in the computation of association rules. They claim that the *minimum support* criterion in the computation of large itemsets may reveal wrong association rules. Readers are suggested to see the example in [AY98a]. They also show that the *minimum support* framework is not suitable in dense data sets in which the number of large itemsets tend to be very huge. Negative association rules and the datasets in which different attributes have varying densities are shown to be good examples of dense data sets. They propose

another concept called *collective strength* for more efficient and reliable computation of large itemsets. The collective strength of an itemset is defined to be a number between 0 and ∞, where 0 indicates a perfect negative correlation and ∞ indicates a perfect positive correlation.

The weaknesses of the minimum support-confidence framework were also noticed by Brin et al. In [BMUT97], implication strength of itemsets was shown to be more accurate than minimum support-confidence. The implication strength of a rule is a number between 0 and ∞, where a value of 1 indicates the rule is as strong as it is expected under statistical assumptions, and a value greater than 1 indicates a presence greater than expected. In [BMS97], the association rules were generalized to correlation rules, and a more accurate measure, *chi–square test*, was proposed to evaluate the strength of an itemset.

2.8 A Discussion on Association Rules

As we pointed out earlier, the major problem in data mining, so in association rules, is that there is no certain criteria to decide which of the discovered rules are really interesting. The interestingness of a rule is generally dependent on the user and on the application. Different measures for interestingness were proposed in the literature, but none of them is applicable in every application. One common point in all of them is that interestingness is directly related to the expectancy of the rule. In other words, a pattern is interesting if it is not known prior to the data mining process or contradicts the beliefs of the user [ST95, ST96b].

Mainly, there are two strategies to increase the interestingness of the discovered rules:

1. Incorporate some heuristics to the algorithm
2. Find a set of rules and prune some of them according to some interest measure after the discovery process

In the first strategy, either constraints are pushed to the algorithm [NLHP98]

or some sort of interest measures are used to eliminate the uninteresting rules during the discovery of association rules. The collective strength [AY98a], implication strength [BMUT97], chi-square test [BMS97] or expected value [AS95] are some of those heuristics pushed into the algorithm. The incorporation of some heuristics into the algorithm results in significant improvement in the computation of the rules, thus it is a preferable method.

The second strategy is necessary, even if the first strategy is also applied in the discovery process, since the interestingness varies from user to user and from application to application. The general methodology is to remove the rules which are known in advance, which contain uninteresting attributes or which are redundant. Brin et al. [BMUT97] suggest to remove the rules that are transitively implied and that are not minimal. In [TKR+95], a *rule cover*, which tries to minimize the antecedents of the rules, is defined to eliminate redundant rules. In the same work a clustering algorithm is also applied on the rules to help the user understand the patterns more easily. The usage of *templates* were proposed in [KMR+94]. Templates specify the attributes that can occur in the antecedent and consequent of a rule, and they can be restrictive or inclusive. This is in fact the same as constrained association rules, but less efficient than the case where the constraints are incorporated into the algorithm.

Finally, we would like to mention the user interaction in discovery of association rules. Although there are efficient algorithms for extracting association rules, the user interaction is at the minimum level. Generally, the user only sets the minimum support and confidence thresholds, and then waits for the results until all the rules are found. Online association rule mining [AY98c, Hid99] tries to avoid this situation by allowing the user to change the thresholds during the discovery process. A semi-automatic miner which is activated every time the data exhibits certain changes was proposed in [ST96a]. In [NLHP98, LNHP99], a two-phase architecture which allows the user to control the search process was proposed. However, the discovery process is still a black-box, which do not allow the user much to be involved in the search process. Data mining is a problem which do not have only one answer, therefore we believe that more user–controlled systems are needed in data mining applications.

Chapter 3

Updating Large Itemsets

Maintenance of association rules is an important problem. When new transactions are added to the set of old transaction database, how can we update the association rules already discovered in the set of old transactions efficiently? Naturally, when new transactions are added to a database, some of the existing frequent patterns may disappear whereas new frequent patterns that did not exist before may also emerge. The straightforward solution is to re-run an algorithm, say *Apriori* [AS94], on the set of whole transactions, i.e., old transactions plus new transactions. However, this process is not efficient since it ignores the previously discovered rules, and repeats all the work done previously. Therefore, algorithms for efficiently updating the association rules were proposed in [CHNW96, CLK97, OS98, SS98a, TBAR97]. These algorithms take the set of association rules in the old database into account, and use this knowledge 1) to remove itemsets that do not exist in the updated database, and 2) to add new rules which were not in the set of old transactions but implied in the updated database. Particularly, when the size of old transactions is large, these algorithms discover the new set of association rules much faster than by re-running an algorithm over the whole database.

In this thesis, we propose an algorithm called $UWEP$ (**U**pdate **W**ith **E**arly **P**runing) that follows the approaches of FUP_2 [CLK97] and *Partition Update* [OS98] algorithms. It works iteratively on the new set of transactions, like the previous algorithms. The advantages of $UWEP$ are that it scans the

existing database at most once and new database exactly once, and it generates and counts the minimum number of candidates in order to determine the set of new large itemsets. Similar to [SON95], in one scan of the database, it creates a *tidlist* for each item in the database, and uses these structures in order to compute the support of supersets of that item. Moreover, it prunes an itemset that will become small from the set of generated candidates as early as possible by a look-ahead pruning. In other words, it does not wait for the kth iteration for pruning a small k-itemset. This look-ahead pruning results in a much smaller number of candidates in the set of new transactions. Another reason for generating a smaller candidate set is the fact that $UWEP$ promotes a candidate itemset to the set of large itemsets only if it is large both in the new set of transactions and in the whole database. This feature yields a much smaller candidate set when some of the old large itemsets are eliminated due to their absence in the new set of transactions, and this can be done without scanning the old database.

This chapter is organized as follows. In Section 3.1, a formal description of updating large itemsets is presented. The related algorithms are discussed in Section 3.2. Section 3.3 presents the $UWEP$ algorithm. Section 3.4 describes the data structures used in $UWEP$. Section 3.5 gives an example execution of the algorithm and its performance comparison with the other algorithms. In Section 3.6, we prove the correctness of the $UWEP$ algorithm, and that it generates and counts a minimum number of candidates. Details of the experiments and performance results on synthetic data are provided in Section 3.7, and a theoretical discussion of the performance comparison of the update algorithms is presented in Section 3.8.

3.1 Formal Problem Description

Table 3.1 summarizes the notation used in the remainder of this chapter. Given $DB, db, |DB|, |db|$, $minsup$ and L_{DB}, the problem of updating association rules is to find the set L_{DB+db} of large itemsets in $DB + db$.

Notation	Definition
DB	The set of old transactions
db	The set of new transactions
$DB + db$	The total set of transactions
$\|A\|$	The number of transactions in the transaction database A
$minsup$	Minimum support threshold
X	A set of items (i.e., one itemset)
$support_A(X)$	Support of X in the set of transactions A
$tidlist_A(X)$	Transaction list of X in the set of transactions A
C_A^k	Set of candidate k-itemsets in a set of transactions A
L_A^k	Set of large k-itemsets in a set of transactions A
$PruneSet$	Set of large itemsets in DB that have 0 support in db
$Unchecked$	Set of large k-itemsets in DB that are not counted in db

Table 3.1: Notation Used in Algorithm $UWEP$

3.2 Previous Algorithms

Updating association rules was first introduced in [CHNW96]. The FUP algorithm proposed by Cheung et al. [CHNW96] works iteratively and its framework is similar to *Apriori* [AS94] and DHP [PCY95a]. At the kth iteration it performs three operations as follows:

1. Scan db for all $X \in L_{DB}^k$. If X's support in the updated database is smaller than the minimum support threshold, remove it from consideration (It is a loser). Otherwise, put it into the set of large itemsets in the updated database.

2. In the same scan, count the supports of itemsets that are in the candidate set of db but not in the set of large itemsets of DB. If the support of an itemset in db is smaller than the minimum support threshold, remove it from the candidate set of db.

3. For the remaining itemsets in the candidate set of db, count their support in DB and decide which of them will be placed in the set of large itemsets of the updated database.

Initially, the candidate set of 1-itemsets of db is the set of items which exist in at least one transaction in db. At the end of the kth iteration, the

new set of candidates are computed from the set of large k-itemsets in the updated database. There are three optimizations employed in FUP, two of which are based on the reduction of transactions (i.e., if X is a small itemset in D, remove X from the transactions in D). The other is the computation of an upper bound value for the support of an itemset, and deciding whether the itemset is small without scanning the database. Formally, an upper bound value for the support of an itemset X is defined as

$$b_X = min(support(Y)) \text{ for all } Y \subset X \text{ and } |Y| = |X| - 1.$$

FUP_2 [CLK97] is a generalization of the FUP algorithm that handles insertions to and deletions from an existing set of transactions. In the case of deletion, the set of candidates are pruned using the upper bound values of the candidate itemsets so that only possibly large itemsets are counted through the scan of the database. The algorithms FUP and FUP_2 scan DB and db as many times as the length of the maximal large itemset in the updated database, and generates a large number of candidates in db since it generates C_{db}^k from L_{DB+db}^{k-1}.

In [SS98a, TBAR97], the concept of *negative border*, that was introduced in [Toi96], is used to compute the new set of large itemsets in the updated database. The negative border consists of all itemsets that were candidates but did not have enough support while computing large itemsets in DB. In other words,

$$NBD(L_k) = C_k - L_k.$$

They assume that the negative border of the set of large itemsets in DB and their counts in DB are also available, and use this knowledge to reduce the number of scans over DB. In [TBAR97], the set of large itemsets in db is first computed by a scan of db. In the same scan, the supports of all itemsets in L_{DB} and $NBD(L_{DB})$ over db are also counted. Then, all itemsets that are large both in DB and db are promoted to the set of large itemsets in $DB + db$. If an itemset X is large in db but small in DB, X and its supersets are checked against DB using the negative border of L_{DB}. If such an item is promoted to the set of large itemsets in $DB+db$, the negative border is computed again, and this process is repeated until there is no change in the negative border. This

algorithm scans db as many times as the length of maximal large itemset in db and DB at most once. However, recomputing negative border again and again reduces its performance. The approach in [SS98a] is very similar to the one in [TBAR97]. It first counts the supports of itemsets in L_{DB} and $NBD(L_{DB})$ over db. If any of the itemsets in the negative border is found to be large in db, then it computes L_{db} and validates those against DB by scanning DB once. Its major advantage is that it does not scan DB if there is no new itemset in db.

The most recent work [OS98] uses the framework in [SON95], and assume that the set of large itemsets in the old database is available. Then, it computes the large itemsets in db by using one of the existing algorithms, namely *Partition* [SON95]. Its final step is counting the support of large itemsets in DB against db, and vice versa. This requires one scan over DB and one scan over db using the Partition [SON95]. The only difference between this algorithm and re-running Partition algorithm is that it does not find large itemsets in DB but assumes that it is available in a file.

3.3 Update with Early Pruning ($UWEP$)

In this section, we will explain how our algorithm works, and the optimizations it employs. The algorithm $UWEP$ is presented in Figure 3.1. Inputs to the algorithm are DB, db, L_{DB} (along with their supports in DB), $|DB|$, $|db|$, and $minsup$. The output of the algorithm is L_{DB+db}, the set of large itemsets in $DB + db$.

We can break down the algorithm $UWEP$ into five steps as identified below.

1. Counting 1-itemsets in db and creating a *tidlist* for each item in db

2. Checking the large itemsets in DB whose items are absent in db and their supersets for largeness in $DB + db$

3. Checking the large itemsets in db for largeness in $DB + db$

UWEP$(DB, db, L_{DB}, |DB|, |db|, minsup)$;
1 C^1_{db} = all 1-itemsets in db whose support is greater than 0
2 $PruneSet = L^1_{DB} - C^1_{db}$
3 $initial_pruning(PruneSet)$ %See Figure 3.2
4 $k = 1$
5 **while** $C^k_{db} \neq \emptyset$ and $L^k_{DB} \neq \emptyset$ **do begin**
6 $\quad Unchecked = L^k_{DB}$
7 $\quad$ **for** all $X \in C^k_{db}$ **do**
8 $\quad\quad$ **if** X is small in db and X is large in DB **then**
9 $\quad\quad\quad$ remove X from $Unchecked$
10 $\quad\quad\quad$ **if** X is small in $DB + db$ **then**
11 $\quad\quad\quad\quad$ remove all supersets of X from L_{DB}
12 $\quad\quad\quad$ **else**
13 $\quad\quad\quad\quad$ add X to L_{DB+db}
14 $\quad\quad$ **end**
15 $\quad\quad$ **else if** X is large both in db and DB **then begin**
16 $\quad\quad\quad$ remove X from $Unchecked$
17 $\quad\quad\quad$ add X to L_{DB+db} and L^k_{db}
18 $\quad\quad$ **end**
19 $\quad\quad$ **else if** X is large in db but small in DB **then begin**
20 $\quad\quad\quad$ find $support_{DB}(X)$ using $tidlists$
21 $\quad\quad\quad$ **if** X is large in $DB + db$ **then**
22 $\quad\quad\quad\quad$ add X to L_{DB+db} and L^k_{db}
23 $\quad\quad$ **end**
24 $\quad$ **for** all $X \in Unchecked$ **do begin**
25 $\quad\quad$ find $support_{db}(X)$ using $tidlists$
26 $\quad\quad$ **if** X is small in $DB + db$ **then**
27 $\quad\quad\quad$ remove all supersets of X from L_{DB}
28 $\quad\quad$ **if** X is large in $DB + db$ **then**
29 $\quad\quad\quad$ add X to L_{DB+db}
30 $\quad$ **end**
31 $\quad k = k + 1$
32 $\quad C^k_{db} = generate_candidate(L^{k-1}_{db})$ %See Figure 3.3
33 **end**

Figure 3.1: Update of Frequent Itemsets

```
   initial_pruning(PruneSet);
1  while PruneSet ≠ ∅ do begin
2      X = first element of PruneSet
3      if X is small in DB + db then
4          remove X and all its supersets from L_DB and PruneSet
5      else
6      begin
7          add the supersets of X in L_DB to the PruneSet
8          add X to L_DB+db and remove X from L_DB
9      end
10     remove X from PruneSet
11 end
```

Figure 3.2: Initial Pruning Algorithm

4. Checking the large itemsets in DB that are not counted over db for largeness in $DB + db$

5. Generating the candidate set from the set of large itemsets obtained at the previous step.

In the first step of the $UWEP$ algorithm (line 1 in Figure 3.1), we count the support of 1-itemsets and create a *tidlist* for each 1-itemset in db. The idea of using *tidlists* was first discussed in [SON95] in order to count the support of candidate k-itemsets. A *tidlist* for an itemset X is an ordered list (ascending or descending) of the transaction identifiers (TID) of the transactions in which the items are present. The support of an itemset X is the length of the corresponding *tidlist*. It is assumed that the transactions are sorted according to TIDs and thus the created *tidlists* are also sorted in the same order of TIDs.

The second part of the algorithm (procedure *initial_pruning* in Figure 3.2) deals with the 1-itemsets whose support is 0 in db but large in DB. In this case, for an itemset X, it is by definition true that $support_{DB+db}(X) = support_{DB}(X)$. If X was previously small in DB, then it is also small in $DB + db$ since its support has not changed and the number of total transactions has increased. On the other hand, if X is large in DB, we have to check whether $support_{DB}(X) \geq minsup \times |DB + db|$ or not. The itemset X could be large or small in the updated database, and we examine each case below.

In the following, we will introduce three lemmas that are useful in pruning the candidate itemsets. Their proofs can be found in [AS94, CHNW96, CLK97, TBAR97].

Lemma 3.1 *All supersets of a small itemset X in a database D are also small in D.*

Proof. Let Y be a superset of X, i.e., $X \subset Y$. If a transaction contains Y, then that transaction certainly contains X in it. Then, $support_{DB+db}(X) \geq support_{DB+db}(Y)$. Hence, if $support_{DB+db}(X) < minsup \times |DB + db|$, then $support_{DB+db}(Y) < minsup \times |DB + db|$, which means Y is small in $DB + db$. □

Now suppose that X is small in the updated database. Then, by Lemma 3.1, any superset of X must also be small in the updated database. $UWEP$ differs from the previous algorithms [CHNW96, CLK97] at this point, by pruning all supersets of an itemset from the set of large itemsets in DB as soon as it is established to be small. In the previous algorithms, a k-itemset is only checked in the kth iteration, but $UWEP$ does not wait until the kth iteration in order to prune the supersets of an itemset in L_{DB} that are small in L_{DB+db}.

Definition 3.1 *Let X be a k-itemset which contains items $I_1, \ldots, I_k$. An immediate superset of X is a $(k+1)$-itemset which contains the k items in X and an additional item I_{k+1}.*

Now, suppose that X is large in the updated database. Then, we add all immediate supersets of X in L_{DB} to the $PruneSet$, which holds the itemsets that must be checked before checking the itemsets in C^1_{db}. Then, for each element in the $PruneSet$, we check whether its support exceeds the minimum support threshold. The operations of pruning and adding immediate supersets are repeated for each itemset in the $PruneSet$. So, all itemsets in L_{DB} that contain a non-existing item in db are removed from L_{DB}, and the ones that are large are added to L_{DB+db} before advancing to the first iteration. This pre-pruning step is particularly useful when the data skewness is present in the

set of transactions. For example, in a supermarket, soup is probably large in winter transactions while it may be small in summer transactions.

Lines 4–33 in Figure 3.1 are used

1. to check whether any candidate itemset in db qualifies to be large in the whole database and to adjust their supports in L_{DB+db}, and
2. to check whether any of the large itemsets in DB which are small in db qualifies to be in the set of L_{DB+db}.

The two **for** loops between lines 4–33 perform these two operations. Let us investigate the first case: checking the candidates in db in the kth iteration.

Lemma 3.2 *Let X be an itemset. If $X \notin L_{DB}$, then $X \in L_{DB+db}$ only if $X \in L_{db}$.*

Proof. Since $X \notin L_{DB}$, $support_{DB}(X) < minsup \times |DB|$. In order to satisfy $X \in L_{DB+db}$, $support_{DB+db}(X) \geq minsup \times |DB + db|$.

Suppose that $X \notin L_{db}$. Then, $support_{db}(X) < minsup \times |db|$. Then,

$support_{DB+db}(X) = support_{DB}(X) + support_{db}(X) < (minsup \times |DB|) + (minsup \times |db|) < minsup \times |DB + db|$.

This is a contradiction. Therefore, $support_{db}(X) \geq minsup \times |db|$, which means that $X \in L_{db}$. □

Corollary 3.1 *Let X be an itemset. If X is small both in DB and db, then X can not be large in $DB + db$.*

Now suppose that X is a candidate k-itemset in db. If it is small in db, then we have to check whether X is in L_{DB} or not. If it is also small in DB (i.e., $X \notin L_{DB}$), X can not be a large itemset in $DB + db$ by Corollary 3.1. Otherwise, we have to check the support of X in $DB + db$. Since we have the support of X in DB and db in hand, we can quickly determine whether it is

large or not. If $(support_{DB}(X) + support_{db}(X)) < minsup\times\ |DB + db|$, then X is small in $DB + db$. By Lemma 3.1, all supersets of X must also be small, thus they are eliminated from L_{DB}. Otherwise, X is large and we add X to L_{DB+db}. Another advantage of our algorithm occurs here by not adding X to the set of L^k_{db} to keep the candidate set smaller, which we will explain later in detail.

Now assume that a candidate k-itemset X is large in db. There are two possibilities: X is either large or small in DB.

Lemma 3.3 *Let X be an itemset. If $X \in L_{DB}$ and $X \in L_{db}$, then $X \in L_{DB+db}$.*

Proof. Since $X \in L_{DB}$, $support_{DB}(X) \geq minsup \times |DB|$, and since $X \in L_{db}$, $support_{db}(X) \geq minsup \times |db|$. If we add these two terms, $support_{DB}(X) + support_{db}(X) \geq minsup \times |DB| + minsup \times |db|$. Thus, $support_{DB+db}(X) \geq minsup \times (|DB| + |db|) \geq minsup \times |DB + db|$. □

If X is large in DB, then X is also large in $DB + db$ by Lemma 3.3. In this case, we add the corresponding supports of X in db and DB, and put X into L_{DB+db} with the new support. If X is small in DB, we have to check whether it is large in $DB + db$ or not. However, we do not have the support of X in DB since it is not large. We can obtain it by scanning DB. In this scan, for each 1-itemset in DB, we determine its support and its *tidlist*, as explained before in this section. We will then use these *tidlists* in order to find the support of longer itemsets whenever they are needed. After counting the support of X in DB, we place X into L_{DB+db} if its support in $DB + db$ is larger than $minsup \times |DB + db|$.

An important issue here is to decide which candidates go to the set of large k-itemsets in db. FUP_2 [CLK97] algorithm places all itemsets that are large in the whole database into L^k_{db} in the kth iteration. Others [OS98, TBAR97, SS98a] place those candidates that are large in db regardless of whether they are small or large in DB. We choose another strategy and put only those candidates into C^k_{db} that are large in db and $DB + db$. In other words, if a k-itemset X is large in db but small in $DB + db$, we do not place it into L^k_{db}. This is the most

Cases	In DB	In db	In $DB + db$	Add to C_{db}^k	Add to L_{DB+db}^k	Prune supersets from L_{DB}
Case 1	Small	Small	Small	No	No	No
Case 2a	Small	Large	Small	No	No	No
Case 2b	Small	Large	Large	Yes	Yes	No
Case 3a	Large	Small	Small	No	No	Yes
Case 3b	Large	Small	Large	No	Yes	No
Case 4	Large	Large	Large	Yes	Yes	No

Table 3.2: Possible Cases in Addition of Transactions

important advantage of $UWEP$ since this significantly reduces the number of candidates in db.

In $UWEP$, there is a possibility that a large k-itemset in DB may not be generated in C_{db}^k, since we include those candidates that are large both in db and $DB + db$. The solution is to keep the set of itemsets that must be verified against db, namely $Unchecked$, which contains the large k-itemsets in DB that are not generated in db. In the beginning of the kth iteration, we place all large k-itemsets in DB to the set of $Unchecked$ (line 6 in Figure 3.1). Whenever we check a candidate k-itemset in C_{db}^k, we will remove it from the set $Unchecked$. When we complete the first **for** loop between lines 7–23 in Figure 3.1, $Unchecked$ contains the large itemsets in DB that are not verified against db. The second **for** loop is used to verify them against db. Since we do not generate them from L_{db}^{k-1}, we do not have their supports in db, therefore we have to compute their support from the *tidlists* of the individual items contained in that itemset. If the total support of any element in $Unchecked$ exceeds the minimum support threshold, it is added to L_{DB+db}. Otherwise, the supersets of that itemset are removed from L_{DB} again by Lemma 3.1.

All possible cases that arrive in adding transactions and the actions taken by $UWEP$ are summarized in Table 3.2.

Figure 3.3 gives the candidate generation procedure that is adopted from [SON95]. For two $(k-1)$-itemsets in L_{db}^{k-1}, if the first $(k-2)$ items are the same, then a candidate k-itemset is generated from those $(k-1)$-itemsets by concatenating the last item in the second itemset to the end of the first itemset,

```
    generate_candidate(L_db^{k-1});
1   C_db^k = ∅
2   for all itemsets X ∈ L_db^{k-1} and Y ∈ L_db^{k-1} do
3      if X_1 = Y_1 ∧ ··· ∧ X_{k-2} = Y_{k-2} ∧ X_{k-1} < Y_{k-1} then begin
4         C = X_1 X_2 ... X_{k-1} Y_{k-1}
5         if all subsets S of C is an element of L_db^{k-1} then begin
6            tidlist_db(C) = tidlist_db(X) ∩ tidlist_db(Y)
7            support_db(C) = |tidlist_db(C)|
8            add C to C_db^k
9         end
10     end
```

Figure 3.3: Candidate Generation Algorithm in *UWEP*

assuming that the last item of the second itemset is greater than the last item in the first itemset. However, a candidate generated in this process is pruned from the set of candidates if any of its $(k-1)$-subsets is not large.

3.4 Data Structures Employed

ITEMSET is a list of item numbers, and *TRANSLIST* is a list of transactions. *ITEMTRANS* is a list of records consisting of an *ITEMSET*, its *TRANSLIST*, and its support in the set of transactions (i.e., length of *TRANSLIST*). C_{DB}^k is a queue of *ITEMTRANS* and L_{DB}^k is a hash table where each entry consists of a queue of *ITEMTRANS*. We use a queue for the entries of L_{DB}^k and C_{DB}^k because we gradually add new itemsets to those sets and thus we should minimize the time for addition. The reason behind using a hash table is to find a specific itemset in a short time. It is not necessary to use a hash table for C_{DB}^k because we create and process C_{DB}^k sequentially.

The most important part of the *UWEP* algorithm is the operation of pruning supersets. Therefore, the data structures to keep the large itemsets in DB are very important for an efficient update algorithm. Finding supersets just before pruning them is costly because we have to find all supersets of an itemset X that are large in DB, and no more. There are two methods for finding

supersets of a k-itemset X if we want to find them just before pruning.

1. Add one item in L_{DB}^{1} and check whether it is in L_{DB}^{k+1}, and repeat this process for all items in L_{DB}^{1}
2. Check for each item in L_{DB}^{k+1} whether it consists of the itemset X

Both of these methods require unnecessary computations. Thus, we compute the supersets of an itemset X during the storage of the old large itemsets in a data structure. We keep a hash table for storing the large itemsets in DB for finding a specific itemset efficiently. Each entry in the hash table consists of a queue of itemsets along with their support and list of supersets that are large in DB. While entering a k-itemset X into the hash table, we find all of its subsets of length $k-1$, and push X into the list of their supersets. Since all of X's subsets are also large, we do not make any unnecessary computation. The hash table for storing large itemsets allows us to efficiently find the entry where an itemset is placed, so inserting supersets is a cheap operation with this strategy. When we want to prune the supersets of an itemset X, we find them stored in a list associated with that itemset. Each itemset in the list of X's supersets is removed from the hash table, and this operation is repeated for each of the supersets of the itemsets in that list.

Example 3.1 *Let A be the itemset we want to prune from L_{DB}. Let AB, AC, AD be its supersets that are large. We remove them from the hash table, and then remove the supersets of AB, AC, and AD. For instance, if the superset list of AB is ABC and ABD, then we also remove ABC, ABD, and their large supersets from the hash table.*

Another source of improvement in the implementation is the usage of an inverted list for the transaction database. When we compute the set of large itemsets in a set of transactions DB initially, we create a file where each item is associated with a list of transaction identifiers in which the item exists. During the update operation, we can use this inverted file for counting the supports of the itemsets that are small in DB but large in db. The inverted file allows us to directly reach the transactions containing a specific item, and computing its

DB		db	
TID	**Items**	**TID**	**Items**
1	A,C,D,E,F		
2	B,D,F		
3	A,D,E	1	A,F
4	A,B,D,E,F	2	B,C,F
5	A,B,C,E,F	3	A,C
6	B,F	4	B,F
7	A,D,E,F	5	A,B,C
8	A,B,D,F	6	A,C,D
9	A,D,F		

Table 3.3: Set of Transactions DB and db

support in DB. This is certainly more efficient than reading all transactions and identifying the transactions containing a specific itemset. Especially for large transaction databases, this optimization improves the I/O time in the computation of large itemsets. $UWEP$ also allows us to update the list of transaction identifiers associated with itemsets: We only add the transaction identifiers of the transactions in db containing that item to the $TRANSLIST$ of that item. (We assume that none of the transaction identifiers in DB are used for the transactions in db, and TIDs in db are greater than those in DB.)

3.5 An Example Execution of the Algorithm

We now introduce an example that illustrates the benefits of our algorithm and compare the number of candidates generated and counted with *Apriori* and FUP_2 algorithms. We will write an itemset $\{A_1, \ldots, A_n\}$ as $A_1, \ldots, A_n$, and a pair $(itemset, support)$ refers to an itemset and its support in the corresponding set of transactions.

Example 3.2 *In Table 3.3, the set of transactions in DB and db are provided.* $|DB| = 9, |db| = 6, |DB + db| = 15$. *The minimum support threshold minsup is set to 0.3. Thus, an itemset X must be present in at least 3 transactions in DB (i.e.,* $\lceil |DB| \times 0.3 \rceil = \lceil 2.7 \rceil = 3$*), in at least 2 transactions in db, and in at*

least 5 transactions in $DB + db$ in order to be a large itemset. The execution of the algorithm for this database is described below.

Initially, we assume that the set of large itemsets in DB are given. In the example database DB, the sets of large k-itemsets along with their counts are as follows.

$$L^1_{DB} = \{(A,7),(B,5),(D,7),(E,5),(F,8)\}$$

$$L^2_{DB} = \{(AB,3),(AD,6),(AE,5),(AF,6),(BD,3),(BF,5),$$
$$(DE,4),(DF,6),(EF,4)\}$$

$$L^3_{DB} = \{(ABF,3),(ADE,4),(ADF,5),(AEF,4),(BDF,3),(DEF,3)\}$$

$$L^4_{DB} = \{(ADEF,3)\}$$

In the first step of the algorithm, db is scanned in order to find the support of 1-itemsets in db. In this scan, we generate the *tidlist* for each 1-itemset. In the example, the candidate 1-itemsets in db, along with their supports, are:

$$C^1_{db} = \{(A,4),(B,3),(C,4),(D,1),(F,3)\}.$$

Note that we do not include E in C^1_{db} since its support is zero in db. On the other hand, E is added to the $PruneSet$ in order to check itemsets including E in L_{DB}. Since the support of E is 5 and is thus large in $DB + db$, we remove it from L_{DB} and include it in L_{DB+db} and add its supersets in L^2_{db} to the $PruneSet$, namely AE, DE, EF. Then for each element of the $PruneSet$, we repeat the same operation. We add AE to L_{DB+db} since its support is also 5. However, the supports of DE and EF are 4, and they fail to qualify to go into L_{DB+db}. In this step, we remove DE and EF and all their supersets from L_{DB}, namely $ADE, DEF, AEF, ADEF$ (By Lemma 3.1). After these pruning operations, the new sets of large itemsets in DB and set of large itemsets in $DB + db$ are as follows.

$$L^1_{DB} = \{(A,7),(B,5),(D,7),(F,8)\}$$

$$L^2_{DB} = \{(AB,3),(AD,6),(AF,6),(BD,3),(BF,5),(DF,6)\}$$

$L^3_{DB} = \{(ABF, 3), (ADF, 5), (BDF, 3)\}$

$L^4_{DB} = \emptyset$

$L_{DB+db} = \{(E, 5), (AE, 5)\}$

In the first iteration, A, B, C, D, F are added to L_{DB+db}. We add all large 1-itemsets in db to L^1_{db}, namely A, B, C, F. We do not include D in L_{db} since it does not qualify to be large in db. After the first iteration,

$L^1_{db} = \{(A, 4), (B, 3), (C, 4), (F, 3)\}$, and

$L^1_{DB+db} = \{(A, 11), (B, 8), (C, 6), (D, 8), (E, 5), (F, 11)\}$

In the second iteration, we begin with the set of candidates in db,

$C^2_{db} = \{(AB, 1), (AC, 3), (AF, 1), (BC, 2), (BF, 2), (CF, 1)\}$, and

$Unchecked = \{AB, AD, AF, BD, BF, DF\}$.

AB is found to be small in db, but large in DB. AB fails to be large in $DB + db$ since $support_{DB+db}(AB) = 4$. By Lemma 3.1, we remove ABF from L_{DB}. The itemset AC is large in db but small in DB. Since we do not have support of AC in DB in hand, we find AC's support in DB by intersecting the *tidlists* of A and C in DB, which is 2. ($tidlist_{DB}(A) = \{1, 3, 4, 5, 7, 8, 9\}$, $tidlist_{DB}(C) = \{1, 5\}$, their intersection is $\{1, 5\}$) Since the total support of AC is 3+2=5, AC is added to L_{DB+db} (Application of Lemma 3.2). AF is small in db, with a total support of 7. Therefore, AF is added to L_{DB+db}, but we do not include it in L^2_{db}. BC is large in db but small in DB. So, we compute the support of BC in DB, which is 1. The total support of BC is 3, so we do not include it in L_{DB+db} nor in L^2_{db}. BF is large both in DB and db. So it is large in $DB + db$ with a support of 7. Since CF is small both in DB and db, it is small in $DB + db$ by Corollary 3.1. Up to this point, we checked each element of C^2_{db}, but not all elements of L^2_{DB}. At this moment,

$Unchecked = \{AD, BD, DF\}$.

We did not compute the supports of these itemsets in db since we did not include D in L^1_{db}, so for each of them we have to compute its support in db

using *tidlists* of the items contained in the itemset. Supports of AD, BD, DF in db are 1, 0, 0, respectively. We find the total support of these itemsets by adding their supports in DB and db. In our case, the supports of AD, BD, DF in $DB + db$ are 7,3,6, respectively. AD and DF are found to be large in the whole database, so we add them to L_{DB+db}. Since BD is small in the whole database, we have to remove its supersets from L_{DB}, namely BDF.

At the end of the second iteration, we find that

$L^2_{db} = \{(AC, 3), (BF, 2)\}$, and

$L^2_{DB+db} = \{(AC, 5), (AD, 7), (AE, 5), (AF, 7), (BF, 7), (DF, 6)\}$

Before proceeding to third iteration, we compute

$C^3_{db} = \emptyset$

$Unchecked = \{ADF\}$

Since, $C^3_{db} = \emptyset$, we proceed with checking the elements of $Unchecked$. The support of ADF is 0 in db and its support in $DB+db$ is 5. Thus, we add ADF into L_{DB+db} and finish the update operation. The final set of large itemsets in $DB + db$ are:

$L^1_{DB+db} = \{(A, 11), (B, 8), (C, 6), (D, 8), (E, 5), (F, 11)\}$

$L^2_{DB+db} = \{(AC, 5), (AD, 7), (AE, 5), (AF, 7), (BF, 7), (DF, 6)\}$

$L^3_{DB+db} = \{ADF, 5\}$

3.5.1 Comparison with the Existing Algorithms

Table 3.4 shows the number of candidates generated and counted by the *Apriori*, FUP_2, and $UWEP$ algorithms over the example database given in Table 3.3. It is worth noting that the *Apriori* algorithm re-runs over the whole set of transactions, and therefore counting candidates over DB and db is irrelevant.

As Table 3.4 shows, our algorithm generates a much smaller number of

		Apriori	FUP_2	$UWEP$
Iteration 1	Candidates generated in db	6	6	5
	Candidates counted in DB	–	1	1
	Candidates counted in db	–	6	6
	Total # of candidates counted	6	7	7
Iteration 2	Candidates generated in db	15	15	6
	Candidates counted in DB	–	2	2
	Candidates counted in db	–	9	9
	Total # of candidates counted	15	11	11
Iteration 3	Candidates generated in db	1	1	0
	Candidates counted in DB	–	0	0
	Candidates counted in db	–	1	1
	Total # of candidates counted	1	1	1

Table 3.4: Number of Candidates Generated and Counted in the Example Database

candidate sets than *Apriori* or FUP_2 in this specific example (We will analyze the general case later in detail). Especially for the second iteration, $UWEP$ achieves $\frac{15-6}{15} = 60\%$ improvement over the two algorithms. Overall, $UWEP$ has a performance improvement of $\frac{22-11}{22} = 50\%$ over the two algorithms. Note that, the candidates counted by $UWEP$ is the same as FUP_2, but the number of candidates generated by FUP_2 is larger than the one generated by $UWEP$.

In the case of running the Partition Update algorithm (*PU*) of [OS98], the number of candidates counted is much greater than that of $UWEP$. In *db* there are four large 1-itemsets and three large 2-itemsets. In order to find them, 11 candidates are generated and counted in *db*. Since we know the support of four of them in DB, *PU* has to count only 3 candidates on DB. However, it has to count 17 large itemsets of DB over *db* since their supports in *db* are not available. Therefore, a total of 3 itemsets are counted in DB and $11 + 17 = 28$ itemsets are counted in *db*. On the other hand, $UWEP$ counts 3 candidates in DB and $6 + 9 + 1 = 16$ candidates in *db*. Even only one scan of *db* and DB is enough for counting itemsets, the number of candidates counted is very high in comparison to the $UWEP$ algorithm, where $UWEP$ achieves a $\frac{28-16}{28} = 43\%$ improvement over *Partition Update* algorithm in the number of candidates counted in *db*.

3.6 Completeness and Efficiency of the Algorithm

The algorithm *UWEP* presented in Figure 3.1 correctly and completely computes the set of large itemsets in the updated database.

Lemma 3.4 *Given a set of old transactions (DB), a set of new transactions (db), and a set of itemsets L_{DB} which are large over DB, the algorithm in Figure 3.1 discovers all the large itemsets over $DB + db$ correctly.*

Proof. Let X be a k-itemset. By Corollary 3.1, X must be large in either DB or db, or both. Thus, in order to compute large itemsets in $DB + db$, we have to check large itemsets in DB against db, and large itemsets in db against DB. Let us investigate these two cases:

Case 1: Checking for all $X \in L_{DB}$ against db

In the initial pruning step (algorithm in Figure 3.2), all itemsets X in L_{DB} such that $support_{db}(X) = 0$ are checked. If X is small in $DB + db$, all of its supersets are removed from consideration since they can not also be large in $DB + db$ by Lemma 3.1. If X is large in $DB + db$, we put it into L_{DB+db}, and its immediate supersets into the $PruneSet$. This process is repeated until the $PruneSet$ is empty. In the end, any large itemset in DB whose support in db is zero is checked against db. Thus, before the **while** loop on line 5 in Figure 3.1, L_{DB} contains the large itemsets in DB whose support in db is greater than zero, and L_{DB+db} contains all large itemsets containing the items whose support is zero in db.

In the kth iteration, $Unchecked$ is initialized to the set of large k-itemsets in DB. Any element of $Unchecked$ that is present in C^k_{db} is checked on lines 9 and 16. If an itemset in $Unchecked$ does not exist in C^k_{db}, then the second **for** loop counts their support in db, and decides which of them are large in the updated database. Therefore, all elements of L^k_{DB} are checked against db, and the ones that are large in $DB + db$ are determined.

Case 2: Checking for all $X \in L_{db}$ against DB

In the *UWEP* algorithm, C_{db}^k contains possibly large itemsets over $DB+db$, instead of possibly large itemsets in db. In the first **for** loop, only those in C_{db}^k that are large over $DB + db$ are put into L_{db}^k (lines 17 and 22). If a k-itemset X is large in db but not in $DB + db$, then it is a waste of effort to put it into L_{db}^k because by Lemma 3.1, it is not possible that a superset of X is large in $DB+db$. Since any superset of X is certainly small in $DB+db$, we do not need to check whether any superset of X is large in db or not (even if a superset of X is large in db, it will be certainly small in the updated database). Since our purpose is to generate the large itemsets in $DB + db$, putting X into L_{db}^k is a waste of effort, and reduces the performance of the algorithm.

Thus, the first **for** loop checks for all the itemsets in C_{db}^k against DB. If any large itemset in C_{db}^k is also large in DB, then we simply put it into L_{DB+db}^k on line 17 by Lemma 3.3. If it is small in DB, then we count its support in DB using *tidlists*, and decide to put it into L_{DB+db}^k and L_{db}^k on line 22. Therefore, all large itemsets in db are checked against DB.

As a consequence of Case 1 and Case 2, the algorithm *UWEP* computes the large itemsets in $DB + db$ correctly and completely. □

Lemma 3.5 *The number of candidates generated and counted by the algorithm UWEP in Figure 3.1 is minimum.*

Proof. The only candidate generation operation is over db. Therefore, to prove that the number of candidates generated is minimum, we only deal with the set of candidates in db. C_{db}^1 contains only the itemsets whose support is greater than zero. This is the minimum bound because to decide which of the itemsets is large in $DB + db$, we have to know at least the support of each item in db. Therefore, C_{db}^1 contains the minimum number of candidates. In the kth iteration, we put only the itemsets that are large over $DB + db$ into L_{db}^k. The completeness of this operation is shown in the proof of Lemma 3.4. We have to put those itemsets that are large over $DB + db$ into L_{db}^k because, their supersets are possibly large over $DB + db$, and we have to check them in order to complete the update operation. Since, we do not include any other itemset in L_{db}^k, this is the minimum bound for a level-wise algorithm. As shown

in Figure 3.3, the candidate set C_{db}^{k+1} is computed from L_{db}^{k}, so the number of candidates generated in db is also minimum.

Since the candidates generated in db is minimum, the number of candidates counted in db is also minimum. The only remaining issue is the number of candidates counted in DB. Since, we only scan DB in order to find the support of an itemset that is not large in DB, this is also the lower bound. Hence, the number of candidates counted is minimum. □

3.7 Experimental Results

In order to measure the performance of $UWEP$, we conducted several experiments using the synthetic data introduced in [AS94]. Before proceeding to the details of the experiments, we would like to present the parameters used in the data generation procedure.

The synthetic data generated in [AS94] mimics the transactions in the retailing environment. Our synthetic data generation procedure is a simple extension of the method used in [AS94]. We generated a transaction database of size $2 \times |DB|$, where the first $|DB|$ transactions were placed into the set of old transactions. From the remaining transactions, we took the first $\frac{|DB|}{10}$ transactions for the first incremental database, took the first $\frac{2 \times |DB|}{10}$ transactions for the second incremental database, and so on. Since all transactions are generated using the same statistical pattern, the transactions in the incremental database exhibit the same regularities in the original database. In the experiments, we used the following parameters. Number of maximal potentially large itemsets=$|L|$=2000, number of transactions=$|D|$=200,000, average size of the transactions=$|T|$=10, number of items=N=1000 and average size of the maximal potentially large itemset=$|I|$=4. We follow the notation $Tx.Iy.Dm.dn$ used in [CHNW96] to denote databases in which $|DB| = m$ thousands, $|db| = n$ thousands, $|T| = x$ and $|I| = y$. Readers not familiar with these parameters are referred to [AS94].

For the first experiment, we measured the speedup gained by $UWEP$ over

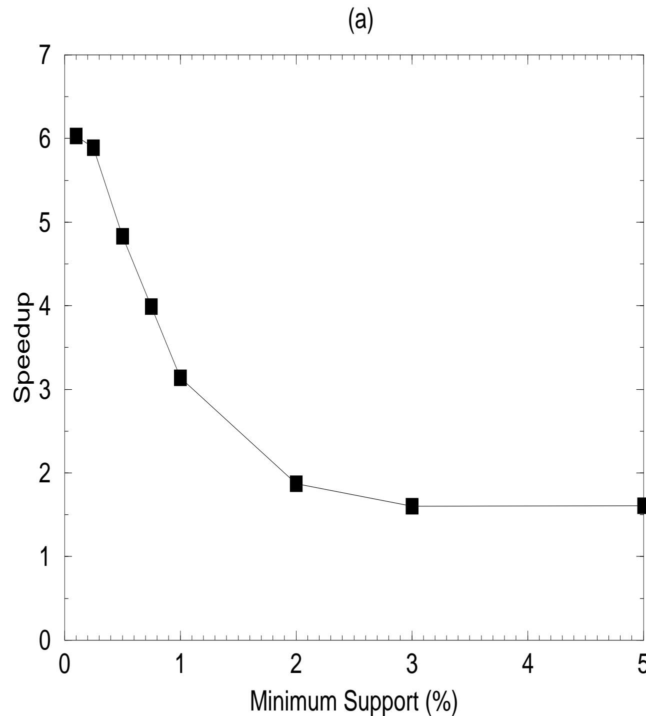

Figure 3.4: Speedup by *UWEP* over *Partition* Algorithm

rerunning Partition algorithm [SON95]. We have chosen *Partition* since it is one of the best association rule algorithms and the same data structures and methodology for finding large itemsets are used in both algorithms. Figure 3.4 shows the results for $T10.I4.D100.d10$. The y-axis in the graph represents $\frac{Execution\ Time\ of\ Partition}{Execution\ Time\ of\ UWEP}$, and x-axis represents different support levels. As it can be seen from Figure 3.4, *UWEP* performs much better than re-running *Partition*. Figure 3.4 shows that at lower support levels, the speedup gain of *UWEP* increases from 1.5 to 6 as the minimum support decreases from 3% to 0.1%. For support levels higher than 3%, the speedup seems to converge to 1.5.

In the second experiment, we measured the effect of the size of the incremental database on the execution time of the algorithms. Figure 3.5 shows the execution times for *UWEP* and *Partition* algorithms for $T10.I4.D100.dn$, where n varies from 10 to 100, with the minimum support set to 0.5%. For smaller sizes of the incremental database, *UWEP* achieves a much better performance than *Partition*. As the size of the new transactions increases, the

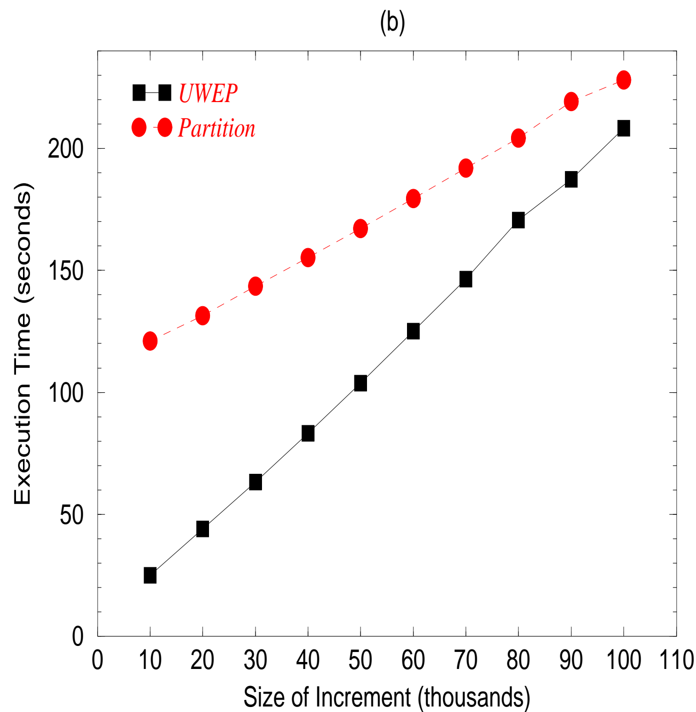

Figure 3.5: Execution Times of $UWEP$ and $Partition$ Algorithms

execution time of $UWEP$ gets closer to that of the $Partition$. On the other hand, despite adding 100% transactions, $UWEP$ still performs better than re-running $Partition$. One interesting feature of $UWEP$ is that its execution time is linear to the size of incremental database under a specified minimum support. In this sense, $UWEP$ can scale up linearly to the size of incremental database whatever the minimum support is.

The third experiment investigates the number of candidates generated and counted for the three update algorithms, $Partition\ Update$, FUP_2, and $UWEP$. For this experiment, we generated an increment database containing a smaller number of items than that in the original database. Table 3.5 shows the number of candidates generated and counted for three algorithms for $T10.I4.D100.d10$ with 900 items in the new set of transactions. The reason behind smaller number of items in the incremental database is to see the effects of data skewness in the update of large itemsets. As Table 3.5 shows, $UWEP$ generates a much smaller number of candidates in comparison to the other two algorithms, between 32%–53% of those generated by FUP_2 and $Partition\ Update$.

		(1)	(2)	(3)	Imprv.	Imprv.
	minsup	*PU*	FUP_2	*UWEP*	on (1)	on (2)
Candidates Generated in *db*	0.75%	100177	99797	53759	46%	46%
	0.5%	146431	161746	90884	38%	44%
	0.1%	351652	511717	239662	32%	53%
Candidates Counted in *db*	0.75%	100341	53762	53762	46%	–
	0.5%	147740	91417	91417	38%	–
	0.1%	379352	251963	251963	34%	–
Candidates Counted in *DB*	0.75%	206	187	187	9%	–
	0.5%	1612	571	571	65%	–
	0.1%	28040	8675	8675	69%	–
Total Candidates Counted	0.75%	100547	53949	53949	46%	–
	0.5%	149352	91988	91988	38%	–
	0.1%	407392	260638	260638	36%	–

Table 3.5: Number of Candidates Generated and Counted on Synthetic Data

The number of candidates counted by *UWEP* is exactly the same as that by FUP_2. However, the *Partition Update* algorithm counts more candidates than *UWEP* counts, up to 69%. The results indicate that *UWEP* performs much better than the other two algorithms when some of the large itemsets in *DB* are absent in *db*, thus in $DB + db$, as well.

3.8 Theoretical Discussion of the Update Algorithms

3.8.1 Number of Candidates

UWEP yields a smaller candidate set in comparison to other update algorithms. FUP_2 [CLK97], which is a generalization of *FUP* [CHNW96], examines a large k-itemset only in the kth iteration and generates the candidate set C_{db}^k from the set of large $(k-1)$-itemsets in the updated database. Then, by means of a few optimizations, it prunes some of the candidates and counts the remaining over *DB* and *db*. *PartitionUpdate*(*PU*) finds the set of large

itemsets in db and then checks large itemsets in DB against db and vice versa. Thus, it generates the candidate set C_{db}^k from the set of large $(k-1)$-itemsets in the incremental database. Similarly, the algorithms in [SS98a, TBAR97] generate the candidate set C_{db}^k from the set of large itemsets L_{db}^{k-1}, with the same number of candidates in PU. On the other hand, $UWEP$ generates the set of candidate set C_{db}^k from the set of itemsets that are large both in db and in the updated database. This results in a much smaller candidate set in comparison to the mentioned algorithms.

In some special cases, the difference between the number of candidates generated by $UWEP$ and by the other update algorithms may converge. When the large itemsets in DB and $DB + db$ are nearly same, $UWEP$ and FUP_2 generates and counts nearly the same number of candidates. However, when the data is skewed and db does not include most of the large itemsets in DB, $UWEP$ outperforms FUP and FUP_2. PU algorithm may perform better when db contains a smaller number of large itemsets. This is also valid for the other update algorithms since they generate the candidates only over db. However, as we proved in Section 3.6, $UWEP$ generates and counts the minimum number of candidate itemsets for a level-wise algorithm. Hence, $UWEP$ is the best algorithm in terms of the number of candidates generated and counted.

3.8.2 Time Complexity

The motivation behind the update algorithms is using the background knowledge to avoid repetition of the computation of old large itemsets. Thus, rerunning an association rule algorithm, say *Apriori* [AS94], performs worse than any update algorithm since the set of old large itemsets are ignored and all the work done previously is repeated.

$UWEP$ takes its power from pruning the large itemsets as early as possible and generating the minimal number of candidates. This smaller number of candidates to be examined brings an advantage to $UWEP$ in terms of time complexity. In the experimental work, we did not compare $UWEP$ and FUP_2 in terms of time complexity because the underlying data structures and methodologies for counting the supports of itemsets are completely different.

In such an experiment, the results solely depend on the difference between the frameworks (*Partition* algorithm used by $UWEP$ and *Apriori* algorithm used by FUP_2) of these two algorithms, instead of the efficiency of the update algorithms. Theoretically, it is expected that $UWEP$ performs better than FUP_2 because it generates a smaller number of candidates than the latter. The differences between the two algorithms are the pruning step of $UWEP$ and the pruning optimizations employed in FUP_2. The pruning operation in $UWEP$ requires less time because of the data structures used for storing large itemsets. Pruning candidate itemsets in FUP_2 is also efficient by means of storing upper bounds for itemsets. However, each candidate itemset is checked for the pruning operation and this may require more time. Especially, in the second iteration, the candidate set is very huge and checking for pruning each itemset is costly. $UWEP$ generates its candidate set carefully and minimizes it instead of generating an extremely large set and then pruning it. Actually, the number of candidates counted is the same. The difference between the candidates generated and candidates counted is the drawback of the FUP_2 algorithm, and this gives an advantage to $UWEP$.

In addition to the improvement on the size of the candidate set, $UWEP$ requires fewer scans over DB and db. FUP_2 requires a scan over the set of transactions for each iteration while $UWEP$ requires at most one scan over DB and exactly one scan over db for the whole computation. Especially for large datasets, this also yields a great improvement over FUP_2. However, as we noted earlier, this is a direct consequence of the underlying frameworks of two algorithms.

$PartitionUpdate(PU)$ algorithm also performs worse than $UWEP$, because it first generates the candidate set on db without using the information about the old large itemsets. Thus, it may generate and count itemsets that have no chance to be large in the whole database. Besides, PU does not prune the large itemsets in DB and checks for each large itemset in DB whether it is large in the whole set of transactions. The only benefit of the $PartitionUpdate$ algorithm comes from using the set of large itemsets in DB from a file instead of computing them again. In terms of number of scans over the database, PU makes an extra scan over db in comparison to $UWEP$ in order to check the

large itemsets in DB against db.

The candidates are generated over db in [SS98a, TBAR97], too. The same arguments in PU apply to these algorithms, and their performances are also worse than $UWEP$. The major disadvantage of the algorithms in [SS98a, TBAR97] is the storage and re-computation of the negative border, that requires much space and time, respectively. Moreover, many scans over the incremental database are required in both algorithms. Number of scans over DB depend on the number and size of new large itemsets, and it may be greater than one.

Chapter 4

Case of Deleted Transactions

In this chapter, we will examine the case where some of the existing transactions in DB are removed from DB. Throughout this chapter, we will use db^- for the set of transactions that will be deleted, and $DB - db$ for the set of transactions that remain in the database after the update.

In order to give an intuition about the problems in deletion, we first investigate the possible cases according to whether an itemset X is small or large in DB and db^-. We will provide examples to illustrate different cases in $DB - db$. In the examples below, we will take $|DB| = 100$, $|db^-| = 10$, and $minsup = 20\%$. For X to be large in the updated database, its support in $DB - db$ must be greater than or equal to 18.

Case 1: X is large in DB and small in db^-.

In this case, X must be large in $DB - db$.

Lemma 4.1 *If X is large in DB and small in db^-, then X must be large in $DB - db$.*

Proof. Since X is large in DB, $support_{DB}(X) \geq |DB| \times minsup$. Since X is small in db^-, $support_{db^-}(X) < |db^-| \times minsup$. If you multiply the second expression by -1 and add two equations, $support_{DB}(X) - support_{db^-}(X) \geq$

$|DB| \times minsup - |db^-| \times minsup$, which is equal to $support_{DB-db}(X) \geq |DB - db| \times minsup$. Thus, X is large in the updated database. □

Case 2: X is large in DB and large in db^-.

X can be large in $DB-db$. Consider the case where $support_{DB}(X) = 30$ and $support_{db^-}(X) = 2$. Then, $support_{DB-db}(X) = 28$, so X is large in $DB - db$.

X can be small in $DB-db$. Consider the case where $support_{DB}(X) = 20$ and $support_{db^-}(X) = 5$. Then, $support_{DB-db}(X) = 15$, so X is small in $DB - db$.

Case 3: X is small in DB and small in db^-.

X can be large in $DB-db$. Consider the case where $support_{DB}(X) = 19$ and $support_{db^-}(X) = 1$. Then, $support_{DB-db}(X) = 18$, so X is large in $DB - db$.

X can be small in $DB-db$. Consider the case where $support_{DB}(X) = 15$ and $support_{db^-}(X) = 1$. Then, $support_{DB-db}(X) = 14$, so X is small in $DB - db$.

Case 4: X is small in DB and large in db^-.

In this case, X must be small in $DB - db$.

Lemma 4.2 *If X is small in DB and large in db^-, then X must be small in $DB - db$.*

Proof. Since X is small in DB, $support_{DB}(X) < |DB| \times minsup$. Since X is large in db^-, $support_{db^-}(X) \geq |db^-| \times minsup$. If you multiply the second by -1 and add two equations, $support_{DB}(X) - support_{db^-}(X) < |DB| \times minsup - |db^-| \times minsup$, which is equal to $support_{DB-db}(X) < |DB - db| \times minsup$. Thus, X is small in the updated database. □

Table 4.1 summarizes the possible cases in deletion of transactions. When we update a set of transactions by deleting some of the transactions, there are two possible cases: The existing large itemsets may become small after update (Case 2 in Table 4.1), or some itemsets that are small in the original database may become large after update (Case 4 in Table 4.1). Because of these reasons, to find the set of large itemsets in the updated database, we

Cases	**In** DB	**In** db^{-}	**In** $DB - db$
Case 1	Large	Small	Certainly large
Case 2	Large	Large	May be small or large
Case 3	Small	Small	May be small or large
Case 4	Small	Large	Certainly small

Table 4.1: Possible Cases in Deletion of Transactions

have to check whether the old large itemsets are still large or not, and whether some itemsets are added to the set of large itemsets or not.

4.1 Existing Approaches

Update of large itemsets in case of deletion of transactions is not studied much. To the best of our knowledge, there are two update algorithms handling deletion of transactions: FUP_2 [CLK97] and Thomas's algorithm [TBAR97].

In [TBAR97], the logic is similar to the case of addition of transactions. They use the negative border of DB in order to efficiently compute the large itemsets in $DB - db$. The paper did not discuss the case of deletions in detail, so we can not provide a detailed analysis of the algorithm here.

FUP_2, for the case of deleted transactions, is presented in Figure 4.1. The candidate set is generated over $DB - db$, and then it is partitioned into two sets: P_k contains the ones that are large in DB and Q_k contains the others. db is scanned to find the support of each candidate itemset. By Lemma 4, candidate itemsets that are large in db are removed from Q_k. Since we have the support of itemsets in P_k in DB and db, it is trivial to find the ones that are large in the updated database. For the itemsets in Q_k, we have to find their supports in $DB - db$ by scanning the updated database. Then, we decide which of the itemsets in the candidate set promote to the set of large itemsets in $DB - db$.

The major disadvantage of FUP_2 is that it scans the databases as many

```
    FUP_2_delete(k)
1   C^k_{DB-db} = generate_candidate(L^{k-1}_{DB-db})
2   if C^k_{DB-db} ≠ ∅ then
3   begin
4      Partition C^k_{DB-db} into P_k = C^k_{DB-db} ∩ L^k_{DB} and Q_k = C^k_{DB-db} − P_k
5      for each X ∈ P_k ∪ Q_k do
6         compute support_{db^-}(X) by scanning db^-
7      for each X ∈ P_k do
8         compute support_{DB-db}(X)
9      Remove candidates which are large in db^- from Q_k
10     for each X ∈ Q_k do
11        compute support_{DB-db}(X) by scanning DB − db
12     Add candidates in P_k ∪ Q_k which are large in DB − db to L^k_{DB-db}
13     Stop if |L^k_{DB-db}| < k + 1
14  end
```

Figure 4.1: FUP_2 Algorithm: Deletion of Transactions

times as the length of the maximal itemset. As we will investigate in Section 4.2, the intuition behind FUP_2 seems to be the best solution in the case of deleted transactions.

4.2 Challenges in Update for Deletion Case

In case of added transactions, a large itemset in $DB + db$ must be large in at least one of DB and db. Thus, finding large itemsets in the incremental database is sufficient to determine the set of large itemsets in $DB+db$. However, in the case of deleted transactions, a new itemset can be added to the set of large itemsets in $DB - db$ if it is small both in db and DB. In other words, a small itemset in db can be large in the updated database. As a consequence, keeping the set of large itemsets in db is not sufficient to find the large itemsets in $DB-db$. The candidate set for any iteration must also include those itemsets that are small in db but large in $DB - db$.

Because of this reason, there are two possible approaches.

1. Keep small itemsets in db as well as the large itemsets.

2. Generate the candidate set C^k_{DB-db} from L^{k-1}_{DB-db}.

The first approach is a costly operation because the set of small itemsets in a database is much larger than the set of large itemsets. If we denote the set of small k-itemsets in db by S^k_{db} and all itemsets of length k by PS_k, then $S^k_{db} = PS_k - L^k_{db}$. Finding all itemsets of length k, eliminating the ones in L^k_{db}, and counting their supports is very expensive in terms of time and space. Thus, this approach is not an efficient solution.

The second approach is used in FUP_2 algorithm [CLK97]. As in case of addition of transactions, they generate the candidates over the updated database instead of the incremental database. The methodology used by [CLK97] seems to be the best solution to this problem without using the knowledge of old large itemsets. Let us investigate the reasons behind this claim.

Let us think L^k_{DB-db} as two separate sets. Let $OldLarge$ be the set of itemsets in $DB - db$ that are large in DB and $OldSmall$ the set of itemsets in $DB-db$ that are small in DB. For any itemset X in $OldSmall$, the supersets of X are also small in DB, and L_{DB} does not contain them. Thus, the candidate set must contain those containing X. The only improvement we can do is reducing the number of itemsets that contain no items from $OldSmall$,i.e., the itemsets that can be generated from the set $OldLarge$.

Example 4.1 *Let* $L^k_{DB-db} = \{A, B, C, D, E\}$. *Let* $OldLarge = \{A, B, C, D\}$ *and* $OldSmall = \{D, E\}$. FUP_2 *puts into* C^k_{DB-db} *all itemsets of length 2 containing the items in* L^2_{DB-db}. *The candidate set for an update algorithm should contain* AD, AE, BD, BE, CD, CE, *and* DE, *because* D *and* E *are previously small and none of the itemsets containing them exists in the set of old large itemsets. The only improvement we can do is to prune any of* AB, AC *and* BC. *One possible solution is not to generate candidates over OldLarge but this does not work when at least one of* AB, AC *and* BC *is small in* DB. *If one of them is small, then it has a chance to be large after the deletion of transactions, so we have to put it into the candidate set.*

Since we know the large $(k + 1)$-itemsets in DB, one possible approach to reduce the size of the candidate set is not to include those that are large in

DB. However, it does not seem doable directly with the *generate_candidate* function, because *generate_candidate* function computes the candidates from L^k_{DB-db} without any other information. Then, we have to modify the candidate generation function to use the information on L^{k+1}_{DB}. This can be done by generating the candidates over $OldLarge$ and removing those that are in L^{k+1}_{DB}. However, this requires generating the candidates and pruning them, that is same as the method used in FUP_2. As a result, there is no way to generate only the candidates over $OldLarge$ which are small in DB.

The best solution for generating candidate set seems to be using L^k_{DB-db} in the *generate_candidate* function. Therefore, we can not make an improvement on the size of the candidate set by the optimization employed in $UWEP$.

Moreover, the early pruning strategy in $UWEP$ does not bring an advantage in case of deleted transactions with respect to the algorithm FUP_2. In other words, early pruning does not result in a smaller candidate set in the later iterations. As it can be seen in Figure 4.1, FUP_2 makes a kind of pruning by dividing C^k_{DB-db} into $P_k = C^k_{DB-db} \cap L^k_{DB}$ and $Q_k = C^k_{DB-db} - P_k$.

To conclude, we can say that the optimizations employed in $UWEP$ do not work in the case of deletions. Without using negative border, FUP_2 seems to be the optimal level-wise algorithm when some of the transactions are deleted from the database. The only drawback of FUP_2 is the number of scans over the databases. This problem can be solved using the framework of $UWEP$, i.e., the framework in the *Partition* algorithm [SON95].

Chapter 5

Conclusion

Discovering association rules is an important class of data mining, and association rules have a wide area of usage. Although many efficient algorithms have been proposed up to now, extracting association rules is still a computationally expensive operation in large databases.

Since it is a time-consuming operation, the maintenance of association rules is also an important issue, especially in dynamic databases in which frequent additions and deletions take place. Instead of computing all the rules again, we proposed an efficient algorithm in this thesis, which uses the previously discovered rules in order to find the set of new association rules. *Update With Early Pruning (UWEP)*, as presented in this thesis, attempts to minimize the number of candidates generated and counted over the incremental database. Specifically, we show that $UWEP$ generates and counts the optimally minimum number of candidates. Moreover, it outperforms the existing update algorithms in terms of the number of scans over the databases. While the other update algorithms make as many passes as the length of maximal large itemset, $UWEP$ requires at most scan over the old database, and exactly one scan over the incremental database.

We presented a theoretical comparison of $UWEP$ with the other update algorithms, and showed that $UWEP$ outperforms the others theoretically. Moreover, experimental results indicate that our theoretical analysis is valid in real

life, too. The experiments on synthetic data, which is assumed to imitate the real life [AS94], supported our claim about the optimality of our algorithm.

5.1 Future Work on *UWEP*

Some possible future work are summarized as follows.

- We said that the number of scans over the databases is limited to one. However, this requires that the memory should be big enough to hold the sets of inverted lists, and the set of candidate itemsets generated. This requirement can be overcome by applying memory buffering techniques, or a partitioning framework applied in the *Partition* algorithm [SON95].
- We studied on the framework of association rules, and proposed an algorithm to update large itemsets efficiently. However, there are data mining tasks which are also based on the generation of large itemsets. We believe that *UWEP* can be modified easily to be applicable to other data mining tasks, such as discovering sequential patterns or deviations.
- We showed that the case of deletions and modifications is more difficult to handle than the case of additions. The more efficient applicability of *UWEP* to those cases is an interesting future work.
- The logic in *UWEP* may be used to develop an efficient algorithm to mine the association rules in a large database. By considering each transaction (or a certain set of transactions) as an incremental database, can we develop a new algorithm to find all the rules valid on the whole database?

Bibliography

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'93)*, pages 207–216, Washington, D. C., USA, May 1993.

[AMS+96] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In Usama Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramaswamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of* 20th *Intl. Conf. on Very Large Databases (VLDB'94)*, pages 487–499, Santiago de Chile, Chile, September 1994.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of* 11th *Intl. Conf. on Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, September 1995.

[AS96] Rakesh Agrawal and John C. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962–969, 1996.

[AY98a] Charu C. Aggarwal and Philip S. Yu. Mining large itemsets for association rules. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 21(1):23–31, March 1998.

[AY98b] Charu C. Aggarwal and Philip S. Yu. A new framework for itemset generation. In *Proceedings of* 17th *Symposium on Principles of Database Systems (PODS'98)*, pages 18–24, Seattle, Washington, USA, June 1998.

[AY98c] Charu C. Aggarwal and Philip S. Yu. Online generation of association rules. In *Proceedings of* 14th *Intl. Conf. on Data Engineering (ICDE'98)*, pages 402–411, Orlando, Florida, USA, February 1998.

[BAG99] Roberto J. Bayardo, Rakesh Agrawal, and Dimitrios Gunopulos. Constraint based rule mining in large, dense databases. In *Proceedings of* 15th *Intl. Conf. on Data Engineering (ICDE'99)*, pages 188–197, Sydney, Australia, March 1999.

[Bay98] Roberto J. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'98)*, pages 85–93, Seattle, Washington, USA, June 1998.

[BMS97] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'97)*, pages 265–276, Tucson, Arizona, USA, May 1997.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Sergey Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'97)*, pages 255–264, Tucson, Arizona, USA, May 1997.

[CFCK98] C. H. Cai, Ada Wai-Chee Fu, C. H. Cheng, and W. W. Kwong. Mining association rules with weighted items. In *Proceedings of 1998 Intl. Database Engineering and Applications Symposium (IDEAS'98)*, pages 68–77, Cardiff, Wales, UK, July 1998.

[CHNW96] David Wai-Lok Cheung, Jiawei Han, Vincent T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental update technique. In *Proceedings of*

12th *Intl. Conf. on Data Engineering (ICDE'96)*, pages 106–114, New Orleans, Louisiana, USA, February 1996.

[CHY96] Ming-Syan Chen, Jiawei Han, and Philip S. Yu. Data mining: An overview from database perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, 1996.

[CLK97] David Wai-Lok Cheung, Sau Dan Lee, and Benjamin Kao. A general incremental technique for maintaining discovered association rules. In *Proceedings of the* 5th *Intl. Conf. on Database Systems for Advanced Applications (DASFAA'97)*, pages 185–194, Melbourne, Australia, April 1997.

[CNFF96] David Wai-Lok Cheung, Vincent Ng, Ada Wai-Chee Fu, and Yongjian Fu. Efficient mining of association rules in distributed databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):911–922, 1996.

[DLM+98] Gautam Das, King-Ip Lin, Heikki Mannila, Gopal Renganathan, and Padhraic Smyth. Rule discovery from time series. In *Proceedings of the* 4th *Intl. Conf. on Knowledge Discovery and Data Mining (KDD'98)*, pages 16–22, New York City, New York, USA, August 1998.

[Fay98] Usama Fayyad. Mining databases: Towards algorithms for knowledge discovery. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 21(1):39–48, March 1998.

[FL96] Scott Fortin and Ling Liu. An object-oriented approach to multilevel association rule mining. In *Proceedings of* 5th *Intl. Conf. on Information and Knowledge Management (CIKM'96)*, pages 65–72, Rockville, Maryland, USA, November 1996.

[FMMT96] Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama. Mining optimized association rules for numeric attributes. In *Proceedings of* 15th *Symposium on Principles of Database Systems (PODS'96)*, pages 182–191, Montreal, Canada, June 1996.

[FPSS96a] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery: An overview. In Usama Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramaswamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 1–31. AAAI/MIT Press, 1996.

[FPSS96b] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. Knowledge discovery and data mining: Towards a unifying framework. In *Proceedings of* 2nd *Intl. Conf. on Knowledge Discovery and Data Mining (KDD'96)*, pages 82–88, Portland, Oregon, USA, August 1996.

[FWS+98] Ada Wai-Chee Fu, Man Hong Wong, Siu Chun Sze, Wai Chiu Wong, Wai Lun Wong, and Wing Kwan Yu. Finding fuzzy sets for the mining of fuzzy association rules for numerical attributes. In *Proceedings of* 1st *Intl. Symposium on Intelligent Data Engineering and Learning (IDEAL'98)*, pages 263–268, October 1998.

[GWS98] Valery Guralnik, Duminda Wijesekera, and Jaideep Srivastava. Pattern directed mining of sequence data. In *Proceedings of the* 4th *Intl. Conf. on Knowledge Discovery and Data Mining (KDD'98)*, pages 51–57, New York City, New York, USA, August 1998.

[HDY99] Jiawei Han, Guozhu Dong, and Yiwen Yin. Efficient mining of partial periodic patterns in time series database. In *Proceedings of* 15th *Intl. Conf. on Data Engineering (ICDE'99)*, pages 106–115, Sydney, Australia, March 1999.

[HF95] Jiawei Han and Yongjian Fu. Discovery of multiple level association rules from large databases. In *Proceedings of* 21st *Intl. Conf. on Very Large Databases (VLDB'95)*, pages 420–431, Zurich, Switzerland, September 1995.

[Hid99] Christian Hidber. Online association rule mining. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'99)*, Philadelphia, PA, USA, May–June 1999.

[HKMT95] Marcel Holsheimer, Martin Kersten, Heikki Mannila, and Hannu Toivonen. A perspective on databases and data mining. In *Proceedings of* 1st *Intl. Conf. on Knowledge Discovery and Data Mining (KDD'95)*, pages 150–155, Montreal, Canada, August 1995.

[HP96] Jia Liang Han and Ashley W. Plank. Background for association rules and cost estimate of mining algorithms. In *Proceedings of* 5th *Intl. Conf. on Information and Knowledge Management (CIKM'96)*, pages 73–80, Rockville, Maryland, USA, November 1996.

[HS95] Maurice Houtsma and Arun Swami. Set-oriented mining of association rules in relational databases. In *Proceedings of* 11th *Intl. Conf. on Data Engineering (ICDE'95)*, Taipei, Taiwan, March 1995.

[KFW98] Chan Man Kuok, Ada Wai-Chee Fu, and Man Hon Wong. Mining fuzzy association rules in databases. *SIGMOD Record*, 27(1):41–46, 1998.

[KLKF98] Flip Korn, Alexandros Labrinidis, Yannis Kotidis, and Christos Faloutsos. Ratio rules: A new paradigm for fast, quantifiable data mining. In *Proceedings of* 24th *Intl. Conf. on Very Large Databases (VLDB'98)*, pages 582–593, New York City, New York, USA, August 1998.

[KMR+94] Mika Klemettinen, Heikki Mannila, Pirjo Ronkainen, Hannu Toivonen, and A. Inkeri Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proceedings of* 3rd *Intl. Conf. on Information and Knowledge Management (CIKM'94)*, pages 401–407, Gaithersburg, Maryland, USA, November 1994.

[LNHP99] Laks V. S. Lakshmanan, Raymond T. Ng, Jiawei Han, and Alex Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'99)*, Philadelphia, PA, USA, May–June 1999.

[Man96] Heikki Mannila. Data mining: Machine learning, statistics, and databases. In *Proceedings of* 8th *Intl. Conf. on Scientific and Statistical Database Management (SSDBM'96)*, pages 2–9, Stockholm, Sweden, June 1996.

[Man97] Heikki Mannila. Methods and problems in data mining. In *Proceedings of* 6th *Intl. Conf. on Database Theory (ICDT'97)*, pages 41–55, Delphi, Greece, January 1997.

[MT96] Heikki Mannila and Hannu Toivonen. Discovering generalized episodes using minimal occurrences. In *Proceedings of* 2nd *Intl. Conf. on Knowledge Discovery and Data Mining (KDD'96)*, pages 146–151, Portland, Oregon, USA, August 1996.

[MTV94] Heikki Mannila, Hannu Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *Proceedings of AAAI'94 Workshop on Knowledge Discovery in Databases (KDD'94)*, pages 181–192, Seattle, Washington, USA, July 1994.

[MTV95] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovering frequent episodes in sequences. In *Proceedings of* 1st *Intl. Conf. on Knowledge Discovery and Data Mining (KDD'95)*, pages 210–215, Montreal, Canada, August 1995.

[NLHP98] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'98)*, pages 13–24, Seattle, Washington, USA, June 1998.

[ORS98] Banu Özden, Sridhar Ramaswamy, and Avi Silberschatz. Cyclic association rules. In *Proceedings of* 14th *Intl. Conf. on Data Engineering (ICDE'98)*, pages 412–421, Orlando, Florida, USA, February 1998.

[OS98] Edward Omiecinski and Ashoka Savasere. Efficient mining of association rules in large dynamic databases. In *Proceedings of* 16th *British National Conference on Databases (BNCOD'98)*, pages 49–63, Cardiff, Wales, UK, July 1998.

[PCY95a] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'95)*, pages 175–186, San Jose, California, USA, May 1995.

[PCY95b] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. Efficient parallel data mining for association rules. In *Proceedings of* 4th *Intl. Conf. on Information and Knowledge Management (CIKM'95)*, pages 31–36, Baltimore, Maryland, USA, November 1995.

[PCY97] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. Mining association rules with adjustable accuracy. In *Proceedings of* 6th *Intl. Conf. on Information and Knowledge Management (CIKM'97)*, pages 151–160, Las Vegas, Nevada, USA, November 1997.

[RMS98] Sridhar Ramaswamy, Sameer Mahajan, and Avi Silberschatz. On the discovery of interesting patterns in association rules. In *Proceedings of* 24th *Intl. Conf. on Very Large Databases (VLDB'98)*, pages 368–379, New York City, New York, USA, August 1998.

[RS98] Rajeev Rastogi and Kyuseok Shim. Mining optimized association rules with categorical and numerical attributes. In *Proceedings of* 14th *Intl. Conf. on Data Engineering (ICDE'98)*, pages 503–512, Orlando, Florida, USA, February 1998.

[SA95] Ramakrishnan Srikant and Rakesh Agrawal. Mining generalized association rules. In *Proceedings of* 21st *Intl. Conf. on Very Large Databases (VLDB'95)*, pages 407–419, Zurich, Switzerland, September 1995.

[SA96a] Ramakrishnan Srikant and Rakesh Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'96)*, pages 1–12, Montreal, Canada, June 1996.

[SA96b] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of* 5th *Intl. Conf. on Extending Database Technology (EDBT'96)*, pages 3–17, Avignon, France, March 1996.

[SON95] Ashoka Savasere, Edward Omiecinski, and Shamkant Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of* 21st *Intl. Conf. on Very Large Databases (VLDB'95)*, pages 432–444, Zurich, Switzerland, September 1995.

[SON98] Ashoka Savasere, Edward Omiecinski, and Shamkant Navathe. Mining for strong negative associations in a large database of customer transactions. In *Proceedings of* 14th *Intl. Conf. on Data Engineering (ICDE'98)*, pages 494–502, Orlando, Florida, USA, February 1998.

[SS98a] N. L. Sarda and N. V. Srinivas. An adaptive algorithm for incremental mining of association rules. In *Proceedings of* 9th *Intl. Conf. on Database and Expert Systems Applications Workshop (DEXA Workshop'98)*, pages 240–245, Vienna, Austria, August 1998.

[SS98b] Li Shen and Hong Shen. Mining flexible multiple–level association rules in all concept hierarchies. In *Proceedings of* 9th *Intl. Conf. on Database and Expert Systems Applications (DEXA'98)*, pages 786–795, Vienna, Austria, August 1998.

[ST95] Avi Silberschatz and Alexander Tuzhilin. On subjective measures of interestingness on knowledge discovery. In *Proceedings of* 1st *Intl. Conf. on Knowledge Discovery and Data Mining (KDD'95)*, pages 275–281, Montreal, Canada, August 1995.

[ST96a] Avi Silberschatz and Alexander Tuzhilin. User-assisted knowledge discovery: How much should the user be involved. In *Proceedings of SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'96)*, Montreal, Canada, June 1996.

[ST96b] Avi Silberschatz and Alexander Tuzhilin. What makes patterns interesting in knowledge discovery systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):970–984, December 1996.

[SVA97] Ramakrishnan Srikant, Quoc Vu, and Rakesh Agrawal. Mining association rules with item constraints. In *Proceedings of the* 3rd *Intl. Conf. on Knowledge Discovery and Data Mining (KDD'97)*, pages 67–73, Newport Beach, California, USA, August 1997.

[TBAR97] Shiby Thomas, Sreenath Bodagala, Khaled Alsabti, and Sanjay Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proceedings of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining (KDD'97)*, pages 263–266, Newport Beach, California, USA, August 1997.

[TKR+95] Hannu Toivonen, Mika Klemettinen, Pirjo Ronkainen, H. Katonen, and Heikki Mannila. Pruning and grouping discovered association rules. In *Proceedings of MLNet Workshop on Statistics, Machine Learning and Discovery in Databases*, pages 47–52, Heraklion, Crete, Greece, April 1995.

[Toi96] Hannu Toivonen. Sampling large databases for association rules. In *Proceedings of 22nd Intl. Conf. on Very Large Databases (VLDB'96)*, pages 134–145, Mumbay, India, September 1996.

[TS98] Shiby Thomas and Sunita Sarawagi. Mining generalized association rules and sequential patterns using sql queries. In *Proceedings of the 4th Intl. Conf. on Knowledge Discovery and Data Mining (KDD'98)*, pages 344–348, New York City, New York, USA, August 1998.

[WTL98] Ke Wang, Soon Hock William Tay, and Bing Liu. Interestingness–based interval merger for numeric association rules. In *Proceedings of the 4th Intl. Conf. on Knowledge Discovery and Data Mining (KDD'98)*, pages 121–128, New York City, New York, USA, August 1998.

[ZO98] Mohammed Javeed Zaki and Mitsunori Ogihara. Theoretical foundations of association rules. In *Proceedings of 3rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'98)*, Seattle, Washington, USA, June 1998.

[ZPLO97] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Wei Li, and Mitsunori Ogihara. Evaluation of sampling for data mining of association rules. In *Proceedings of the 7th Intl. Conf. Workshop on Research Issues in Data Engineering (RIDE'97)*, Birmingham, UK, April 1997.

[ZPOL97a] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In *Proceedings of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining (KDD'97)*, pages 283–286, Newport Beach, California, USA, August 1997.

[ZPOL97b] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.