<table>
<thead>
<tr>
<th>Title</th>
<th>A general incremental technique for maintaining discovered association rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>Cheung, DW; Lee, SD; Kao, B</td>
</tr>
<tr>
<td>Citation</td>
<td>The 5th International Conference on Database Systems for Advanced Applications, Melbourne, Australia, 1-4 April 1997. In Conference Proceedings, 1997, p. 185-194</td>
</tr>
<tr>
<td>Issued Date</td>
<td>1997</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hdl.handle.net/10722/57231">http://hdl.handle.net/10722/57231</a></td>
</tr>
<tr>
<td>Rights</td>
<td>This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.; ©1997 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.</td>
</tr>
</tbody>
</table>
A General Incremental Technique for Maintaining Discovered Association Rules

David W. Cheung  S.D. Lee  Benjamin Kao
Department of Computer Science
The University of Hong Kong
Pokfulam Road, Hong Kong
\{dcheung,sdleekao\}@cs.hku.hk

Abstract
A more general incremental updating technique is developed for maintaining the association rules discovered in a database in the cases including insertion, deletion, and modification of transactions in the database. A previously proposed algorithm FUP can only handle the maintenance problem in the case of insertion. The proposed algorithm FUP2 makes use of the previous mining result to cut down the cost of finding the new rules in an updated database. In the insertion only case, FUP2 is equivalent to FUP. In the deletion only case, FUP2 is a complementary algorithm of FUP which is very efficient when the deleted transactions is a small part of the database, which is the most applicable case. In the general case, FUP2 can efficiently update the discovered rules when new transactions are added to a transaction database, and obsolete transactions are removed from it. The proposed algorithm has been implemented and its performance is studied and compared with the best algorithms for mining association rules studied so far. The study shows that the new incremental algorithm is significantly faster than the traditional approach of mining the whole updated database.

Keywords: Association Rules, Data Mining, Knowledge Discovery, Large Databases, Maintenance.

1 Introduction
In recent years, data mining has attracted much attention in database research. This is due to its wide applicability in many areas, including the retail industry and the finance sector [6]. The availability of automated tools has enabled the collection of large amount of data. These large databases contain information that is potentially useful for making market strategies and financial forecasts. Data mining is the task to find out such useful information from large databases. The information includes association rules, characteristic rules, classification rules, generalized relations, discriminant rules, etc. [9]

Of the various data mining problems, mining of association rules is an important one [3]. A classical example is about the retail industry. Typically, a record in the sales data describes all the items that are bought in a single transaction, together with other information such as the transaction time, customer-id, etc. Mining association rules from such a database is to find out, from the huge amount of past transactions, all the rules like “A customer who buys item X and item Y is also likely to buy item Z in the same transaction”, where X, Y and Z are initially unknown. Such rules are very useful for marketers to develop and implement customized marketing programs and strategies.

Recently, many interesting works have been published in association rules mining including mining of quantitative association rules and multi-level association rules, and parallel and distributed mining of association rules [2, 3, 4, 5, 8, 10, 12, 13, 14].

A feature of data mining problems is that in order to have stable and reliable results, a giant amount (often of the order of gigabytes) of data has to be collected and analyzed. The large amount of input data and mining results poses a maintenance problem. While new transactions are being appended to a database and obsolete ones are being removed, rules or patterns already discovered also have to be updated. In this paper we examine the problem of maintaining discovered association rules. We propose a new incremental algorithm FUP2 in this paper which can efficiently handle all the update cases including insertion, deletion and modification of transactions.

Previous works
The problem of mining association rules was first introduced in [2]. In that paper it was shown that the problem can be decomposed into two subproblems:


185
1. Find out all large itemsets, which are the sets of items that are contained in a sufficiently large number of transactions, with respect to a threshold minimum support.

2. From the set of large itemsets found, find out all the association rules that have a confidence value exceeding a threshold minimum confidence.

Since the solution to the second subproblem is straightforward [3], major research efforts have been spent on the first subproblem. Among the algorithms proposed to solve the first subproblem efficiently, the Apriori [3] and DHP [12] algorithms are the most successful. The Apriori algorithm finds out the large itemsets iteratively. In each iteration, it generates a number of candidate large itemsets and then verify them by scanning the database. The key success is the use of the apriori_gen function to generate a small number of candidate itemsets. DHP improves over Apriori by further reducing the number of candidate itemsets using a hashing technique.

While both Apriori and DHP efficiently discover association rules from a database, the rules maintenance problem is not addressed. The problem of maintaining association rules is first studied in [4]. That paper proposes the FUP algorithm, which can update the association rules in a database when new transactions are added to the database. It is based on the framework of Apriori and it also finds new large itemsets iteratively. The idea is to store the counts of all the large itemsets found in a previous mining operation. Using these stored counts and examining the newly added transactions, the algorithm can generate a very small number of candidate new large itemsets. The overall count of these candidate itemsets are then obtained by scanning the original database. Consequently, all new large itemsets are found. The FUP algorithm is very efficient. However, the algorithm does not handle the case of deleting transactions from the database. The modification of transactions is not addressed, either.

In this paper we propose a new algorithm, called FUP2, that can update the existing association rules when transactions are added and deleted from the database. It is a generalization of the FUP algorithm [4]. Like FUP, FUP2 makes use of the previous mining result to cut down the amount of work that has to be done to discover the new set of rules.

The remaining of this paper is organized as follows. Section 2 gives a detailed description of the problem. The new algorithm is described in Section 3. A performance study of FUP2 is presented in Section 4. We discuss some implementation issues of FUP2 in Section 5 and this paper is ended with a conclusion in Section 6.

2 Problem Description

2.1 Mining of association rules

Let \( I = \{i_1, i_2, \ldots, i_m\} \) be a set of literals, called items. Let \( D \) be a database of transactions, where each transaction \( T \) is a set of items such that \( T \subseteq I \). For a given itemset \( X \subseteq I \) and a given transaction \( T \), we say that \( T \) contains \( X \) if and only if \( X \subseteq T \). The support count of an itemset \( X \) is defined to be \( \sigma_X = \text{number of transactions in } D \text{ that contain } X \). We say that an itemset \( X \) is large, with respect to a support threshold of \( s\% \), if \( \sigma_X \geq |D| \times s\% \), where \(|D|\) is the number of transactions in the database \( D \). An association rule is an implication of the form \( "X \Rightarrow Y" \), where \( X \subseteq I \), \( Y \subseteq I \) and \( X \cap Y = \emptyset \). The association rule \( X \Rightarrow Y \) is said to hold in the database \( D \) with confidence \( c\% \) if no less than \( c\% \) of the transactions in \( D \) that contain \( X \) also contain \( Y \). The rule \( X \Rightarrow Y \) has support \( s\% \) in \( D \) if \( \sigma_{X \cup Y} = |D| \times s\% \). For a given pair of confidence and support thresholds, the problem of mining association rules is to find out all the association rules that have confidence and support greater than the corresponding thresholds. This problem can be reduced to the problem of finding all large itemsets for the same support threshold [2].

Thus, if \( s\% \) is the given support threshold, the mining problem is reduced to the problem of finding the set \( L = \{X \subseteq I \land \sigma_X \geq |D| \times s\% \} \). For the convenience of subsequent discussions, we call an itemset that contains exactly \( k \) items a \( k \)-itemset. We use the symbol \( L_k \) to denote the set of all \( k \)-itemsets in \( L \).

2.2 Update of association rules

After some update activities, old transactions are deleted from the database \( D \) and new transactions are added. We can treat the modification of existing transactions as deletion followed by insertion. Let \( \Delta^- \) be the set of deleted transactions and \( \Delta^+ \) be the set of newly added transactions. We assume, naturally, that \( \Delta^- \subseteq D \). Denote the updated database by \( D' \). Note that \( D' = (D - \Delta^-) \cup \Delta^+ \).

We denote the set of unchanged transactions by \( D^- = D - \Delta^- = D' - \Delta^+ \).

<table>
<thead>
<tr>
<th>database</th>
<th>support count of itemset ( X )</th>
<th>Large ( k )-itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta^+ )</td>
<td>( \delta_X^+ )</td>
<td>—</td>
</tr>
<tr>
<td>( D^- )</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>( \Delta^- )</td>
<td>( \delta_X^- )</td>
<td>—</td>
</tr>
<tr>
<td>( D = \Delta^- \cup D^- )</td>
<td>( \sigma_X )</td>
<td>( L_k )</td>
</tr>
<tr>
<td>( D' = D^- \cup \Delta^+ )</td>
<td>( \sigma_X' )</td>
<td>( L'_k )</td>
</tr>
</tbody>
</table>

Table 1: Definitions of several symbols

As defined in the previous subsection, \( \sigma_X \) is the support count of itemset \( X \) in the original database \( D \). The set of large itemsets in \( D \) is \( L \).
and $L_k$ is the set of $k$-itemsets in $L$. Define $\sigma'_X$ to be the new support count of an itemset $X$ in the updated database $D'$, and $L'$ to be the set of large itemsets in $D'$. $L'_k$ is the set of $k$-itemsets in $L'$. We further define $\delta'_X$ to be the support count of itemset $X$ in the database $D'$ and $\delta_X$ to be that of $\Delta^-$. These definitions are summarized in Table 1. We define $\delta_X = \delta'_X - \delta_X$, which is the change of support count of itemset $X$ as a result of the update activities. Thus, we have:

**Lemma 1** \[ \sigma'_X = \sigma_X + \delta'_X - \delta_X = \sigma_X + \delta_X \]

**Proof.** By definition. \qed

As the result of a previous mining on the old database $D$, we have already found $L$ and $\sigma_X \forall X \in L$. Thus, the update problem is to find $L'$ and $\sigma'_X \forall X \in L'$ efficiently, given the knowledge of $D$, $D'$, $\Delta^-$, $\Delta^+$, $L$ and $\sigma_X \forall X \in L$.

### 3 The FUP2 algorithm

In this section, we introduce the FUP2 algorithm step by step. We first focus on the special case for transaction deletion only ($\delta^+ = \emptyset$). This special case can be considered as a complement of the FUP algorithm [4], which handles transaction insertion only. Next, we generalize the deletion algorithm to handle the case for transaction deletion as well as insertion ($|\Delta^+| \geq 0$).

#### 3.1 The special case for transaction deletion only ($\delta^+ = \emptyset$)

For the delete-only case, we have $\Delta^+ = \emptyset$ and hence $D' = D^- \cup \emptyset = D^- = D - \Delta^-$. Note also that $\delta_X = 0 \forall X \subseteq I$.

To discover the large itemsets in the updated database $D'$, the FUP2 algorithm executes iteratively. In the $k$-th iteration, all the large $k$-itemsets in $D'$ are found as follows. As in Apriori [3], we form a set of candidates $C_k$ which is a superset of $L_k$. In the first iteration, $C_1$ is exactly the set $I$. In subsequent iterations, $C_k$ is calculated from $L'_{k-1}$, the large itemsets found in the previous iteration, using the same apriori-gen function as in Apriori [3]. All the itemsets in $L'_k$ are guaranteed to be contained in $C_k$.

Next, we use the old large $k$-itemsets $L_k$ from the previous mining result to divide the candidate set $C_k$ into 2 parts: $P_k = C_k \cap L_k$ and $Q_k = C_k - P_k$.

In words, $P_k$ ($Q_k$) is the set of candidate itemsets that are previously large (small) with respect to $D$. Again, our goal is to select those itemsets that are currently large (w.r.t. $D'$). We treat the candidates in these two partitions separately.

With this partitioning, for all candidates $X \in P_k$, we already know its support count $\sigma_X$ from the previous mining results. We find out $\delta'_X$ by scanning $\Delta^-$. Then, we can obtain the updated support count $\sigma'_X$ using Lemma 1. Thus, a candidate $X$ from $P_k$ goes to $L'_k$ if and only if $\sigma'_X \geq |D'| \times s\%$.

For the candidates in $Q_k$, we only know that they were not large in the original database $D$. We do not know their support counts. However, since they were not large, we know that $\sigma_X < |D| \times s\% \forall X \in Q_k$. We can make use of this information to tell which candidates from $Q_k$ may be large and which will not.

**Lemma 2** If $X \notin L$ and $\delta_X = 0$, then $\sigma'_X < |D'| \times s\%$ if $\sigma_X \geq |\Delta^-| \times s\%$.

**Proof.** Since $X \notin L$, we have $\sigma_X < |D| \times s\%$. Hence, $\sigma'_X = \sigma_X - \delta_X + \delta'_X < |D| \times s\% - |\Delta^-| \times s\% + 0 = (|D| - |\Delta^-|) \times s\% = |D'| \times s\%$.

That is to say, for each candidate in $Q_k$, if it is large in $\Delta^-$, then it cannot be large in $L'_k$. We first scan $\Delta^-$ and obtain $\delta_X$ for each $X \in Q_k$. Then, we delete those candidates for which $\delta_X \geq |\Delta^-| \times s\%$, thus leaving in $Q_k$ those that are small in $\Delta^-$. Note that we are not checking all small itemsets in $\Delta^-$. We are only checking for those small itemsets of $\Delta^-$ that are in $Q_k$. The number of such itemsets is not large, as $Q_k \subseteq C_k$. For the candidates $X$ that remain in $Q_k$, we scan $D^-$ to obtain their new support counts $\sigma'_X$. Finally, we add to $L'_k$ those candidates $X$ from $Q_k$ for which $\sigma'_X \geq |D'| \times s\%$.

Thus, we have discovered which candidates from $P_k$ and $Q_k$ are large and put them into $L'_k$. Moreover, we have also found $\sigma'_X$ for each $X \in L'_k$. We have completed one iteration. In the subsequent iterations, large itemsets of larger sizes are discovered. The iterations go on until either $C_k = \emptyset$ or $|L'_k| < k + 1$ for some $k$. The steps of the $k$th iteration are summarized as follows.

1. Obtain a candidate set $C_k$ of itemsets. Halt if $C_k = \emptyset$.
2. Partition $C_k$ into $P_k$ and $Q_k$, where $P_k = C_k \cap L_k$ and $Q_k = C_k - P_k$.
3. Scan $\Delta^-$ to find out $\delta_X$ for each $X \in P_k \cup Q_k$.
4. For each $X \in P_k$, Calculate $\sigma'_X$.
5. Delete from $Q_k$ those candidates $X$ where $\delta_X \geq |\Delta^-| \times s\%$. (Application of Lemma 2.)
6. Scan $\Delta^-$ to find out $\sigma'_X$ of the remaining candidates $X \in Q_k$.
7. Add to $L'_k$ those candidates $X$ from $P_k \cup Q_k$ for which $\sigma'_X \geq |D'| \times s\%$.
8. Halt if $|L'_k| < k + 1$.

As found in previous works, the speed of the Apriori algorithm depends very much on the size of the candidate set. To improve performance, our FUP2 algorithm makes use of the information $L_k$ and $\sigma_X \forall X \in L_k$ to reduce the size of the candidate set. It scans the updated database $D'$ with a candidate set ($Q_k \subseteq C_k$) which is significantly smaller than that ($C_k$) of Apriori.
(see Section 4). The algorithm, however, does scan the deleted transactions \( \Delta^- \) with the same number of candidates as Apriori does. We can reduce this number of candidates by the following optimization.

### 3.2 An optimization

Observe that in step 5 of the FUP2 algorithm, we are removing from \( Q_k \) those candidates which are \textit{large} in \( \Delta^- \). Those that are \textit{small} in \( \Delta^- \) always remain. Now, there is a way to determine whether a candidate itemset \( X \) is \textit{small} prior to knowing \( \delta_X \), thus saving the work of finding the value of \( \delta_X \).

**Lemma 3** For any itemsets \( X \) and \( Y \) such that \( X \subseteq Y \), \( \delta_X \geq \delta_Y \).

**Proof.** Any deleted transaction that contains \( Y \) must also contain \( X \).

**Corollary 1** All supersets of a small 1-itemset are small.

Hence, if we remember which 1-itemsets are small in \( \Delta^- \) during the first iteration, then in the subsequent iterations, we can quickly determine if a candidate from \( Q_k \) is small in \( \Delta^- \) without finding its support count in \( \Delta^- \). Thus, we can optimize the above algorithm by adding the following steps:

1. For each candidate \( \delta_X \) of \( Q_k \), if \( \delta_X \) contains any item which is a \textit{small} 1-itemset in \( \Delta^- \), move it to the set \( R_k \). All the candidates so moved to \( R_k \) are those that are small in \( \Delta^- \) by corollary 1.
2. Move all candidates from \( R_k \) to \( Q_k \).

This modification significantly reduces the number of candidates during the scan of \( \Delta^- \). The only additional cost is to remember the set of \textit{small} 1-itemset in the first iteration. This requires extra memory space of size linear to \( |I| \). The extra CPU time required is negligible, since we have to find \( \delta_X \) for all 1-itemsets \( X \) anyway. So, the additional cost of this optimization is relatively inexpensive. The number of the candidates for scanning \( \Delta^- \) is not affected by this optimization, but the number of candidates for scanning \( \Delta^- \) is significantly reduced. So, this optimization speeds up the performance of the algorithm at negligible cost.

Let us illustrate this special case of the FUP2 algorithm with the example shown in Table 2. The original database \( D \) contains 5 transactions and we set the support threshold to 25%. So, itemsets with a support count \( \sigma_X \) no less than \( 5 \times 25\% = 1.25 \) are large. The large itemsets in \( L \) are shown in the same table. For convenience, we write \( XYZ \) for the itemset \( \{X, Y, Z\} \) when no ambiguity arises. One transaction is deleted, leaving 4 transactions in the final database \( D' \). Now, let us apply the FUP2 algorithm to see how \( L' \) is generated.

### Transactions: \( (I = \{A, B, C, D, E\}) \)

\[
\begin{array}{c|ccc|c}
\hline
\Delta^- & A & B & E \\
\hline
D^- & A & B & C \\
& A & B & D \\
& & B & D \\
& C & D \\
\hline
\end{array}
\]

### Large itemsets (support threshold \( s = 25\% \))

\( D = \Delta^- \cup D^- \)

<table>
<thead>
<tr>
<th>( \sigma_X )</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>AB</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_X )</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 2:** An example for \( \Delta^+ = \emptyset \)

In the first iteration, \( C_1 = I = \{A, B, C, D, E\} \). Of these candidates, only \( E \) was not large in \( D \). So, after partitioning, \( P_1 = \{A, B, C, D\} \) and \( Q_1 = \{E\} \). Next, we scan \( \Delta^- \) and update the support counts of the candidates in \( P_1 \). Only \( A \) and \( B \) occur in \( \Delta^- \). So, the counts are updated as \( \sigma'_A = \sigma'_B = 2, \sigma'_D = 3 \). In the same scan of \( \Delta^- \), we find out that \( \delta_E = 1 \geq 0.25 = |\Delta^-| \times 25\% \). Hence, \( \sigma_X \) is large in \( \Delta^- \), and it was \textit{small} in \( D \). It cannot be \textit{large} in \( D' \) by Lemma 2. So, it is removed from \( Q_1 \). This leaves \( Q_1 \) empty, hence we need not scan \( D^- \) at all in this iteration. Since all the remaining candidates in \( P_1 \cup Q_1 \) have a support count in \( D^- \) no less than \( |D'| \times 25\% = 1 \), they all fall into \( L_1 \). We remember that \( C \) and \( D \) are small in \( \Delta^- \) for optimization.

In the second iteration, we first obtain \( C_2 \) by applying the \textsf{apriori-gen} function on \( L_1' \). This gives \( C_2 = \{AB, AC, AD, BC, BD, CD\} \), of which only \( AB \) was large in \( D \). So, the partitioning results in \( P_2 = \{AB\} \) and \( Q_2 = \{AC, AD, BC, BD, CD\} \). Next, we scan \( \Delta^- \) and update the count \( \sigma'_{AB} = \sigma'_{AC} = 2 - 1 = 1 \). All the candidates in \( Q_2 \) contain either item \( C \) or \( D \), which are small in \( \Delta^- \). So, we know that all the candidates in \( Q_2 \) are definitely small in \( \Delta^- \) (corollary 1) and hence potentially large in \( D' \) (Lemma 2). There is no need to find out \( \delta_X \) for these candidates \( X \in Q_2 \). So, the next job is to scan \( D^- \) to obtain \( \sigma'_X \) for the candidates in \( Q_2 \). This gives \( \sigma'_{AC} = \sigma'_{AD} = \sigma'_{BC} = \sigma'_{BD} = \sigma'_{CD} = 1 \). Consequently, \( AB, AC, AD, BC, BD, CD \) are large in \( D' \) and hence are included in \( L_2' \).

In the third iteration, \textsf{apriori-gen} gives a candidate set \( C_3 = \{ABC, ABD, ABCD, BCD\} \). None of these candidates were large. So, \( P_3 = \emptyset \) and \( Q_3 = C_3 \). Since all the candidates contain item \( C \) or item \( D \), we know that they are all small in \( \Delta^- \) without having to find out their support counts in \( \Delta^- \). So, there is no need to scan \( \Delta^- \) in this iteration! We only have to scan \( D^- \) to obtain the support counts in \( D' \) for the candidates. The results are \( \sigma'_{ABC} = 1, \sigma'_{ABD} = \sigma'_{ACD} = \sigma'_{BCD} = 0 \). Only \( ABC \) goes to \( L_3' \).
There is only 1 large itemset found in this iteration. This is insufficient to generate any candidates in the next iteration. Hence, the algorithm stops after 3 iterations.

Table 3 compares the size of candidates when Apriori is applied on \( D' \) and when FUP2 is employed. While Apriori scans \( D' \) three times, with a total of 15 candidate itemsets, FUP2 scans \( D' \) only twice, with a total of 9 candidates only. Although FUP2 has to scan \( \Delta^- \) with 6 candidate sets, the time spent on this is insignificant, as \( |D^-| \gg |\Delta^-| \) in most practical applications. In our example, FUP2 has reduced the number of candidates for scanning \( D^- \) by \( \frac{15 - 9}{15} = 40\% \) — a significant improvement.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Apriori</th>
<th>FUP2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>scan ( D^- )</td>
<td>scan ( D^- )</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>15</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 3: Size of candidates for the example

3.3 The general case for transaction deletion and insertion

Now, we extend the algorithm introduced in the previous subsection to handle the general case for transaction insertion as well as deletion. We no longer assume that \( \Delta^+ = \emptyset \). So, \( \delta_X^+ \) may be positive for any \( X \subseteq I \). Consequently, Lemma 2 cannot be applied and corollary 1 is no longer useful.

As before, we find out \( L'_k \) and \( \sigma_X' \forall X \in L'_k \) in the \( k \)-th iteration. In each iteration, we first form a candidate set \( C_k \) and then partition it into two parts \( P_k \) and \( Q_k \) before. Again, for each candidate \( X \in P_k \), we know \( \sigma_X \) from the previous mining result. So, we only have to scan \( \Delta^- \) and \( \Delta^+ \) to update the support count for the candidates in \( P_k \). In the FUP2 algorithm, we choose to scan \( \Delta^- \) to find \( \delta_X \) for each candidate \( X \) first. As we scan \( \Delta^+ \), we can deduct the support count at the same time, and remove a candidate from \( P_k \) as soon as its support count drops below \( |D'| \times s\% - |\Delta^+| \). This is because such a candidate has no hope to have \( \sigma_X' \geq |D'| \times s\% \), as \( \delta_X^+ \leq |\Delta^+| \). Next, we scan \( \Delta^- \) to find \( \delta_X^- \) for each candidate \( X \) that remains in \( P_k \). Finally, we calculate \( \sigma_X' \) for each candidate in \( P_k \) using Lemma 1, and add those with \( \sigma_X' \geq |D'| \times s\% \) to \( L'_k \).

For the candidates \( X \in Q_k \), again we do not know \( \sigma_X \), but we know that \( \sigma_X < |D| \times s\% \). By a generalization of Lemma 2, we are able to prune some candidates from \( Q_k \) without knowing their counts in \( \Delta^- \).

Lemma 4 If \( X \notin L' \) and \( \delta_X = \delta_X^+ - \delta_X^- \leq |\Delta^+| - |\Delta^-| \times s\% \), then \( X \notin L' \).

Proof. If \( X \notin L' \), then \( \sigma_X < |D| \times s\% \). Hence, \( \sigma_X' = \sigma_X + (\delta_X^+ - \delta_X^-) < |D| \times s\% + (|\Delta^+| - |\Delta^-|) \times s\% \). Thus \( X \notin L' \) by the definition of \( L' \).

So, for each candidate \( X \in Q_k \), we obtain the values of \( \delta_X^+ \) and \( \delta_X^- \) during the scans of \( \Delta^+ \) and \( \Delta^- \). Then, we calculate \( \delta_X \) and remove those with \( \delta_X \leq (|\Delta^+| - |\Delta^-|) \times s\% \), because Lemma 4 tells us that they will not fall into \( L'_k \). For the remaining candidate in \( Q_k \), we scan \( D^- \) and obtain their support counts in \( D^- \). Adding this count to \( \delta_X^+ \) gives \( \sigma_X' \). We add those candidates with \( \sigma_X' \geq |D'| \times s\% \) from \( Q_k \) to \( L'_k \). This finishes the iteration. This algorithm scans \( \Delta^- \) and \( \Delta^+ \) with a candidate set of size \( |P_k \cup Q_k| = |C_k| \), the same as Apriori does. However, it scans \( D^- \) with a candidate set much smaller than the initial size of \( Q_k \subseteq C_k \), thereby saving time, assuming that \( |D^-| \gg |\Delta^-| \). The algorithm saves a lot of time when compared to Apriori.

3.4 Optimizations

In the algorithm described in the above paragraphs, the databases \( \Delta^- \) and \( \Delta^+ \) have to be scanned with a candidate set of size \( |C_k| \). As an improvement to the algorithm, we can reduce this size by finding bounds on the values of \( \delta_X^+ \) and \( \delta_X^- \) for each candidate \( X \) prior to the scans of \( \Delta^- \) and \( \Delta^+ \). The idea comes from an extension of Lemma 3.

Lemma 5 For any itemsets \( X \) and \( Y \) such that \( X \subseteq Y \), \( \delta_X^- \geq \delta_Y^- \) and \( \delta_X^+ \geq \delta_Y^+ \).

Proof. Any transaction in \( \Delta^- \) that contains \( Y \) also contains \( X \). The same is true for transactions in \( \Delta^+ \).

Using this lemma, at the \( k \)-th iteration \( (k \geq 2) \), we can obtain an upper bound \( b_X^- \) for \( \delta_X^- \) of candidate \( Y \) before scanning \( \Delta^- \). The bound is taken to be the minimum of \( \delta_X^- \) for all \( X \subseteq Y \land |X| = |Y| - 1 \). Note that since \( Y \) is a candidate generated by apriori-gen [3], all its size-(\( k - 1 \)) subsets \( X \) must be in \( L'_{k-1} \) and hence \( C_{k-1} \); thus \( \delta_X^- \) has been found in the previous iteration. A bound \( b_X^+ \) can be similarly obtained for \( \delta_X^+ \) for each candidate \( Y \) before scanning \( \Delta^+ \). Lower bounds for \( \delta_X^+ \) and \( \delta_X^- \) are zero, of course.

Now, before we scan \( \Delta^- \), we do not know the values of \( \delta_X^+ \) and \( \delta_X^- \) for each candidate \( X \). We cannot apply Lemma 4 directly. However, combining lemmas 4 and 5, we can do some pruning at this stage: For each candidate \( X \) in \( Q_k \), if \( b_X^+ \leq (|\Delta^+| - |\Delta^-|) \times s\% \), then \( X \) cannot be in \( L' \). So, we can remove such \( X \) from \( Q_k \). Similarly, we may use the bound \( b_X^- \) to remove the candidates \( X \) in \( P_k \) which satisfy \( \delta_X + b_X^- < |D| \times s\% \). This reduces the number of candidates before scanning \( \Delta^- \) at a negligible cost.
After scanning $\Delta^-$ and before scanning $\Delta^+$, we know the value of $\delta^-_X$, but not $\delta^+_X$ for a candidate $X$. Combining lemmas 4 and 5 gives us the following pruning: Delete from $Q_k$ those candidate for which $b_X^--\delta^-_X \leq (|\Delta^+| - |\Delta^-|) \times s\%$. For the candidates $X$ in $P_k$, those satisfying $\sigma_X + b_X^+-\delta^-_X < |D| \times s\%$ are deleted. Thus, the number of candidates is reduced at a negligible cost before scanning $\Delta^+$. We still have not made use of the bound $b_X^-$. It is employed in the following optimization which corresponds to the optimization introduced in Section 3.2. We note that for a candidate $X$ in $Q_k$, we actually do not need to find $\delta^-_X$, since the deleted transactions contribute nothing to the final support count $\sigma_X^-$. However, the value of $\delta^-_X$ helps us to remove some candidates from $Q_k$ before scanning $D^-$. So, it helps to improve performance. For a candidate $X$ satisfying $b_X^- - \delta^-_X \geq (|\Delta^+| - |\Delta^-|) \times s\%$, whatever the value of $\delta^+_X$ be, we have $b_X^- - \delta^-_X \geq b_X^- - \delta^-_X \geq (|\Delta^+| - |\Delta^-|) \times s\%$. Thus, we do not need to find $\delta^+_X$ for these candidates. As in Section 3.2, we remove such candidates before we scan $\Delta^+$, thus reducing the size of the candidates in this scan.

A candidate $X$ in $R_k$ may be finally found to be in $L_k$. However, $\delta^-_X$ is not available for such a candidate. This causes troubles in the calculation of $b_X^-$ for the candidates in the next iteration. As a remedy, we assign $b_X^-$ to $\delta^-_X$ for all candidates $X$ in $R_k$. The bounds so calculated will still be valid, though not optimal. In the scan of $\Delta^+$, we cannot directly apply Lemma 4 to the candidates in $R_k$ directly. We can only prune out those candidates $X$ from $R_k$ for which $\delta^+_X \leq (|\Delta^+| - |\Delta^-|) \times s\%$. In the deletion-only case (Section 3.2), we introduced the set $R_k$ to optimize the algorithm without paying much cost. This is not true for the general case. Although $R_k$ reduces the candidate set for the scan of $\Delta^+$, it also causes the candidate set in the scan of $D^-$ to be larger. Moreover, a candidate $X$ that gets moved to $R_k$ will not have its count ($\delta^+_X$) in $\Delta^+$ tallied. If we want to apply Lemma 4 to test if $X$ can be ignored in the scan of $D^-$, only a trivial lower bound (zero) of $\delta^+_X$ is used. Therefore, the test and thus the pruning is less effective. The tradeoffs of whether to use $R_k$ is thus on the amount of work saved in scanning $\Delta^-$ and the effectiveness of the pruning (Lemma 4). Naturally, if $|\Delta^-|$ is large, using $R_k$ can save much work. Our algorithm therefore applies $R_k$ only when $|\Delta^-| > |\Delta^+|$. Here is the final version of our FUP2 algorithm, for iteration $k$ where $k \geq 2$.

For the first iteration, set $C_1 = I$, $b_X^+ = |\Delta^+|$ and $\delta^+_X = |\Delta^-|$.

1. Obtain a candidate set $C_k$ of itemsets. Halt if $C_k = \emptyset$.
2. Calculate $b_X^+$ for each $X \in C_k$.
3. Partition $C_k$ into $P_k$ and $Q_k$.
4. For each $X \in P_k$, remove it if $\sigma_X + b_X^+ < |D| \times s\%$.
5. For each $X \in Q_k$, remove it if $b_X^+ \leq (|\Delta^+| - |\Delta^-|) \times s\%$.
6. If $|\Delta^-| \leq |\Delta^+|$, let $R_k = \emptyset$. Otherwise, calculate $b_X^+$ for each $X \in Q_k$ and if $b_X^- - \delta^-_X \geq (|\Delta^+| - |\Delta^-|) \times s\%$, move it to $R_k$ and assign $b_X^-$ to $\delta^-_X$.
7. Scan $\Delta^+$ to find $\delta_X^+$ for each $X \in P_k \cup Q_k$.
8. Delete from $P_k$ those candidates $X$ where $\sigma_X + b_X^+ - \delta^-_X < |D| \times s\%$.
9. Delete from $Q_k$ those candidates with $b_X^- - \delta^-_X \leq (|\Delta^+| - |\Delta^-|) \times s\%$.
10. Scan $\Delta^+$ to find $\delta_X^-$ for each $X \in P_k \cup Q_k \cup R_k$.
11. For each candidate $X \in P_k$, calculate $\sigma_X^-$. For each candidate $X \in Q_k$, delete $X$ if $\delta_X^- - \delta^-_X \leq (|\Delta^+| - |\Delta^-|) \times s\%$.
12. For each candidate $X \in R_k$, delete $X$ if $\delta_X^- \leq (|\Delta^+| - |\Delta^-|) \times s\%$.
13. Scan $D^-$ and get the count of each $X \in Q_k \cup R_k$. Then, add this count to $\delta_X^+$ to get $\sigma_X^+$. Add to $L'_k$ those candidates $X$ from $P_k \cup Q_k \cup R_k$ where $\sigma_X^+ \geq |D| \times s\%$.
16. Halt if $|L'_k| < k + 1$.

It is interesting to note that this algorithm reduces to Apriori [3] if we set $\Delta^- = D^- = \emptyset$, FUP [4] if we set $\Delta^+ = \emptyset$ and the transaction deletion algorithm in Section 3.2 if we set $\Delta^+ = \emptyset$. So, it is a generalization of these three algorithms.

A further improvement can be made by applying the DHP technique [12]. The technique can be introduced into the FUP2 algorithm to hash the counts of itemsets in $D'$. This brings the benefits of the DHP algorithm into the FUP2 algorithm immediately. We call this DHP-enhanced update algorithm FUP2ʰ, to distinguish it from FUP2.¹ Let us illustrate this final FUP2 algorithm with the example in Table 4. This example is the same as the previous one, except that we have $\Delta^+$ containing one transaction $\{C,D\}$ this time. Large itemsets and their counts in $D'$ are shown in the same table.

In the first iteration, we have $C_1 = I = \{A,B,C,D,E\}$, $P_1 = \{A,B,C,D\}$, $Q_1 = \{E\}$ and $R_1 = \emptyset$. Note that for this iteration, $b_X^+ = |\Delta^+| = 1$ and $\delta_X^- = |\Delta^-| = 1$ for all $X \subseteq I$. Next, $\Delta^-$ is scanned and we find $\delta_A^- = \delta_B^- = \delta_C^- = 1$ and $\delta_D^- = \delta_E^- = 0$. After pruning (step 9), $Q_1 = \emptyset$. Then, $\Delta^+$ is scanned and we have $\delta_A^+ = \delta_B^+ = 0$; $\delta_C^+ = \delta_D^+ = 1$. Since both $Q_k$ and $R_k$ are now empty, steps 12–14 can be skipped. $D^-$ need not be scanned in this iteration. Finally, $\sigma_A^+ = \sigma_B^+ = 2$, $\sigma_C^+ = 3$ and $\sigma_D^+ = 4$. All of them are large in $D'$. So, $L'_1 = \{A,B,C,D\}$.

¹We remark that the DHP algorithm requires extra memory to store a big hash table. Algorithm FUP2ʰ should therefore be applied only when memory is plentiful.
Transactions: \( \{I = \{A, B, C, D, E\}\} \)

\[
D = \begin{pmatrix}
\Delta^- & A & B & C & E \\
\Delta^- & A & B & C & D \\
\Delta^+ & C & D & \\
\end{pmatrix}
\]

Large itemsets (support threshold \( s = 25\% \))

in \( D' = D^- \cup \Delta^+ \):

<table>
<thead>
<tr>
<th>Itemsets(X)</th>
<th>( A \times A \times B \times C \times D \times CD )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_X )</td>
<td>2    2    3    3    4    2</td>
</tr>
</tbody>
</table>

Table 4: An example for \( |\Delta^+| > 0 \)

In the second iteration, \( C_2 = \{ AB, AC, AD, BC, BD, CD \} \), \( Q_2 = \{ AC, AD, BC, BD, CD \} \) and \( P_3 = \{ AB \} \). Since \( b_{AC}^+ = b_{AD}^+ = b_{BC}^+ = b_{BD}^+ = 0 \) (because \( \delta_{AC}^+ = \delta_{AD}^+ = \delta_{BC}^+ = \delta_{BD}^+ = 0 \)), the corresponding candidates are removed from \( Q_2 \) in step 5, leaving \( Q_2 = \{ CD \} \). Next, we scan \( \Delta^- \) and obtain \( \delta_{AB}^+ = 1 \), \( \delta_{CD}^- = 0 \). Since \( b_{AB}^+ = 0 \), \( AB \) is removed from \( P_2 \) in step 8, leaving \( P_2 = \emptyset \). Then, \( \Delta^+ \) is scanned to get \( \delta_{CD}^- = 1 \), followed by the scan of \( D^- \). Finally \( \sigma_{CD}^- \) is found to be 2, enough for \( CD \) to be large. Thus \( L_2^+ = \{ CD \} \). This is the last iteration, since \( |L_2^+| = 1 < 3 \) is insufficient to generate a \( C_3 \) in the third iteration.

Hence, we find that in the updated database \( D' \), \( L' = \{ A, B, C, D, CD \} \). The large itemset \( AB \in L' \) is now obsolete and the new large itemset \( CD \) is added to \( L' \). Note that FUP2 scans \( D^- \) only once, to obtain the count of \( CD \) in the second iteration. If we apply the Apriori algorithm on \( D' \) instead, we will have to scan \( D^- \) twice, with 11 candidates from \( C_1 \) and \( C_2 \). So, FUP2 reduces the candidate size by \( \frac{11-1}{11} = 91\% \), a very significant improvement over Apriori.

4 Performance Analysis

To assess the performance of our new algorithms, Apriori, DHP, FUP2 and FUP2\( \# \) are implemented on an RS/6000 workstation (model 410) running AIX. Several experiments are conducted to compare their performance.

4.1 Generation of synthetic data

In the experiments, we used synthetic data as the input databases to the algorithms. The data are generated using the same technique as introduced in [3] and modified in [12]. Readers are referred to these papers for a detailed description. Table 5 gives a list of the parameters used in the data generation method. To model the change of association rules as a result of inserting and deleting transactions, we slightly modified the data generation method as follows.

| \( |\Delta^-| \) | number of deleted transactions |
| \( |D^-| \) | number of unchanged transactions |
| \( |\Delta^+| \) | number of added transactions |
| \( |T| \) | mean size of transactions |
| \( |L| \) | mean size of potentially large itemsets |
| \( |L| \) | number of potentially large itemsets |
| \( N \) | number of items |

Table 5: Parameters for data generation

We split the data generation procedure into 2 steps. In the first step, a set \( L \) of potentially large itemsets is generated. In the second step, a subset of \( L \) is used to generate the database transactions.

To model a change of association rules, we choose two integers \( p \) and \( q \) in the range from zero to \( |L| \), such that \( p+q \geq |L| \).

2 We use the first \( p \) potentially large itemsets from \( L \) to generate \( \Delta^- \) and the last \( q \) potentially large itemsets from \( L \) to generate \( \Delta^+ \). \( D^- \) is generated from the whole \( L \). As a result, the first \( p \) potentially large itemsets in \( L \) have a higher tendency to be large in \( D = \Delta^- \cup D^- \) than in \( D' = \Delta^- \cup \Delta^+ \). They correspond to large itemsets that turn obsolete due to the updates. Similarly, the last \( q \) potentially large itemsets have a higher tendency to be large in \( D' \) than in \( D \). They correspond to new large itemsets in the updated database. The middle \( p+q-|L| \) potentially large itemsets take part in the generation of all of \( \Delta^- \), \( D^- \) as well as \( \Delta^+ \). So, they would be large in both \( D \) and \( D' \). They represent the association rules that remain unchanged as a result of the update. By varying the values of \( p \) and \( q \), we can control the degree of similarity between \( D \) and \( D' \).

In the following we use the notation \( T_x.y.D_i-j+k \), modified from the one used in [3], to denote an experiment using databases with the following sizes: \( |D| = |\Delta^-| + |D^-| = i \) thousand, \( |\Delta^-| = j \) thousand, \( |\Delta^+| = k \) thousand, \( |T| = x \) and \( |L| = y \). In the experiments, we set \( |L| = 2100 \), \( |L| = 1000 \) and \( p = q = 2000 \).

3 For DHP and FUP2\( \# \), we use a hash table of 4096 entries. The hash table is used to prune size-2 candidates (i.e. \( C_2 \) only).

In each experiment, we first use DHP to find out the large itemsets in \( D \). Then, we run FUP2 and FUP2\( \# \), supplying to them the databases \( \Delta^- \), \( D^- \) and \( \Delta^+ \) and the large itemsets and their support...

\[ \text{This condition is used to model rules that stay in the old as well as new databases.} \]

\[ \text{There are several other parameters for the data generation procedure reported in [4] and [12]. For example,} \]

\[ S_q \text{ is the clustering size used in the generation of potentially large itemsets,} \]

\[ P \text{ is the size of the pool of potentially large itemsets from which transactions are generated,} \]

\[ M_f \text{ is the multiplying factor associated with the pool. Following [4],} \]

\[ \text{we set} \ S_q = 5, P = 50, M_f = 200. \text{Readers are referred to} \]

\[ \text{[12] for a detailed explanation of these parameters.} \]
counts in $D$. The time taken is noted. To compare with the performance of Apriori and DHP, we run these two algorithms on the updated database $D'$, and note the amounts the time they have spent. The time taken by the algorithms are then compared.

### 4.2 Comparing the four algorithms

The four algorithms are tested against the setting $T10.I4.D100–5+5$. The support thresholds is varied between 1.0 and 3.0. The results are plotted in Figure 1. It is found that FUP2 is 1.83 to 2.27 times faster than Apriori, while FUP2$\bar{N}$ is 1.99 to 2.96 times faster than DHP and is 2.05 to 3.40 times faster than Apriori. To explain the performance gain, let us examine the number of candidate itemsets generated by each algorithm in the scan of $D'$ for the particular instance with support threshold 2.0 (see Table 6). The total number of candidates generated by FUP2 is 38% of that of Apriori. The candidate size of FUP2$\bar{N}$ is 28% of that of DHP and is 21% of that of Apriori. This significant reduction in the number of candidates is the main reason for the performance gain. Clearly, FUP2$\bar{N}$ is very efficient because it combines the techniques of both FUP2 and DHP to greatly reduce the number of candidates.

![Figure 1: Comparison of the four algorithms](image)

**Table 6: Number of candidate itemsets**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Apriori</th>
<th>DHP</th>
<th>FUP2</th>
<th>FUP2$\bar{N}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99</td>
<td>99</td>
<td>98</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>4095</td>
<td>4095</td>
<td>1794</td>
<td>272</td>
</tr>
<tr>
<td>3</td>
<td>3414</td>
<td>2018</td>
<td>3018</td>
<td>3018</td>
</tr>
<tr>
<td>4</td>
<td>805</td>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>497</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>495</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>258</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>92</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>10797</td>
<td>7897</td>
<td>3905</td>
<td>3188</td>
</tr>
<tr>
<td><strong>DHP</strong></td>
<td>100%</td>
<td>76.1%</td>
<td>37.4%</td>
<td>27.1%</td>
</tr>
<tr>
<td><strong>FUP2</strong></td>
<td>113%</td>
<td>100%</td>
<td>49.4%</td>
<td>27.7%</td>
</tr>
</tbody>
</table>

Figure 2: Effect of $|\Delta^-|$ and $|\Delta^+|$

### 4.4 Varying the number of deleted and added transactions independently

Another experiment is conducted to find out how the size of $\Delta^-$ affects the performance of the algorithms. We use the setting $T10.I4.D100–x+10$ for the experiment. The support threshold is 2%. In other words, we use an initial database of 100 thousand transactions. Ten thousand transactions are added to the database and $x$ thousand are deleted. Figure 3 shows the results of this experiment. As the number of deleted transactions increases, the amounts of time taken by Apriori and DHP decrease, since the size of the final database decreases. For example, at $x = 1.0$, FUP2 is 4.5 times faster than Apriori. As $x$ increases, the number of transactions that FUP2 and FUP2$\bar{N}$ have to handle increases; therefore, these algorithms take more and more time as $x$ grows. However, FUP2 and FUP2$\bar{N}$ still outperform Apriori and DHP for $x < 30$. Beyond that, Apriori and DHP take less time to fin-
ish. This means that as long as the number of deleted transactions is less than 30% of the original database, the incremental algorithms win. Practically, the original database \( D \) in a data mining problem is very large. The amount of updates should be much less than 30% of \( D \).

![Figure 3: Effect of |\( \Delta^- \)|](image)

A similar experiment is done using the setting T10.I4.D100-10+2 and the same support threshold of 2%. This time, we keep the size of \( \Delta^- \) constant and vary the size of \( \Delta^+ \). The results are plotted in Figure 4. As \( x \) increases, \(|\Delta^+|\) increases. So, the execution time of Apriori and DHP increases with \( x \). They do not run faster than FUP2 and FUP2\( \delta \) even when \( x \) is as large as 40.

![Figure 4: Effect of |\( \Delta^+ \)|](image)

Examining Figure 4 more closely, we notice that the execution time of FUP2 and FUP2\( \delta \) is quite steady in the range 1.0 \( \leq x \leq 7.5 \). For \( x > 15 \), the execution time increases with \( x \). This is because the greater the value of \( x \), the more the transactions the algorithms have to handle. However, in the range 7.5 \( \leq x \leq 15 \), the execution time drops as \( x \) increases! Also, if we examine Figure 3 more carefully, we can also notice sharper rises in the execution times of FUP2 and FUP2\( \delta \) in that range of \( x \).

To understand this phenomenon, recall that in iteration \( k \), if an itemset \( V \) was not large in \( D \) but is in \( C_k \), it is put in \( Q_k \). Suppose that \( V \) is also small in \( D' \). Then, since \( V \) is small in both \( D \) and \( D' \), it does not occur frequently in \( D \) and \( D' \). Statistically, \( \delta^+_V \) and \( \delta^-_V \) are small in magnitude and they are close to each other. So, \( \delta^+_V = \delta^+_V - \delta^-_V \) has a very small magnitude. It may be positive or negative. When Lemma 4 is applied to prune \( Q_k \) in step 12 of FUP2, a candidate \( X \) in \( Q_k \) is pruned if \( \delta_X \leq (|\Delta^+| - |\Delta^-|) \times s\). So, when \(|\Delta^+| - |\Delta^-| \gg 0 \), \( V \) has a very high change of being deleted from \( Q_k \). If \(|\Delta^+| - |\Delta^-| \geq 0 \) but is small in magnitude, \( V \) may escape the pruning if \( \delta_Y \) is large enough, although there is still a high chance that \( V \) is pruned away. If, however, \(|\Delta^+| - |\Delta^-| < 0 \), then \( V \) will only be pruned away if \( \delta_Y \) is negative enough, but the chance of this is low. Hence, as \(|\Delta^+| - |\Delta^-| \) increases from a negative value to a small positive value (e.g., as \( x \) in Figure 4 varies from 10 to 15 thousands), the chance that \( V \) gets pruned increases.

As there are many itemsets that behave like \( V \), the drop in execution time of FUP2 and FUP2\( \delta \) is very dramatic when \(|\Delta^+| \) increases from slightly below \(|\Delta^-| \) to slightly above \(|\Delta^-| \). A similar result occurs as \(|\Delta^-| \) decreases from slightly above \(|\Delta^+| \) to slightly below it.

### 4.5 Scale-up experiment

To find out if FUP2 and FUP2\( \delta \) work also for large databases, experiments with scale-up databases are conducted. We use the setting T10.I4.Dx=10+10. Again, we use a support threshold of 2%. The results are shown in Figure 5. The execution times of all the four algorithms increase linearly as \( x \) increases. This shows that FUP2 and FUP2\( \delta \) are scalable and can work with very large databases.

![Figure 5: Scale-up experiment](image)

### 5 Discussion

Our new incremental algorithms make use of certain information to achieve their high performance. This information includes the old large itemsets and their support counts, the transactions that are not changed by the update \((D')\), and the transactions that are inserted or deleted \((\Delta^+\cup\Delta^-)\). Is it reasonable to assume that such information be available? The answer is yes.
The large itemsets and their support counts in the original database come from the results of a previous mining activity. We assume that this information is stored. As the association rules can be calculated from these counts efficiently, it is more desirable to store the counts rather than the association rules. Storing the counts enables us to maintain of the association rules efficiently.

A database system that supports recovery keeps all updates into the log files. Consequently, it is possible to retrieve from the log files all the deleted and newly added transactions since the last mining. By identifying the newly inserted transactions in the current updated database (e.g., with the help of transaction IDs), we can select from the updated database those transactions which have remained unchanged since the last mining activity. Thus, we can obtain the set of unchanged transactions. Hence $\Delta^-$, $D^-$ and $\Delta^+$ are available.

6 Conclusions

We studied an efficient incremental updating technique for the maintenance of association rules discovered by database mining. This technique updates the association rules when old transactions are removed from the database and new transactions are added to it. It uses the information available from a previous mining to reduce the amount of work that has to be done to discover the association rules in the updated database. It is a generalization of two previous algorithms: Apriori [3] and FUP [4]. Performance studies show that the new technique is significantly faster than mining the updated database from scratch. The new technique works well over wide ranges of system parameter values. In particular, it works well for updates of a wide range of insertion sizes and small to moderate deletion sizes.

References


