Chapter 2

# ARM++: A HYBRID ASSOCIATION RULE MINING ALGORITHM

*Zahir Tari and Wensheng Wu*
Royal Melbourne Institute of Technology

**Abstract**

Most of the approaches for association rule mining focus on the performance of the discovery of the frequent itemsets. They are based on the algorithms that require the transformation of data from one representation to another, and therefore excessively use resources and incur heavy CPU overhead. This chapter proposes a hybrid algorithm that is resource efficient and provides better performance. It characterizes the trade-offs among data representation, computation, I/O, and heuristics. The proposed algorithm uses an array-based item storage for the candidate and frequent itemsets. In addition, we propose a comparison algorithm (CmpApr) that compares candidate itemsets with a transaction, a filtering algorithm (FilterApr) that reduces the number of comparison operations required to find frequent itemsets. The hybrid algorithm (ARM++) integrates filtering methods within the Partition algorithm [7]. Performance analyses from our implementation indicate that ARM++ has better performance and scales linearly.

## 1 BACKGROUND

We are living in an information age that is overwhelmed by enormous amount of data and information. Data mining within the database community, also known as *knowledge discovery* by the AI community, is the science of automated extraction of useful information or hidden patterns from large databases. Data mining is a new, multidisciplinary field ranging across database technology, statistics, artificial intelligence, machine learning, etc. It normally processes data that have already been collected, such as records of all transactions in a bank, and does not involve the data collection strategy itself.

Data mining is not concerned with a small set of data, as these can be well handled by classical statistical analysis techniques. Data mining focuses on new

problems that may arise with large data repositories, such as finding a target within a massive dataset in a short time, finding hidden (i.e. not explicit) relationships amongst a huge volume of information within data repositories (e.g., analysis of emails to detect terrorist threats). Such relationships found through the use of data mining techniques are called *models* or *patterns*. Descriptive models characterize the general properties of the data in the database, while predictive models perform inferences on the current data for predictions. One typical financial application using data mining is the profiling of customer behavior. A bank keeps transaction records of its customers and can use data mining technology to cluster customers into levels of high credit risk, medium credit risk, and trust, which may help them to advertise suitable new products and bank loan approval.

There are many data mining tasks and algorithms. These are often classified into four components [11]:

- Models (pattern structures): these model the underlying structures in a database.
- Score functions: the role is to decide how well the developed model fits with the data.
- Optimization and search methods: these relate to the optimization of the score function and searching over many models and structures.
- Data management strategies: These deal with efficient access and use of data during the search/optimization.

Data mining systems are categorized as follows [12]:

- Classification according to the types of databases to be mined: object-oriented databases, object-relational databases, spatial database, temporal databases and time-series databases, text databases and multimedia databases, heterogeneous databases and legacy databases, and the World Wide Web.
- Classification according to the types of knowledge to be mined: characterization, discrimination, association, classification, clustering, outlier analysis, and evolution analysis.
- Classification according to the types of techniques utilized: machine learning, statistics, pattern recognition, visualization, trees, networks and rules, etc.
- Classification according to the types of applications: finance, telecommunications, DNA, stock markets, etc.

## 2 MOTIVATION

This chapter focuses on a specific area of data mining, namely, mining of association or relationships between data items. The problem of mining association rules was introduced in [1] and can be defined as follows. Given a set of transactions, where each transaction is a set of items, an association rule is an expression of the form $X \Rightarrow Y$, where $X$ and $Y$ are sets of items. There are two measurements of an association rule; *confidence* and *support*. The *confidence* of a

rule represents the percentage of transactions that contain $Y$ out of those that contain $X$. The *support* of a rule is the percentage of transactions that contain both $X$ and $Y$. The problem of mining association rules becomes then a two-step process [1–3]. The first step consists of finding all sets of items (called *itemsets*) that have transaction support above minimum support. The *support* for an itemset is the number of transactions that contain the itemset. Itemsets with minimum support are called *frequent* itemsets, and otherwise *small* itemsets. The second step uses the frequent itemsets to generate the desired rules. For a given *frequent* itemset $Y = \{I_1, I_2,....,I_k\}$, $k \geq 2$, it generates all rules that use items from the set $\{I_1,I_2,...,I_k\}$. The antecedent of each of these rules will be a subset $X$ of $Y$, and the consequent will be the itemset $Y$-$X$. If the confidence, i.e., the ratio of the support of $Y$ divided by the support of $X$, is greater than a confidence factor $c$, it is an association rule; otherwise, it is not.

Because the number of candidate itemsets and that of transactions are both very large, all the frequent itemsets can be found only in an iterative way, where the itemset with $k$ items is defined as a $k$-itemset. In this way, iteration means each frequent $k$-itemset is generated in an increasing order of $k$. To obtain better performance, different algorithms and data structures have been designed [1-7] to reduce the number of iterations, the number of candidate itemsets, the number of transactions in each iteration, the number of items in each transaction, and the method of comparison between candidate itemsets and transactions to accelerate the identification of a candidate itemset in a transaction. In particular, a lot of work on the efficiency of association rule mining was done in the context of the following approaches: Apriori [2], AprioriTid [2], and Partition [7]. These approaches aim to reduce the execution time by applying heuristics and transforming the data into different representations. However, the transformation of the data from the original form to another will require extra resources and CPU time. On one hand, the required resources are not guaranteed to be available. For example, there might not be enough disk space to hold the transformed data for Partition. This results in the failure of the execution. On the other hand, the time savings from the new data representation might not be able to compensate for the time spent on the transformation. This depends somewhat on the characteristics of the data. To our knowledge, none of the existing algorithms performs as well as others with all the simulation data of different characteristics.

In this chapter, we propose three algorithms, namely, *CmpApr, FilterApr,* and *ARM++*, that aim to improve the performance of association rule mining algorithms at difference stages of the construction of the frequent itemsets. After an evaluation of the performance of the existing algorithms, as shown in Section 5, our findings is that *ARM++* provides a better performance. This gain in performance is mainly related to the fact that ARM++ applies new heuristics in the early stage of association rule mining and changes the data structure when the transformation is beneficial and the resources are available in the late stage. In the early stage, to improve the performance of the existing algorithms, e.g., *Apriori* [2], we come up with two heuristics: (1) a new comparison method, which is implemented in *CmpApr*; and (2) the inherent relations between the data items used to reduce the comparison of unnecessary items, which is implemented in *FilterApr*.

After a detailed analysis of these two heuristics, we realized that the second heuristic reduces the number of comparisons to such an extent that the original

beneficial comparison method in *CmpApr* has a negative impact on the execution time in *FilterApr*. Based upon the fact that *FilterApr* is much faster than *CmpApr*, we choose *FilterApr* as the algorithm for the early stage of the algorithm. In the late stage of the algorithm, we use the existing *Partition* [7]. However, we start the conversion of the data only when the estimated transformed data can be held in the memory, thereby minimizing the possible overhead of data I/O operation and extra requirement of disk space.

This chapter is organized as follows. Section 3 reviews some of the major approaches for association rule mining. Section 4 is dedicated to the implementation of the array-based data structure (*ArrayApr*). In Section 5 we describe in detail the three different algorithms, that is, *CmprApr*, *FilterApr*, and *ARM++*. Section 6 provides a detailed analysis of the performance of our algorithms, and finally future extensions of these algorithms are given in Section 7.

# 3 RELATED WORK

The discovery of frequent itemsets and the construction of association rules are two sub-problems of association rule mining. Our focus here is on the frequent itemset searching of the first sub-problem. The three major data representations used by existing algorithms to store the database are item-lists, candidate-lists, and TID-lists. We describe them and discuss the impact of these data representations on the performance of the algorithms that use them.

## 3.1 Existing Approaches

**AIS [1]**

The problem of association rules was first introduced in [1] along with an algorithm that was later called AIS [2]. To find frequent sets, AIS creates candidates "on-the-fly" while it reads the database. Several passes are necessary, and during one pass, the entire database is read, one transaction after the other. Adding items to sets that were found to be frequent in previous passes creates a candidate. Such sets are called *frontier sets*. The candidate that is created by adding an item to a frontier set $F$ is called a *1-extension* of $F$ because one item was added to $F$. To avoid duplicate candidates, only items that are larger than the largest item in $F$ are considered for 1-extensions. To avoid generating candidates that do not even occur in the database, AIS does not build 1-extensions on blind faith, but only when they are encountered while reading the database.

Associated with every candidate, a counter is maintained to keep track of the frequency of the candidate in the database. When a candidate is first created, this counter is set to 1, and when the candidate is found subsequently in other transactions, this counter is incremented. After a complete pass through all transactions, the counts are examined, and candidates that meet the minimum support requirement become the new frontier sets. This is a simplification because determination of which expansions to include as candidates becomes trickier in the presence of k-extensions and support estimation. For k-extensions, for example, only maximal frequent sets become frontier sets [1].

Unfortunately, the AIS candidate generation strategy creates a large number of candidates, and sophisticated pruning techniques are necessary to decide whether an extension should be included in the candidate set. The methods include a technique called *pruning function optimization* and estimating support for a prospective candidate based on relative frequencies of its subsets. Pruning functions use the fact that a sum of carefully chosen weights per item can rule out certain sets as candidates without actually counting them. An example is the total transaction price. If fewer transactions than the fraction required for minimum support exceed a price threshold, then sets that are more expensive cannot possibly be frequent. These decisions can be fairly costly; moreover, they have to be made repeatedly for many subsets for each transaction. If an unlikely candidate set is rejected, this decision has to be made for every transaction the set appears in.

**SETM (Set Oriented Mining)**

The SETM algorithm [5] uses only standard database operations to find frequent sets. For this reason, it uses its own data representation to store every itemset supported by a transaction along with the transaction's ID (TID). SETM repeatedly modifies the entire database to perform candidate generation, support counting, and remove infrequent sets.

SETM has a few advantages over AIS because it creates fewer candidates. However, the problem with the SETM algorithm is that candidates are replicated for every transaction in which they occur, which results in huge sizes of intermediate results. Moreover, the itemsets have to be stored explicitly, i.e., by listing their items in ascending order. Using candidate IDs would save space, but then the join could not be carried out as an SQL operation. What is even worse is that these huge relations have to be sorted twice to generate the next larger frequent sets.

**Apriori, AprioriTid, and AprioriHybrid Algorithms [2–4,6–8]**

The vast number of candidates in AIS caused its authors to design a new candidate generation strategy called *apriori-gen* as part of the algorithms *Apriori* and *AprioriTid* [2]. *Apriori-gen* has been so successful in reducing the number of candidates that it has been used in every algorithm proposed since it was published [3,4, 6–8]. The underlying principle, based on the a priori property, is to generate only those candidates for which all subsets have been previously determined to be frequent. In particular, a (k+1)-candidate will be accepted only if all its k-subsets are frequent. Upon reading a transaction $T$ in the counting phase of pass $k$, *Apriori* has to determine all the $k$-candidates supported by $T$ and increment the support counters associated with these candidates.

The major problem for *Apriori* is that it always has to read the entire database in every pass, although many items and many transactions are no longer needed in later passes of the algorithm. In particular, the items that are not frequent and the transactions that contain fewer items than the current candidates are not necessary. Removing them would obviate the expensive effort to try to count sets that cannot possibly be candidates.

The shortcoming of *Apriori*, that it could not remove unwanted parts of the database during later passes, has led to the design of *AprioriTid* [2], which uses a

different data representation than the item-lists used by *Apriori*. *AprioriTid* can be considered an optimised version of SETM that does not rely on standard database operations and uses *apriori-gen* for faster candidate generation. Therefore, comparing *Apriori* and *AprioriTid* is more interesting because they both generate the same number of candidates and differ mainly in their underlying data representation.

While *Apriori* avoids swapping data to disk, it does not weed out useless items in later passes and hence wastes time on futile attempts to count support of sets involving these items. *AprioriTid*, on the other hand, prunes the data set as described in the previous section and as a result outruns *Apriori* in later passes. Unfortunately, in the second iteration, as a consequence of the candidate-list representation, the data usually do not fit in memory, so swapping is necessary.

**Partition [7]**

While all the algorithms presented so far are more or less variations of the same scheme, the *Partition* algorithm takes a different approach. *Partition* tries to address two major shortcomings of previous algorithms. The first problem with the previous algorithms is that the number of passes over the database is not known beforehand, regardless of which representation is used. Therefore, the number of I/O operations is not known and is likely to be very large. *AprioriTid* tries to circumvent this problem by buffering the database, but then the database size is limited by the size of main memory. The second problem lies with pruning the database in the later passes, i.e., removing unnecessary parts of the data. AIS and *Apriori* fail to optimize the Item-lists structure. Candidate-lists do permit pruning the database, but they cause problems because of their unpredictably large intermediate results in the early passes.

The approach taken in *Partition* [7] to solve the first problem (unpredictably large I/O-cost) is to divide the database into equally sized horizontal Partitions. An algorithm to determine the frequent sets is run on each subset of transactions independently, producing a set of *local frequent itemsets* for each partition. The partition size is chosen such that an entire partition can reside in memory. Hence, only one read is necessary for this step, and all passes access only the buffered data. To address the second problem (failure to reduce the database size in later passes), *Partition* uses a new "TID-list" data representation both to determine the frequent itemsets for each partition and to count the global supports during the counting phase. TID-lists invert the candidate-list representation by associating with each itemset $X$ a list of all the TIDs for those transactions that support the set. The TID-lists for a $k$-candidate can be computed easily by intersecting the TID-lists of two of its $(k\text{-}1)$-subsets. All TID-lists are sorted so that this intersection can be computed efficiently with a merge-join, which only requires traversing the two lists once.

Like candidate-lists, TID-lists change in every pass and may have to be swapped to disk if there is not enough memory available to store them. Again, the size of intermediate results can be larger than the original data size, and this figure is not known. The reason is the same as that for candidate-lists, with the difference that in *Partition*, TIDs are replicated for every candidate set instead of replicating candidate identifiers for every transaction.

## 3.2 The ARM++ Approach

If we need to select an algorithm for later iterations of the frequent itemset discovery, which algorithm should we choose? Both *AprioriTid* and *Partition* outperform *Apriori* in the later iterations mainly due to their underlying data structures. All three algorithms generate the same number of candidates and frequent itemsets. For *Partition*, if a *k*-frequent itemset is in a transaction *t*, to make this count, it needs only one comparison of the TID-lists of the two (*k*-1) frequent subsets. In contrast, *AprioriTid* needs two comparisons to detect the existence of two subsets of the *k*-frequent itemset, in addition to the overhead of the access to the two subsets through the auxiliary data structure. If the data for both algorithms are kept in memory, *Partition* beats *AprioriTid* in terms of performance. With the increasing number of iterations, the gap of the number of comparisons between *Partition* and *Apriori* gets wider.

In this chapter, we propose three new algorithms, varying in the comparison methods, transaction filtering, and transaction transformation. The underlying data structure is described in *ArrayApr*, which stores candidate and frequent itemsets with the proposed array-based data representation rather than the commonly used hash-tree representation [2-4,6–7]. ARM++ is a hybrid algorithm. It is a combination of *FilterApr* and *Partition* [7], where *FilterApr* is used in the early passes (*FilterApr* phase) and *Partition* in the subsequent passes (*Partition* phase). The pivot point is, whenever the estimated TID-list of *Partition* can be held in memory, we switch from *FilterApr* to *Partition*. A brief overview of these algorithms is shown in Table 2.1, and their interdependencies are described in Figure 2.1:

**Table 2.1.** An Overview of the Proposed Algorithms

| Itemset Representation | Data Representation | Comparison Method | Original Algorithm | New Algorithms |
|---|---|---|---|---|
| Array | Item-list | Itemset vs. Trans | *ArrayApr* | *CmpApr* |
| Array | Item-list | Sub-trans vs. Itemset | *ArrayApr* | *FilterApr* |
| Array | Item-list | Sub-trans vs. Itemset | *FilterApr* | *ARM++* |
| | TID-list | Merge-Join | *Partition* | |

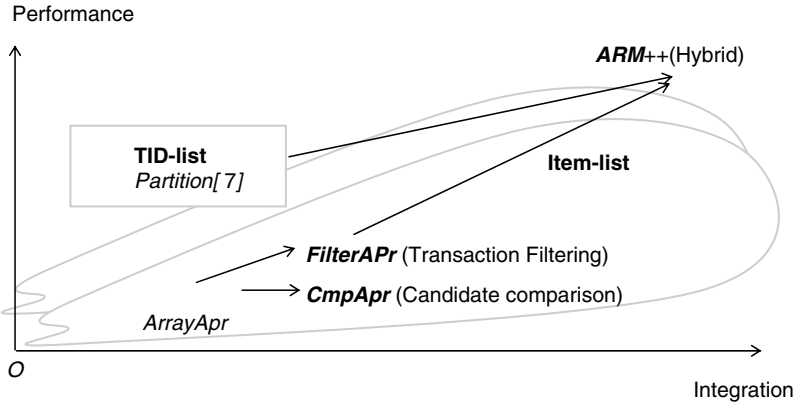


**Figure 2.1.** Algorithm Evolution Diagram

- ***CmpApr*** employs a new comparison method, *candidate comparison,* which compares candidate itemsets against a transaction instead of comparing subsets of the transaction with the itemsets. The new array-based data representation of candidate itemsets provides fast access to the items of the itemsets for the new comparison method.
- ***FilterApr*** harnesses the power of our new transaction filtering, which sharply reduces the number of comparison operations required to find the frequent itemsets among the candidates.
- ***ARM++*** integrates *FilterApr* with *Partition*. This new hybrid algorithm is the last of our series of optimizations. This new hybrid algorithm aims to make the best use of the available resources, i.e., the memory and secondary storage, to achieve the minimum execution time.

# 4 ARRAYAPR DATA STRUCTURE

In this section, we first introduce the array-based itemset storage and later show its application in the generation of the candidate and frequent itemsets (Figure 2.2). In contrast to *Apriori*, which uses a tree to store the candidates (that have to be tested against a transaction) in order to reduce the number of comparisons, *ArrayApr* uses the hash function to reach the candidates that are supported by the transaction. Then we explore the functions of the hash-tree during the counting phase and see how they are implemented with the array structure. We have used the data generation technique proposed in [2] to measure the performance of *ArrayApr.* Results of such evaluation are presented in Section 5.

## 4.1 Arrays: Itemset-counter Table, Hash Table, and Sibling Table

In *ArrayApr*, as in *Apriori*, itemsets are stored separately. However, they are stored in different structures. Our array structure contains three tables: a Hash table, an Itemset-counter table, and a Sibling table. The hash table is part of a hash function, which, given an itemset, can calculate that itemset's mapping address in the Itemset-counter table. After comparing the itemset with its counterpart in the Itemset-counter, we know whether it exists in the itemset-counter table. The sibling table stores the clustering information of itemsets in a bitmap representation. For an itemset in the Itemset-counter table, if the next one in the table is its sibling, its corresponding bit in Sibling Table is "1"; otherwise, it is "0."

For example, {1, 4, 5, 6}, {1, 4, 5, 7}, {1, 4, 5, 9}, and {1, 4, 6, 9} are candidate 4-itemsets. The layout of their storage is shown in Figure 2.2a. After scanning through the database and counting their supports, we assume all are frequent. We copy from the candidate array structure all the frequent itemsets and their clustering information into the frequent Itemset-counter table and the frequent Sibling table, respectively. Then we initialize the hash table based on the itemsets in the Itemset-counter table. After the creation of the frequent itemset

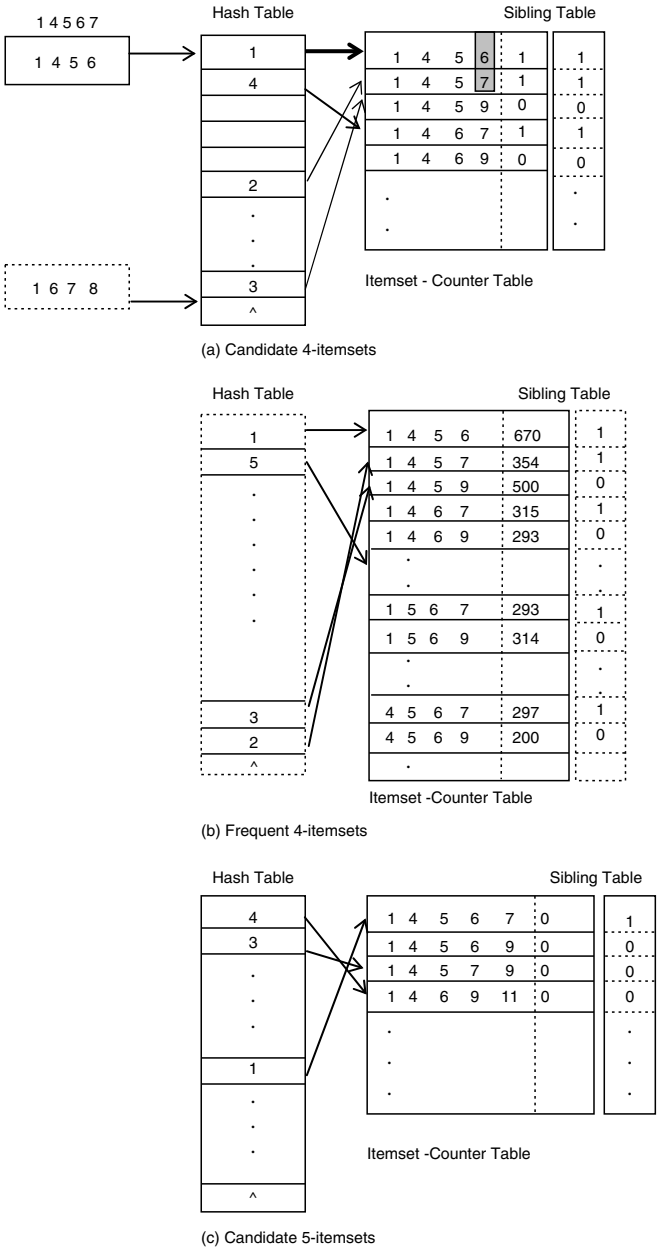


**Figure 2.2.** Array-based storage of itemsets

array structure, we delete the candidate array structure. The layout of the frequent 4-itemsets is depicted in Figure 2.2b.

The next step is to generate 5-candidates. We scan through the Sibling table of frequent 4-itemsets. If there are siblings, we invoke *apriori-gen* to create the 5-candidates. Instead of generating all the 5-candidates and then detecting their candidacy, immediately after we generate a candidate, we check its candidacy.

In our example, {1, 4, 5, 6}, {1, 4, 5, 7}, and {1, 4, 5, 9} are siblings. First, {1, 4, 5, 6, 7} is created in Phase I of *apriori-gen*. In Phase II, {1, 4, 6, 7}, {1, 5, 6, 7}, and {4, 5, 6, 7} are checked against the frequent Itemset-counter table through the frequent hash table for their existence. We assume all are frequent. The 5-candidate {1, 4, 5, 6, 7} is inserted into the new candidate Itemset-counter table, with its counter and sibling bit initialized to zero. Then we generate another potential 5-candidate {1, 4, 5, 6, 9} in Phase I of *apriori-gen*. In Phase II, {1, 4, 6, 9}, {1, 5, 6, 9}, and {4, 5, 6, 9} are checked against the frequent array structure. We know {1, 4, 6, 9} is there. We assume the other two are both frequent. Hence, {1, 4, 5, 6, 9} is appended to the new candidate Itemset-counter table, with its counter and sibling initialized to zero. Because {1, 4, 5, 6, 9} is an immediate sibling of {1, 4, 5, 6, 7}, the bit corresponding to {1, 4, 5, 6, 7} is set to "1" in the new candidate Sibling table. The last step is the processing of {1, 4, 5, 7, 9}. We assume it is also a candidate. It is appended to the new candidate array structure, with its counter and sibling bit reset. After the creation of the candidate array structure, the frequent Itemset-counter table is reserved for the rule discovery, while the frequent Hash table and frequent Sibling table are deleted. The candidate 5-itemsets are shown in Figure 2.2c.

## 4.2 Counting

So far, we have discussed the generation of frequent and candidate itemsets with the array structure. Next, we investigate the functions of the hash-tree in the counting phase and see how the array structure can provide the same functionality. We use the example shown in Figure 2.3 to illustrate the functions of the hash tree in the phase of counting. Internal nodes of such a tree are implemented as hash tables to allow fast selection of the next node. To reach the leaf for a set, start with the root and hash on the first item of the set. Reaching the next internal node, hash on the second item and so on until a leaf is found.

Consider now the transaction $T$ = {1, 4, 5, 6, 7}. *Apriori* needs to identify whether the combinatorial subsets with 4 items of $T$ are candidates. The set of subsets $SS$ of $T$ is {*s1*, *s2, s3*, *s4*}, where *s1*={1, 4, 5, 6}, *s2*={1, 4, 5, 7}, *s3*={1, 4, 6, 7}, and *s4*={4, 5, 6, 7}. Assume that all are candidates. So there are four candidates, *c1*, *c2, c3*, and *c4*, where *c1*=*s1*, *c2*=s2, c3=s3, *and* c4=s4.

Assume further that *c1* and *c2* are stored in a leaf node $LN_1$. Inside $LN_1$, there is another candidate, {1, 4, 5, 9}, which also has the prefix {1, 4, 5} but is not supported by *T*. Similarly, *c3* is stored in a leaf node $LN_2$ along with another

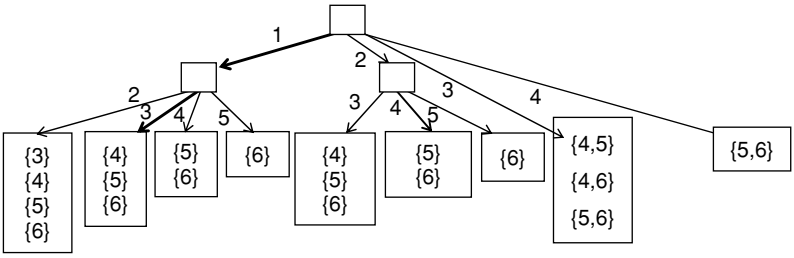


**Figure 2.3.** Hash tree structure for candidates

candidate {1, 4, 6, 9}. And *c4* is stored in a leaf node $LN_3$ with another candidate {4, 5, 6, 9}.

In order to identify the candidacy of the first two subsets, *s1* and *s2, Apriori* reaches $LN_1$ from the root by traversing first the edge labeled with item 1, then the one with item 4, and last the one with item 5. The edge selections are implemented as the hash-table loop-ups. *Apriori* tests items 1, 4, and 5 once to reach the leaf. Then it checks all the candidates in the leaf to determine whether they are supported by *T*. The first three items (1, 4, and 5) do not have to be considered any more, but for all the larger items *i* in a candidate set, we have to check whether $i \in T$. Here, the sets of the larger items are stored as item-lists, while the transaction is in the form of a bitmap. In our example, after reaching $LN_1$, we need one comparison to identify a candidate. So after another three comparisons, *s1* and *s2* are found to be candidates, the counters of *c1* and *c2* are increased by 1 separately, while there is no match for {1, 4, 5, 9}. For *s3*, after reaching $LN_2$, we need another two comparisons. Also, we need another six comparisons for *s4* after reaching $LN_3$. In Figure 2.4, the paths to locate sets {1, 4, 5, 6}, {1, 4, 5, 7}, {1, 4, 6, 7}, and {4, 5, 6, 7} are marked with bold arrows. The associated items are in bold.

The above example demonstrates three functions of the hash tree in *Apriori*:

- Store the candidate/frequent itemsets: *c1, c2, c3,* and *c4* are stored in the hash tree.
- Identify the status of a set of items, i.e., whether it is a candidate/frequent itemset: *s1, s2, s3,* and *s4* are candidates.
- Further, if an itemset is a candidate, locate the position of the candidate and its counter. The counters of *c1, c2, c3*, and *c4* are found and incremented.

In contrast to *Apriori*, we employ the Itemset-counter array to store the itemsets, along with the auxiliary Hash table and Sibling table to achieve the same functionality provided by the hash-tree:

- All the frequent itemsets and candidate itemsets are stored in the Array structure.
- For any given set of items, if it is a candidate/frequent itemset, the hash function maps it to a bucket within its hash table that points to an itemset in the

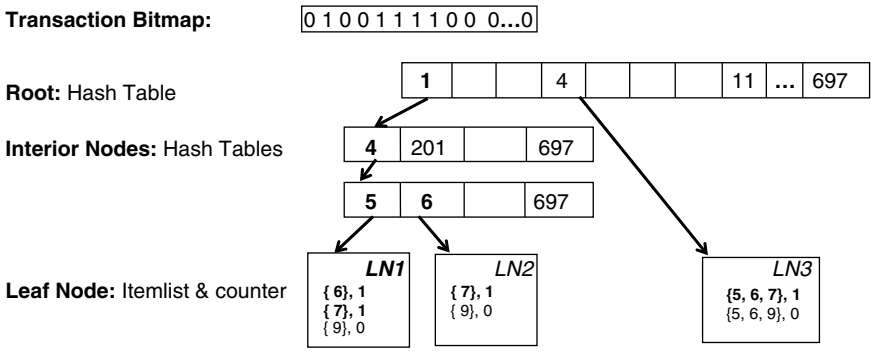


**Figure 2.4.** *Apriori* Hash Tree storage for itemsets

Itemset-counter table. The given itemset and the one in the table are the same, e.g., {1, 4, 5, 6} in Figure 2.2a. Otherwise, the hash function maps it to an empty bucket within its hash Table, e.g., {1, 6, 7, 8} in Figure 2.2a.

- Because both the itemset and its counter are stored together in the Itemset-counter table, once the itemset is located, the counter can be incremented quickly.

Let us use the same example as that for *Apriori*, $T$ = {1, 4, 5, 6, 7} to illustrate how the Array structure works. When comparing the transaction against the candidate itemset, instead of initializing a bitmap for each transaction, we generate clusters of possible candidate itemsets. For $SS$ = {*s1*, *s2, s3*, *s4* | *s1*={1, 4, 5, 6}, *s2*={1, 4, 5, 7}, *s3*={1, 4, 6, 7}, *s4*={4, 5, 6, 7}}, there are three clusters: *cluster1* = {1, 4, 5, 6, [7]}, *cluster2* = {1, 4, 6, 7}, and *cluster3* = {4, 5, 6, 7}. Because in a cluster, all the itemsets are the same except for their last items, we need only store item 7 instead of {1, 4, 5, 7} in *cluster1*. Then we compare each cluster with the candidate itemsets. For *cluster1*, the hash function leads *s1* to its corresponding position in the Itemset-counter table with four comparisons. The counter of the itemset increases by 1. Next, for *s2*=[7], there are two ways to check its candidacy. If the sibling chain is short, say, less than 4, we compare item 7 with the last item of the next itemset along the sibling chain until these items match, or until 7 is less than the last item of the next itemset along the chain. If the chain is long, for all the itemsets in the cluster, the hash function generates their addresses in the Itemset-counter table all at once, so we can check directly their existence in the Itemset-counter table. Because both the candidate itemsets and the itemsets in the clusters are stored in ascending order, the two methods generate the same results.

In our example, the sibling chain is three itemsets long, less than 4. Hence, on the fifth comparison, we compare 7 with the last item of {1, 4, 5, 7}. They match, so the counter of the next itemset increases by 1. Because there is nothing left in *cluster1*, we move on to *cluster2*. The hash function maps subset {1, 4, 6, 7} to its corresponding entry in the Itemset-counter table. With four comparisons, we match the subset with the itemset and increase the counter by 1. In the same manner, with four comparisons, we locate and increase the counter of {4, 5, 6, 7}.

# 5 ARM++: A HYBRID ALGORITHM FOR ASSOCIATION RULES

This section presents three new algorithms, which vary in their comparison methods, transaction filtering, and transaction transformation. As in *ArrayApr,* the candidate and frequent itemsets in all the new algorithms are stored with the new array-based data representation rather than the common hash-tree representation [2,3,4,6,7].

## 5.1 Methods of Comparison: *CmpApr*

For a transaction and a set of candidate itemsets, there are two ways to compare them. Existing *Apriori*-based algorithms [2,6] only compare the transaction against the candidate itemset by hashing the items in the transaction against the hash-tree. Up to now, all our discussions have been based upon this method,

namely, *subset comparison*. For example, in *ArrayApr*, in the $k^{th}$ iteration, given a transaction, for all subsets that are $k$-candidates, $k$ comparisons are needed to determine the candidacy of each subset. However, for the subsets that are not candidates, the comparison stops after the first mismatch between the candidate and the subset, so the number of comparisons might be less than $k$ for each subset. In this example, we assume that six comparisons are needed to determine the candidacy of a subset in the sixth iteration, no matter whether it turns out to be a candidate or not. With a transaction of 20 items, for the subset comparison method, ignoring the overhead of hashing, the number of item comparisons is $\binom{20}{6} * 6 = 232,560$.

However, there is another comparison method, namely, *candidate comparison*. It compares the candidate itemsets against the transaction. The transaction is initialized in a bitmap. We assume that the number of comparisons between a $k$-candidate and the transaction is $k$, though it might be less if the candidate is not supported by the transaction. We continue with the previous example. If there are 8,192 candidates in the $6^{th}$ iteration, the number of item comparisons is 8,192 * 6 = 49,152. In this case, it is obvious that candidate comparison performs better than its counterpart. Also, the candidate comparison method does not have the hashing overhead. The description of *candidate_compare( )* routine is given in Function 1 (see below).

Nevertheless, candidate comparison does not guarantee a smaller number of comparisons. For the same transaction, in the third iteration with 28,000 candidates, the subset comparison generates $\binom{20}{3} * 3 = 3,420$ comparisons, while candidate comparison requires 28,000 *3 = 84,000 comparisons.

***Candidate_compare***

```
1)  m=1
2)  while m <= | C_k|          % | C_k| is the number of candidates in C_k %
3)    if all items i in c_m ∈ T  % c_m is the m^th candidate in C_k %
4)      c_m.count ++
5)      m++
6)      while c_m is sibling of c_m-1     % skip the first k-1 items of the
                                          sibling candidates %
7)        if k^th item in c_m ∈ T
8)          c_m.count++
9)        end-if
10)       m++
11)     end-while
12)   else                        % skip all the sibling candidates %
13)     m++                       % because none is supported by T %
14)     while c_m is sibling of c_m-1
15)       m++
16)     end-while
17)   end-if
18) end-while
end.
```

**Function 1:** Candidate_compare

In our candidate comparison method, the comparison of sibling candidates within a cluster can be accelerated in the same fashion as described in subsection 2.2. After we find that a sibling candidate is supported by a transaction, its siblings only need to check whether or not their last items are in the transaction bitmap. This process is implemented in steps 6-11 of Function 1. Similarly, in steps 14–16, once we find that a candidate is not supported, all the comparisons of its siblings with the transaction are skipped. The candidate comparison benefits from our array structure, since, when we compare the items in a candidate with a transaction, all the items are stored adjacently.

Our new algorithm, *CmpApr*, is described in Algorithm 1 (see below). It is based upon both the subset comparison (step 8–11) and the candidate comparison (step 6). From the above example, we can see that in the early iterations, when we have a large number of candidates and a comparatively small number of subsets in a transaction, the subset comparison method is better. In the later iterations, when we have a small number of candidates and comparatively large number of subsets in a transaction, the candidate comparison method is preferable. Fortunately, when we start to process a transaction, we know the number of items in the transaction, the length of candidates, and the number of candidates. For a transaction with $|T|$ items in the $k^{th}$ iteration, we can precalculate the number of subsets, $\binom{|T|}{k}$. If it is smaller than the number of candidates, we select the traditional subset comparison method; otherwise, we use our candidate comparison method. Preference is given to the latter when the number of subsets equals the number of candidates, because the overhead of the hashing function is larger than that of the initialization of the transaction into a bitmap. The condition statement of step 5 incorporates the above selection criteria.

***CmpApr***

1) $L_0 = \emptyset$, $k = 1$
2) $C_1 = \{ \{i\} \mid i \in I \}$
3) **while** ( $C_k \neq \emptyset$ ) **do**
   % *count support* %
4) **forall** transactions $T \in D$
5) **if** ($estCmp(|T|, k) >= |C_k|$) % In *CmpApr*, $estCmp(|T|, k) = \binom{|T|}{k}$%
6) *candidate_compare*( $C_k$, $T$)
7) **else**
   % *ArrayApr* body: subset comparison %
8) $C_t$ = subset($C_k$, $T$)
9) **forall** $c \in C_t$
10) $c.count$ ++
11) **end-forall**
12) **end-if**
13) **end-forall**
14) $L_k = \{c \in C_k \mid c.count \geq n * s_{min}\}$
15) $C_{k+1}$ = generate_candidates($L_k$ )
16) $k$++
17) **end-while**
18) **return** $L = \bigcup_k L_k$

**end.**

**Algorithm 1** *CmpApr*

## 5.2 Online Transformation: *FilterApr*

This subsection describes the *FilterApr* algorithm, which is used for the subset comparison. This algorithm introduces two layers of filtering. The first is called *transaction transformation*, which occurs while the transactions are being read; the other is called *subset transformation*, which happens during the subset generation from transactions.

Within an iteration, if an item in a transaction is not part of the frequent itemsets supported by the transaction, it is *useless* since it contributes nothing to the generation of frequent itemsets; otherwise, it is *useful*. Processing the data without the useless items is vitally important. As mentioned earlier, *AprioriTid* and *Partition* outperform *Apriori* in the later iterations in that their underlying data structures, itemset-list and TID-list, store only the useful data. During the counting phase, both algorithms save the overhead of computation associated with items of no interest, whilst *Apriori* cannot efficiently trim the item-list structure and has to process the subsets containing useless items. Because *FilterApr* reads and then drops the useless items before checking the candidacy of the subsets of the transactions, the number of the comparisons in *FilterApr* is much less than that in *ArrayApr*, though the filtering in *FilterApr* is not as efficient as the built-in pruning of the useless items in itemset-list and TID-list.

### 5.2.1 Transaction Transformation

The essence of transaction transformation is to screen out useless items before the real processing. We achieve this by building a set of transaction filers derived from the candidate itemsets.

The items in a transaction that do not appear in any of the supporting frequent itemsets in the $k$th iteration can be dropped in the $k$th iteration. However, we have a problem in applying this property to practice. Before we finish the $k$th iteration, we don't know which candidate is frequent. A workable and less stringent property is that the items in a transaction that do not appear in any of the candidate itemsets in the $k$th iteration can be removed. Before the start of the $k$th iteration, we can build an *item filter* with only those items that appear in the $k$-candidates. The filter is implemented as a bitmap. In the $k$th iteration, all items that do not belong to the filter will be discarded; only items that exist in it will be processed.

For example, suppose we have only four candidate itemsets {1, 4, 5, 6}, {1, 4, 5, 7}, {1, 4, 5, 9}, and {1, 4, 6, 9} in the fourth iteration. A transaction $T$ = {2, 3, 4, 5, 6, 7, 9, 10}, with *item filter,* will be trimmed down to {4, 5, 6, 7, 9}. However, if we investigate the above example more carefully, we find there is no item 1 in the transaction, whereas item 1 is the very first item of all the candidate itemsets. This means that none of the itemsets is supported by the transaction. Therefore, without item 1, all the items in T are useless. Our example shows that the set of possible candidate items at a particular position of all candidate itemsets can determine the potential usefulness of an item in a transaction.

We call all the possible items at a position $j$ of the candidate $k$-itemsets the *necessary candidate items* of position $j$, denoted by $I_j$, where,

$$I_j = \{i_j \mid \ i_j \text{ is the } j^{\text{th}} \text{ item of } c \cap c \in C_k\}.$$

In our example, the necessary candidate items of position 1, $I_1$, is {1}, $I_2$ is {4}, $I_3$ is {5, 6}, and $I_4$ is {6, 7, 9}.

In order to use the necessary candidate items to filter the transactions, let us consider the procedure of the generation of the subsets of a transaction. In the $k$th iteration, from the start of a transaction $T$, the first item $t_1$ in $T$ can only be the first item of a subset. For $t_1$ to be useful, the subset or one of the subsets, in which $t_1$ is the first item, must be a candidate. Hence, the first useful item $t_1$ must belong to $I_1$, i.e., $t_1 \in I_1$. The second transaction item $t_2$ can be either the first or the second item of a subset. For $t_2$ to be useful, the subset or one of the subsets, in which $t_2$ is the either the first item or the second item, must be a candidate. Hence, the second useful item $t_2$ must belong to either $I_1$ or $I_2$, i.e., $t_2 \in I_1 \cup I_2$. Hence, for the useful $m^{\text{th}}$ item in transaction $T$, $t_m$, we have

$$t_m \in \bigcup_{1}^{m} \boldsymbol{I}_j, \text{ where } m < k.$$

The useful $k^{\text{th}}$ item and the useful items after it in a transaction have to appear in our item filter. Hence, we have

$$t_n \in \bigcup_{1}^{k} \boldsymbol{I}_j, \text{ where } n \geq k \cap n \leq |\mathrm{T}|.$$

If we look from the other side of the same transaction $T^R$, that is, from the end going backwards, the last useful item of a transaction, $t^R{}_1$, can only be the last item of some of the candidate itemsets, i.e., $t^R{}_1 \in I_k$. The second-to-last useful item of a transaction, $t^R{}_2$, can be either the last or the second-to-last of some of the candidate itemsets, i.e., $t^R{}_2 \in I_k \cup I_{k-1}$. Hence, for the $m^{th}$-to-last useful item in the transaction, $t^R{}_m$, we have

$$t^R{}_m \in \bigcup_{k}^{k-m+1} \boldsymbol{I}_j, \text{ where } m < k.$$

The $k^{\text{th}}$-to-last useful item and the useful items before it in a transaction have to appear in our item filter. Hence, we have

$$t^R{}_n \in \bigcup_{k}^{1} \boldsymbol{I}_j, \text{ where } n \geq k \cap n \leq |\mathrm{T}^R|.$$

Based upon our analysis of the subset generation from the transaction, we can derive the *possible transaction items* at position $j$ of a transaction from the necessary candidate items. The formal definition is in Figure 2.5. A graphical representation is shown in Figure 2.6.

We define *transaction_transform( )* in Function 2, as shown below. Forward possible transaction items are used in steps 1–9, the item filter is used in steps 11–19, and backward possible transaction items are used in steps 20–28. Before

Forward possible transaction items: $T_m = \bigcup_{1}^{m} Ij, (m = 1, 2, \ldots, k-1)$

Item filter: $T_k = \bigcup_{1}^{k} Ij,$

Backward possible transaction items: $T^R{}_m = \bigcup_{k}^{k-m+1} Ij, (m = 1, 2, \ldots, k-1)$

**Figure 2.5.** Transaction Transformation Filters

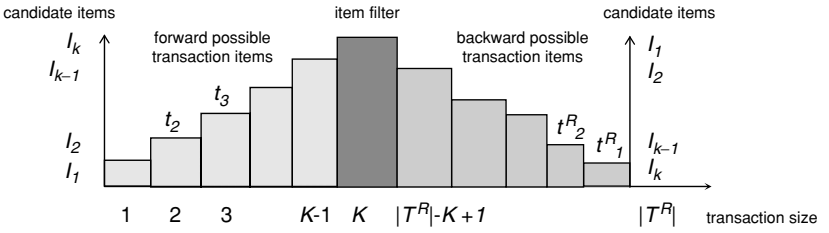


**Figure 2.6.** Possible Transaction Items

the start of the *k*th iteration, we can generate the possible transaction item filters from the candidate itemsets in the form of a bitmap. When we read a transaction, we apply the possible transaction item filters by invoking *transaction_transform( )* to remove the useless items from the transaction. After the invocation, if the length of the transformed transaction is not less than *k,* we continue to count its support; otherwise, the transaction is discarded, since it will not support any *k*-candidates. This process is implemented in step 30.

***Transaction_transform***

```
 1) m=1, j =1                    % Phase I: Forward possible transaction items %
 2) while (m<k ∩ j< |T|)         % transaction T %
 3)    if (t_j ∈ T_m)            % useful item %
 4)       m++, j++
 5)    else                      % useless, discarded %
 6)       mark t_j to be discarded
 7)       j++
 8)    end-if
 9) end-while

10) if (m is k ∩ j < |T|)        % k potentially useful items, items not
                                    transformed %
11)       while (j ≤ |T|)        % Phase II: Item filter %
12)          if (t_j ∈ T_k)
13)             m++, j++
14)          else
15)             mark t_j to be discarded
16)             j++
17)          end-if
18)       end-while
19)       adjust T to remove discarded item

20)       m=1, j=| T|            % Phase III: Backward possible transaction
                                    items %
21)       while (m<k ∩ j≥0)
22)          if (t_j ∈ T^R_m)    % useful item %
23)             m++, j--
24)          else                % useless, discarded %
25)             mark t_j to be discarded
```

```
26)            j--
27)          end-if
28)        end-while
29)  end-if
30)  adjust T to remove discarded item
31)  return T
end.
```

**Function 2:** *Transaction_transform*

Transaction transformation works on the transactions based upon the possible transaction items, which are generated from necessary candidate items according to the relationship between the items at a particular position in the transaction and the items at a particular position in the candidate itemsets.

### 5.2.2 Subset Transformation

Transaction transformation finishes before the generation of the subset. The next layer of filtering, *subset transformation*, works on the subsets generated from the transactions to reduce the combinatorial subset space for the support counting. We discover the inter-item relationships between the adjacent items of the candidate itemsets and use these heuristics to avoid the generation of useless subsets, which turn out to be small itemsets.

In the previous example, with only four candidates at the fourth iteration, namely, {1, 4, 5, 6}, {1, 4, 5, 7}, {1, 4, 5, 9}, and {1, 4, 6, 9}, and a transaction $T = \{1, 4, 5, 6, 7, 9\}$, the transaction transformation cannot trim $T$ any more. The subsets generated from $T$ with four items are

$$s_1 = \{1, 4, 5, 6\}, s_2 = \{1, 4, 5, 7\}, s_3 = \{1, 4, 5, 9\}, s_4 = \{1, 4, 6, 7\},$$
$$s_5 = \{1, 4, 6, 9\}, s_6 = \{1, 5, 6, 7\}, s_7 = \{1, 5, 6, 9\}, s_8 = \{1, 6, 7, 9\}.$$

From the candidate itemsets, we know that after the first item, 1, the only possible second item is 4. So only those subsets with the second item as 4 are generated. We have $s_1$, $s_2$, $s_3$, $s_4$, and $s_5$ left. After the second item 4, the possible third items are 5 or 6. The remaining five subsets have no problems. After the third item 5, the possible fourth items are 6, 7, or 9. $s_1$, $s_2$, and $s_3$ survive the test. After another third item, 6, the only possible fourth item is 9. $s_4$ is discarded and $s_5$ is generated. In the example, after our *possible subset item* test, subset $s_4$, $s_6$, $s_7$, and $s_8$ are discarded "on-the-fly" instead of being passed on to the hashing function to check their candidacy.

The heuristics behind the usage of inter-item relationships are these: when we generate a subset from the first item to the last, the set of $(j+1)^{\text{th}}$ possible subset items can be limited based upon the known $j^{\text{th}}$ item. In the example shown above, without the knowledge of the third item, we can only use the set of necessary candidate items at position 4, i.e., $I_4 = \{6, 7, 9\}$. We cannot filter any item. Once we know that the third item is 6, the fourth possible subset item is 9, so we can filter out $s_4$.

In order to save the inter-item relationship, we apply the module-$2^n$ ($n \geq 0$) operation on the item at the $(j-1)^{\text{th}}$ ($j > 1$) position of a candidate itemset. If the result is $i$, we add the $j^{th}$ item of the candidate to the $i^{\text{th}}$ set of possible subset

items. Actually, we split $I_j$, the sets of the necessary candidate items at the $j^{th}$ position, into $2^n$ sets of *possible subset items* (*PSI*). We denote the $i^{th}$ set of possible subset items at position $j$ by $PSI_{ji}$. There is an exception for $I_1$: it will not be divided, since there are no items before the first. The number, $2^n$, into which the possible subset items split the necessary candidate items is called the *splitting factor*. For fast detection, we select the splitting factor as a number to the power of 2.

In the $k^{th}$ iteration, similar to the *k possible transaction item* filters created for the transaction transformation, we build $k*2^n$ *possible subset item* filters, which are also in the form of a bitmap. The possible subset item filters of our previous example, with the splitting factor of 2, are shown in Figure 2.7. The dashed lines mark the module operations on the items.

The splitting factor is a measurement of how thoroughly *PSI*s represent the inter-item relationships among the candidate itemsets. With a splitting factor of 1, *PSI*s reduce to the possible candidate itemsets. The larger the splitting factor, the more fully *PSI*s represent the inter-item relationship, and the better they screen out useless subsets. However, the memory requirement of *PSI*s increases linearly with the splitting factor. The trade-off of the space-and-time problem of the splitting factor is further investigated with experimental results in subsection 6.2. Subset transformation is based upon the set of PSIs and is described below as Function 3. For each subset, *subset_transform* marks its usefulness.

***Subset_Transform***

```
 1) set c useful
 2) m=2                          % start from the second item %
 3) while (m ≤ k)
 4)    previous = c_{m-1} MOD 2^n % calculate which set of PSI %
 5)    if c_m in PSI_{m, previous} % subset item in the Possible Subset Items %
 6)       m++                    % check next subset item %
 7)    else
 8)       set c useless
 9)       break                  % skip to next subset %
10)    end-if
11) end-while
end.
```

**Function 3:** Subset_Transform

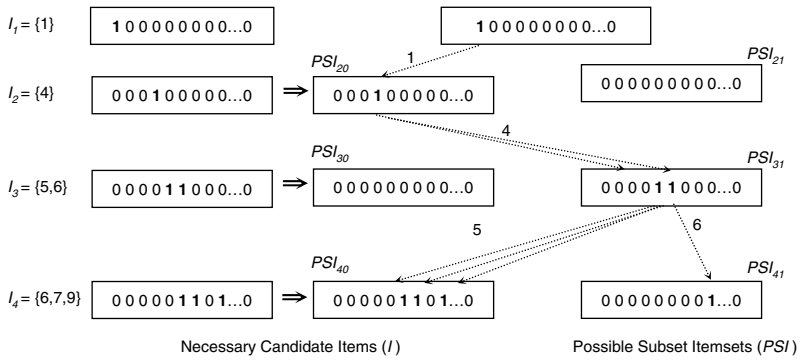


**Figure 2.7.** Possible Subset Items

To integrate transaction transformation and subset transformation, we come up with a new routine, *filterCount( )*. It applies the transaction transformation in step 3 and the subset transformation in step 8, as defined in Function 4. Firstly, the transaction transformation reduces the number of items in the transactions to be processed in the counting phase. Secondly, the subset transformation reduces the number of subsets to be detected for candidacy.

***filterCount( )***
1) **forall** transactions $T \in D$
2) % Transaction transformation %
3) $T$ = *transaction_transform*($T$)
4) **if** $|T| \geq k$
5) $C_t$ = subset($C_k$, $T$)
6) **forall** $c \in C_t$
7) % Subset Transformation %
8) *subset_transform*($c$)
9) **if** $c$ is useful
10) $c$.count ++
11) **end-if**
12) **end-forall**
13) **end-if**
14) **end-forall**
**end.**

**Function 4:** *filterCount*()

To end this section, we propose a new algorithm, *FilterApr*, as shown in Algorithm 2 below. It uses *filterCount( )* in step 4 to find all the frequent itemsets.

***FilterApr***
1) $L_0 = \emptyset$, $k = 1$
2) $C_1 = \{ \{i\} \mid i \in I \}$ % all *1*-itemsets %
3) **while** ( $C_k \neq \emptyset$ ) **do**
4) *filterCount*()
5) $L_k = \{c \in C_k \mid c.count \geq n * s_{min}\}$
6) $C_{k+1}$ = *apriori_gen*($L_k$)
7) $k$++
8) **end-while**
9) **return** $L = \bigcup_k L_k$
**end.**

**Algorithm 2**: FilterApr

## 5.3 *ARM*++: A Fast Algorithm

In this section, we combine *FilterApr* with *Partition* [7] to propose a new algorithm, *ARM*++, as defined in Algorithm 3. It is a hybrid of *FilterApr* in the early passes (*FilterApr* phase) and *Partition* in the subsequent passes (*Partition* phase).

The pivot point is that whenever the estimated TID-list of *Partition* can be held in memory, we switch from *FilterApr* to *Partition*.

### 5.3.1 Implementation of *ARM++*: Partition Phase

Being similar to *Partition*, in steps 24–31 of the partition phase, *ARM++* works with the TID-list representation. The count for a candidate is determined immediately after it has been generated from two frequent sets. To compute the count, the TID-lists of the two frequent sets are joined using a merge-join.

One minor difference between *ARM++* of the partition phase and *Partition* is that *ARM++* uses the same Array structure to store frequent sets and the same candidate generation technique as those in *ArrayApr*. Use of the same data structure and the candidate generation code further simplifies the comparison between TID-lists and item-lists, because our results are not obscured by different storage and candidate generation procedures.

### 5.3.2 No Partitioning of data

The very reason *Partition* divides the data into several parts is that it cannot keep all the TID-lists in memory, especially in the early iterations. With the iteration number increasing, the number of candidates decreases sharply. Also, with the length of the candidate itemsets increasing, they are less likely to be supported by transactions. Hence, in the later iterations, it is possible to cache all the TID-lists in memory if it is not possible in the early iterations.

When the size of the TID-lists exceeds the amount of free memory, the data that cannot be held in memory will be swapped onto the disk by the virtual memory system. This process is not only time-consuming but also not always possible. Given a large database that occupies nearly all the disk space, there might not be enough space for the swapping area. For example, with our 79.6 MB simulation data containing only 1,000,000 transactions, for support as low as 0.25%, with an average transaction size as long as 20 and an average itemset length of 6, in the third iteration, there are 12,933 frequent itemsets. The minimum length of the TID-lists is 2,500, and each TID takes 4 bytes. Hence, we need a minimum of 123 MB to store the TID-lists before the start of the third iteration. With physical memory of 64 MB, and free disk space of 64 MB, my computer cannot run *Partition*, since there is not enough space to store the data in the format of a TID-list. After eight or more iterations, the memory requirement to store the TID-lists of the candidate itemsets drops to no more than 25 MB, so one partition is enough. In this case, my computer can run *Partition* from the eighth iteration.

Based upon the above analysis, with large databases on the disk, it is likely that we do not have enough free space to store the intermediate TID-lists. So we implement *ARM++* as a hybrid of *FilterApr* and *Partition*. In the early iterations, before the TID-lists can be held in memory in step 6, we adopt *FilterApr*. Once we can start *Partition* without splitting the data, we transform the data from item-list format into TID-list in steps 9–22 and switch to *Partition*. In the partition phase, *ARM++* has only one partition, so the whole TID-list is held in memory; there is no extra disk space needed to store the intermediate TID-list,

as in the case of multiple partitioning. Another advantage is that we can test the performance of the TID-list data structure against that of the item-list in the later iterations without the impact of partitioning.

***ARM++***

1) $L_0 = \emptyset$, $k = 1$
2) $C_1 = \{ \{i\} \mid i \in I \}$
3) transformed-to-TID = false
4) **while** ( $C_k \neq \emptyset$ ) **do**
5) **if**( $\sum_{|C_{k+1}|} \min_k |C_k|$ > available mem) % the estimated size TID-lists vs. avail. mem.%
6) *filterCount*()
7) **else**
% transfer from item-list to TID-list %
8) **if** NOT transformed-to-TID
9) **forall** transactions $T \in D$
10) $T$ = *transaction_transform*( $T$ )
11) **if** $|T| \geq k$
12) $C_t$ = subset($C_k$, $T$)
13) **forall** $c \in C_t$
14) *subset_transform*($c$)
15) **if** $c$ is useful
16) $c$.count ++
17) T($c$) + = T.id % add transaction id to tid-list %
18) **end-if**
19) **end-forall**
20) **end-if**
21) **end-forall**
22) transformed-to-TID = true
23) **else**
% *Partition Phase*%
24) **forall** candidates $c$ of size $k$
25) $T(c)$ = generate_TID_list($c$)
26) **if** ($|T(c)| \geq n * S_{min}$)
27) $L_k = L_k \cup \{c\}$
28) **else**
29) drop_candidate($c$)
30) **end-if**
31) **end-forall**
32) **end-if**
33) $L_k = \{c \in C_k \mid c.count \geq n * s_{min}\}$
34) $C_{k+1}$ = *apriori_gen*( $L_k$ )
35) $k$++
36) **end-while**
37) **return** $L = \bigcup_k L_k$

**end.**

**Algorithm 3:** *ARM++*

### 5.3.3 Estimation of the size of intermediate TID-list data

When we implement the above strategy, we need to determine the size of the TID-lists of all ($k$+1)-candidates before the start of the ($k$+1)$^{th}$ iteration. We can calculate the potential maximum size of the data when we use *apriori-gen* to generate the $k$+1 candidates.

After the $k^{th}$ scan, we already know the support of each $k$-frequent itemset. Based upon the first property of *a priori*, the support of any $k$+1 frequent itemset is equal to or less than that of its child $k$-frequent itemset with the smallest support. In *Partition,* the support for a candidate is generated at the same time the candidate is generated. If the count is no less than the minimum support, the candidate becomes a frequent itemset; otherwise, it is discarded. The length of the TID-list of a frequent/candidate itemset $c_{k+1}$ is actually its support. Hence, the maximum possible length of the TID-list of the candidate, $|c_{k+1}|_{max}$, is the minimum of all the supports of the $k^-$ containing frequent itemsets of the candidate, i.e., $|c_{k+1}|_{max} = \min_k |C_k|$, where $c_k \subset c_{k+1}$. For example, given four 3-frequent itemsets {3 169 377}, {3 169 555}, {3 337 555}, and {169 337 555} with their supports, i.e., 326, 327, 333, and 310, respectively, the support of {3 169 377 555} cannot exceed 310.

Before the start of the ($k$+1)$^{th}$ iteration, we have gathered all the supports for $k$-frequent itemsets. In step 34, when we derive $k$+1 candidates from $k$-frequents, for each generated candidate, we can calculate the maximum possible length of its TID-list, $|c_{k+1}|_{max=} \min_k |C_k|$. The sum of such lengths associated with all candidates, $\sum_{|C_{k+1}|} \min_k |C_k|$, is the estimation of the size of the TID-list data of the ($k$+1)$^{th}$ iteration. In step 5, if the sum is equal to or less than the size of the free memory, we know if we start to transform the data from item-list to TID-list in the $k$+1 iteration, we do not need to swap the resulting data. In this case, while we count the supports of the candidates in the $k$+1 iteration using modified *FilterApr* in steps 9–22, if a transaction includes some candidates, we save the ID of the transaction into the TID-list buffers associated with the candidates. After the $k$+1 iteration, we enter the partition phase of *ARM++*.

### 5.3.4 Combining 1-itemsets and 2-itemsets counting

Let us consider the performance of the TID-list and item-list. It is in the later iterations that the savings on the computation of irrelevant items give *Partition* an edge over *FilterApr*. However, in the second iteration, *FilterApr* outperforms *Partition*. Consider a database of $m$ = 1,000 items, all of which we assume to be frequent, when the support is very low. This means that all 2-combinations of those items, $m$*($m$-1)/2, at the level of 500,000 candidates have to be evaluated by *Partition* in pass 2. Assume further that there are 10,000,000 transactions with an average of 20 items. The average length of a TID-list for a *1*-itemset is therefore 10,000,000*20/1,000 = 200,000 TIDs. One merge-join to count a candidate requires as many comparisons as there are items in the longer list; thus, 500,000 * 200,000 = $10^{11}$ comparisons are necessary during pass 2. This figure is usually

even larger because the lists that are longer than average cause more comparisons than assumed here. We can estimate the number of hash operations performed by *FilterApr*. In iteration 2, with so large a number of candidate itemsets, *FilterApr* would use subset comparison based upon the Array structure. Again, we assume that all items are frequent. The approximation of the comparison is $(_2^{20})$* 10,000,000 = 3.8* $10^9$.

As shown in the above example, in the second iteration, both the *Apriori* and *Partition* require a large number of comparisons to locate the candidate itemsets. We can optimize the counting in the second iteration by counting the support for 2-candidates directly, saving all the comparison overheads. Further, the direct counting can be done in the first scan of the database. We can combine the *1*-itemset and *2*-itemset counting in the first iteration, saving the I/O cost of one scan of the data. The performance results of all the above algorithms, *ArrayApr, CmpApr, FilterApr,* and *ARM*++, in Section 5 are generated with this optimization.

## 6 PERFORMANCE ANALYSIS

This section illustrates the performance of the proposed algorithms. In particular, we demonstrate the effects of online transformation of transactions, which significantly reduce the CPU overhead in the early iterations. Also, we present the efficiency of TID-lists in the later iterations whenever the resources needed for execution are available. We evaluate the algorithms with two different methods. The first is based upon the execution time of different algorithms listed in Figures 2.8, 2.9, and 2.10. It gives preference to the actual execution time of the different parts of the algorithms. However, the implementation tools and underlying execution environment also have direct impact on the execution time. This makes the comparison result of algorithms tested on different platforms obscured by factors other than the algorithms themselves. The second method is based upon the number of integer comparisons involved in the algorithm of the frequent itemset discovery, as specified in Table 2.2. Because it is independent of the implementation tools and testing platform, this method genuinely reflects the efficiency of the algorithm.

All our algorithms use the *a priori* [2] optimization to reduce the number of candidate itemsets. In addition, *CmpApr* adopts different comparison methods to reduce the number of comparisons. *FilterApr* reduces the combinatorial search space by cutting the number of items in the transactions as well as the number of subsets of the transactions. In the early iterations, *FilterApr* outperforms

**Table 2.2.** Number of comparisons to determine the candidacy of itemsets

| Algorithm | No. Subset Comparisons | No. Candidate Comparisons | No. TID Comparisons | Total | Time (Sec.) |
|---|---|---|---|---|---|
| *ArrayApr* | 3,735,752,027 | 0 | 0 | 3,735,752,027 | 20,952.00 |
| *CmpApr* | 518,788,343 | 1,924,005,176 | 0 | 2,442,793,519 | 1,856.92 |
| *FilterApr* | 191,467,720 | 0 | 0 | 191,467,720 | 174.66 |
| ARM++ | 85,722,627 | 0 | 17,432,961 | 103,155,588 | 108.42 |
| Item-list ideal | 137,343,148 | 0 | 0 | 137,343,148 | N/A |

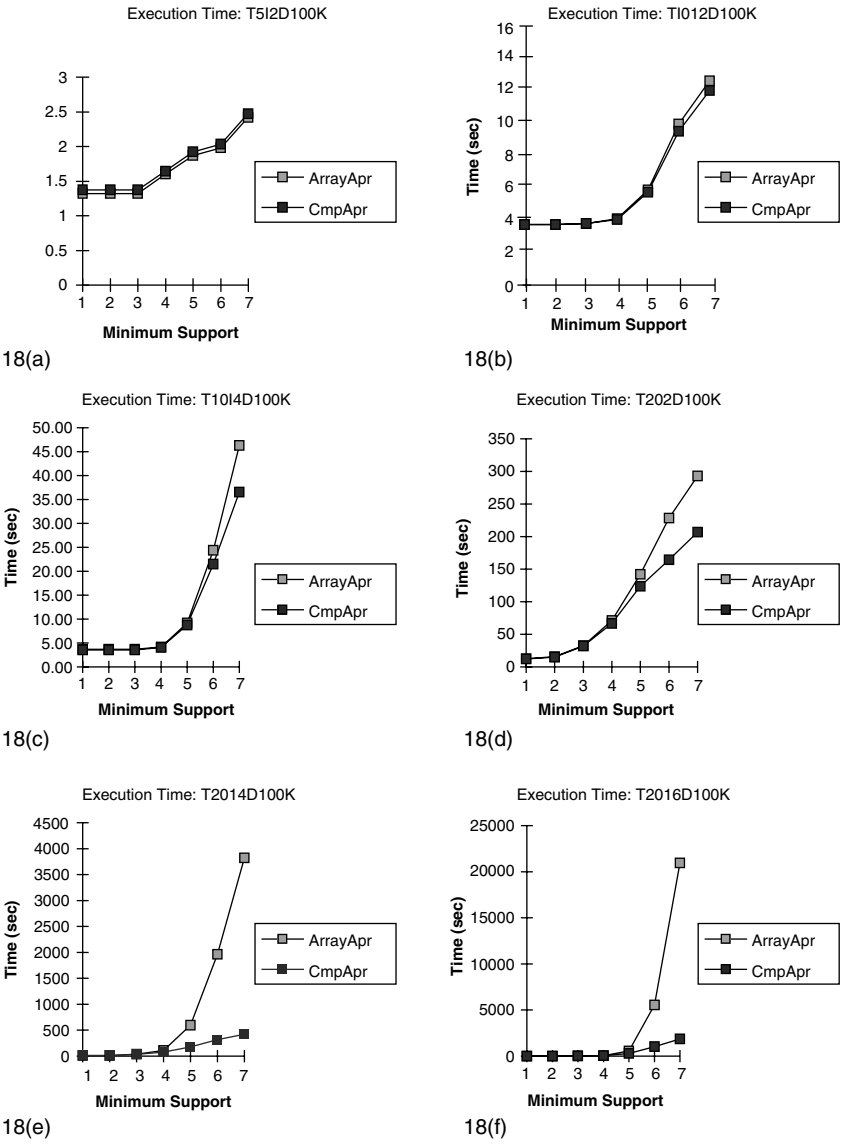


**Figure 2.8.**

*Partition,* which might require too much space to hold the intermediate result, thereby making it impossible to execute. However, *Partition* [7] needs only one comparison to determine the existence of a candidate itemset in a transaction, while *FilterApr* needs $n$ comparisons in the $n^{\text{th}}$ iteration. That is the reason why *Partition* outperforms *FilterApr* in later iterations. As a compromise of *FilterApr* and *Partition*, *ARM++* also considers the availability of resources. It executes *FilterApr* in the early iterations when resources are not enough for *Partition*. Then it shifts to *Partition* whenever the resources are available for execution.

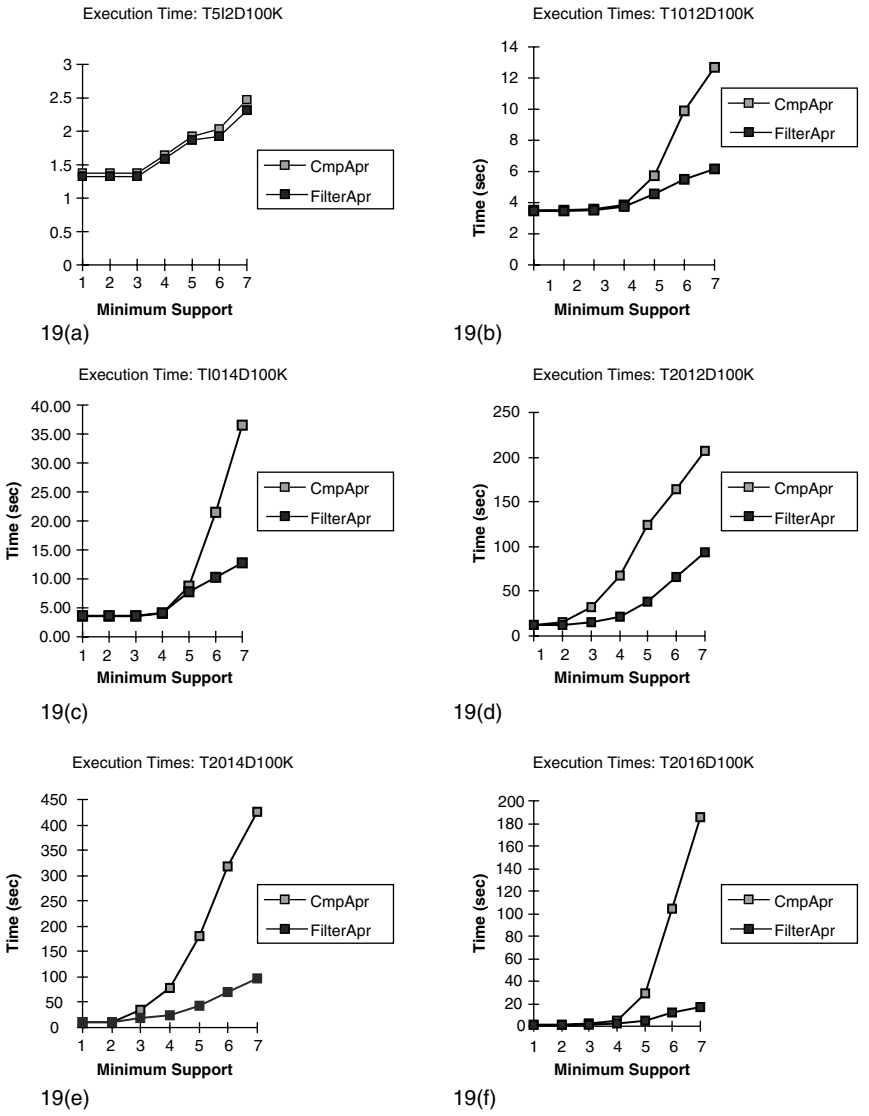


**Figure 2.9.** Execution times for *CmpApr* and *FilterApr*

## 6.1 Compare Candidate Comparison and Subset Comparison

In this section, we compare the performance of *ArrayApr* and that of *CmpApr*. *ArrayApr* is our implementation of the *a priori* optimization on the candidate itemsets stored in the array structure. Its performance shows the effect of *a priori* without any other heuristics. In addition to the subset comparison used in *ArrayApr*, *CmpApr* selectively uses candidate comparison to reduce the number of comparisons and thus reduces the overall computation time.

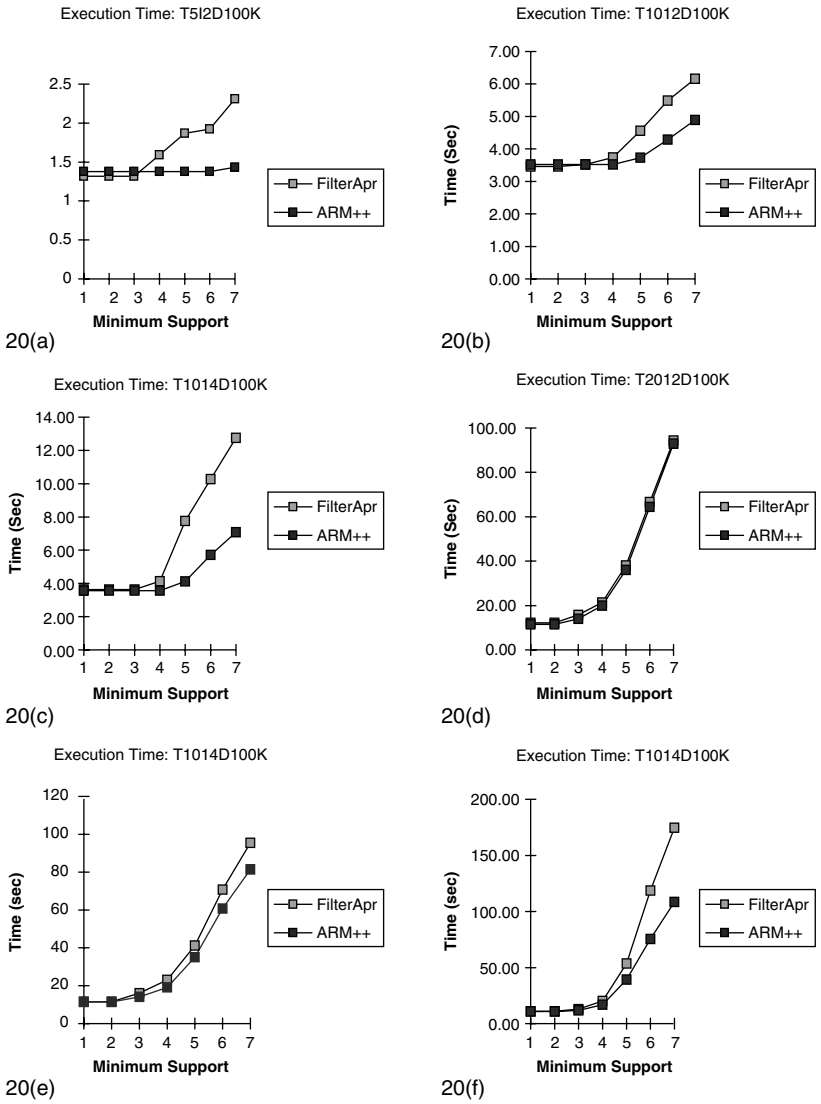


**Figure 2.10.** Execution times for *FilterApr* and *ARM++*

***Execution time.*** Figure 2.8 shows the execution time for the six synthetic datasets of 100,000 transactions given in Table 2.3 for decreasing values of minimum support. In the figure, as well as in Figures 2.9 and 2.10, the values of 1, 2, 3, 4, 5, 6, and 7 on the *X*-axis represent the minimum support threshold of 2%, 1.5%, 1%, 0.75%, 0.5%, 0.33%, and 0.25%, respectively. As value on the *X*-axis increases from 1 to 7, the minimum support drops from 2% to 0.25%, and the execution times of the algorithms increase. This is because with the decrease of the minimum support, the total numbers of candidate itemsets and of frequent itemsets increase, both of which take more time to generate. Moreover, if we compare Figure 2.8(a) through Figure 2.8(f), we find that as the average length of transactions increases, the execution time increases. Further, for the same average length

**Table 2.3.** Synthetic Data Sets

| Name | T | I | D | Data Size(corr=0.5, c=0.5) |
|---|---|---|---|---|
| T5.I2.100K | 5 | 2 | 100,000 | 2.33 MB |
| T10.I2.100K | 10 | 2 | 100,000 | 4.19 MB |
| T10.I4.100K | 10 | 4 | 100,000 | 4.23 MB |
| T20.I2.100K | 20 | 2 | 100,000 | 7.99 MB |
| T20.I4.100K | 20 | 4 | 100,000 | 7.97 MB |
| T20.I6.100K | 20 | 6 | 100,000 | 7.97 MB |
| T5.I2.500K | 5 | 2 | 500,000 | 11.6 MB |
| T10.I2.500K | 10 | 2 | 500,000 | 20.9 MB |
| T10.I4.500K | 10 | 4 | 500,000 | 21.1 MB |
| T20.I2.500K | 20 | 2 | 500,000 | 39.9 MB |
| T20.I4.500K | 20 | 4 | 500,000 | 39.8 MB |
| T20.I6.500K | 20 | 6 | 500,000 | 39.8 MB |
| T5.I2.1M | 5 | 2 | 1,000,000 | 23.3 MB |
| T10.I2.1M | 10 | 2 | 1,000,000 | 41.9 MB |
| T10.I4.1M | 10 | 4 | 1,000,000 | 42.3 MB |
| T20.I2.1M | 20 | 2 | 1,000,000 | 79.9 MB |
| T20.I4.1M | 20 | 4 | 1,000,000 | 79.7 MB |
| T20.I6.1M | 20 | 6 | 1,000,000 | 79.6 MB |
| T10.I4.2M | 10 | 4 | 2,000,000 | 84.6 MB |
| T10.I4.5M | 10 | 4 | 5,000,000 | 211 MB |
| T10.I4.10M | 10 | 4 | 10,000,000 | 423 MB |

of transactions, with the increase of the average length of itemsets, the execution time also increases. Both these outcomes result from the increase of the numbers of frequent itemsets and of candidate itemsets.

With a small average length of transactions, small average length of itemsets, and high minimum support rate, the numbers of candidate itemsets and of frequent itemsets are much less than those with large average length of transaction, large average length of itemsets, and low minimum support rate. We can see that the "easiest" dataset is T5I2D100K at the highest support setting of 2%, while the "hardest" is T20I6D100K at the lowest support setting (0.25%).

It is with the hard dataset that the effect of reduction of the algorithm search space can show up. With the easy dataset, the gain in the reduction of the number of candidate and frequent itemsets is so small that it might not offset the extra complexity introduced. As to the performance comparison of *ArrayApr* and *CmpApr,* for example, the execution times of *CmpApr* on T5I2D100K are slightly longer than those of *ArrayApr* with all settings of minimum support. With T10I2D100K, the speed gain of *CmpApr* over *ArrayApr* is marginal. It is with the hardest dataset that the true efficiency of *CmpApr* is fully represented. Therefore, in the following discussion, we will focus on the performance of the algorithms on the hardest dataset, both in terms of the time and the number of comparisons that occurred.

As shown in Figure 2.8(f), the improvement in execution times for the hardest dataset is quite significant. The execution time improves from 20,952 seconds to 1,856.92 seconds. Since both *ArrayApr* and *CmpApr* use the same candidate itemset generation technique and process the same transactions, the latter mainly benefits from the candidate comparison method. Subset comparisons are much more expensive than candidate comparison because of the overhead of hashing func-

tions. Before each subset comparison, the position of the subset has to be calculated based upon the content of the subset. The longer the subset, the higher the overhead of hashing. Though special optimization has been implemented on the hashing calculation, it is still very expensive, considering the fact that it is required for each subset. In contrast, for candidate comparison, the candidates are stored in the array structure sequentially. The comparisons are conducted in the order of the candidate itemset, so there is no extra cost in determining the positions of candidate itemsets.

## 6.2 Transform Transactions and Subsets

This subsection compares the performance of *CmpApr* and that of *FilterApr*. *CmpApr* uses a different comparison method to reduce the number of item comparisons as well as the cost of each comparison, while *FilterApr* reduces both the number of items in the transactions and the subsets of transactions.

***Execution time.*** Figure 2.9 shows the execution times of both *CmpApr* and *FilterApr*. With all the data sets, *FilterApr* outperforms *CmpApr*. Especially for the hardest data set, the execution time drops significantly from 1,856.92 seconds to 174.66 seconds. Since *FilterApr* only uses the comparatively slower subset comparison, the improvement is mainly due to the significant reduction in the number of subsets of transactions. There is overhead associated with the transaction transformation, which processes data at the speed of 1 MB/second. We derive this number by subtracting the sequential input throughput with transformation of about 4 MB/second from the measured raw sequential of about 5 MB/second. Compared with the time saved, this optimization is very effective. One good feature about subset filtering is that the longer the subset, the more information about the interrelationship between the adjacent items, the more powerful the transaction transformation, and the more significant the reduction on the execution time.

***Number of comparisons.*** In subset comparison, for each subset of a transaction, we need to determine whether it is a candidate or not. The subsets of a transaction are generated combinatorially from the items of the transaction. Transaction transformation reduces the number of subsets by filtering out the useless items and the unnecessary subsets generated from the retained items. In Table 2.2, the total number of item comparisons of *FilterApr* is 7.84% of that of *CmpApr*, and execution time of *FilterApr* is 9.4% of that of *CmpApr*. We can see that the filters increase the hit-ratio by removing over 92% of futile item comparisons.

## 6.3 Integrate FilterApr and Partition

Here we compare the performance of *FilterApr* and that of *ARM++*. Because *FilterApr* and *ARM++* share the same algorithm in the early iterations, actually we compare the performance of *FilterApr* and the *Partition* phase of *ARM++* in the later iterations. Although *FilterApr* has employed several new optimizations to improve its performance, in the later iteration, the TID-list underlying *ARM++*

beats the item-structure behind *FilterApr*. In the $k^{th}$ iteration, to compute a count for a candidate itemset, *ARM++* needs only one comparison of the TIDs of its two sub-itemsets, while *FilterApr* needs o($k$) comparisons, i.e., the comparisons of the $k$ items of the subset with the items in one or multiple hash buckets. Recall *itemset-list* also needs o($k$) comparisons. Inherently, the TID-list is the best among the three possible structures in later iterations. However, the size of the TID-list is in proportion to the number of the transactions in the database. For data with the same support for the same number of frequent itemsets, the length of TID-lists of the database with 10,000,000 transactions would be 100 times those of the database with 100,000 transactions. In contrast, *FilterApr* needs no extra memory to hold the database. It works on the original database, and the processing can be accelerated with the help of filters.

***Execution time.*** Figure 2.11 shows the execution times of both *FilterApr* and *ARM++*. For the hardest data set, the execution time drops from 174.66 seconds down to 108.42 seconds. As discussed above, the improvement is due to the adoption of *Partition* in the later iterations of execution. Similar to the subset comparison, there is hashing overhead associated with the TID comparison. Therefore, TID comparison is much quicker than the candidate comparison. However, in *ARM++*, there is an overhead to transform the data from the item-structure to the TID-structure when switching from *FilterApr* to *Partition*. As to the memory requirement, the filters in *FilterApr* requires less than 104 KB memory. When *ARM++* switching to the *Partition* phase, we use all the available 64 MB of memory to store the intermediate TID-lists.

***Number of comparisons.*** In Table 2.2, for *ARM++*, there is a new column, *No. TID Comparisons*, to summarize the number of item comparisons based upon the TID-lists. The total number of item comparisons of *ARM++* is 53.88% of that of *FilterApr*, and the execution time of *ARM++* is 62.07% of that of *FilterApr*. We can see that the adoption of the TID-list increases the hit-ratio by removing over 46% of the unnecessary item comparisons.

In the first column, we list an *Item-list Ideal* algorithm, which represents the perfect algorithm based upon item-structure where none of the item comparisons is related to any small itemsets. Actually, the total number of item comparisons of *ARM++* is only 75% of that of the *Item-list Ideal* algorithm. This demonstrates that the underlying TID-list data structure can provide a more efficient comparison method than the item-structure in the later iterations.

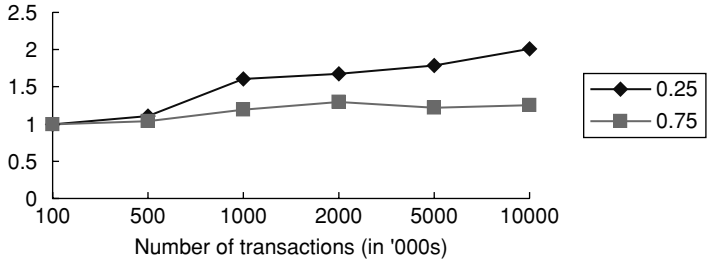


**Figure 2.11.** Number of transactions scale-up: ARM++

***Scalability.*** Figure 2.11 shows the scalability of the *ARM++* when the number of transactions scales up. The number of transactions ranges from 100 K, 500 K, 1 M, 2 M, 5 M, up to 10 M. The minimum supports of the experiments are set to 0.25% and 0.75%. *ARM++* scales linearly with the increasing number of transactions.

## 6 CONCLUSIONS

Based upon our study of association rule mining, we have proposed a sequential algorithm, *ARM++*, which achieves better performance with the available resources and displays near-linear scale-up behavior. We believe that *ARM++* is the first attempt to integrate different algorithms based upon the available resources. In the early iterations, it requires fewer resources than *Partition*, and in the late iterations, it performs faster than *FilterApr*. In our analysis of different algorithms, we compare both their execution times and the number of comparisons involved. The execution of different algorithms at different stages is performance oriented and resource based. The flexibility of the approach enables us to integrate the latest research result in the association rule mining and related field.

Unfortunately, the utilization of computer power was limited to a single machine due to the sequential nature of our algorithm, so our future work will consist of extending the proposed algorithms in a context of heterogeneous environments. Several algorithms have been proposed [3,4,6] that aim to reduce execution time by running on multiple machines and minimizing costly intercommunication. However, all these algorithms are designed for parallel machines or homogeneous network environments, where the performance of each node or machine is the same or similar and the connection is reliable and fast. In a heterogeneous network environment, the power of each machines varies, and the throughput of network connection between different machine varies. Because usually there are multiple jobs running at the same time, the local resources, i.e., CPU, memory, disk, and communication resources, change over time. The challenge is to maximize performance while using minimum resources.

## ACKNOWLEDGMENT

This project is supported by the ARC (Australian Research Council) Linkage-project LP0347217 (titled “Designing a Scalable and Robust Infrastructure for Highly Dynamic Web Services”) and SUN Microsystems.

## REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami (1993): Mining Association Rules between Sets of Items in Large Databases. *Proc. SIGMOD International Conference on Management of Data*, Washington, DC, pp. 207–216.

[2] R. Agrawal and R. Srikant (1994): Fast Algorithms for Mining Association Rules. *Proc. Very Large Database International Conference*, Santiago, pp. 487–498.
[3] R. Agrawal and J.C. Shafer (1996): *Parallel Mining of Association Rules*: *Design, Implementation and Experience*. Research Report RJ10004, IBM Almaden Research Center, San Jose.
[4] D. W. Cheung, V. T. Ng, A. W. Fu, and Y. Fu (1996): Efficient Mining of Association Rules in Distributed Databases. *IEEE Trans. Knowledge Data Eng*. *8*(6), 911–921.
[5] M. Houtsma, and Arun Swami (1995): Set-Oriented Mining for Association Rules in Relational Databases. *IEEE Int. Conf. on Data Engineering* pp. 25–33.
[6] J.S. Park, M-S Chen, and P. S. Yu (1995): An Effective Hash-Based Algorithm for Mining Association Rules. *Proc. SIGMOD Int. Conf. Management of Data*, pp. 175–186.
[7] A. Savasere, E. Omiecinski, and S. Navathe (1995): An Efficient Algorithm for Mining Association Rules in Large Databases. *Proc. Very Large Database Int. Conf*., Zurich, pp. 432–444.
[8] R. Srikant and R. Agrawal (1996): Mining Quantitative Association Rules in Large Relational Tables. *Proc. SIGMOD Int. Con. on Management of Data*, Montreal, pp. 1–12.
[9] http://www. almaden.ibm.com/cs/quest/syndata.html
[10] R. Srikant and R. Agrawal: Mining Quantitative Association Rules in Large Relational Tables. *SIGMOD 96*, Montreal, pp. 1–12.
[11] D. Hand, H. Mannila, and P. Smyth (2001): *Principles of Data Mining*. MIT Press, Cambridge, MA.
[12] J. Han and M. Kamber (2001): *Data Mining—Concepts and Techniques*. Academic Press, New York.