LGen — A Lattice-Based Candidate Set Generation Algorithm for I/O Efficient Association Rule Mining

Chi-Lap Yip K. K. Loo Ben Kao David Cheung C. K. Cheng

Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong.
{*clyip, kkloo, kao, dcheung, ckcheng*}*@csis.hku.hk*

Abstract

Most algorithms for association rule mining are variants of the basic `Apriori` algorithm [1]. One characteristic of these Apriori-based algorithms is that candidate itemsets are generated in rounds, with the size of the itemsets incremented by one per round. The number of database scans required by Apriori-based algorithms thus depends on the size of the largest large itemsets. In this paper we devise a more general candidate set generation algorithm, `LGen`, which generates candidate itemsets of multiple sizes during each database scan. We show that, given a reasonable set of suggested large itemsets, `LGen` can significantly reduce the number of I/O passes required. In the best cases, only two passes are sufficient to discover all the large itemsets irrespective of the size of the largest ones.

Keywords: Data mining, association rules, lattice, Apriori, LGen

1 Introduction

Data mining has recently attracted considerable attention from database practitioners and researchers because of its applicability in many areas, such as decision support, market strategy and financial forecasts. Combining techniques from the fields of machine learning, statistics and databases, data mining enables us to find out useful and invaluable information from huge databases.

Mining of association rules is a research topic that has received much attention among the various data mining problems. Many interesting works have been published recently on this problem and its variations [10, 1, 4, 3, 6, 7, 8, 9, 11]. The retail industry provides a classic example application. Typically, a sales database of a supermarket stores, for each transaction, all the items that are bought in that transaction, together with other information such as the transaction time, customer-id, etc. The association rule mining problem is to find out all inference rules such as: "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 not known beforehand. Such rules are very useful for marketers to develop and to implement customized marketing programs and strategies.

The problem of mining association rules was first introduced in [10]. In that paper it was shown that the problem could be decomposed into two subproblems:

1. Find out all *large itemsets* and their support counts. A large itemset is a set of items which are contained in a sufficiently large number of transactions, with respect to a support threshold *minimum support*.
2. From the set of large itemsets found, find out all the association rules that have a confidence value exceeding a confidence threshold *minimum confidence*.

Since the solution to the second subproblem is straightforward [1], major research efforts have been spent on the first subproblem. Most of the algorithms devised to find large itemsets are based on the `Apriori` algorithm [1]. The `Apriori` algorithm finds out the large itemsets iteratively. In the i^{th} iteration, `Apriori` generates a number of candidate itemsets of size i.[1] `Apriori` then scans the database to find the support count of each candidate itemset. Itemsets whose support counts are smaller than the minimum support are discarded. `Apriori` terminates when no more candidate set can be generated.

The key of the `Apriori` algorithm is the `Apriori_Gen` function [1] which wisely generates only those candidate itemsets that have the potential of being large. However, at each database scan, only candidate itemsets of the same size are generated. Consequently, the number of database scans required by Apriori-based algorithms depends on the size of the largest large itemsets. As an example, if a database contains a size-10 large itemset, then at least 10 passes over the database are required. For large databases containing gigabytes of transactions, the I/O cost is dauntingly big.

The goal of this paper is to analyze and to improve the I/O requirement of the `Apriori` algorithm. In particular, we generalize `Apriori_Gen` to a new candidate set generation algorithm, `LGen`, based on Lattice Theory. The main idea is to relax `Apriori`'s restriction that candidate itemsets generation must start from size one and that at each database scan, only candidate itemsets of the same size are generated. Instead, `LGen` takes a (partial) set of multiple-sized large itemsets *as a hint* to generate a set of multiple-sized candidate itemsets. This approach allows us to take advantage of an educated guess, or a *suggestion*, of a set of large itemsets.

(**Example 1**) As a simple example, suppose the itemset $\{a, b, c, d\}$ and all its subsets are large in a database. `Apriori` will require four passes over the data to generate the itemset $\{a, b, c, d\}$ (see Figure 1(a)). However, if one already knew that the itemsets $\{a, b, c\}$, $\{a, b, d\}$, and $\{c, d\}$ were large, then we can use this piece of information to help us generate the itemset $\{a, b, c, d\}$ early. One simple strategy is to expand the *lattices* "rooted" at $\{a, b, c\}$, $\{a, b, d\}$, and $\{c, d\}$ to collect all of their subsets (see Figure 1(b)). Note that these subsets are large as well. We can then divide these large itemsets into groups according to their sizes, and apply `Apriori_Gen` to each group. This strategy will thus generate candidate itemsets of various sizes.

[1] The size of an itemset is the number of items the itemset contains.

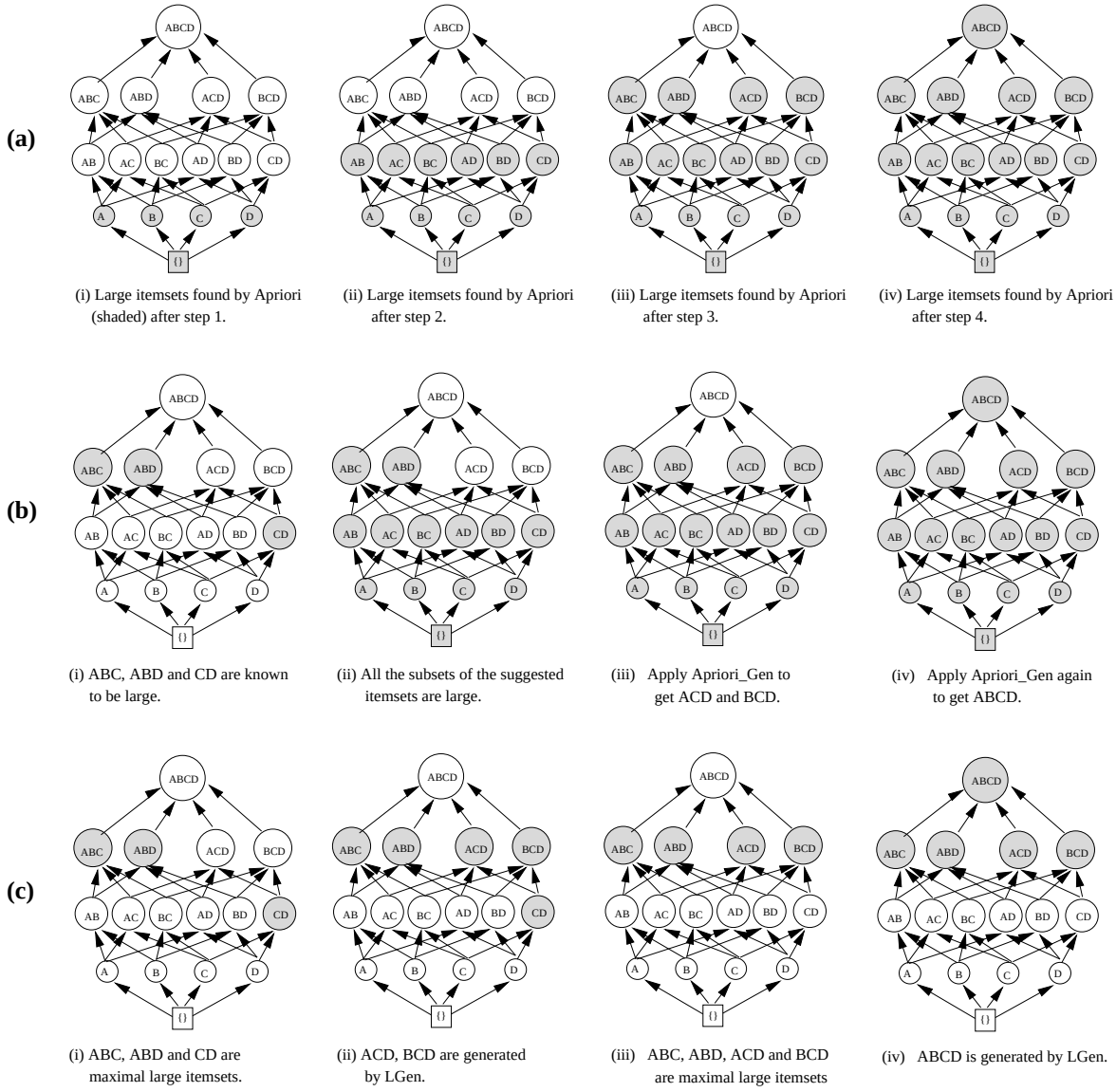

Figure 1: (a) `Apriori_Gen`, (b) a simple strategy, and (c) `LGen`

In our simple example, the itemsets $\{a, c, d\}$ and $\{b, c, d\}$ will be generated in the first iteration, among others. The database is then scanned to count the supports of the candidate itemsets. The large ones will be added to the set of large itemsets, and the whole process repeats. Finally, the itemset $\{a, b, c, d\}$ is generated in the second iteration.

Although the simple strategy can generate large itemsets early, it may not be very efficient. The reason is that a fairly large number of large itemsets need to be considered when generating candidate itemsets (e.g., all shaded nodes in Figure 1(b ii)). Also, it will generate many candidate itemsets that are already known to be large (e.g., when we apply `Apriori_Gen` to those size-2 large itemsets in the first iteration, the itemsets $\{a, b, c\}$ and $\{a, b, d\}$ are generated again).

Our candidate set generation algorithm `LGen` adopts the strategy of generating *large* can-

didate itemsets as soon as possible, using a suggested set of large itemsets as a hint. However, instead of using all large itemsets known so far to generate a batch of candidate itemsets, `LGen` uses only the *maximal* large itemsets for candidate generation. An (already known) large itemset is maximal if it is not a proper subset of another (already known) large itemset. (For example, the itemsets $\{a, b, c\}$, $\{a, b, d\}$, and $\{c, d\}$ are maximal in Figure 1(b i).) The advantages of `LGen` over the simple strategy are twofold: First, the set of maximal large itemsets is much smaller than the set of all large itemsets. This makes the candidate generation procedure much more efficient. Second, it guarantees that no redundant candidate itemsets (i.e., those that are already known to be large) are generated. Figure 1(c) illustrates the candidate generation procedure of `LGen`. Note that in Figure 1(c), the itemset $\{a, b, c, d\}$ is generated in the second iteration, once we are given that the *suggested itemsets* $\{a, b, c\}$, $\{a, b, d\}$, and $\{c, d\}$ are large.

In this paper we present the `LGen` candidate set generation function and the `FindLarge` algorithm which uses `LGen` to discover large itemsets in a database. We prove their correctness and show that replacing `Apriori` and `Apriori_Gen` by `FindLarge` and `LGen` allows us to significantly reduce the amount of I/O cost required for mining association rules. We study the various properties of the algorithms and address the following questions:

- Will `FindLarge` generate more candidate itemsets than `Apriori`? A naive algorithm which generates all possible itemsets as the candidate set can of course discover all large itemsets in one single database scan. This algorithm, however, is clearly infeasible because the candidate set would be way too large. We will prove that `FindLarge` does not generate more candidate itemsets than `Apriori` does. It only generates them earlier and in fewer passes.

- Where can I find the set of suggested large itemsets for `FindLarge`? To apply `FindLarge` successfully, one needs to supply it with suggested large itemsets as a "hint". Although these suggested itemsets are not necessary, their presence will significantly improve the performance of `FindLarge`. These suggested itemsets can be obtained in different ways. For example, a supermarket may keep a database of transactions of the past twelve months. At the end of every month, a new set of transactions are added to the database and transactions that are more than a month old are removed (see Figure 2). To mine the association rules of the updated database, we can use the set of large itemsets found from the old database as the suggested set and apply `FindLarge`. Since the updated database and the old one overlap by more than 90% (11/12), our performance study shows that `FindLarge` uses significantly fewer passes compared with `Apriori`. Alternatively, one can take a sample of the database, apply `Apriori` on the sample to get a rough estimate of the large itemsets, and use it as the suggested set for `FindLarge`. We will discuss how sampling is used to assist `FindLarge` in Section 4.

- How much performance gain can `FindLarge` achieve over `Apriori`? We performed an extensive simulation study on the performance of `FindLarge`. We found that the number

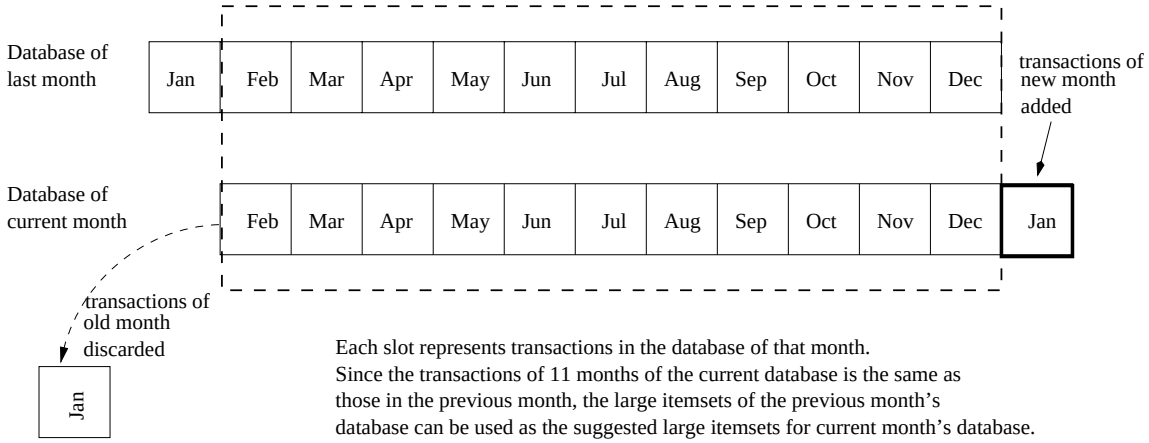

Figure 2: Obtaining suggested large itemsets from an old database.

of I/O passes saved by `FindLarge` over `Apriori` depended on the *accuracy* of the suggested large itemsets. As an extreme case, if the suggested itemsets cover all the large itemsets in the database, then `FindLarge` requires only 2 database scans. This number is independent of the size of the largest large itemsets. The saving is thus significant for a database whose association rules contain a non-trivial number of items. In general, `FindLarge` outperforms `Apriori` if the suggested set covers more than 20% of the set of large itemsets. As we will see, this coverage (and much better ones) can be easily obtained by sampling techniques.

- If I use sampling to obtain a set of suggested large itemsets, how many samples shall I take? If we take a very small sample, then the large itemsets discovered from it would not be a good estimate of the large itemsets of the whole database. `FindLarge` would not perform much better than `Apriori` because the "hint" is not good enough. On the other hand, if we take a vary large sample, then the I/O cost of applying `Apriori` on the large sample to obtain the large itemsets estimate would be substantial, essentially wiping out the benefit achieved by `FindLarge`. We will show that taking a 10% sample of the database is usually sufficient to get a good estimate. Sampling plus `FindLarge` is thus a viable option for fast association rule mining.

The rest of this paper is organized as follows. In Section 2 we take a closer look of the `Apriori` algorithm and the `Apriori_Gen` function. In Section 3 we describe the `LGen` algorithm for candidate generation and show how `Apriori` can be extended to `FindLarge`. We prove the correctness of `LGen` and `FindLarge` and that `FindLarge` does not generate more candidate itemsets than `Apriori` does. Section 4 describes our simulation study and presents the evaluation results. In particular, we discuss how the accuracy of the set of suggested large itemsets affects the performance of `FindLarge`, and how sampling is used to obtain a good estimate of the large itemsets. Finally, we conclude the paper in Section 5.

2 The Apriori algorithm

Conceptually, finding large itemsets from database transactions involves keeping a count for every itemset. However, since the number of possible itemsets is exponential to the number of items in the database, it is impractical to count every subset we encounter in the database transactions. The `Apriori` algorithm tackles this combinatorial explosion problem by using an iterative approach to count the itemsets. First, itemsets containing only one item (1-itemsets or singletons) are counted, and the set of large 1-itemsets (L_1) is found.[2] Then, a set of possibly large 2-itemsets is generated using the function `Apriori_Gen`. Since for an itemset of size n to be large, all its size $n-1$ subsets must also be large, `Apriori_Gen` only generates those 2-itemsets whose size one subsets are all in L_1. This set is the candidate set of size-2 itemsets, C_2.[3] For example, if $L_1 = \{\{c\}, \{e\}, \{g\}, \{j\}\}$, C_2 would be $\{\{c,e\}, \{c,g\}, \{c,j\}, \{e,g\}, \{e,j\}, \{g,j\}\}$.

After C_2 is generated, the database is scanned once again to determine the support counts of the itemsets in C_2. Those with their support counts larger than the support threshold are put into the set of size-2 large itemsets, L_2. L_2 is then used to generate C_3 in a similar manner: all size-two subsets of every element in C_3 must be in L_2. So, if L_2 in our previous example turns out to be $\{\{c,e\}, \{c,g\}, \{c,j\}, \{g,j\}\}$, C_3 would be $\{\{c,g,j\}\}$. Note that the itemset $\{c,e,j\}$ is not generated because not all of its size-two subsets are in L_2. Again, the database is scanned once more to find L_3 from C_3. This candidate set generation–verification process is continued until no more candidate itemsets can be generated. Finally, the set of large itemsets is equal to the union of all the L_i's.

The iterative nature of the `Apriori` algorithm implies that at least n database passes are needed to discover all the large itemsets if the biggest large itemsets are of size n. Since database passes involve slow disk access, to increase efficiency, we should minimize the number of database passes during the mining process. One solution is to generate bigger-sized candidate itemsets as soon as possible, so that their supports can be counted early. With `Apriori_Gen`, unfortunately, the only piece of information that is useful for generating new candidate itemsets during the n-th iteration is the size-$(n-1)$ large itemsets, L_{n-1}. Information from other L_i's $(i < n-1)$ and C_i's $(i \leq n-1)$ are not useful because it is already subsumed in L_{n-1}. As a result, we cannot generate candidate itemsets larger than n.

Now, suppose one is given a set of suggested large itemsets S. We can use this set as additional information to generate large itemsets early. During the first iteration, besides counting the supports of size-1 itemsets, we can also count the supports of the elements in S as well. After the first iteration, we thus have a (partial) set of large itemsets of various sizes, L. These itemsets include all large singletons, as well as those itemsets in S that are verified large. We can now follow the principle of `Apriori` to generate candidate itemsets based on L. The only problem remains is how to generalize `Apriori_Gen` to compute a set of multiple-sized candidate itemsets from a set of multiple-sized large itemsets (L) *efficiently*.

[2] We use the notation L_k to denote the set of size-k large itemsets.

[3] We use the notation C_k to denote the set of size-k candidate itemsets.

```
1   function FindLarge(SuggestedLarge)
2       Set of large itemsets with associated counters, MaxLargeItemsets := ∅
3       Iteration := 1
4       CandidateSet := (all 1-itemsets) ∪ (⋃_{s∈SuggestedLarge} ↓ s)
5       Scan database and count occurrence frequency of every set in CandidateSet
6       NewLargeItemsets := large itemsets in CandidateSet
7       while (NewLargeItemsets ≠ ∅)
8           Iteration := Iteration+1
9           MaxLargeItemsets := Max(MaxLargeItemsets ∪ NewLargeItemsets)
10          CandidateSet := LGen(MaxLargeItemsets,Iteration)
11          Count occurrence frequency of every set in CandidateSet
12          NewLargeItemsets := large itemsets in CandidateSet
13      end while
14      return all subsets of elements in MaxLargeItemsets
```

Figure 3: Finding large itemsets

3 LGen

We generalize `Apriori_Gen` to a new candidate set generation function called `LGen` based on Lattice Theory. The main idea of `LGen` is to generate candidate itemsets of bigger sizes early using information provided by a set of suggested large itemsets. Before we describe our algorithm formally, let us first illustrate the idea with an example.

(**Example 2**) Suppose we have a database whose large itemsets are $\{a, b, c, d, e, f\}$, $\{d, e, f, g\}$, $\{e, f, g, h\}$, $\{h, i, j\}$ and all their subsets. Assume that the sets $\{a, b, d, e, f\}$, $\{e, f, g, h\}$, and $\{d, g\}$ are suggested large. During the first iteration, we count the supports of the singletons as well as those of the suggested itemsets. Assume that the suggested itemsets are verified large in the first iteration. In the second iteration, since $\{a, b, d, e, f\}$ is large, we know that its subset $\{d, e\}$ is also large. Similarly, we can infer from $\{e, f, g, h\}$ that $\{e, g\}$ is also large. Since $\{d, g\}$ is also large, we can generate the candidate itemset $\{d, e, g\}$ and start counting it. Similarly, the candidate itemset $\{d, f, g\}$ can also be generated this way. Therefore, we have generated some size-3 candidate itemsets before we find out all size-two large itemsets.

Our algorithm for finding large itemsets, `FindLarge` is shown in Figure 3. The method is similar to `Apriori` except that:

- it takes a set of suggested itemsets, *SuggestedLarge* as input and counts their supports during the first database scan, and
- it replaces the `Apriori_Gen` function by the more general `LGen` function which takes the set of maximal large itemsets (*MaxLargeItemsets*) as input.

The algorithm consists of two stages. The first stage consists of a single database scan (lines 4–6). Singletons, as well as the suggested large itemsets and their subsets ($\bigcup_{s \in SuggestedLarge} \downarrow s$), are counted. Any itemsets found to be large at this first stage is put into the set of newly found

```
1   function LGen(MaxLargeItemsets,n)
2       CandidateSet:= ∅
3       repeat
4           NewCandidates:= LGFixedSize(MaxLargeItemsets,n)
5           CandidateSet:= CandidateSet ∪ NewCandidates
6           n := n + 1
7       until (n > 1+size of the biggest itemset in MaxLargeItemsets)
8       return CandidateSet
```

Figure 4: Generating candidate itemsets for a certain iteration

large itemsets (*NewLargeItemsets*).

The second stage of `FindLarge` is iterative. The iteration continues until no more new large itemsets can be found. At each iteration, `FindLarge` generates a set of candidate itemsets based on the large itemsets it has already discovered. As we have argued in Section 1, we could apply `Apriori_Gen` on the whole set of large itemsets already found. However, the drawback is that the set of large itemsets could be large, and that it would result in the generation of many redundant candidate itemsets. Instead, `FindLarge` first *canonicalizes* the set of large itemsets into a set of *maximal* large itemsets (*MaxLargeItemsets*), and passes the maximal set to `LGen` to generate candidate itemsets. The function `Max`() (line 9) performs the canonicalization.

We can consider canonicalization as a way of compressing the information contained in the set of large itemsets. The idea is that suppose we know that a set s is large, we immediately know that all of its subsets are also large. Considering the set of itemsets with the subset operator as a lattice, we borrow notations from lattice theory [5][2] and denote the set of all subsets of s by its downset: $\downarrow s = \{x \mid x \subseteq s\}$. We can then represent the set of all large itemsets L by a union of downsets: $L = \bigcup_{s \in \max(L)} \downarrow s$, where $\max(L)$, the set of maximal elements of L, is defined as $\max(L) = \{x \in L \mid \forall y \in L\ [x \subseteq y \Rightarrow y \subseteq x]\}$.

In Example 2 (page 7) where $\{a, b, c, d, e, f\}$, $\{d, e, f, g\}$, $\{e, f, g, h\}$, $\{h, i, j\}$ and all their subsets are large, $\max(L) = \{\{a, b, c, d, e, f\}, \{d, e, f, g\}, \{e, f, g, h\}, \{h, i, j\}\}$. Hence, only 4 itemsets are needed to represent L, which contains 86 large itemsets.

The set of maximal large itemsets found is then passed to `LGen` to generate candidate itemsets (line 10). The crux is how to do the generation based on the compressed maximals only. We remark that the `Apriori` algorithm with `Apriori_Gen` is in fact displaying a special case of candidate generation with canonicalization. Recall that in `Apriori`, at the beginning of the the n-th iteration, the set of large itemsets already discovered is $\bigcup_{k=0}^{n-1} L_k$. Canonicalizing the set gives $\max(\bigcup_{k=0}^{n-1} L_k) = L_{n-1}$. Interestingly, `Apriori_Gen` generates the candidate set C_n based solely on L_{n-1}.

The function `LGen` is shown in Figure 4. By simple induction, one can show that at the beginning of the n-th iteration of `FindLarge`, all large itemsets whose sizes are smaller than n are known. Hence, when `LGen` is called at the n-th iteration of `FindLarge`, it only generates candidate itemsets that are of size n or larger. To generate fix-sized candidate itemsets, `LGen`

1 function `LGFixedSize`(*MaxLargeItemsets*,n)
2 *CandidateSet* := $\emptyset$
3 foreach $i, j \in$ *MaxLargeItemsets*, $i \neq j$,
4 if ($|(i \cap j)| \geq n-2$)
5 $$NewCandidates := \left\{ \{a_1, a_2, ..., a_n\} \;\middle|\; \begin{array}{l} a_1, a_2, ..., a_{n-2} \in i \cap j, \\ a_{n-1} \in i - j,\ a_n \in j - i, \\ \forall i \in [1, n]\ \exists t \in \mathit{MaxLargeItemsets} \text{ s.t.} \\ \{a_1, a_2, ..., a_n\} - \{a_i\} \subseteq t \\ \nexists u \in \mathit{MaxLargeItemsets} \text{ s.t.} \\ \{a_1, a_2, ..., a_n\} \subseteq u \end{array} \right\}$$
6 *CandidateSet* := *CandidateSet* $\cup$ *NewCandidates*
7 end if
8 end foreach
9 return *CandidateSet*

Figure 5: Generating candidate itemsets of a fixed size

calls the helper function, `LGFixedSize` (Figure 5). Essentially, given a target candidate itemset size n, `LGFixedSize` examines every pair of maximal large itemsets i, j whose intersection is at least $n-2$ in size (lines 3–4). It then generates candidate itemsets of size n by picking $n-2$ items from the intersection between i and j, one item from the set difference $i-j$, and another item from $j-i$. A candidate itemset so generated is then checked to see if all of its size-$(n-1)$ subsets are already known to be large. (That is, if all of them are subsets of certain maximal large itemsets.) If not, the candidate itemset is discarded. The candidate itemsets so generated are collected in the set *NewCandidates* as shown in line 5 of Figure 5.

(**Example 3**) Let us consider the example shown in Figure 1(c). Suppose we are given that the itemsets $\{a, b, c\}$, $\{a, b, d\}$, and $\{c, d\}$ are maximal large itemsets (Figure 1(c i)). To generate size-3 candidate itemsets, `LGen` needs to consider pairs whose intersections contain at least $3 - 2 = 1$ item (e.g., $\{a, b, c\}$ and $\{c, d\}$). Let $i = \{a, b, c\}$ and $j = \{c, d\}$, we have $i \cap j = \{c\}$, $i - j = \{a, b\}$, and $j - i = \{d\}$. The itemsets generated based on this i, j pair are $\{c, a, d\}$ and $\{c, b, d\}$. Since all of their size-2 subsets are subsets of some maximal itemsets, they are included in the candidate set. Now, suppose that the support counts of $\{a, c, d\}$ and $\{b, c, d\}$ are larger than the support threshold, then in the second iteration, the maximal large itemsets are $\{a, b, c\}$, $\{a, b, d\}$, $\{a, c, d\}$, and $\{b, c, d\}$. One can check that if we take any two of them to generate size-4 candidate itemsets, the set $\{a, b, c, d\}$ will be generated.

Indeed, `LGen` is a generalization of `Apriori_Gen`. When it is known that two size-$(n-1)$ subsets of a size-n itemset are large, `Apriori_Gen` examines that size-n itemset to see whether all its size $n-1$ subsets are large. Since only large itemsets of size $n-1$ are known when `Apriori_Gen` is called to generate itemsets of size n, size n candidate itemsets are, in practice, found by taking the union of two itemsets i and j of size $n-1$ before checking whether all its size-$(n-1)$ subsets are large. Note that for the union of two size-$(n-1)$ itemsets to be of size n, their intersection must be of size $n-2$. Hence, this is equivalent to taking $n-2$ items from the intersection of the two itemsets, one item from $i-j$, and another item from $j-i$.

`LGen` generalizes this idea. Given a maximal itemset of size k, $k > n-1$, we know that all its ${}_kC_{n-1}$ subsets of size $n-1$ are large. With two such maximal itemsets p and q, we know that all possible pairs, one from the ${}_{|p|}C_{n-1}$ subsets of size $n-1$, and another from the ${}_{|q|}C_{n-1}$ subsets of size $n-1$, will be examined if `Apriori_Gen` is used. Yet, since all the size-$(n-2)$ intersections of these pairs are subsets of the intersection of p and q, and the set differences between the pairs are always subsets of $p-q$ or $q-p$, to form an itemset of size n, we can take $n-2$ items from the intersection between p and q, one item from $p-q$, and one item from $q-p$. The size-n itemsets so formed can then be checked to see whether they are eligible to be a candidate itemset, that is, whether all their size-$(n-1)$ subsets are large. This way, all possible size-n candidates `Apriori` examines that are not known to be large (i.e., not already a subset of p or q) can be examined without expanding the maximal itemsets to their size-$(n-1)$ subsets.

3.1 Theorems

In this subsection we summarize a few properties of `LGen` in the following theorems. The proofs are included in the Appendix of this paper. We use the symbol S to represent the set of suggested large itemsets, $\downarrow y$ to represent the downset of any itemset y (i.e., $\downarrow y = \{x \mid x \subseteq y\}$), so $\bigcup_{s \in S} \downarrow s$ is the set of all itemsets suggested implicitly or explicitly by the suggested set S. Also, we use C_{LGen} to represent the set of all candidate itemsets generated by `LGen` in `FindLarge` and $C_{Apriori}$ to represent the set of all candidate itemsets generated by `Apriori_Gen` in the `Apriori` algorithm.

Theorem 1 *Given a set of suggested large itemsets S, $C_{Apriori} \subseteq C_{LGen} \cup (\bigcup_{s \in S} \downarrow s)$.*

Since `FindLarge` (which uses `LGen`) counts the supports of all itemsets in $C_{LGen} \cup (\bigcup_{s \in S} \downarrow s)$, Theorem 1 says that any candidate itemset that is generated by `Apriori` will have its support counted by `FindLarge`. Hence, if `Apriori` finds out all large itemsets in the database, `FindLarge` does too. In other words, `FindLarge` is correct.

Theorem 2 *$C_{LGen} \subseteq C_{Apriori}$.*

Theorem 2 says that the set of candidate itemsets generated by `LGen` is a subset of that generated by `Apriori_Gen`. `LGen` thus does not generate any unnecessary candidate itemsets and waste resources in counting bogus ones. However, recall that `FindLarge` counts the supports of the suggested large itemsets in the first database scan for verification. Therefore, if the suggested set contains itemsets that are actually small, `FindLarge` will count their supports superfluously. Fortunately, the number of large itemsets in a database is usually order-of-magnitude fewer than the number of candidate itemsets. The extra support counting is thus insignificant compared with the support counting of all the candidate itemsets. `FindLarge` using `LGen` thus requires similar counting effort as `Apriori` does.

Theorem 3 *If $S = \emptyset$ then $C_{LGen} = C_{Apriori}$.*

Theorem 3 says that without suggested large itemsets, `FindLarge` reduces to `Apriori`. In particular, they generate exactly the same set of candidate itemsets.

4 Experiments

To evaluate the performance of `FindLarge` using `LGen` as the candidate set generation algorithm, we performed extensive simulation experiments. Our goals are to study the amount of I/O saving that can be achieved by `FindLarge` over `Apriori`, and how sampling can be used to construct a good set of suggested large itemsets. In this section we present some representative results from the experiments.

4.1 Synthetic Database Generation

In the experiments, we followed the approach of [1, 9] and used synthetic data as the test databases. Here, let us briefly describe the synthetic database generation procedure. Readers are referred to [9] for more details. To generate a transaction database D, we first generate a pool W of potentially large itemsets. Each itemset in W is generated by first determining the itemset size from a Poisson distribution with mean $|I|$. The $|W|$ itemsets are divided into groups of $S_q + 1$ itemsets. For each group, items in the first itemset are picked randomly from the set of N items. Then in each of the following S_q itemsets, some fraction p of items from the first itemset are duplicated. The fraction p is determined from a Poisson distribution with mean equal to 0.5. The rest are picked randomly. After W is generated, it is used to generate the database D. The size of each transaction in D is generated from a Poisson distribution with mean $|T|$. Next, a random itemset from W is chosen and its items are added to the transaction being generated. If the transaction has acquired the desired size, we move on to generate the next transaction. If not, we pick another itemset from W and repeat the above procedure until the transaction has got the desired number of items. Table 1 summarizes the parameter setting of our baseline experiment.

Parameter	Description	Value
$\|D\|$	database size in number of transactions	131,072
N	number of items	1,000
$\|W\|$	number of potentially large itemsets	2,000
$\|I\|$	avg. size of potentially large itemsets	4
$\|T\|$	avg. size of transactions	20

Table 1: Parameters of the database generation model

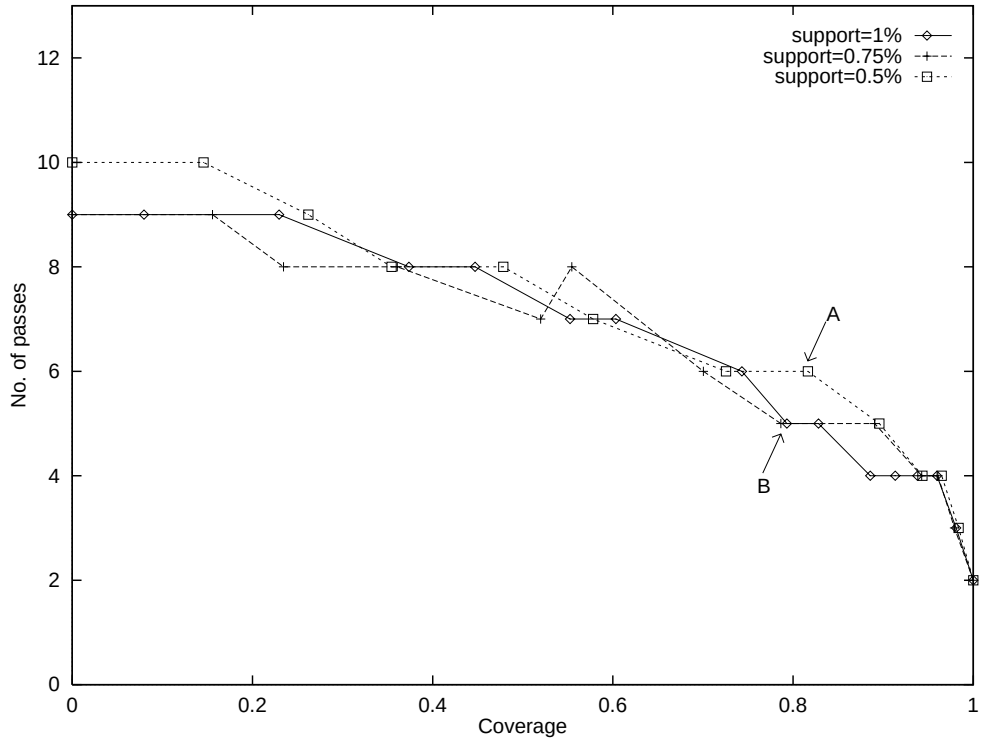

Figure 6: Number of I/O passes vs. coverage under different support threshold.

4.2 Coverages and I/O Savings

For each database instance generated, we first discovered the set of large itemsets, L, using `Apriori`. Our first set of experiments studies how the "coverage" of the suggested large itemsets affects the performance of `FindLarge`. By coverage, we mean the fraction of large itemsets in L that are suggested.[4] To model coverage, we drew a random sample from L to form a set of suggested large itemsets S. We define the *coverage* of a suggested set S over the set of large itemsets L by

$$coverage = \frac{|\bigcup_{s \in (S \cap L)} \downarrow s|}{|L|}.$$

Since in our first set of experiments, S was drawn from L, we have $S \cap L = S$. After we had constructed S, we ran `FindLarge` on the database using S as the suggested set. Finally, we compared the number of I/O passes each algorithm had taken.

Note that with the way we generated the suggested set S, no element in S was small. In practice, however, the suggested itemsets could contain small itemsets. Since small suggested itemsets are discarded in the first database scan of `FindLarge` (see Figure 3), their presence does not affect the number of I/O passes required by `FindLarge` and hence they were not modeled in this set of experiments.

We generated a number of database instances according to the above-mentioned model.

[4] If an itemset is suggested, then all of its subsets are also implicitly suggested.

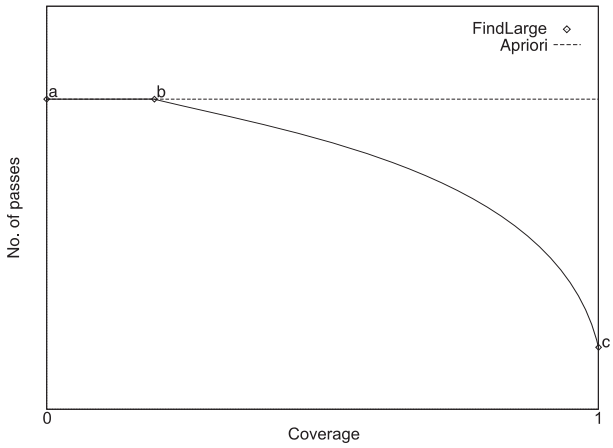

Figure 7: A representative I/O vs. coverage curve.

For each database instance, we applied `Apriori` to discover the large itemsets under different support thresholds. Also, we generated a number of suggested sets of various coverages. We then applied `FindLarge` to the different database instances with different suggested sets under different support threshold settings. The number of I/O passes that `Apriori` and `FindLarge` had taken were subsequently compared. Figure 6 shows the result obtained from one typical database instance. In the following we refer to this particular database as $\mathcal{D}$.

In Figure 6 three sets of points ('$\diamond$', '$+$', '$\square$') are shown corresponding to the support thresholds 1%, 0.75%, and 0.5% respectively. Each point shows the number of I/O passes `FindLarge` took when applied to $\mathcal{D}$ with a particular suggested set of a certain coverage. For example, the $\square$ point labeled A shows that when `FindLarge` was applied with a suggested set whose coverage was 81.6%, 6 passes were required. Note that the points shown in Figure 6 take on discrete values. The lines connecting points of the same kind are there for legibility reason only and should not be interpreted as interpolation.

For the database $\mathcal{D}$, when the support threshold was set to either 1.0% or 0.75%, the size of the largest large itemsets was 9. `Apriori` took 9 I/O passes. This is shown in Figure 6 by the '$\diamond$' and '$+$' points when coverage equals 0. (Recall that `FindLarge` reduces to `Apriori` when the suggested set is null.) When the support threshold was lowered to 0.5%, the support counts of certain size-10 itemsets also exceeded the support threshold. In that case, the size of the largest large itemsets was 10. `Apriori` thus took 10 I/O passes.

One general observation from Figure 6 is that the higher the coverage of the suggested set has with respect to the set of large itemsets, the smaller the number of I/O passes `FindLarge` is required. In fact, all of the data points we obtained from our experiment exhibit a typical curve as shown in Figure 7.

In general, we can divide the curve into four stages:

At point a (coverage = 0). When the suggested set does not cover any large itemsets, `FindLarge` with `LGen` degenerates to `Apriori`. The number of I/O passes required by the two

algorithms are thus the same.

Between points a and b. In this region, `FindLarge` takes the same number of passes as `Apriori` does. With a very small coverage, there are only few large itemsets suggested, and these itemsets usually consist of only a small number of items.[5] In this case, `LGen` is unable to provide the advantage of generating large-sized candidate itemsets early. Hence, no saving in I/O is obtained. The length of the line $\overline{ab}$, fortunately, is usually small. For example, in our simulation experiments, the line $\overline{ab}$ in general spans only from coverage = 0 to coverage = 20%. As we will see later, getting a suggested set with a larger-than-20% coverage is easily obtainable by sampling techniques.

Between points b and c. The number of I/O passes required by `FindLarge` decreases gradually as the coverage of the suggested set increases. This is because as more large-sized itemsets are suggested, `LGen` is better able to generate large-sized candidate itemsets early. As an example, when mining the database instance $\mathcal{D}$ with the support threshold set at 1% using a suggested set whose coverage is 0.743 (point labeled B in Figure 6), `LGen` generated candidate itemsets of sizes ranging from 2 to 7 *early in pass number 2*. Again, the larger the coverage, the fewer I/O passes does `FindLarge` need.

We also observe that the amount of I/O saving increases more rapidly when the coverage is approaching 1. This is because with a very large coverage, the suggested set contains many *top-sized*, or maximal large itemsets. This greatly facilitates the generation of other not-yet-discovered maximal large itemsets as candidates early. Since `FindLarge` terminates once all the maximals are found, only very few passes are required.

At point c (coverage = 100%). `FindLarge` only needs two passes over the database when the suggested set covers all large itemsets. The first pass is used to find out the support counts of the suggested itemsets. Since these itemsets are the only large itemsets in the database, `LGen` will generate the negative border[6] [12] as candidates. In the second pass, `FindLarge` counts the supports of the candidate itemsets. Since none of the candidate is large, `FindLarge` terminates after only two passes over the database.

4.3 Candidate Set Sizes

As we have discussed in Section 3, `FindLarge` essentially checks the same number of candidate itemsets as `Apriori` does, but in fewer passes. Therefore, under `FindLarge`, the *average* number of support counts that need to be determined per pass is larger than that of `Apriori`. Since support counts take space, an interesting question is: will `FindLarge` require more memory than `Apriori`? Surprisingly, our simulation results show that `FindLarge` does not require more memory. In fact, in some cases, it uses a little bit less.

[5] A size-k itemset suggested implicitly suggests $2^k - 1$ non-empty itemsets. So if the suggested set contains large-sized itemsets, it would possess a good coverage.

[6] The negative border is the set of itemsets which are small but all of their subsets are large.

Pass	`Apriori`	`FindLarge` with support=1%, coverage=8%	`FindLarge` with support=1%, coverage=74%	`FindLarge` with support=1%, coverage=98%
1	1,000	1,234	3,461	4,118
2	220,780	220,687	220,335	220,173
3	937	858	366	56
4	777	732	159	-
5	519	504	23	-
6	229	227	3	-
7	84	84	-	-
8	19	19	-	-
9	2	2	-	-
Total	224,347	224,347	224,347	224,347

Table 2: Candidate set size for each pass when database $\mathcal{D}$ is mined using `Apriori` and `FindLarge`.

As an example, we mined the database $\mathcal{D}$ using `Apriori` and `FindLarge` with three suggested sets of different coverages (8%, 74%, and 98%). Table 2 shows the number of support counts tallied during each pass of the mining algorithms. Note that the numbers shown in Table 2 equal the candidate sets' sizes except for those listed under the first pass of `FindLarge`. This is because `FindLarge` counts the supports of all subsets of the suggested itemsets in the first pass, besides those of the singletons. These support counts are also included in the table. For example, during the first pass when `FindLarge` mined the database $\mathcal{D}$ with a suggested set S of coverage 8%, it counted the supports of 1,000 singletons as well as 234 itemsets suggested (explicitly or implicitly) by S.

From the table, we observe that the size of the candidate set generated in the second pass dominates the others. It thus determines the memory requirement of the algorithm. A characteristic of `FindLarge` is to redistribute the counting job so that more support counts are processed in the first pass in order to save work in the later passes. Since it is very unlikely that the number of singletons that exist in a database is larger than the number of size-2 candidate itemsets, in general, `FindLarge` does not require more memory than `Apriori` does.

4.4 Sampling

In a previous study [12], it was shown that sampling was a cost-effective technique for finding an approximate solution to the mining problem. We can therefore use sampling to obtain a good suggested set if one is not readily available.

We performed a set of simulation experiments to study how sampling should be done in order to obtain a good suggested set. In the experiments, a number of databases were generated. For each database, we extracted a fraction f of transactions as samples. We then mined the samples using `Apriori`. The resulting large itemsets discovered were then used as the suggested

suggested large set obtained by mining the sample with support = 1%				
f	1/16	1/8	1/4	1/2
avg. coverage	0.920	0.934	0.945	0.964
avg. I/O cost	6.2	5.8	7.9	10.2

Table 3: I/O cost vs. sampling size.

suggested large set obtained by mining the sample with support = 0.75%				
f	1/16	1/8	1/4	1/2
avg. coverage	0.975	0.990	0.997	1.000
avg. I/O cost	5.6	4.8	5.3	6.5

Table 4: I/O cost vs. sampling size using a smaller support threshold for the samples.

set for `FindLarge`. We repeated the experiment a number of times, each with a different set of samples. For each experiment, we recorded the coverage of the suggested set derived from the samples, and the total I/O cost spent. The I/O cost included scanning the samples by `Apriori` and scanning the database by `FindLarge`. We express the I/O cost in terms of database scans. As an example, suppose $f = 0.1$ and `Apriori` takes 10 passes over the samples, then the amount of I/O spent in `Apriori` applied over the samples is equivalent to $0.1 \times 10 = 1$ database scan. If `FindLarge` takes 5 passes over the database, then the total I/O cost would be $1 + 5 = 6$ database scans. For each value of f, the I/O costs spent in different experiment runs (corresponding to different sets of samples taken) were averaged. Similarly, we also determined the average coverages of the different sets of samples. The average I/O cost is then compared with the number of I/O passes taken by `Apriori` when it was applied to the database directly.

Table 3 shows the result of a typical experiment. In the experiment, the database was generated according to our baseline setting (Table 1). The support threshold was set to 1% and the number of I/O passes that `Apriori` took was 9. From Table 3, we see that even with a small set of samples ($f = 1/16$), the expected coverage of the suggested set derived from it exceeded 90%. Also, `FindLarge` saved $(9 - 6.2)/9 = 31\%$ of I/O compared with `Apriori`. This performance gain was even improved to 36% when the sample size was increased to 1/8 of the database. Further increment of the sample size, however, became counter-productive, as is shown in the table when $f = 1/4$ and $f = 1/2$. This is because mining a large sample with `Apriori` is too costly in I/O.

In [12], it was shown that mining the samples with a slightly smaller support threshold than that is required for the database improves the accuracy of the estimate. We therefore re-ran our experiments following the idea. Table 4 shows the result for the same database used in Table 3, except that the samples were mined by `Apriori` with a support threshold of 0.75%. From the table, we see that lowering the samples' support threshold improved the I/O cost across the board. For example, only 4.8 database scans on average was required when $f = 1/8$.

5 Conclusion

This paper described a new algorithm `FindLarge` for discovering large itemsets in a transaction database. `FindLarge` uses a new candidate set generation algorithm `LGen` which takes a set of multiple-sized large itemsets to generate multiple-sized candidate itemsets. Given a reasonably accurate suggested set of large itemsets, `LGen` allows big-sized candidate itemsets to be generated and processed early. This results in significant I/O saving compared with traditional Apriori-based mining algorithms.

We proved a number of theorems about `FindLarge` and `LGen`. In particular, we showed that `FindLarge` is correct and that `LGen` never generates redundant candidate itemsets. Hence the CPU requirement of `FindLarge` is compatible with `Apriori`.

In order to evaluate the I/O performance of `FindLarge`, we conducted extensive experiments. We showed that the better coverage the suggested set has, the fewer I/O passes `FindLarge` requires. In the best case, when the suggested set covers all large itemsets, `FindLarge` takes only two passes over the database.

To obtain a good suggested set, sampling techniques can be applied. We showed that a small sample is usually sufficient to generate a suggested set of high coverage. `FindLarge` is thus an efficient and practical algorithm for mining association rules.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of the Twentieth International Conference on Very Large Databases*, pages 487–499, Santiago, Chile, 1994.

[2] Garrett Birkhoff. *Lattice Theory*, volume 25 of *AMS Colloquium Publications*. AMS, 1984.

[3] David W. Cheung, Jiawei Han, Vincent T. Ng, Ada Fu, and Yongjian Fu. A fast distributed algorithm for mining association rules. In *Proc. Fourth International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.

[4] David W. Cheung, Jiawei Han, Vincent T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proceedings of the Twelfth International Conference on Data Engineering*, New Orleans, Louisiana, 1996. IEEE computer Society.

[5] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990. ISBN 0 521 36584 8.

[6] Jiawei Han and Yongjian Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the 21st VLDB Conference*, pages 420–431, Zurich, Switzerland, 1995.

[7] Mika Klemettinen, Heikki Mannila, Pirjo Ronkainen, Hannu Toivonen, and A. Inkeri Verkamo. Finding interesting rules from large sets of discovered association rules. In nabil R. Adam, Bharat K. Bhargava, and Yelena Yesha, editors, *Third International Conference on Information and Knowledge Management (CIKM'94)*, pages 401–407, Seattle, Washington, November 1994. ACM Press.

[8] J. S. Park, M. S. Chen, and P. S. Yu. Efficient parallel data mining for association rules. In *Proc. 1995 International Conference on Information and Knowledge Management*, Baltimore, MD, November 1995.

[9] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. ACM SIGMOD international Conference on Management of Data*, San Jose, California, May 1995.

[10] T. Imielinski R. Agrawal and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, page 207, Washington, D.C., May 1993.

[11] Ramakrishnan Srikant and Rakesh Agrawal. Mining quantitative association rules in large relational tables. In H. V. Jagadish and Inderpal singh Mumick, editors, *Proc. ACM SIGMOD international Conference on Management of Data*, Montreal, Canada, June 1996.

[12] Hannu Toivonen. Sampling large databases for association rules. In *Proceedings of the 22th Conference on Very Large Data Bases (VLDB)*, September 1996.

6 Appendix

6.1 Definitions and notations

The set of maximal elements of a set of sets L related by the subset relationship in a partial order $\langle L, \subseteq \rangle$ is:

$$\max(L) = \left\{ x \in L \;\middle|\; \forall y \in L \; [x \subseteq y \Rightarrow y \subseteq x] \right\}$$

The down set of $x \in L$ in a partial order $\langle L, \subseteq \rangle$ is:

$$\downarrow x = \left\{ y \;\middle|\; y \subseteq x \right\}$$

It is the complete lattice of sets induced by the maximal element x.

The set of sets in L whose size is n is defined as:

$$szeq(L, n) = \left\{ x \in L \;\middle|\; |x| = n \right\}$$

Similarly, the set of sets in L whose size is less than or equal to n is defined as:

$$szle(L, n) = \left\{ x \in L \;\middle|\; |x| \leq n \right\}$$

The set of subsets of X whose size is n is defined as:

$$subsetszeq(X, n) = \left\{ x \subseteq X \;\middle|\; |x| = n \right\}$$

6.2 Algorithm showcase

Given the set of large itemsets L, the set of candidate itemsets of size n generated by `Apriori_Gen` is:

$$AG(L, n) = \bigcup_{\substack{i,j \in szeq(L, n-1) \\ i \neq j}} \left\{ i \cup j \;\middle|\; \begin{array}{l} |i \cup j| = n, \\ subsetszeq(i \cup j, n-1) \subseteq szeq(L, n-1) \end{array} \right\} \tag{1}$$

Note that we do not need to know the complete set of large itemsets L when generating candidates of size n. Only those elements in L that is of size $n-1$ need be known.

Given a set of suggested large itemsets that have been verified large, S, and the set of large itemsets L, the set of candidate itemsets of size n generated by `LGen` when large itemsets of size one less than k is known ($k \leq n$) is:

$$LG(S, L, n, k) = \bigcup_{\substack{i,j \in \max[S \cup szeq(L, k-1)] \\ i \neq j}} \left\{ x \cup y \cup z \;\middle|\; \begin{array}{ll} x \subseteq i - j, & |x| = 1, \\ y \subseteq i \cap j, & |y| = n - 2, \\ z \subseteq j - i, & |z| = 1, \\ subsetszeq(x \cup y \cup z, n-1) \subseteq S \cup szle(L, k-1) & \\ x \cup y \cup z \notin S & \end{array} \right\} \tag{2}$$

Again, the complete set of large itemsets L need not be known when we invoke $LG(S, L, n, k)$. We only need to know S and all the large itemsets of size $k-1$ (i.e., $szeq(L, k-1)$); these should be known by `FindLarge` at iteration k. Note also that S is a subset of L since its elements

has been verified large. Also, in the first iteration of `FindLarge`, all the subsets of the maximal elements of S have been verified large. That is,

$$S = \bigcup_{s \in \max(S)} \downarrow s \tag{3}$$

6.3 Things to prove

In the following discussion, we assume that `Apriori_Gen` is correct, that is, it can find all possible large itemsets:

$$\left[\bigcup_{n=2}^{\infty} AG(L,n) \right] \cup szeq(L,1) \supseteq L \tag{4}$$

The upper limit of the union is taken to be infinity for notational convenience. Since $AG(L,n)$ is empty when n is larger than one more than the size of the largest element in L, the upper limit can be as small as $1 + \max\{|x| \mid x \in L\}$.

We attempt to prove that:

- `LGen` will generate only a subset of candidate itemsets generated by `Apriori_Gen`

$$\forall S \subseteq L \;\; \forall n \geq 2 \;\; \forall k \leq n \;\; [LG(S,L,n,k) \subseteq AG(L,n) - S] \tag{5}$$

- When large itemsets of size less than n is known and `LGen` is called to generate candidate itemsets of size n, it will generate those itemsets `Apriori_Gen` generates except those already verified to be large.

$$\forall S \subseteq L \;\; \forall n \geq 2 \;\; [LG(S,L,n,n) = AG(L,n) - S] \tag{6}$$

- `FindLarge` examines the same candidate sets as `Apriori` does, except those itemsets already verified large.

$$\forall S \subseteq L \;\; \left[\bigcup_{k=2}^{\infty} \bigcup_{n=k}^{\infty} LG(S,L,n,k) = \bigcup_{n=2}^{\infty} AG(L,n) - S \right] \tag{7}$$

- `FindLarge` can find all possible large itemsets.

$$\forall S \subseteq L \;\; \left[\bigcup_{k=2}^{\infty} \bigcup_{n=k}^{\infty} LG(S,L,n,k) \cup S \cup szeq(L,1) \supseteq L \right] \tag{8}$$

or

$$\forall S \subseteq L \;\; \left[\bigcup_{n=2}^{\infty} LG(S,L,n,n) \cup S \cup szeq(L,1) \supseteq L \right] \tag{9}$$

- When no suggested large itemset is verified to be really large, `LGen` reduces to `Apriori_Gen`.

$$\forall n \geq 2 \;\; [LG(\emptyset,L,n,n) = AG(L,n)] \tag{10}$$

6.4 Rewriting AG

We now rewrite AG to a form that facilitates our proof afterwards. From Equation 1, we see that $|i| = |j| = n-1$ since $i, j \in szeq(L, n-1)$. Also, because

$$\begin{aligned} |i \cup j| &= |i| + |j| - |i \cap j| && (11) \\ &= |i \cap j| + |i - j| + |j - i| && (12) \end{aligned}$$

With the condition that $|i \cup j| = n$ in Equation 1, we have, substituting into Equation 11,

$$n = (n-1) + (n-1) - |i \cap j|$$

and thus $|i \cap j| = n-2$. Noticing that $i \neq j$, from Equation 12 we have $|i - j| = |j - i| = 1$.

Thus, we can rewrite Equation 1 as:

$$AG(L,n) = \bigcup_{\substack{i,j \in szeq(L,n-1) \\ i \neq j}} \left\{ x \cup y \cup z \;\middle|\; \begin{array}{ll} x \subseteq i - j, & |x| = 1, \\ y \subseteq i \cap j, & |y| = n-2, \\ z \subseteq j - i, & |z| = 1, \\ subsetszeq(x \cup y \cup z, n-1) \subseteq szeq(L, n-1) \end{array} \right\} \quad (13)$$

Note its similarity of this equation with Equation 2, the equation for the LG algorithm.

6.5 Proof

6.5.1 `LGen` will generate only a subset of candidate itemsets generated by `Apriori_Gen` (Equation 5)

For all $S \subseteq L$, $n \geq 2$, and $k \leq n$, suppose $w \in LG(S, L, n, k)$. By definition of LG, we know that $w \notin S$. Also, we know that

$$\exists i, j \in \max\left[S \cup szeq(L, k-1)\right], i \neq j,$$

such that

$$\exists x, y, z \left[\begin{array}{ll} w = x \cup y \cup z & \\ x \subseteq i - j, & |x| = 1, \\ y \subseteq i \cap j, & |y| = n-2, \\ z \subseteq j - i, & |z| = 1, \\ subsetszeq(w, n-1) \subseteq S \cup szle(L, k-1) & \end{array} \right]$$

Now we have two cases, $k < n$ and $k = n$. Note that $k > n$ is not possible by definition of LG.

I. $k < n$

In this case, we have

$$\begin{aligned} subsetszeq(w, n-1) &\subseteq [S \cup szle(L, k-1)] \\ &= szeq(S, n-1) \\ &\subseteq szeq(L, n-1) \end{aligned}$$

since $k < n$ and every element in $subsetszeq(w, n-1)$ is of size $n-1$.

We need to choose $i', j' \in szeq(L, n-1)$ for AG that correspond to $i, j \in \max\left[S \cup szeq(L, k-1)\right]$ for LG. Indeed, for each x, y, z corresponding to each pair of i, j, we can choose $i' = x \cup y$ and $j' = y \cup z$ and we will have $w \in AG(L, n) - S$ for $k < n$.

II. $k = n$

In this case, we have

$$\begin{aligned} subsetszeq(w, n-1) &\subseteq [S \cup szle(L, n-1)] \\ &= szeq(L, n-1) \end{aligned}$$

since all elements in $subsetszeq(w, n-1)$ are of size $n-1$ and $S \subseteq L$.

We just need to choose $i', j' \in szeq(L, n-1)$ for AG that correspond to $i, j \in \max[S \cup szeq(L, n-1)]$ for LG. Similar to Case I, for each x, y, z corresponding to each pair of i, j, we choose $i' = x \cup y$ and $j' = y \cup z$ and we will have $w \in AG(L, n) - S$ for $k = n$.

Combining the result for both cases, we have $LG(S, L, n, k) \subseteq AG(L, n) - S$, and Equation 5 follows.

6.5.2 When large itemsets of size less than n is known and `LGen` is called to generate candidate itemsets of size n, it will generate those itemsets `Apriori_Gen` generates except those already verified to be large. (Equation 6)

We can prove Equation 6 by proving that $AG(L, n) - S \subseteq LG(S, L, n, n)$ for all $S \subseteq L$ and $n \geq 2$; the other side of inclusion, $LG(S, L, n, n) \subseteq AG(L, n) - S$, has already been asserted by Case II in Section 6.5.1.

Now suppose $v \in AG(L, n) - S$. It follows directly that $v \notin S$. Also, we know that

$$\exists i', j' \in szeq(L, n-1), i' \neq j',$$

such that

$$\exists x, y, z \left[\begin{array}{ll} v = x \cup y \cup z & \\ x \subseteq i' - j', & |x| = 1, \\ y \subseteq i' \cap j', & |y| = n-2, \\ z \subseteq j' - i', & |z| = 1, \\ subsetszeq(v, n-1) \subseteq szeq(L, n-1) & \end{array} \right]$$

To find $i, j \in \max[S \cup szeq(L, k-1)], i \neq j$ in LG that corresponds to i', j' in AG, we choose $i \in \max(S)$ such that $i' \subseteq i$ and $j \in \max(S)$ such that $j' \subseteq j$. We know that $i \neq j$ since otherwise $v = i \cup j = i \in\downarrow i \subseteq S$, which contradicts our assumption that $v \notin S$. Hence, we know that $v \in LG(S, L, n, n)$. Thus, $AG(L, n) - S \subseteq LG(S, L, n, n)$ and Equation 6 follows.

6.5.3 `FindLarge` examines the same candidate sets as `Apriori` does, except those itemsets already verified large. (Equation 7)

From Equation 5, we have, for all $S \subseteq L$, $n \geq 2$, and $k \leq n$,

$$LG(S, L, n, k) \subseteq AG(L, n) - S$$

Taking the union over all valid k and n, we have

$$\bigcup_{k=2}^{\infty} \bigcup_{n=k}^{\infty} LG(S, L, n, k) \subseteq \bigcup_{n=2}^{\infty} AG(L, n) - S$$

Since Equation 6 states that when $n = k$, the two sides of Equation 5 will be equal, that is, $LG(S, L, n, n) = AG(L, n) - S$, we have

$$\bigcup_{k=2}^{\infty} \bigcup_{n=k}^{\infty} LG(S, L, n, k) = \bigcup_{n=2}^{\infty} AG(L, n) - S$$

and Equation 7 follows.

6.5.4 `FindLarge` can find all possible large itemsets. (Equations 8 and 9)

Taking the union of $S \cup szeq(L,1)$ to both sides of Equation 7, we have

$$\forall S \subseteq L \quad \left[\bigcup_{k=2}^{\infty} \bigcup_{n=k}^{\infty} LG(S,L,n,k) \cup S \cup szeq(L,1) = \bigcup_{n=2}^{\infty} AG(L,n) \cup szeq(L,1) \right]$$

Substituting Equation 4, we have

$$\forall S \subseteq L \quad \left[\bigcup_{k=2}^{\infty} \bigcup_{n=k}^{\infty} LG(S,L,n,k) \cup S \cup szeq(L,1) \supseteq L \right]$$

which is Equation 8.

Similarly, by taking the union over all n and taking the union of $S \cup szeq(L,1)$ to both sides of Equation 6, we have

$$\forall S \subseteq L \quad \left[\bigcup_{n=2}^{\infty} LG(S,L,n,n) \cup S \cup szeq(L,1) = \bigcup_{n=2}^{\infty} AG(L,n) \cup szeq(L,1) \right]$$

Substituting Equation 4, we have

$$\forall S \subseteq L \quad \left[\bigcup_{n=2}^{\infty} LG(S,L,n,n) \cup S \cup szeq(L,1) \supseteq L \right]$$

which is Equation 9.

6.5.5 When no suggested large itemset is verified to be really large, `LGen` reduces to `Apriori_Gen`. (Equation 10)

When $S = \emptyset$, we have

$$\max[\emptyset \cup szeq(L,n-1)] = szeq(L,n-1)$$

Also, the condition $x \cup y \cup z \notin S$ in Equation 2 is satisfied vacuously. And since

$$\begin{aligned} & subsetszeq(x \cup y \cup z, n-1) \subseteq S \cup szle(L,n-1) \\ \Rightarrow \quad & subsetszeq(x \cup y \cup z, n-1) \subseteq szeq(L,n-1) \end{aligned}$$

when $S = \emptyset$, Equation 10 follows.