Fast Algorithms for Discovering the Maximum Frequent Set

by

Dao-I Lin

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 1998

Approved: _____

Zvi M. Kedem

To my daughter, Shang Lin

Acknowledgements

I am most grateful to my advisor, Professor Zvi M. Kedem, for guiding me through the doctoral studies. He has been very considerate with a sometimes very frustrated student. His continued support and deep involvement have been invaluable during the preparation of this thesis.

I would like to specially thank Sridhar Ramaswamy for his encouragement and very valuable comments and suggestions. Especially, he suggested a very intuitively appealing name for the algorithm forming the core of the thesis: "Pincer-Search." Many in-depth discussions with him have been of great help in my research.

I would like to thank Dennis Shasha for showing me some important papers in this area. I would like to thank Thomas Anantharaman for being on my thesis committee.

I would like to thank Rakesh Agrawal, Ramakrishnan Srikant, and Roberto J. Bayardo for kindly providing me with experimental data.

I would like to thank my parents, Chin-Yung Lin and Yeh San-May Lin, for

their unconditional support over the years. I would like to thank my father-in-law, Bor-Jen Wu, and my mother-in-law, Wen-Hao Lian, for taking good care of my daughter during the time I studied. Finally, I thank my wife, Ling-Hsiang Wu, for encouraging me to pursue my goal of getting a doctoral degree.

This research was partially supported by the National Science Foundation under grant number CCR-94-11590.

Contents

List of Figures

Chapter 1

Introduction

Knowledge discovery in databases (KDD) has received increasing attention and has been recognized as a promising new field of database research. It is defined by Fayyad *et al.* [FPSU96] as "the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data". The key step in the knowledge discovery process is the data mining step, which is "consisting of applying data analysis and discovery algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data" [FPSU96]. This thesis addresses an important pattern discovery algorithm, which can be applied to many data mining applications.

A key component of many data mining problems is formulated as follows. Given a large database of sets of items (representing market basket data, alarm signals,

etc.), discover all frequent *itemsets* (sets of items), where a frequent itemset is one that occurs in at least a user-defined percentage (minimum *support*) of the database. Depending on the semantics attached to the input database, the frequent itemsets, and the term "occurs," we get the key components of different data mining problems such as the discovery of association rules (e.g., [AS94] [HF95] [MTV94]), theories (e.g., [GMS97]), strong rule (e.g., [P91]), episodes (e.g., [MT95] [MT96c]), and minimal keys (e.g., [GMS97]).

Typical algorithms for finding the *frequent set*, i.e., the set of all frequent itemsets, operate in a *bottom-up* breadth-first fashion (e.g., [AIS93a]) [AS94] [AS96] [BMUT97] [HF95] [MT97] [MTV94] [ORS98] [PCY95] [SON95]). The computation starts from frequent 1-itemsets (minimal length frequent itemsets at the bottom) and then extends one level up in every pass until all maximal (length) frequent itemsets are discovered. *All* frequent itemsets are *explicitly examined* and discovered by these algorithms. When *all* maximal frequent itemsets are short, these algorithms perform reasonably well. However, performance drastically decreases when *any* of the maximal frequent itemsets becomes longer, because a maximal frequent itemset of size l implies the presence of $2^l - 2$ non-trivial frequent itemsets (its nontrivial subsets) as well, each of which is explicitly examined by such algorithms. In data mining applications where items are correlated, maximum frequent itemsets could be long [BMUT97].

Therefore, instead of examining and "assembling" all the frequent itemsets, an alternative approach might be to "shortcut" the process and attempt to search for maximal frequent itemsets "more directly," as they immediately specify all frequent itemsets. Furthermore, it suffices to know only the maximal frequent set in many data mining applications, such as the minimal key discovery and the theory extraction.

Finding the *maximum frequent set* (or MFS), the set of all maximal frequent itemsets, is essentially a search problem in a hypothesis search space (a lattice of subsets). The search for the maximum frequent set can proceed from the 1-itemsets to n-itemsets (bottom-up) or from the n-itemsets to 1-itemsets (top-down).

We present a novel *Pincer-Search* algorithm, which searches for the MFS from *both bottom-up and top-down directions*. It performs well even when the maximal frequent itemsets are long.

The bottom-up search is similar to *Apriori* [AS94] and *OCD* [MTV94] algorithms. However, the top-down search is novel. It is implemented efficiently by introducing an auxiliary data structure, the *maximum frequent candidate set* (or MFCS), as explained later. By incorporating the computation of the MFCS in our algorithm, we are able to efficiently approach the MFS from both top-down and bottom-up directions. Unlike the bottom-up search that goes up one level in each pass, the MFCS can help the computation "move down" many levels in the

top-down direction in one pass.

In this thesis, we apply the MFCS concept to the association rule mining. In fact, the MFCS concept can be applied in solving other data mining problems as long as the problem has the closure properties to be discussed later. The monotone specialization relation discussed in [MT97] and [M82] was addressing the same properties.

Popular benchmark databases designed by Agrawal and Srikant [AS94] have been used in [AS96], [ORS98], [PCY95], [SON95], and [ZPOL97]. We use these same benchmarks to evaluate the performance of our algorithm. In most cases, our algorithm not only reduces the number of passes of reading the database but also reduces the number of candidates (for whom support is counted). In such cases, both I/O time and CPU time are reduced by eliminating the candidates that are subsets of maximal frequent itemsets found in the MFCS.

The organization of the rest of the thesis is as follows. The procedures for mining association rules, the cost of the processes, and the properties that can be used to reduce the cost will be discussed in Chapter 2. In this chapter, we will also discuss the traditional one-way search algorithms. Chapter 3 will discuss the concept of our two-way search algorithm, called Pincer-Search, and the implementation of this algorithm. Two technical issues and the techniques to address them will be discussed. Chapter 4 presents the results of our experiments on synthetic

and real-life census and stock market databases. Chapter 5 concludes this thesis.

Chapter 2

Association Rule Mining

2.1 The Setting of the Problem

This section briefly introduces the association rule mining problem. To the extent feasible, we follow the terminology of [AIS93a].

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct items. The itemsets could represent different items in a supermarket or different alarm signals in telecommunication networks [HKMRT96]. A *transaction* T is a set of items in I. A transaction could represent some customer purchases of some items from a supermarket or the set of alarm signals occurring within a time interval. A database D is just a set of transactions. A set of items is called an *itemset*. Note that the items in an itemset are assumed to be stored in some domain dependent total order. For instance, if an item is represented as a number, then the items are stored in a

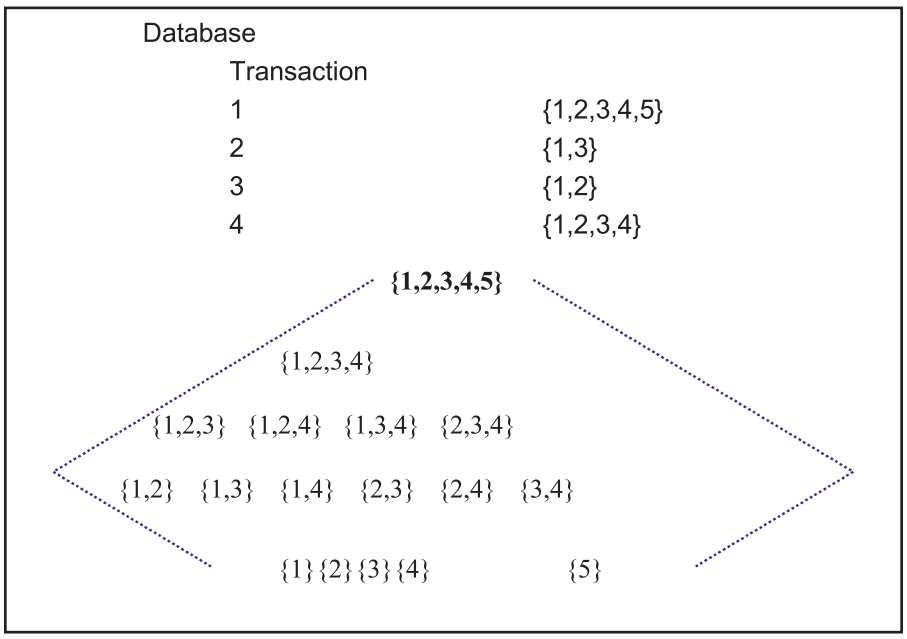

Figure 2.1: An Example Database

numerical order. If an item is represented as a string, then the items are stored in a lexicograpgical order. The number of items in an itemset is called the *length* of an itemset. Itemsets of some length k are referred to as k-itemsets.

A transaction T is said to *support* an itemset $X \subseteq I$ iff it contains all items of X, i.e., $X \subseteq T$. The fraction of the transactions in D that support X is called the *support* of X, denoted as support(X). There is a user-defined *minimum support* threshold, which is a fraction, i.e., a number in [0,1]. An itemset is *frequent* iff its support is above the minimum support. Otherwise, it is *infrequent*.

Example See Fig. 2.1. There are five distinct items, 1 through 5, in the database. There are four transactions. If the minimum support is set to 0.5, then the frequent

itemsets are {1}, {2}, {3}, {4}, {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}, {1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}, and {1,2,3,4}, since they occur in at least half of the database (half of the transactions). For instance, itemset {1,2} is frequent, since three out of the four transactions (transaction 1, 3, and 4) contain items 1 and 2 (i.e., the support of {1,2} is 0.75). On the other hand, itemset {2,5} is infrequent, since only one out of the four transactions contains items 2 and 5 (i.e., the support of {2,5} is 0.25). □

An *association rule* has the form $R : X \rightarrow Y$, where X and Y are two non-empty and non-intersecting itemsets. The *support for rule* R is defined as support($X \cup Y$). A *confidence* factor (represented by percentage), defined as support($X \cup Y$)/support(X) (assume support(X) > 0), is used to evaluate the strength of such association rules. The semantics of the confidence of a rule indicates how often it can be expected to apply, while its support indicates how trustworthy this rule is.

For example, if the minimum confidence is set to 100%, then the association rule {1,4} $\rightarrow$ {2,3} holds. But the {1,2} $\rightarrow$ {3,4} does not hold because its confidence is 67%.

The goal of association rule mining is to discover all rules that have support and confidence greater than some user-defined minimum support and minimum confidence thresholds, respectively.

The normally followed scheme for mining association rules consists of two stages [AS94]:

1. the discovery of frequent itemsets, followed by

2. the generation of association rules.

Example Consider the database as in Fig. 2.1. Suppose the minimum support is set to 75% and the minimum confidence is set to 100%. The first step discovers the frequent set, which is {{1}, {2}, {3}, {1,2}, {1,3}}. The second step generates the association rules. In this example, two association rules will be generated: $\{2\} \rightarrow \{1\}$ and $\{3\} \rightarrow \{1\}$. □

Because the association rule generation step is rather straightforward and also because the frequent itemsets discovery step is the most time consuming process, we explicitly focus the thesis on the discovery of frequent itemsets.

2.2 The Maximum Frequent Set

Among all the frequent itemsets, some will satisfy the property that they have no proper superset that are themselves frequent. Such itemsets are *maximal frequent itemsets*. In other words, an itemset is a maximal frequent itemset if and only if it is frequent and no proper superset of it is frequent. Obviously, an itemset is

frequent if and only if it is a subset of a maximal frequent itemset. The *maximum frequent set* (or MFS) is the set of all the maximal frequent itemsets.

Example See Fig. 2.1. When the minimum support is set to 50%, within all the frequent itemsets, the itemset {1,2,3,4} is a maximal frequent itemset, since it is frequent and no proper superset of it is frequent. Therefore, the MFS is {{1,2,3,4}}. In general, there will be more than one maximal frequent itemset in the MFS. □

Since an itemset is frequent if and only if it is a subset of a maximal frequent itemset, one can discover the MFS first and then generate the subsets of the MFS and count their supports by reading the database once. A *complete hash-tree*, to be discussed in Section 3.7, can be built for this purpose, and then the supports can easily be counted by reading the database once.

Therefore, the problem of discovering the frequent set can be transformed into the problem of discovering the maximum frequent set. The maximum frequent set forms a border between frequent and infrequent sets. Once the maximum frequent set is known, the frequent and infrequent sets are known.

2.3 A Common Scheme for Discovering the Frequent Set

A typical frequent set discovery process follows a standard scheme. Throughout the execution, the set of all itemsets is partitioned, perhaps implicitly, into three

sets:

1. *frequent:* This is the set of those itemsets that have been discovered so far as frequent

2. *infrequent:* This is the set of those itemsets that have been discovered so far as infrequent

3. *unclassified:* This is the set of all the other itemsets.

Initially, the frequent and the infrequent sets are empty. Throughout the execution, the frequent set and the infrequent set grow monotonically at the expense of the unclassified set. The execution terminates when the unclassified set becomes empty, i.e., when every itemset is either in the frequent set or in the infrequent set. In other words, the execution terminates when all maximal frequent itemsets are discovered.

Consider *any process* for classifying itemsets and some point in the execution where some itemsets have been classified as frequent, some as infrequent, and some are still unclassified. Two closure properties can be used to immediately classify some of the unclassified itemsets:

Property 1: If an itemset is infrequent, all it supersets must be infrequent, and they do not need to be examined further

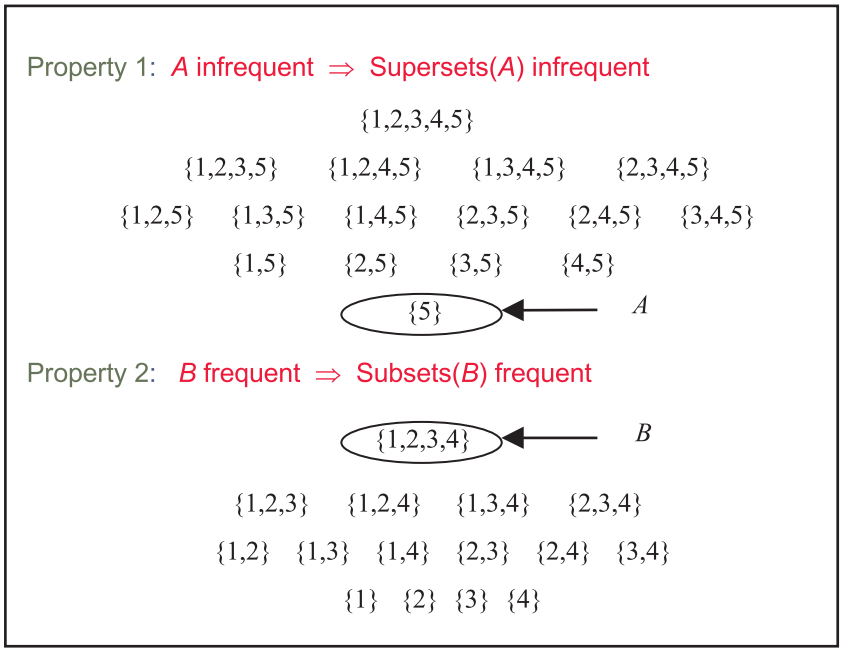

Figure 2.2: Two Closure Properties

Property 2: If an itemset is frequent, all its subsets must be frequent, and they do not need to be examined further

Example Consider the database as in Fig. 2.1 and see Fig. 2.2. The itemset {5} is infrequent and therefore itemset {2,5} must be infrequent, as the support of {2,5} must be equal or less than the support of {5}. In other words, there will be an equal number or fewer transactions containing both items 2 and 5 than those containing item 5. Conversely, if itemset {1,2,3,4} is frequent, then itemset {1,2,3} must be frequent, as there will be an equal number or more transactions containing items 1, 2, and 3 than those containing items 1, 2, 3, and 4. □

In general, it is possible to search for the maximal frequent itemsets either *bottom-up* or *top-down*. If all maximal frequent itemsets are expected to be short (close to 1 in size), it seems efficient to search for them bottom-up. If all maximal frequent itemsets are expected to be long (close to n in size) it seems efficient to search for them top-down.

We first sketch a realization of the most commonly used approach of discovering the MFS. This is a *bottom-up* approach. It consists of repeatedly applying a *pass*, itself consisting of two steps. At the end of pass k all frequent itemsets of size k or less have been discovered.

As the *first step* of pass $k+1$, itemsets of size $k+1$ each having two frequent k-subsets with the same first $k-1$ items are generated. Some of these itemsets are *pruned*, as they do not need to be processed further. Specifically, itemsets that are supersets of infrequent itemsets are pruned (and discarded), as of course they are infrequent (by Property 1). The remaining itemsets form the set of the *candidates* for this pass.

As the *second step*, the support for these itemsets is computed and they are classified as either frequent or infrequent. The support of a candidate is computed by reading the database.

Example Fig. 2.3 shows an example of this bottom-up approach. Consider the same database as in Fig. 2.1. All five 1-itemsets ({1}, {2}, {3}, {4}, {5}) are

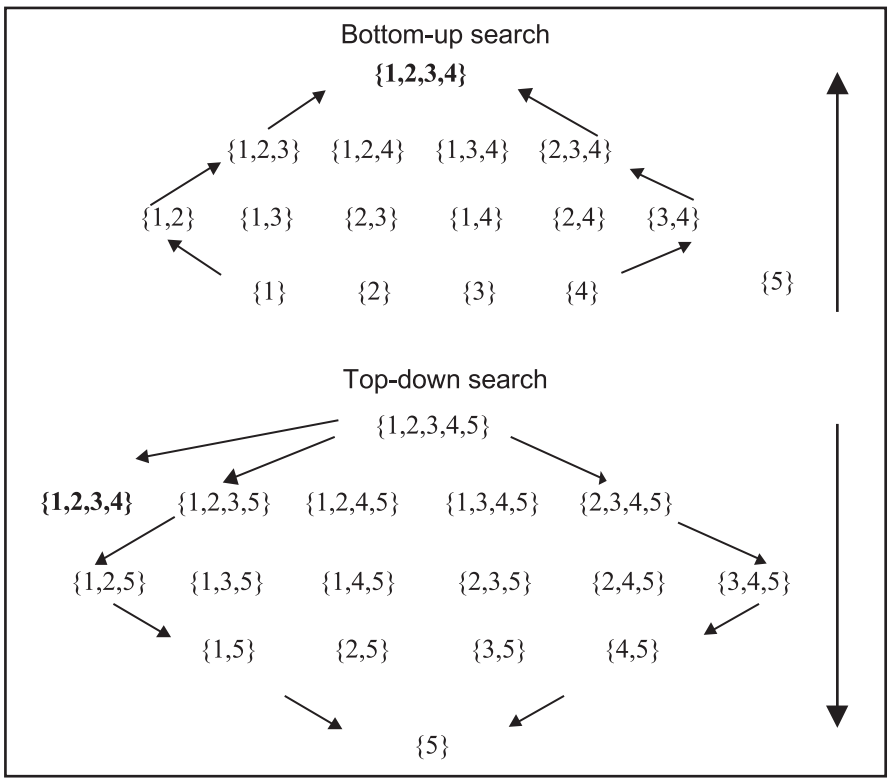

Figure 2.3: One-Way Searches

candidates in the first pass. After the support counting phase, the 1-itemset {5} is determined to be infrequent. By Property 1, all the supersets of {5} need not be considered. So the candidates for the second pass are {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}. The same procedure repeats until all the maximum frequent itemsets (in this example, only one: {1,2,3,4}) are obtained. □

In this bottom-up approach, *every* frequent itemset must have been a candidate at some pass and is therefore *explicitly* considered. As we can see from this example, every frequent itemset (subsets of {1,2,3,4}) need to be visited before the

maximal frequent itemsets are reached. Therefore, this approach is efficient only when *all* maximal frequent itemsets are short.

When *some* maximum frequent itemsets happen to be long, this method will be inefficient. In this case, it might be more efficient to search for those long maximum frequent itemsets using a top-down approach.

A *top-down* approach starts with the single n-itemset and decreases the size of the candidates by one in every pass. When a k-itemset is determined to be infrequent, all of its $(k-1)$-subsets will be examined in the next pass. On the other hand, if a k-itemset is frequent, then all of its subsets must be frequent and need not be examined (by Property 2).

Example See Fig. 2.3 and consider the same database as in Fig. 2.1. The 5-itemset {1,2,3,4,5} is the only candidate in the first pass. After the support counting phase, it is infrequent. The candidates for the second pass are all the 4-subsets of itemset {1,2,3,4,5}. In this example, itemset {1,2,3,4} is frequent and all the others are infrequent. By Property 2, all subsets of {1,2,3,4} are frequent and need not be examined. The same procedure repeats until all maximal frequent itemsets are obtained (i.e., after all infrequent itemsets are visited). □

In this top-down approach, *every* infrequent itemset is *explicitly* examined. As shown in Fig. 2.3, every infrequent itemset (itemset {5} and its supersets) needs to be visited before the maximal frequent itemsets are obtained.

Note that, in a "pure" bottom-up approach, only Property 1 above is used to prune candidates. This is the technique that many algorithms (e.g., [AS94] [AS96] [BMUT97] [HF95] [MT97] [MTV94] [ORS98] [PCY95] [SON95]) use to decrease the number of candidates. In a "pure" top-down approach, only Property 2 is used to prune candidates. This is the technique used in [ZPOL97] and [MT96a].

2.4 Typical Algorithms for Frequent Set Discovery

We briefly discuss existing frequent set discovery algorithms in a roughly chronological order.

2.4.1 AIS and SETM Algorithms

The problem of association rule mining was first introduced in [AIS93a]. An algorithm called AIS was given for discovering the frequent set. To find the frequent itemsets, AIS creates candidates while reading the database. During each pass, the entire database is read. A candidate is created by adding items to those itemsets, called *frontier itemsets*, that were found frequent in the last pass. To avoid generating candidates that do not occur in the database, a candidate is generated only when a transaction containing the candidate is read. New candidates are generated by extending the frontier itemsets with the other items in a transaction.

An expected support factor is used to determine when to stop extending the

frontier itemsets. This expected support for a candidate itemset $X+Y$ is calculated by the formula $f(I_1) \cdot f(I_2) \cdots f(I_k) \cdot (x-c)/\text{dbsize}$, where X is a frontier itemset, $Y = \{I_1, I_2, \ldots, I_k\}$, $X+Y$ is a k-extention of X, X appears in a total of x records, c is the number of records that have been processed in the pass in which $X+Y$ is first considered as a candidate, $f(I_j)$ is the support of itemset $\{I_j\}$, and finally dbsize is the size of the database. In here, k-extention of X means adding k items, which are not in X, into the itemset X.

For instance, if itemset {1,2} is a frontier itemset, for transaction {1,2,3,4,6}, we could have the following candidates:

1. {1,2,3} expected frequent: continue extending
2. {1,2,3,4} expected infrequent: do not extend any further
3. {1,2,3,6} expected frequent: cannot extend any further
4. {1,2,4} expected infrequent: do not extend any further
5. {1,2,6} expected frequent: cannot extend any further

Itemset {1,2,3,4,6} was not considered, since {1,2,3,4} was expected to be infrequent. Similarly, {1,2,4,6} was not considered, since {1,2,4} was expected to be infrequent. Itemsets {1,2,3,5} and {1,2,5} were not considered, since the item 5 was not in the transaction.

Two complicated heuristics, *remaining tuples optimization* and *pruning function optimization*, were used to prune candidates. Unfortunately, this algorithm still generates too many candidates.

SETM [HS93] algorithm was later designed to use only standard SQL commands to find the frequent set. However, like AIS, SETM also creates candidates on-the-fly while reading the database. Both algorithms are not efficient, since they generate and count too many unnecessary candidates.

2.4.2 Apriori and OCD Algorithms

The Apriori algorithm [AS94] is a typical bottom-up approach algorithm, which perform much better than AIS and SETM. It repeatedly uses *Apriori-gen* algorithm to generate candidates and then count their supports by reading the entire database once. The algorithm is described in Fig. 2.4.

Apriori-gen is a candidate generation algorithm. Its underlying approach is based on the Property 1 (see page 11) mentioned above. The candidate generation algorithm is schematically described in Fig. 2.5.

The *join* procedure (see Fig. 2.6) of the Apriori-gen algorithm combines two frequent k-itemsets, which have the same $(k-1)$-prefix, to generate a $(k+1)$-itemset as a new preliminary candidate.

Following the *join* procedure, the *prune* procedure (see Fig. 2.7) is used to

Algorithm: Apriori algorithm
Input: a database and a user-defined minimum support
Output: all frequent itemsets
1. $L_0 := \emptyset$; $k := 1$;
2. $C_1 := \{\{i\} \mid i \in I\}$
3. Answer := $\emptyset$
4. **while** $C_k \neq \emptyset$
5. read database and count supports for C_k
6. L_k := {frequent itemsets in C_k}
7. C_{k+1} := *Apriori-gen*(L_k)
8. $k := k + 1$
9. Answer := Answer $\cup$ L_k
10. **return** Answer

Figure 2.4: Apriori Algorithm

Algorithm: Apriori-gen Algorithm
Input: L_k, the set containing frequent itemsets found in pass k
Output: new candidate set C_{k+1}
1. call *join* procedure to generate preliminary candidate set
2. call *prune* procedure to get final candidate set

Figure 2.5: Apriori-gen Algorithm

remove from the preliminary candidate set all itemsets c such that some k-subset of c is not in the frequent set L_k. In other words, supersets of an infrequent itemset are pruned.

Concurrently with the Apriori algorithm, Mannila *et al.* [MTV94] proposed an OCD algorithm, which used the same closure property to eliminate candidates. The candidate generation process is also divided into two steps. First, the prelim-

Algorithm: The *join* procedure of the Apriori-gen algorithm
Input: L_k, the set containing frequent itemsets found in pass k
Output: preliminary candidate set C_{k+1}
/* The itemsets in L_k are sorted */
1. **for** i **from** 1 **to** $|L_k - 1|$
2. **for** j **from** $i+1$ **to** $|L_k|$
3. **if** $L_k.itemset_i$ and $L_k.itemset_j$ have the same $(k-1)$-prefix
4. C_{k+1} := $C_{k+1} \cup \{L_k.itemset_i \cup L_k.itemset_j\}$
5. **else**
6. **break**

Figure 2.6: Join Procedure

Algorithm: The *prune* procedure of the Apriori-gen algorithm
Input: Preliminary candidate set C_{k+1} generated from the *join* procedure above
Output: final candidate set C_{k+1} which does not contain any infrequent subset
1. **for** all itemsets c in C_{k+1}
2. **for** all k-subsets s of c
3. **if** $s \notin L_k$
4. **delete** c from C_{k+1}

Figure 2.7: Prune Procedure

inary candidate set is calculated as $C'_k = \{X \cup X' \mid X, X' \in L_{k-1}$ and $|X \cap X'| = k-2\}$. Second, the actual candidate set is calculated as $C_k = \{X \in C'_k \mid X$ contains k members of $L_{k-1}\}$. But, in general, the first step might produce a superset of the candidates produced in the *join* procedure of the Apriori algorithm.

The Apriori-gen algorithm has been very successful in reducing the number of candidates and has been used in many subsequent algorithms, such as *Partition*

[SON95], *DHP* [PCY95], *Sampling* [T96b], *DIC* [BMUT97], and *Clique* [ZPOL97].

2.4.3 DHP and Partition Algorithm

DHP [PCY95] tried to improve Apriori by using a hash filter to count the upper-bound of the support for the next pass. The upper-bound can be used to eliminate candidates. This algorithm is targeting on reducing the number of candidates in the second pass, which could be very large.

We use an example to show how the hash filter works. Suppose the frequent 1-itemsets are {1}, {2}, {3}, {5} in a database of five items, 1, 2, 3, 4, and 5. In the first pass, while a transaction is examined, DHP not only uses this transaction to update the support of all 1-itemsets in this transaction but also updates the counts in a hash table for 2-itemsets. For instance, suppose the hash function is defined as $h(\{x, y\}) = (10x + y) \bmod 7$. A transaction {1,3,5} will allow DHP to increment supports for 1-itemsets {1}, {3}, and {5}. DHP will also update the counts in index $h(\{1, 3\})$, index $h(\{1, 5\})$, and index $h(\{3, 5\})$ of the hash table. That is, DHP will update index 6, index 1, and index 0 of the hash table.

After the database is read once, we can look at the hash table and create a hash filter. If the count in a bucket is less than the product of the minimum support and the database size, then 2-itemsets in this bucket must be infrequent. In this case, the value is set to 0 in the filter. Otherwise, the value is set to 1 in the filter.

This filter is later used to prune candidates. Note that if the count in a bucket is above the threshold, it is not guaranteed that the itemsets in that bucket are frequent unless there is only one itemset has been hashed to that bucket.

By using this hash filter, some candidates can be pruned before reading the database in the next pass. However, according to the investigations done in [AS96], this optimization may not be as good as using a two-dimensional array as discussed in [SA95a]. Furthermore, like Apriori, DHP considers every frequent itemset.

Savasere *et al.* [SON95] proposed a Partition algorithm. This paper addresses two issues in previous algorithms. The first issue is that the algorithms discussed so far require reading the database many times (as many times as the length of the longest frequent itemset). The second issue is that most of the records in the database are not useful in the later passes, since many of the records may not even contain the items in the candidates. In other words, a record that does not contain any item in any candidates can be removed without affecting the support counting process.

The first issue is addressed by horizontally dividing the database into equal sized *partitions*, which can fit in main memory. Each partition is processed independently to produce a *local frequent set* for that partition in the first pass. The process in each partition is using a bottom-up approach similar to Apriori but with a different data structure (to be discussed later). After all local frequent sets are

discovered, their union, called *global candidate set*, forms a superset of the actual frequent set. It relies on the fact that if an itemset is frequent then it must be frequent in at least one of the partition. Similarly, if an itemset is not frequent in any partition, then it must be infrequent.

During the second pass, the database is read again to produce the actual support for the global candidate set. Therefore, the entire process takes only two passes.

The computation in each partition is using a regular bottom-up approach. It will extend the length of the candidates by one in every loop until no more candidates can be generated. To prevent reading the database each time the length of the candidate is incremented, the database is transformed into a new data struture, called TID-list. Each candidate stores a list of the transaction IDs that support this candidate. The database needs to be partitioned into a size that fits into the main memory. The use of the TID-list implicitly addresses the second issue, since only those transactions that support current candidates will be in the TID-list. However, this TID-list incurs additional overhead, since the transaction ID for a transaction containing m items may appear, in the worst case, in $\binom{m}{k}$ TID-lists for the kth pass ($\binom{m}{k}$ means choose k items from those m items).

There are three major problems with this Partition approach. First, it requires the choice of a good partition size to get a good performance. If the partition

is too big, then the TID-list might grow too fast and no longer be able to fit in the main memory. But, if the partition is too small, then there will be too many global candidates and most of them will turn out to be infrequent. Second, it is negatively impacted by data skew, which causes the local frequent set to be very different from each other. Then, the global candidate set will be very large. Third, this algorithm will consider more candidates than Apriori. If there are long maximal frequent itemsets, this algorithm is infeasible.

2.4.4 Sampling Algorithm

Toivonen [T96b] proposed to consider only some samples of the database and discover an *approximate frequent set* by using a standard bottom-up approach algorithm, such as the OCD algorithm or the Apriori algorithm. This random sampling approach can solve the data skew problem in the Partition algorithm. Sampling algorithm is a *guess-and-correct* algorithm [MT96a] [MT97]. It guesses an answer in the first pass, then corrects the answer in subsequent passes.

Unlike Partition, which look at the entire database, Sampling looks only at a part of the database in the first pass. Therefore, a frequent itemset found in the sample database may not be actually frequent (false positive) and an infrequent itemset found in the sample database may turn out to be frequent (false negative). The false positive itemsets can easily be removed once the entire database is read

and the actual supports are known. The missing frequent itemsets (false negative itemsets) require a more complicated way to be recovered.

To reduce the number of false negative itemsets, the minimum support can be set to a lower value. However, to guarantee that no false negative itemset appears, some itemsets constituting so called *negative border* need to be examined. Given a set S of itemsets closed with respect to the set inclusion relation (i.e., satisfying both Property 1 and Property 2), the negative border $\mathrm{Bd}^-(S)$ of set S consists of those minimal itemsets not in S. From subset lattice point of view, this negative border contains the neighbor itemsets of the *local* maximal frequent itemsets. The problem of computing the negative border can be transformed into a *hypergraph traversal* problem [MT97].

If no itemset in the negative border turns out to be frequent, then there is no false negative itemset. Otherwise, some false negative itemsets may need to be recovered. They can be recovered by further extending the negative border.

The performance of this Sampling algorithm relies on the sample database. This algorithm will consider at least the same candidates as Apriori. Therefore, it is still inefficient when the frequent itemsets are long.

2.4.5 A-Random-MFS, DIC, Eclat, MaxEclat, TopDown, Clique, and MaxClique Algorithms

A randomized algorithm called *A-Random-MFS* for discovering the maximum frequent set was presented by Gunopulos *et al.* [GMS97]. The randomized algorithm alone cannot guarantee completeness. Therefore, a complete algorithm that can discover all maximal frequent itemsets requires repeatedly calling the randomized algorithm until no new maximal frequent itemset can be found. It requires computing a set containing *minimal orthogonal elements* which is similar to the negative border in the Sampling algorithm. This computation is in general non-trivial. There is no known polynomial algorithm for this.

The proposed randomized algorithm contains a loop to keep randomly extending the length of a frequent itemset until it become infrequent. It is assuming that the database fits in the main memory. However, it is not clear how this algorithm can be used to handle the cases when the database resides on the disk. When the database cannot fit in the main memory, it is expensive to determine whether the extended itemset is frequent or not, since extending the length by one require reading the database once.

Brin *et al.* [BMUT97] proposed a dynamic itemset counting (DIC) algorithm which combines candidates with different lengths into one pass. This algorithm focuses on reducing the number of passes of reading the database. DIC works like

a train running over the database with stops at every M transactions (M is an adjustable parameter). The track (database) is a circle. The database is divided horizontally into segments of equal size M.

DIC starts counting the supports of 1-itemsets while the first segment is read. It starts counting the supports of some 2-itemsets while the second segment is read. Those 2-itemsets, generated by combining *local* frequent 1-itemset found in the first segment, will be counted. After reading the second segment of the database, additional 2-itemsets could become candidates because additional 1-itemsets may become *locally* frequent with respect to the first and the second segments. Some 2-itemsets may become locally frequent and they may be used to generate some 3-itemsets as candidates. The supports of those 1-itemsets, 2-itemsets, and 3-itemsets will be counted together starting from the third segment. The process continues until no frequent itemsets can be found. Property 1 is also used here for candidate pruning.

In general, this algorithm can reduce the number of passes the database is read. It can perform well when the data is fairly homogenous throughout the file, and M is reasonably small. However, like Partition, this approach is sensitive to data skew. When the data is non-homogenous, this algorithm may suffer because too many local frequent itemsets will turn out to be infrequent globally. This algorithm will consider more candidates than Apriori, therefore, it is also infeasible for finding

long frequent itemsets.

Concurrently with our work, Zaki *et al.* [ZPOL97] discussed bottom-up and top-down lattice traversals, which are similar to what we have discussed in Section 2.3. In addition to the pure bottom-up and pure top-down traversals, they proposed a hybrid traversal, which contains a look-ahead phase followed by a pure bottom-up phase. The look-ahead phase consists of extending the frequent 2-itemsets one item at a time until the extended itemset becomes infrequent. After the look-ahead phase, an Apriori-like bottom-up traversal is then executed.

Note that, their hybrid method looks ahead only at some long candidate itemsets during the initialization stage (in the second pass). As will be discussed in the next chapter, our approach, in contrast, looks ahead at long candidate itemsets throughout the entire execution.

Like A-Random-MFS algorithm, this hybrid traversal requires the database to be stored in main memory. TID-list was used for this purpose. However, as we have discussed, using the TID-list has some weaknesses.

They presented two different ways to cluster the itemsets after the frequent 2-itemsets are discovered. *Equivalence class* clustering groups itemsets together if they have the same first item (same 1-prefix). *Maximal uniform hypergraph clique* clustering groups itemsets together if they are in the same maximal uniform hypergraph clique. The maximal uniform hypergraph clique is similar to our MFCS (to

be discussed in the next chapter). For every cluster generated, they then applied the lattice traversals (either bottom-up, top-down, or hybrid) to each cluster (a lattice for each cluster). The maximal uniform hypergraph clique approach provides a more precise approximation of the maximal frequent itemsets but at a higher computation cost. The equivalence class approach requires less computation, but usually can not approximate the maximal frequent itemsets precisely.

They proposed six algorithms based on the combinations of the different lattice traversing methods and the different clustering methods. *Eclat* uses equivalent class clustering with bottom-up traversal. *MaxEclat* uses equivalent class clustering with hybrid traversal. *Clique* uses maximal uniform hypergraph clique clustering with bottom-up traversal. *MaxClique* uses maximal uniform hypergraph clique clustering with hybrid traversal. *TopDown* uses maximal uniform hypergraph clique clustering with top-down traversal. Equivalent class clustering with top-down traversal was not discussed, since it is too inefficient.

One of the most important differences between MaxClique and Pincer-Search is that MaxClique only calculates the maximal uniform hypergraph clique in the second pass. However, simply considering the maximal uniform hypergraph clique based on the frequent 2-itemsets may be very inaccurate. In contrast, our Pincer-Search algorithm repeatedly maintains the upper-bound of the frequent itemsets (MFCS) throughout the entire process. The approximation is dynamically adjusted

based on all available information and therefore is very accurate. Actually, the MFCS in the Pincer-Search algorithm is the most accurate approximation one can get when no additional information is available.

Another important difference is that they used a bottom-up approach to calculate the maximal uniform hypergraph clique. Conceptually, it keeps applying Apriori-gen until no more candidates can be generated. In contrast, Pincer-Search uses a top-down approach. As will be discussed in the next chapter, this top-down approach has the advantage that it is suitable for incremental updates. It updates the MFCS only when a new infrequent itemset is discovered.

In fact, MaxClique can be viewed as a special case of our Pincer-Search algorithm, if we discard the inplementation details.

2.5 Other Related Work

General survey papers regarding data mining problems can be found in, e.g., [FPS96] [FPSU96] [M97] [PBKKS97].

In addition to the algorithms discussed so far, there has been extensive research relating to the problem of association rule mining such as [BMS97], [GKMT97], [HCC92], [HF95], [MT96b], [ORS98], [S96], [SA95b], [SA96b], [SVA97], [T96a], and [KMRTB94]. Similar candidate pruning techniques has been applied to discover sequential patterns (e.g., [MT95] [MT96c] [SA95a] [SA96a] [Z97]) and episodes (e.g.,

[MT95] [MT96c]). Some other papers concentrate on designing parallel algorithms on share-nothing parallel environment (e.g., ([AS96] [HKK97]) and share-memory parallel environment (e.g., [ZPOL96]). The discovery of frequent set is a key process in solving these problems. A good algorithm for discovering frequent set could be applied in solving these problems.

Mannila and Toivonen [MT97] [GKMT97] analyze the complexity of the (levelwise) bottom-up breadth-first search style algorithms. As our algorithm does not fit in this model, their complexity low bound does not apply to it.

Our work was inspired by the notion of *version space* in Mitchell's machine learning paper [M82]. We found that if we treat a newly discovered frequent itemset as a new *positive training instance*, a newly discovered infrequent itemset as a new *negative training instance*, the candidate set as the *maximally specific generalization* (S), and the MFCS as the *maximally general generalization* (G), then we will be able to use a two-way approaching strategy to discover the maximum frequent set (*generalization* in his terminology) efficiently. We will discuss this in detail in the next chapter.

Chapter 3

Fast Algorithms for Discovering the Maximum Frequent Set

3.1 Our Approach to Reducing the Number of Candidates and the Number of Passes

As discussed in the last chapter, the bottom-up approach is good for the case when *all* maximal frequent itemsets are short and the top-down approach is good when *all* maximal frequent itemsets are long. If some maximal frequent itemsets are long and some are short, then both one-way search approaches will not be efficient.

To design an algorithm that can efficiently discover both long and short maximal frequent itemsets, one might think of simply running both bottom-up and

top-down programs at the same time. However, this naive approach is not good enough. We can actually do much better than that.

Recall that the bottom-up approach described above uses only Property 1 to reduce the number of candidates and the top-down approach uses only Property 2 to reduce the number of candidates. Conceivably, a process that relies on both properties to prune candidates could be much more efficient than a process that relies on *only* the first or the second.

In our approach of combining the top-down and bottom-up searches, we rely on *both* properties to prune candidates and make use of the information gathered in one direction to prune more candidates during the search in the other direction.

If some maximal frequent itemset is found in the top-down direction, then this itemset can be used to eliminate (possibly many) candidates in the bottom-up direction. The subsets of this frequent itemset can be pruned because they are frequent (Property 2). Of course, if an infrequent itemset is found in the bottom-up direction, then it can be used to eliminate some candidates in the top-down direction (Property 1). This "two-way search approach" can fully make use of both properties and thus speed up the search for the maximum frequent set. We call this "two-way search approach" as *Pincer-Search* method. This two-way search approach was first introduced in [L96] [L97] and formalized in [LK97] [LK98].

Let us use an example to explain the concept of this two-way search approach.

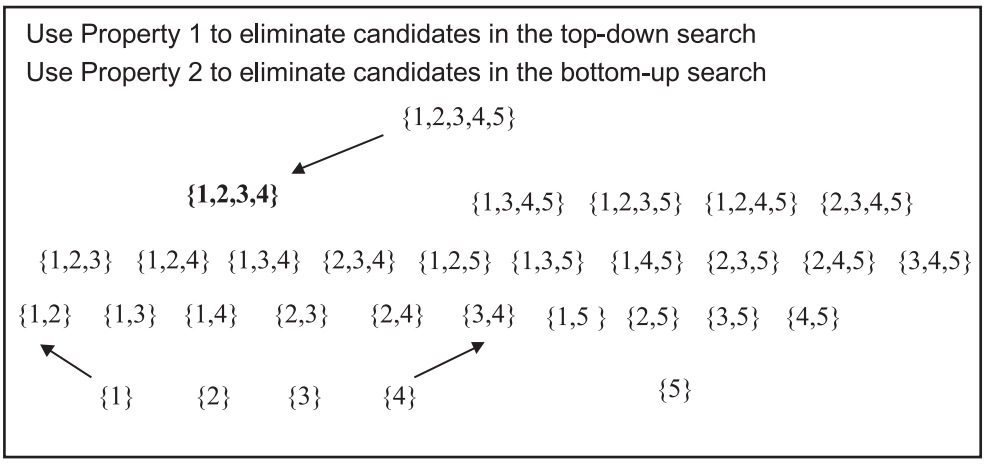

Figure 3.1: Two-Way Search

Considering the same database as in Fig. 2.1, Fig. 3.1 shows the process of the Pincer-Search algorithm. Through this example, we will see that by combining both bottom-up and top-down searches in each pass and by using both properties to eliminate candidates, we could actually use less candidates and less passes than both bottom-up and top-down approaches in discovering the maximum frequent set.

Example See Fig. 3.1. In the first pass, all five 1-itemsets are the candidates for the bottom-up search and the 5-itemset {1,2,3,4,5} is the candidate for the top-down search. After the support counting phase, infrequent itemset {5} is discovered by the bottom-up search and this information is shared with the top-down search. This infrequent itemset {5} not only allows the bottom-up search to eliminate its supersets as candidates but also allows the top-down search to eliminate its

supersets as candidates in the second pass. In the second pass, the candidates for the bottom-up search are {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, and {3,4}. Itemsets {1,5}, {2,5}, {3,5}, and {4,5} are not candidates, since they are supersets of {5}. The only candidate for the top-down search in the second pass is {1,2,3,4}, since all the other 4-subsets of {1,2,3,4,5} are supersets of {5}. After the second support counting phase, {1,2,3,4} is discovered to be frequent by the top-down search. This information is shared with the bottom-up search. All of its subsets are frequent and need not be examined. In this example, itemsets {1,2,3}, {1,2,4}, {1,3,4}, and {2,3,4} will not be candidates for our bottom-up or top-down searches. After that, the program can terminate, since there are no candidates for either bottom-up or top-down searches. □

In this example, our two-way approaching method actually considers less candidates than both one-way bottom-up and top-down approaches. Interestingly, this two-way approaching method also uses fewer passes of reading the database than either bottom-up or top-down approaches. The "pure" bottom-up approach would have taken four passes in this example. The "pure" top-down approach would have taken five passes. Our Pincer-Search method used only two passes. In fact, our two-way approaching method will always use at most as many passes as the minimum of the passes used by bottom-up approach and top-down approach.

Reducing the number of candidates is of critical importance for the efficiency of

the frequent set discovery process, since the cost of the entire process comes from reading the database (I/O time) to generate the supports of candidates (CPU time) and the generation of new candidates (CPU time). The support counting of the candidates is the most expensive part. Therefore, the number of candidates dominates the entire processing time. Reducing the number of candidates not only can reduce the I/O time but also can reduce the CPU time, since fewer candidates need to be counted and generated.

Therefore, it is important that Pincer-Search reduces both the number of candidates and the number of passes. A realization of this two-way search algorithm is discussed next.

3.2 Two-Way Search by Using the MFCS

We have designed a combined two-way search algorithm for discovering the maximum frequent set. It relies on a new data structure during its execution, the *maximum frequent candidate set*, or MFCS for short, which we define next.

Definition 1 *Consider some point during the execution of an algorithm for finding the* MFS. *Some itemsets are frequent, some infrequent, and some unclassified. The maximum frequent candidate set* (MFCS) *is the set of all maximal itemsets that are not known to be infrequent. To be more specific, it is a minimum cardinality set of itemsets such that the union of all the subsets of its elements contains all*

the frequent itemsets but does not contain any infrequent itemsets, that is, it is a minimum cardinality set satisfying the conditions

$$\text{FREQUENT} \subseteq \cup\{2^X \mid X \in \text{MFCS}\}$$

$$\text{INFREQUENT} \cap \{2^X \mid X \in \text{MFCS}\} = \emptyset$$

where FREQUENT *and* INFREQUENT*, stand respectively for sets of all frequent and infrequent itemsets (classified as such so far). (2^X is the power set of X.)*

Thus obviously at any point of the algorithm MFCS is a superset of the MFS. When the algorithm terminates, the MFCS and the MFS are equal.

The computation of our algorithm follows the bottom-up breadth-first search approach. We base our presentation on the Apriori algorithm, and for greatest ease of exposition we present our algorithm as a modification to that algorithm.

Briefly speaking, in each pass, in addition to counting supports of the candidates in the bottom-up direction, the algorithm also counts supports of the itemsets in the MFCS: this set is adapted for the top-down search. This will help in pruning candidates, but will also require changes in candidate generation, as explained later.

Consider a pass k, during which itemsets of size k are to be classified. If some itemset that is an element of the MFCS, say X, of cardinality greater than k is found to be frequent in this pass, then all its subsets must be frequent. Therefore, all of its subsets of cardinality k can be pruned from the set of candidates

considered in the bottom-up direction in this pass. They, and their supersets will never be candidates throughout the rest of the execution, potentially improving performance. But of course, as the maximum frequent set is finally computed, they "will not be forgotten."

Similarly, when a new infrequent itemset is found in the bottom-up direction, the algorithm will use it to update the MFCS. The subsets of the MFCS must not contain this infrequent itemset.

Figure 3.2 conceptually shows the combined two-way search. the MFCS is initialized to contain a single element, the itemset of cardinality n containing all the elements of the database. As an example of its utility, consider the first pass of the bottom-up search. If some m 1-itemsets are infrequent after the first pass (after reading the database once), the MFCS will have one element of cardinality $n - m$. This itemset is generated by removing the m infrequent items from the initial element of the MFCS. In this case, the top-down search goes down m levels in one pass. In general, unlike the search in the bottom-up direction, which goes up one level in one pass, *the top-down search can go down many levels in one pass.*

By using the MFCS, we will be able to discover some maximal frequent itemsets in early passes. This early discovery of the maximal frequent itemsets can reduce the number of candidates and the passes of reading the database which in turn can reduce the CPU time and I/O time. This is especially significant when the

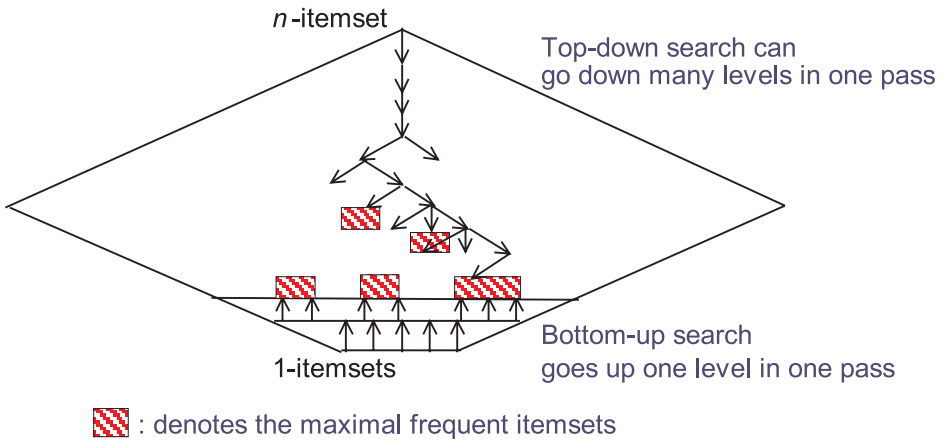

Figure 3.2: The Search Space of Pincer-Search

maximal frequent itemsets discovered in the early passes are long.

For our approach to work efficiently, we need to address two issues. First, how to update the MFCS efficiently? Second, once the subsets of the maximal frequent itemsets found in the MFCS are removed, how do we generate the correct candidate set for the subsequent passes in the bottom-up direction?

3.3 Updating the MFCS Efficiently

Consider some itemset Y that has been "just" classified as infrequent. By the definition of the MFCS, it will be a subset of one or more itemsets in the MFCS and we need to update the MFCS such that its subsets will not contain Y. To update the MFCS, we will do the following process for every superset of Y that is in the MFCS. We replace every such itemset (say X) by $|Y|$ itemsets, each obtained

by removing from X a single item (element) of Y. A newly generated itemset is added to the MFCS only when it is not a subset of any itemset in the MFCS. We do this for each newly discovered infrequent itemset. Formally, we have the *MFCS-gen* algorithm as in Fig. 3.3 (shown here for pass k).

Algorithm: *MFCS-gen*
Input: Old MFCS and the infrequent set S_k found in pass k
Output: New MFCS
1. **for** all itemsets $s \in S_k$
2. **for** all itemsets $m \in$ MFCS
3. **if** s is a subset of m
4. MFCS := MFCS $\setminus \{m\}$
5. **for** all items $e \in$ itemset s
6. **if** $m \setminus \{e\}$ is not a subset of any itemset in the MFCS
7. MFCS := MFCS $\cup \{m \setminus \{e\}\}$
8. **return** MFCS

Figure 3.3: MFCS-gen Algorithm

Example See Fig. 3.4. Suppose {{1,2,3,4,5,6}} is the current ("old") value of the MFCS and two new infrequent itemsets {1,6} and {3,6} are discovered. Consider first the infrequent itemset {1,6}. Since the itemset {1,2,3,4,5,6} (element of the MFCS) contains items 1 and 6, one of its subsets will be {1,6}. By removing item 1 from itemset {1,2,3,4,5,6}, we get {2,3,4,5,6}, and by removing item 6 from itemset {1,2,3,4,5,6} we get {1,2,3,4,5}. After considering itemset {1,6}, the MFCS becomes {{1,2,3,4,5}, {2,3,4,5,6}}. Itemset {3,6} is then used to update this MFCS. Since {3,6} is a subset of {2,3,4,5,6}, two itemsets {2,3,4,5} and {2,4,5,6} are gen-

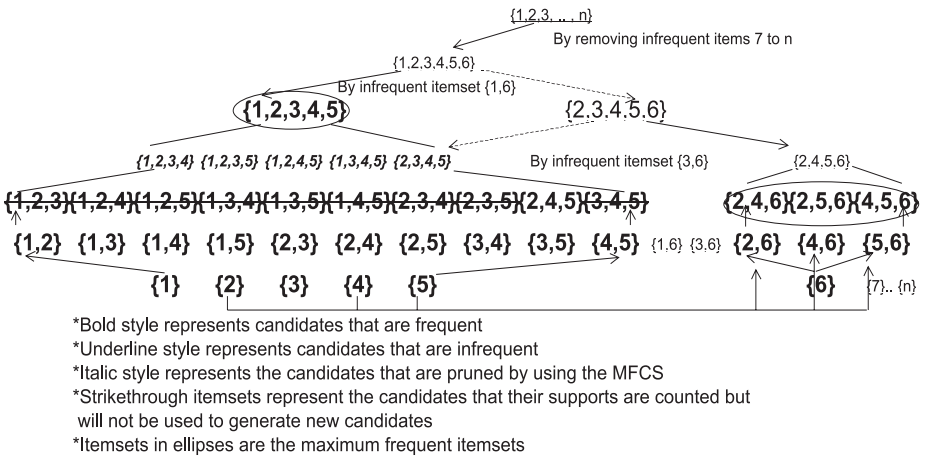

Figure 3.4: Pincer-Search

erated to replace {2,3,4,5,6}. Itemset {2,3,4,5} is a subset of itemset {1,2,3,4,5} in the new MFCS, and it will not be added to the MFCS. Therefore, the MFCS becomes {{1,2,3,4,5}, {2,4,5,6}}. The top-down arrows in Fig. 3.4 show the updates of the MFCS. □

Lemma 1 *The algorithm MFCS-gen correctly updates the* MFCS.

Proof: The algorithm excludes all the infrequent itemsets, so the final set will not contain any infrequent itemsets as subsets of its elements. Step 7 removes only one item from the itemset m: the longest subset of the itemset m that does not contain the infrequent itemset s. Since this algorithm always generates longest itemsets, the number of the itemsets will be minimum at the end. Therefore, this algorithm generates the MFCS correctly. □

3.4 New Candidate Generation Algorithms

Recall that, as discussed in Section 2.4.2, a preliminary candidate set will be generated after the *join* procedure is called. In our algorithm, after a maximal frequent itemset is added to the MFS, all of its subsets in the frequent set (computed so far) will be removed. We show by example that if the original *join* procedure of the Apriori-gen algorithm is applied, some of the needed itemsets could be missing from the preliminary candidate set. Consider Fig. 3.4. Suppose the original frequent itemset L_3 is {{1,2,3}, {1,2,4}, {1,2,5}, {1,3,4}, {1,3,5}, {1,4,5}, {2,3,4}, {2,3,5}, {2,4,5}, {2,4,6}, {2,5,6}, {3,4,5}, {4,5,6}}. Assume itemset {1,2,3,4,5} in the MFCS is determined to be frequent. Then all 3-itemsets of the original frequent set L_3 will be removed from it by our algorithm, except for {2,4,6}, {2,5,6}, and {4,5,6}. Since the Apriori-gen algorithm uses a $(k-1)$-prefix test on the frequent set to generate new candidates, and no two itemsets in the current frequent set {{2,4,6}, {2,5,6}, {4,5,6}} share a 2-prefix, no candidate will be generated by applying the *join* procedure on this frequent set. However, the correct preliminary candidate set should be {{2,4,5,6}}.

Based on the above observation, we need to recover some missing candidates.

3.4.1 New Preliminary Candidate Set Generation Procedure

In our new preliminary candidate set generation procedure, the *join* procedure of the Apriori-gen algorithm is first called to generate a temporary candidate set, which might be incomplete. When it is incomplete, a *recovery* procedure will be called to recover those missing candidates.

All missing candidates can be obtained by restoring some itemsets to the current frequent set. The restored itemsets are extracted from the MFS, which implicitly maintains all frequent itemsets discovered so far.

The first group of itemsets that needs to be restored contains those k-itemsets that have the same $(k-1)$-prefix as some itemset in the current frequent set.

Consider then in pass k, an itemset X in the MFS and an itemset Y in the current frequent set such that $|X| > k$. Suppose that the first $k-1$ items of Y are in X and the $(k-1)$st item of Y is equal to the jth item of X. We obtain the k-subsets of X that have the same $(k-1)$-prefix as Y by taking one item of X that has an index greater than j and combining it with the first $k-1$ items of Y, thus getting one of these k-subsets. After these k-itemsets are found, we recover candidates by combining them with itemset Y as shown in Fig. 3.5.

Example See Fig. 3.4. The MFS is {{1,2,3,4,5}} and the current frequent set is {{2,4,6}, {2,5,6}, {4,5,6}}. The only 3-subset of {{1,2,3,4,5}} that needs to be restored for itemset {2,4,6} to generate a new candidate is {2,4,5}. This is

```
Algorithm: The recovery procedure
Input: C_{k+1} from join procedure, L_k, and current MFS
Output: a complete candidate set C_{k+1}
1. for all itemsets l in L_k
2.  for all itemsets m in MFS
3.   if the first k-1 items in l are also in m
4.    /* suppose m.item_j = l.item_{k-1} */
5.    for i from j+1 to |m|
6.     C_{k+1} := C_{k+1} ∪ {{l.item_1, l.item_2, ..., l.item_k, m.item_i}}
```

Figure 3.5: Recovery Algorithm

because it is the only subset of {{1,2,3,4,5}} that has the same length and the same 2-prefix as itemset {2,4,6}. By combining {2,4,5} and {2,4,6}, we recover the missing candidate {2,4,5,6}. No itemsets need to be restored for itemsets {2,5,6} and {4,5,6}. □

The second group of itemsets that need to be restored consists of those k-subsets of the MFS having the same $(k-1)$-prefix but having no common superset in the MFS. A similar *recovery* procedure can be applied after they are restored.

3.4.2 New Prune Procedure

After the recovery stage, a preliminary candidate set will be generated. We can then proceed to the prune stage. Instead of checking to see if all k-subsets of an itemset X are in L_k, we can simply check to see if X is a subset of an itemset in the current MFCS as shown in Fig. 3.6. In comparison with the *prune* procedure

of Apriori-gen, we use one fewer loop.

Algorithm: New *prune* procedure
Input: current MFCS and C_{k+1} after *join* and *recovery* procedures
Output: final candidate set C_{k+1}
1. **for** all itemsets c in C_{k+1}
2. **if** c **is not** a subset of any itemset in the current MFCS
3. **delete** c from C_{k+1}

Figure 3.6: New Prune Algorithm

3.4.3 Correctness of the New Candidate Generation Algorithm

In summary, our candidate generation process contains three steps as described in Fig. 3.7.

Algorithm: New candidate generation algorithm
Input: L_k, current MFCS, and current MFS
Output: new candidate set C_{k+1}
1. call the *join* procedure as in the Apriori algorithm
2. call the *recovery* procedure if necessary
3. call the new *prune* procedure

Figure 3.7: New Candidate Generation Algorithm

Lemma 2 *The new candidate generation algorithm generates correct candidate set.*

Proof: Recall the candidate generation process as in Apriori-gen. There are four possible cases when we combine two frequent k-itemsets, say I and J, which have

the same $(k-1)$-prefix, to generate a $(k+1)$-itemset as a new preliminary candidate. In this proof, we will show that, even though that we remove the subsets of the MFS from the current frequent set, our new candidate generation algorithm will handle all these cases correctly.

Case 1: $\{I$ is not a subset of $X \mid$ for all $X \in$ MFS$\}$ and $\{J$ is not a subset of $Y \mid$ for all $Y \in$ MFS$\}$. Both itemsets are in the current frequent set. The *join* procedure will combine them and generate a preliminary candidate.

Case 2: $\{I$ is a subset of $X \mid X \in$ MFS$\}$ and $\{J$ is not a subset of $Y \mid$ for all $Y \in$ MFS$\}$. I is removed from the current frequent set. However, combining I and J will generate a candidate that needs to be examined. The *recovery* procedure discussed above will recover candidates in this case.

Case 3: $\{I$ and J are both subsets of some $X \mid X \in$ MFS$\}$. Don't combine them, since the candidate that they generate will be a subset of X and must be frequent.

Case 4: Not in Case 3 and $\{I$ is a subset of some $X \mid X \in$ MFS$\}$ and $\{J$ is a subset of some $Y \mid Y \in$ MFS$\}$ and $X \neq Y$. Both I and J are removed from the current frequent set. However, by combining them,

a necessary candidate will be generated. Similar *recovery* procedure as discussed above will recover missing candidates of this case.

Our preliminary candidate generation algorithm considers the same combinations of the frequent itemsets as does in the Apriori-gen algorithm. This preliminary candidate generation algorithm will generate all the candidates, as the Apriori-gen algorithm does, except those that are subsets of the MFS. By the *recovery* procedure, some subsets of the MFS will be restored in the later passes when necessary.

Lemma 1 showed that MFCS will be maintained correctly in every pass. Therefore, our new *prune* procedure will make sure no superset of infrequent itemsets is in the preliminary candidate set. Therefore, the *new* candidate generation algorithm is correct. □

3.5 The Pure Pincer-Search Algorithm

We now present our complete algorithm (see Fig. 3.8), *The Pincer-Search Algorithm*, which relies on the combined approach for determining the maximum frequent set. Lines 9 to 12 constitute our *new* candidate generation procedure.

The MFCS is initialized to contain one itemset, which consists of all the database items. The MFCS is updated whenever new infrequent itemsets are found (line 8). If an itemset in the MFCS is found to be frequent, then its subsets

Algorithm: The Pincer-Search algorithm
Input: a database and a user-defined minimum support
Output: MFS which contains all maximal frequent itemsets

1. $L_0 := \emptyset$; $k := 1$; $C_1 := \{\{i\} \mid i \in I\}$
2. MFCS := $\{\{1, 2, \ldots, n\}\}$; MFS := $\emptyset$
3. **while** $C_k \neq \emptyset$
4. read database and count supports for C_k and MFCS
5. remove frequent itemsets from MFCS and add them to MFS
6. L_k := {frequent itemsets in C_k} \ {subsets of MFS}
7. S_k := {infrequent itemsets in C_k}
8. call the *MFCS-gen* algorithm if $S_k \neq \emptyset$
9. call the *join* procedure to generate C_{k+1}
10. if any frequent itemset in C_k is removed in line 6
11. call *recovery* procedure to recover candidates to C_{k+1}
12. call new *prune* procedure to prune candidates in C_{k+1}
13. $k := k + 1$
14. **end-while**
15. **return** MFS

Figure 3.8: The Pincer-Search algorithm

will not participate in the subsequent support counting and candidate set generation steps. Line 6 will exclude those itemsets that are subsets of any itemset in the current MFS, which contains the frequent itemsets found in the MFCS. If some itemsets in L_k are removed, the algorithm will call the *recovery* procedure to recover missing candidates (line 11).

Theorem 1 *The Pincer-Search algorithm generates all maximal frequent itemsets.*

Proof: Lemma 1 showed that our candidate generation algorithm will generate candidate set correctly. The Pincer-Search algorithm will explicitly or implicitly discover all frequent itemsets. The frequent itemsets are *explicitly* discovered when

they are discovered by the bottom-up search (i.e., they were in the L_k set at some point). The frequent itemsets are *implicitly* discovered when the top-down search discovers their frequent supersets (which are maximal) earlier than the bottom-up search reaches them. Furthermore, only the maximal frequent itemsets will be added to the MFS in Line 5. Therefore, the Pincer-Search algorithm generates all maximal frequent itemsets. □

3.6 The Adaptive Pincer-Search Algorithm

3.6.1 Delay the Use of the MFCS

In general, one may not want to use the "pure" version of the Pincer-Search algorithm. For instance, in some cases, there may be too many infrequent 2-itemsets. In such cases, it may be too costly to maintain the MFCS. The algorithm we have implemented is in fact an adaptive version of the algorithm. This adaptive version does not maintain the MFCS, when doing so would be counterproductive. It delays the maintenance of the MFCS until a later pass when the expected cost of calculating the MFCS is acceptable. This is also the algorithm whose performance is being evaluated in Chapter 4. Thus the very small overhead of deciding when to use the MFCS is accounted in the performance evaluation of our adaptive Pincer-Search algorithm.

Another adaptive approach is to generate all candidates as the Apriori algo-

rithm, but not to count the support of the candidates that are subsets of any itemset in the current MFS. This approach simplifies the pure Pincer-Search algorithm in such a way that it need not do the candidate recovery process mentioned in Section 3.4. A flag, indicating whether a candidate should be counted or not, can be easily maintained. Based on the way of doing the recovery process, the cost of the recovery process can be estimated by the number of the current frequent set and the number of the current MFS. When the estimated cost exceeds some threshold, we can switch from the recovery procedure to the candidate generation procedure that generates all candidates. This way, we can still omit the support counting phase, which is the most time-consuming process.

3.6.2 Relaxing the MFCS

Recall that the MFCS is defined as a minimum cardinality set of itemsets such that the union of all the subsets of its elements contains all the frequent itemsets but does not contain any infrequent itemsets. Because of the "minimum cardinality" in the definition, the procedure to maintain the MFCS requires the dropping of those itemsets that are subsets of some other itemsets in the MFCS. If we remove this requirement, we could have more itemsets in the MFCS in the earlier passes, but in return, we might be able to approach the MFS in a more aggressive way.

Consider the same example as in Fig. 3.4. We did not add itemset {2,3,4,5}

into the MFCS, since it is a subset of the itemset {1,2,3,4,5}, which is already in the MFCS. If we relax the definition of the MFCS and allow itemset {2,3,4,5} to be an element in the Relax-MFCS, then we could have the following three cases.

Case 1: If itemset {1,2,3,4,5} is infrequent and itemset {2,3,4,5} is frequent, then we can use {1,2,3,4} to eliminate bottom-up candidates in this pass.

Case 2: If both itemsets are infrequent, then we can use infrequent itemset {2,3,4,5} to update itemset {1,2,3,4,5}. This helps the MFCS moving down faster.

Case 3: If itemset {1,2,3,4,5} turns out to be frequent and of course itemset {2,3,4,5} is frequent, then the support counting for itemset {2,3,4,5} seems to be wasted when comparing with the pure Pincer-Search algorithm.

The full effect of relaxing the MFCS, and when to relax the MFCS requires further study.

3.7 Counting the Supports of All Frequent Itemsets Efficiently

In those cases when we need to know the supports of all frequent itemsets, we can build a *complete hash-tree*, as shown in Fig. 3.9, for the maximum frequent set. This tree can be viewed as a special case of the hash-tree [AS94] in the Apriori algorithm with the leaf nodes containing only one itemset. Unlike their hash-tree which stores only the itemsets with same length, this complete hash-tree store all the subsets of the maximal frequent itemsets (of different length).

The support counting process can be done as the Apriori algorithm with only a minor change. We add a field for storing the supports of the internal nodes. The support in every node is incremented whenever the node is visited in the forward direction (Apriori only increments the leaf nodes.) To increment the supports of candidates supported by a transaction, all combinations of the items in the transaction are considered.

Let us consider an example to demonstrate how the support counting process works. Suppose the maximum frequent set is {{1,2,3}, {2,3,4}}. The complete hash-tree is shown in Fig. 3.9. The database contains three transactions: {1,2,3,5}, {1,2,3,4}, and {2,3,4}. All combinations of the items in each transaction are enumerated in an increasing (left justified) lexicographical order and the tree is visited in that order. If an item in the transaction is not in the tree, then stop and try the

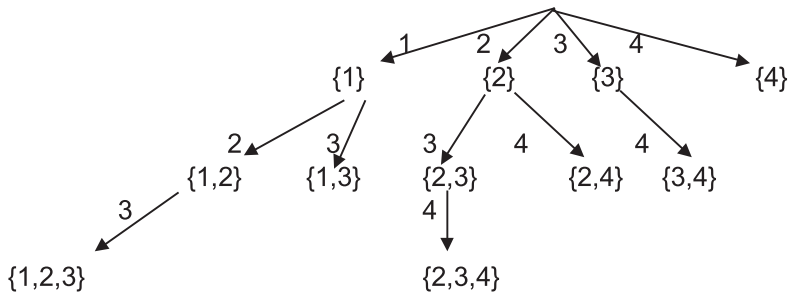

Figure 3.9: Complete Hash-Tree for Final Support Counting

next enumerated sequence. In other words, the tree is traversed in a depth-first fashion. The support is incremented by one when a node is visited in the forward direction (i.e., it is not incremented during backtracking).

For instance, the (left justified) lexicographical enumeration for transaction {1,2,3,5} is 1, 12, 123, 1235, 125, 13, 135, 2, 23, 235, 25, 3, and 35. This means the traversal of the tree is node {1}, node {2}, and node {3}. There is no branch for 5 from node {3}, so backtrack to node {2}. There is no branch for 5 from node {2} either, so backtrack to node {1}. Then try branch 3, backtrack to root, and then visit node {2}, and so on. Similarly, the enumeration for transaction {1,2,3,4} is 1, 12, 123, 1234, 124, 13, 134, 2, 23, 234, 24, 3, 34, and 4. The enumeration for transaction {2,3,4} is 2, 23, 234, 24, 3, 34, and 4.

The supports of all frequent itemsets can be counted efficiently this way by reading the database only once.

Chapter 4

Performance Evaluation

For the Pincer-Search algorithm to be effective, the top-down search needs to reach the maximal frequent itemsets faster than the bottom-up search. A reasonable question can be informally stated: "Can the search in the top-down direction proceed fast enough to reach a maximal frequent itemset faster than the search in the bottom-up direction?" There can be no categorical answer, as this really depends on the distribution of the frequent and infrequent itemsets. However, according to both [AS94] and our experiments, a large fraction of the 2-itemsets will usually be infrequent. These infrequent itemsets will cause the MFCS to go down the levels very fast, allowing it to reach some maximal frequent itemsets after only a few passes. Indeed, in our experiments, we have found that, in most cases, many of the maximal frequent itemsets are found in the MFCS in very early

passes. For instance, in the experiment on database T20.I15.D100K (Fig. 4.6), all maximal frequent itemsets containing up to 17 items are found in 3 passes only!

The performance evaluation presented compares our adaptive Pincer-Search algorithm to the Apriori algorithm. We restrict this performance comparison because it is sufficiently instructive to understand the characteristics of the new algorithm's performance.

4.1 Preliminary Discussion

4.1.1 Auxiliary Data Structures Used

Since we are interested in studying the effect of using the MFCS to reduce the number of candidates and the number of passes, we didn't use more efficient data structures, such as hash tables (e.g., [AS94] [PCY95]), to store the itemsets. We simply used a link-list data structure to store the frequent set and the candidate set in each pass. The databases used in performance evaluation, are the synthetic databases used in [AS94], the census databases similar to [BMUT97], and the stock transaction databases from New York Stock Exchange, Inc. [NYSE97].

Also, as done in [ORS98] and [SA95a], we used a one-dimensional array and a two-dimensional array to speed up the process of the first and the second pass correspondingly. The support counting phase runs very fast by using an array, since no searching is needed. No candidate generation process for 2-itemsets is needed

because we use a two-dimensional array to store the support of all combinations of those frequent 1-itemsets. We start using the link-list data structure after the third pass. For a fair comparison, in all the cases, the number of candidates shown in the figures does not include the candidates in the first two passes. The number of the candidates in the Pincer-Search algorithm includes the candidates in the MFCS.

4.1.2 Scattered and Concentrated Distributions

We first concentrated on the synthetic databases, since they allow us experimenting different distributions of the frequent itemsets. For the same number of frequent itemsets, their distribution can be *concentrated* or *scattered*. In concentrated distribution, the frequent itemsets, having the same length, contain many common items: the frequent items tend to cluster. If the frequent itemsets do not have many common elements, the distribution is scattered. By using the synthetic data generation program as in [AS94], we can generate databases with different distributions by adjusting various parameters. We will present experiments to examine the impact of the distribution type on the performance of the two algorithms.

In the first set of experiments, the number of the maximal frequent itemsets $|L|$ is set to 2000, as in [AS94]. The frequent itemsets found in this set of experiments are rather scattered. To produce databases having a concentrated distribution

of the frequent itemsets, we decrease the parameter $|L|$ to 50 in the second set of experiments. The minimum supports are set to higher values so that the execution time will not be too long.

4.1.3 Non-Monotone Property of the Maximum Frequent Set

For a given database, both the number of candidates and the number of frequent itemsets increase as the minimum support decreases. However, this is *not* the case for the number of the maximal frequent itemsets. For example, when minimum support is 9%, the maximum frequent set may be {{1,2}, {1,3}, {2,3}}. When the minimum support decreases to 6%, the maximum frequent set could become {{1,2,3}}. The number of the maximal frequent itemsets decreased from three to one.

This "nonmonotonicity" does not help bottom-up breadth-first search algorithms. They will have to discover the entire frequent itemsets before the maximum frequent set is discovered. Therefore, in those algorithms, the time, the number of candidates, and the number of passes will monotonically increase when the minimum support decreases.

However, when the minimum support decreases, the length of some maximal frequent itemsets may increase and our MFCS may reach them faster. Therefore, our algorithm *does have* the potential to benefit from this nonmonotonicity.

4.2 Experiments

The test databases are generated synthetically by an algorithm designed by the *IBM Quest* project [AABMSS96]. The synthetic data generation procedure is described in detail in [AS94], whose parameter settings we follow. The number of items N is set to 1000. $|D|$ is the number of transactions. $|T|$ is the average size of transactions. $|I|$ is the average size of maximal frequent itemsets. Thus, e.g., T10.I4.D100K specifies that the average size of transactions is ten, the average size of maximal frequent itemsets is four, and the database contains one hundred thousand transactions.

4.2.1 Scattered Distributions

The results of the first set of experiments are shown in Fig. 4.1, Fig. 4.2, and Fig. 4.3. For the experiments on database T10.I4.D100K, see Fig. 4.2, the best improvement occurs when minimum support is 0.5%. In this experiment, Pincer-Search ran about 1.7 times faster than the Apriori algorithm. The improvement came from reducing the number passes of reading the database and the number of candidates.

In the experiment on database T5.I2.D100K, see Fig. 4.1, Pincer-Search used more candidates than Apriori. That is because of the number of additional candidates used in the MFCS is more than the number of extra candidates pruned

relying on the MFCS. The maximal frequent itemsets, found in the MFCS, are so short that not too many subsets can be pruned. However, the I/O time saved compensated for the extra cost. Therefore, we still get an improvement.

Depending on the distribution of the frequent itemsets, it is also possible that our algorithm might spend time counting the support of the candidates in the MFCS while still not finding any maximal frequent itemsets from the MFCS. For instance, our algorithm took more time than the Apriori algorithm in the case when the minimum support is set to 0.75% and the database is T10.I4.D100K. However, since there were only a few candidates in the MFCS, the difference is quite small.

Fig. 4.3 shows the results of experiments on database T20.I6.D100K. The best improvement in these experiments occured when minimum support is 0.5%. In this case, the Pincer-Search ran about 1.4 times faster than the Apriori algorithm. The reduction of the number of passes and the number of candidates contribute to the improvement.

4.2.2 Concentrated Distributions

In the second set of experiments we study the relative performance of the two algorithms on databases with concentrated distributions. The results are shown in Fig. 4.4, Fig. 4.5, and Fig. 4.6. In the first experiment, see Fig. 4.4, we use

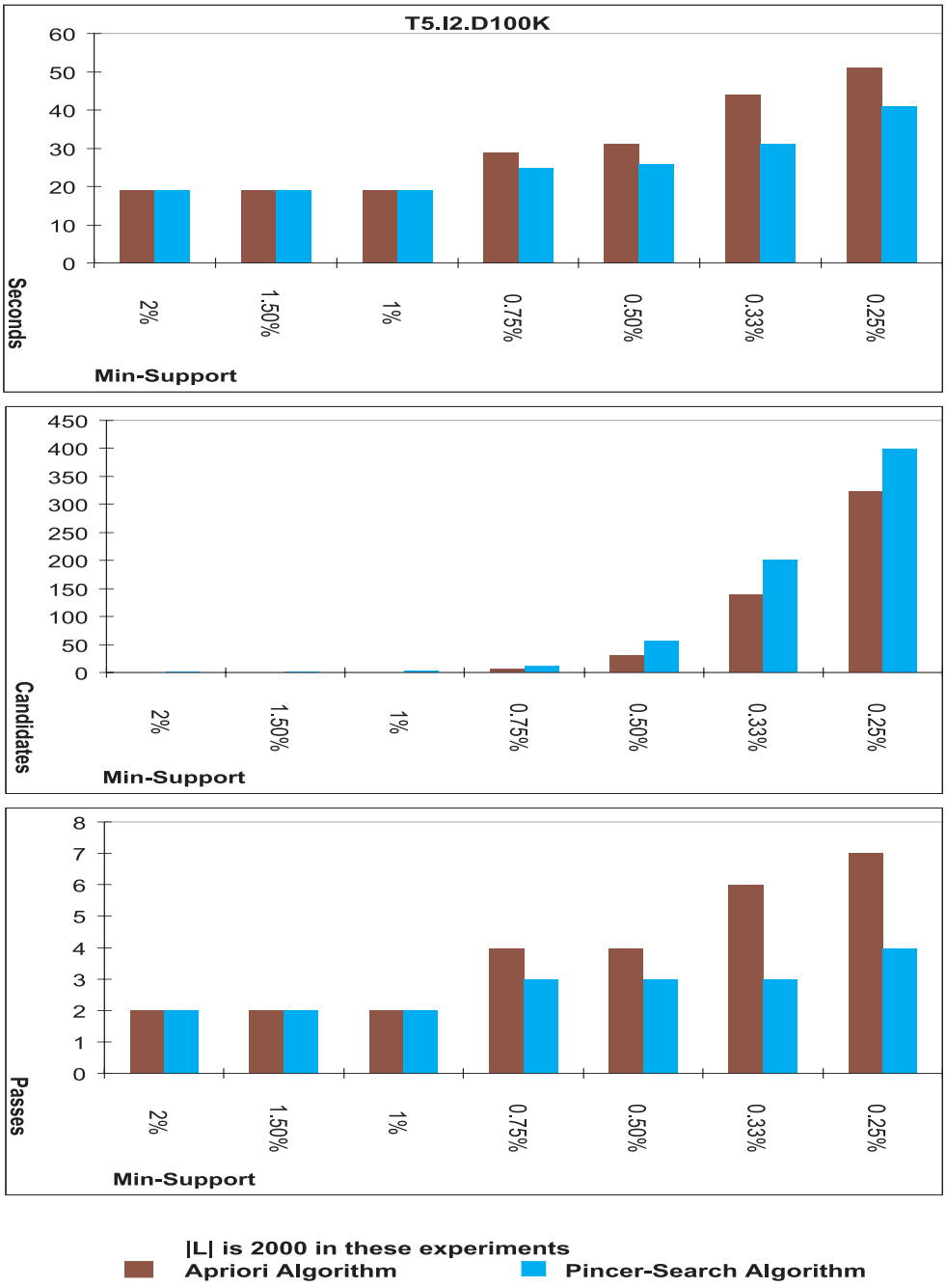

Figure 4.1: Scattered Distribution T5.I2.D100K

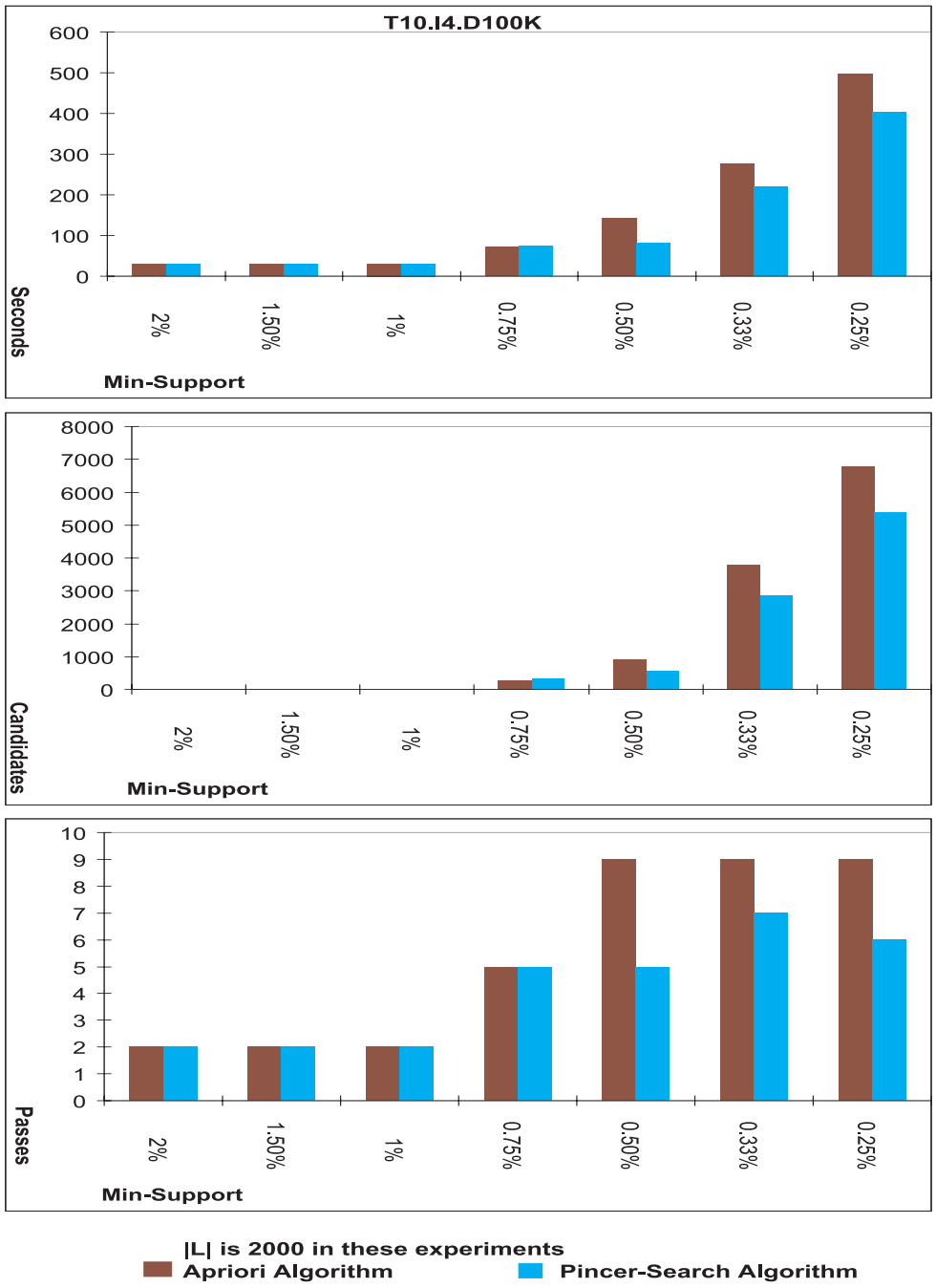

Figure 4.2: Scattered Distribution T10.I4.D100K

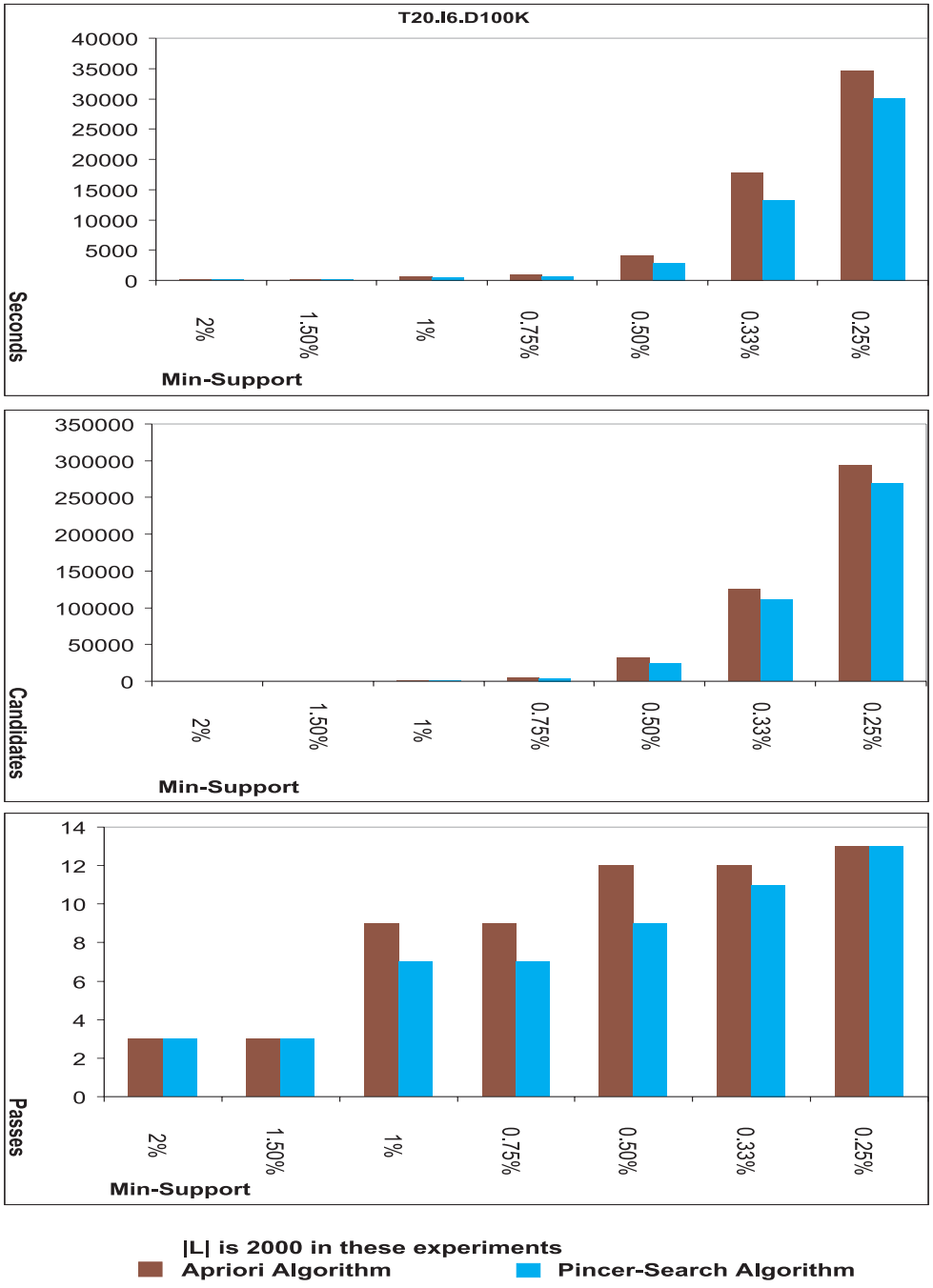

Figure 4.3: Scattered Distribution T20.I6.D100K

the same parameters as the T20.I6.D100K database in the first set of experiments, but the parameter $|L|$ is set to 50. The improvements of Pincer-Search begin to increase. When the minimum support is 18%, our algorithm runs about 2.3 times faster than the Apriori algorithm.

The non-monotone property of the maximum frequent set, considered in Section 4.1.3, impacts on this experiment. When the minimum support is 12%, both Apriori algorithm and Pincer-Search algorithm took eight passes to discover the maximum frequent set. But, when the minimum support decreases to 11%, the maximal frequent itemsets become longer. This forced the Apriori algorithm to take more passes (nine passes) and consider more candidates to discover the maximum frequent set. In contrast, the MFCS allowed our algorithm to reach the maximal frequent itemsets faster. Pincer-Search took only four passes and considered fewer candidates to discover all maximal frequent itemsets.

We further increased the average size of the frequent itemsets in the next two experiments. The average size of the maximal frequent itemsets was increased to 10 in the second experiment and database T20.I10.D100K was used. The results are shown in Fig. 4.5. The best case, in this experiment, is when the minimum support is 6%. Pincer-Search ran approximately 23 times faster than the Apriori algorithm. This improvement mainly came from the early discovery of maximal frequent itemsets which contain up to 16 items. Their subsets were not generated

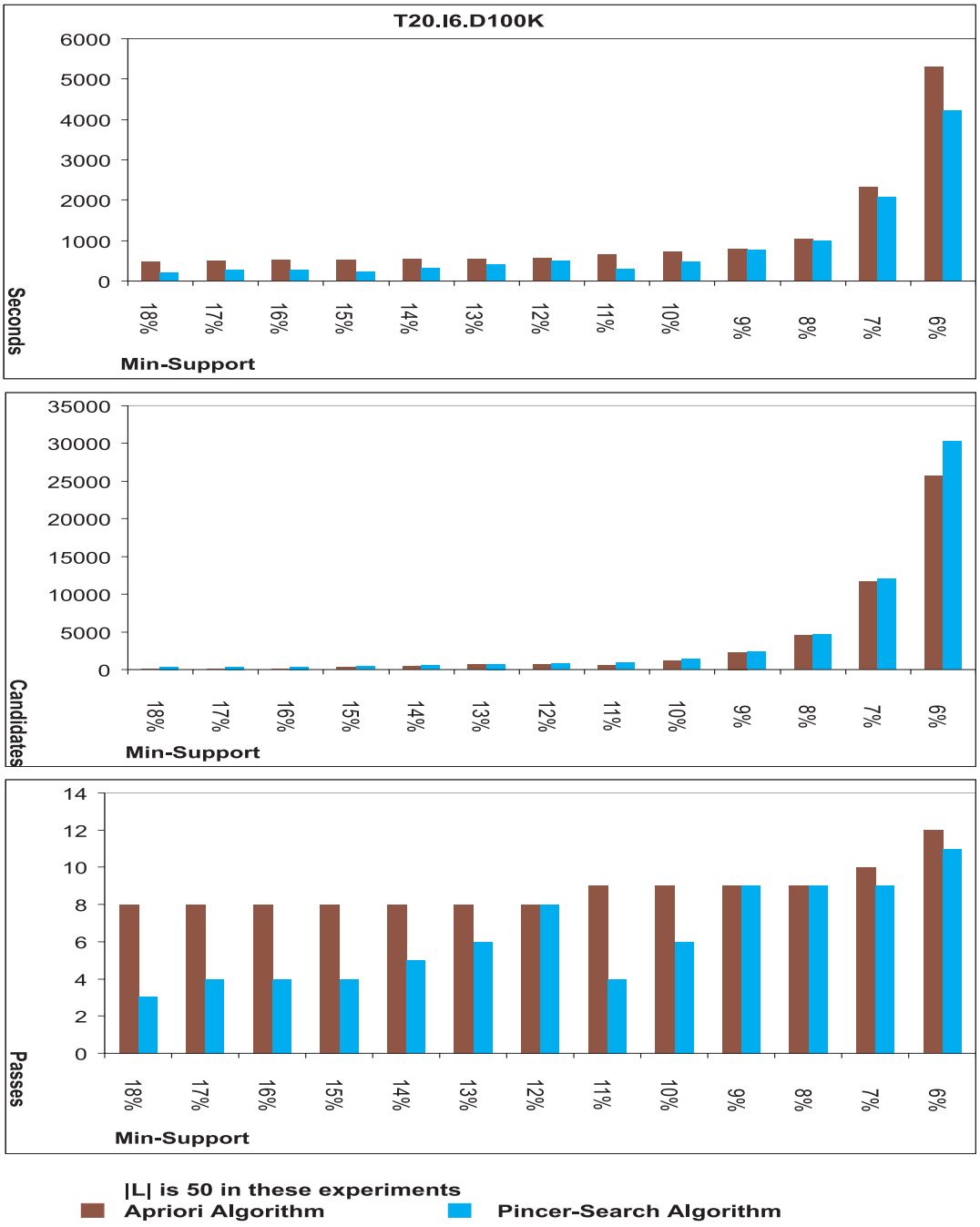

Figure 4.4: Concentrated Distribution T20.I6.D100K

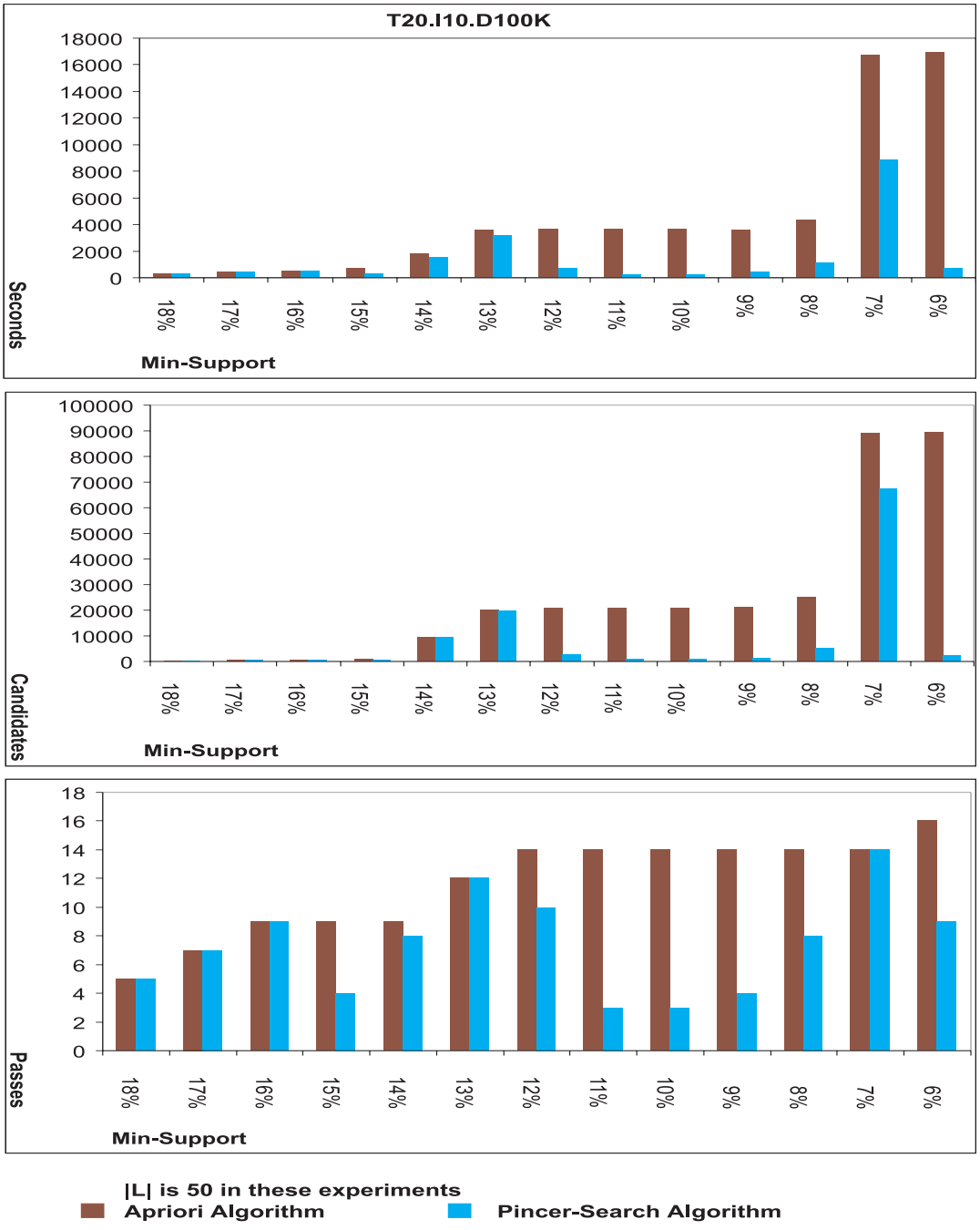

Figure 4.5: Concentrated Distribution T20.I10.D100K

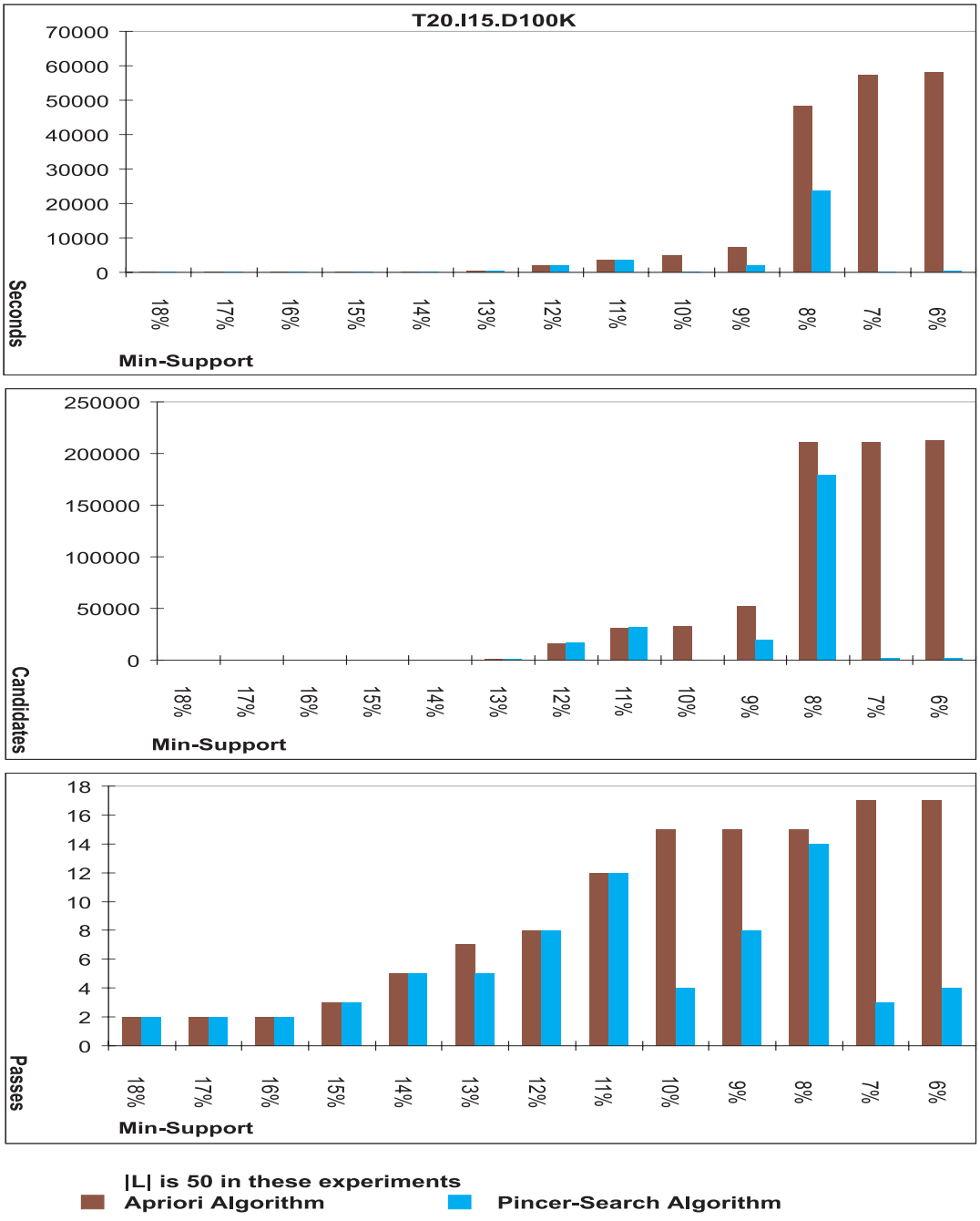

Figure 4.6: Concentrated Distribution T20.I15.D100K

and counted in our algorithm. As shown in this experiment, the reduction of the number of candidates can significantly decrease both I/O time and CPU time.

The last experiment ran on database T20.I15.D100K. As shown in Fig. 4.6, Pincer-Search took as few as three passes to discover all maximal frequent itemsets which contain as many as 17 items. This experiment shows improvements of more than two orders of magnitude when the minimum supports are 6% and 7%. One can expect even greater improvements when the average size of the maximal frequent itemsets is further increased.

4.2.3 Using a Hash-Tree

Our first implementation of the Pincer-Search algorithm did not use a hash-tree for the support counting phase. On Sridhar Ramaswamy's suggestion, we incorporated a hash-tree into our implementation. Using a hash-tree did improve the overall performance a lot. Both Apriori algorithm and Pincer-Search algorithm benefit from using it. The speed up is about four to eight times. In addition to using the hash-tree, the databases were converted from a text format into a binary format. This also improves the I/O time for about 20 to 30%. Fig. 4.7–Fig. 4.12 show the results of the same experiments as previous sections, but with the help of the hash-tree.

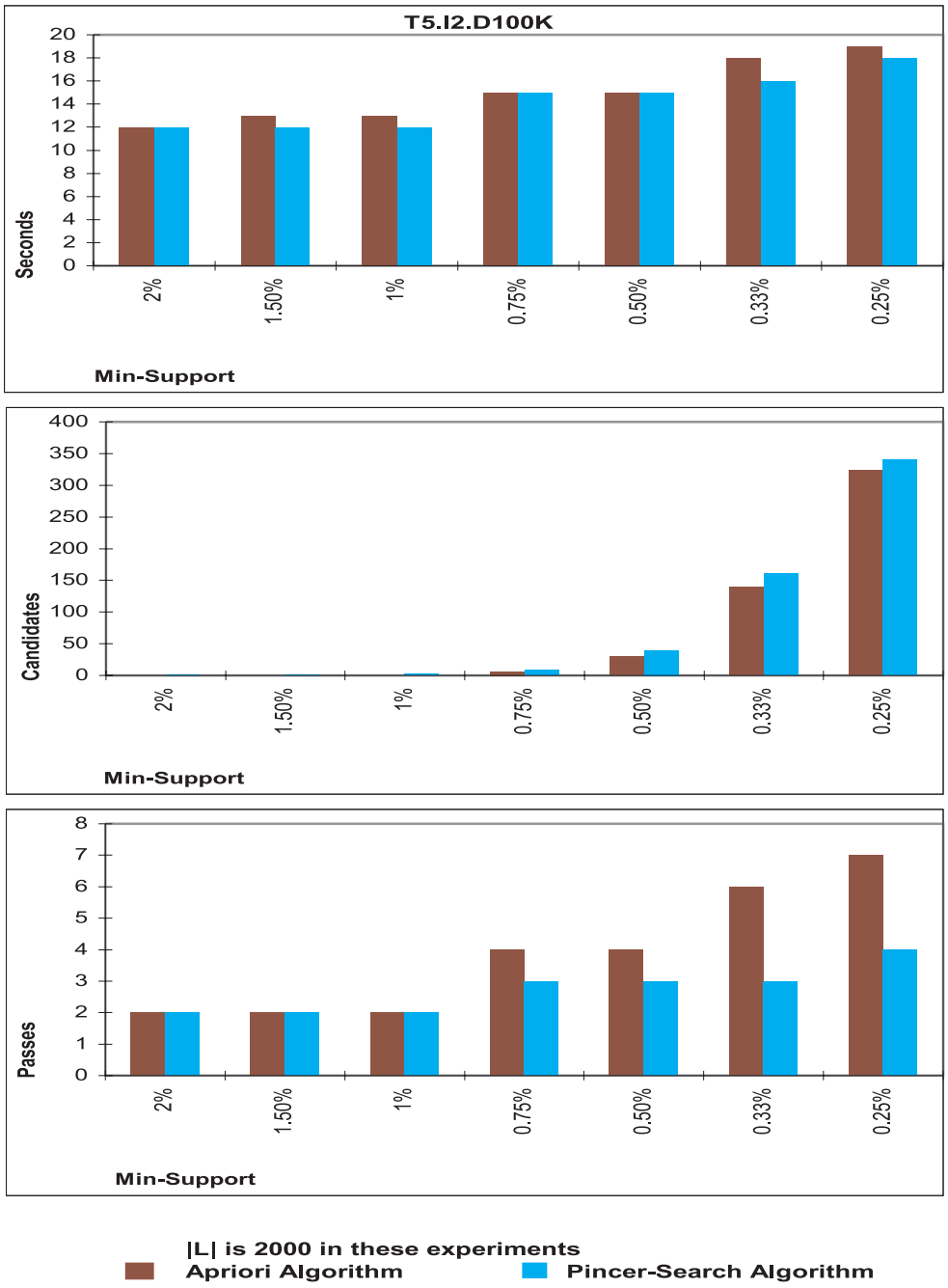

Figure 4.7: Scattered Distribution T5.I2.D100K (Using a Hash-Tree)

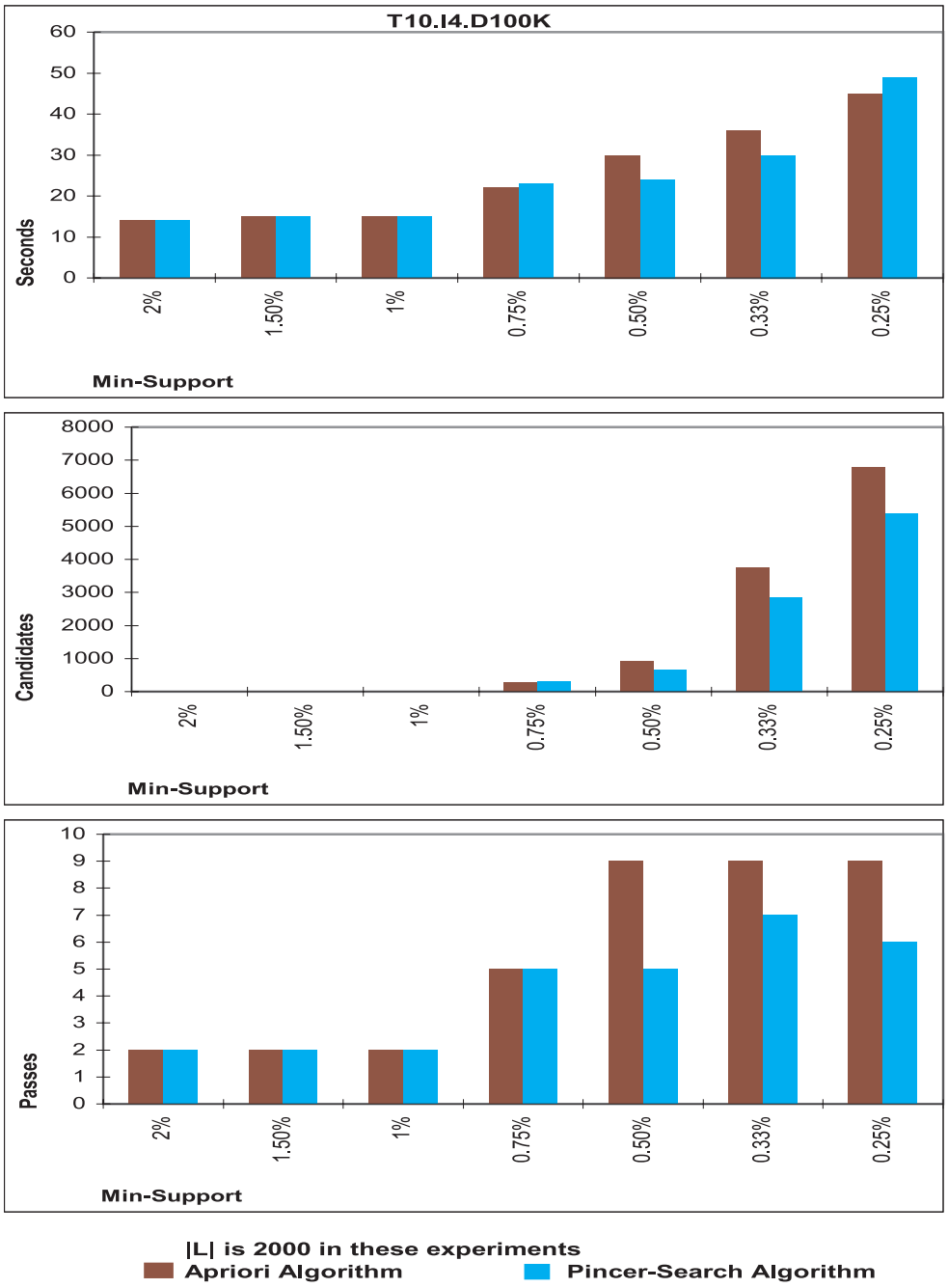

Figure 4.8: Scattered Distribution T10.I4.D100K (Using a Hash-Tree)

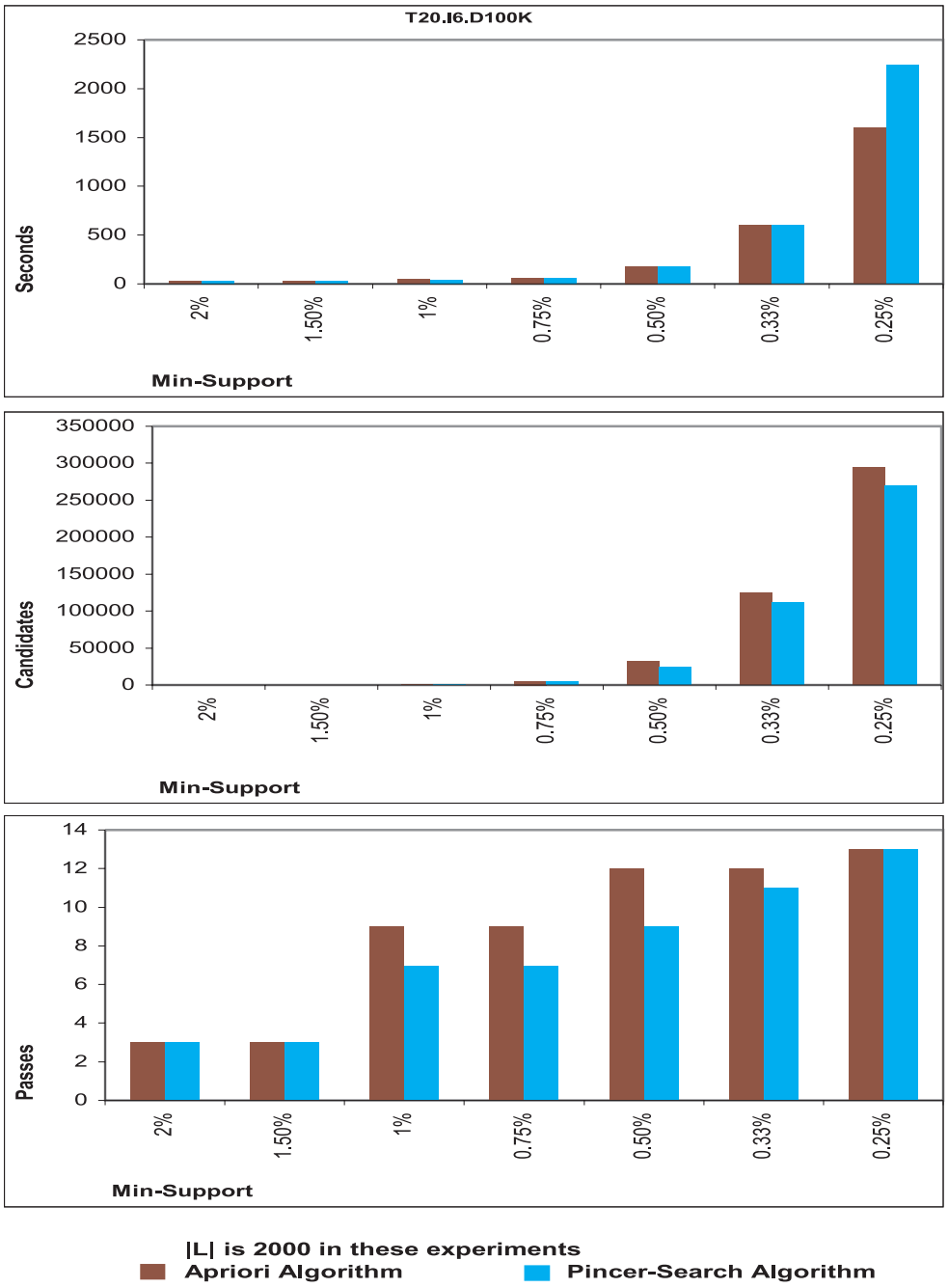

Figure 4.9: Scattered Distribution T20.I6.D100K (Using a Hash-Tree)

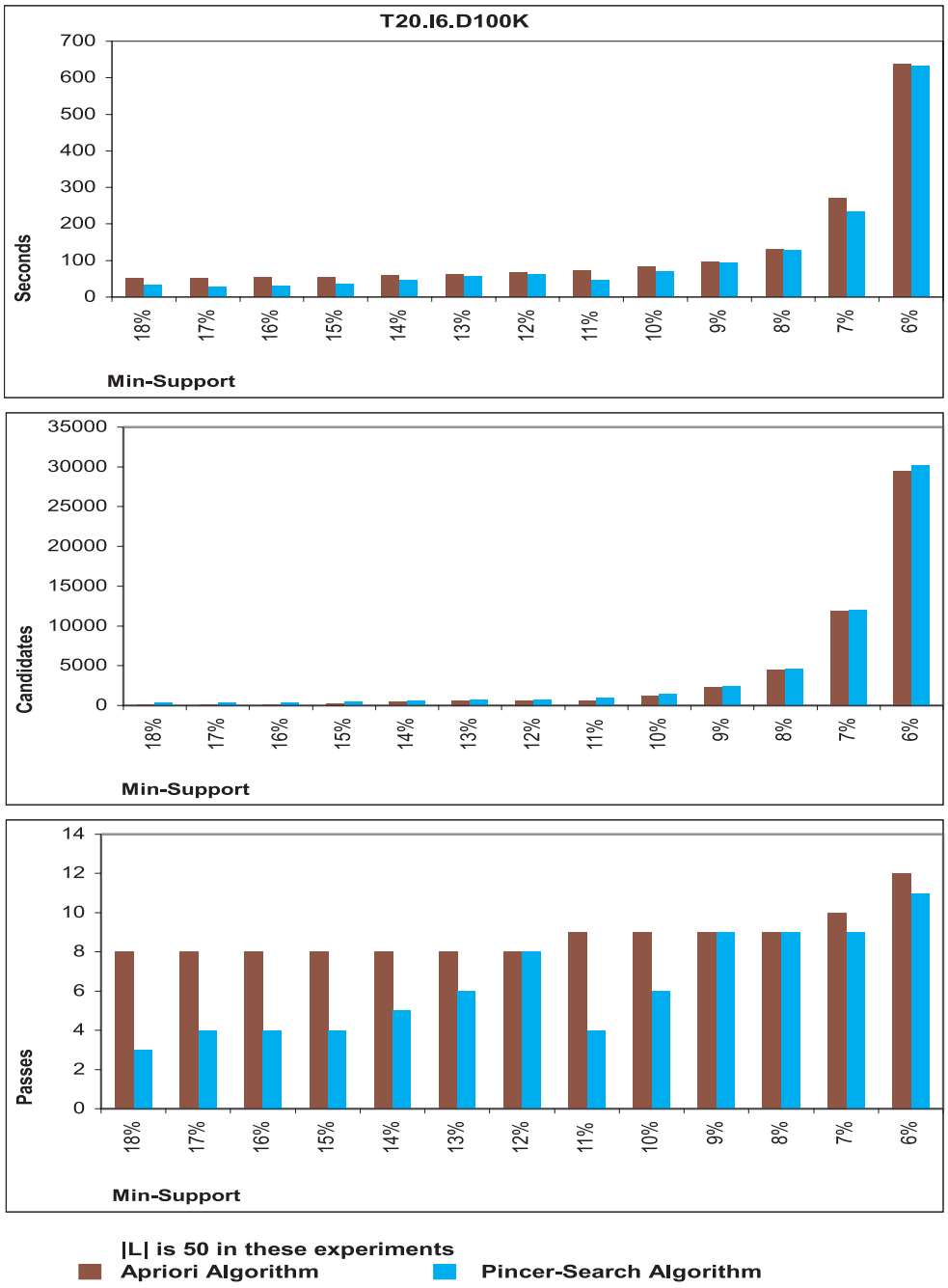

Figure 4.10: Concentrated Distribution T20.I6.D100K (Using a Hash-Tree)

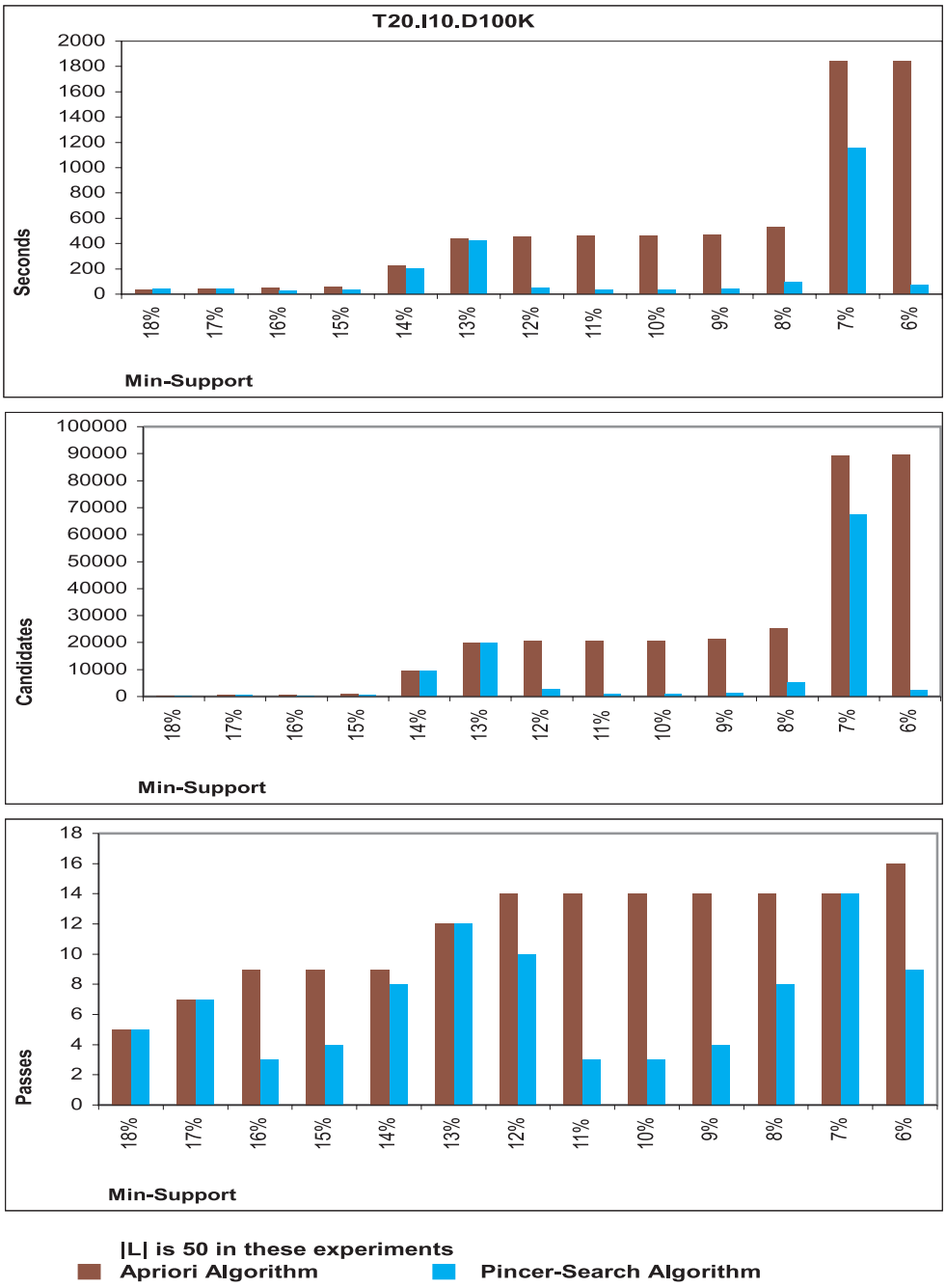

Figure 4.11: Concentrated Distribution T20.I10.D100K (Using a Hash-Tree)

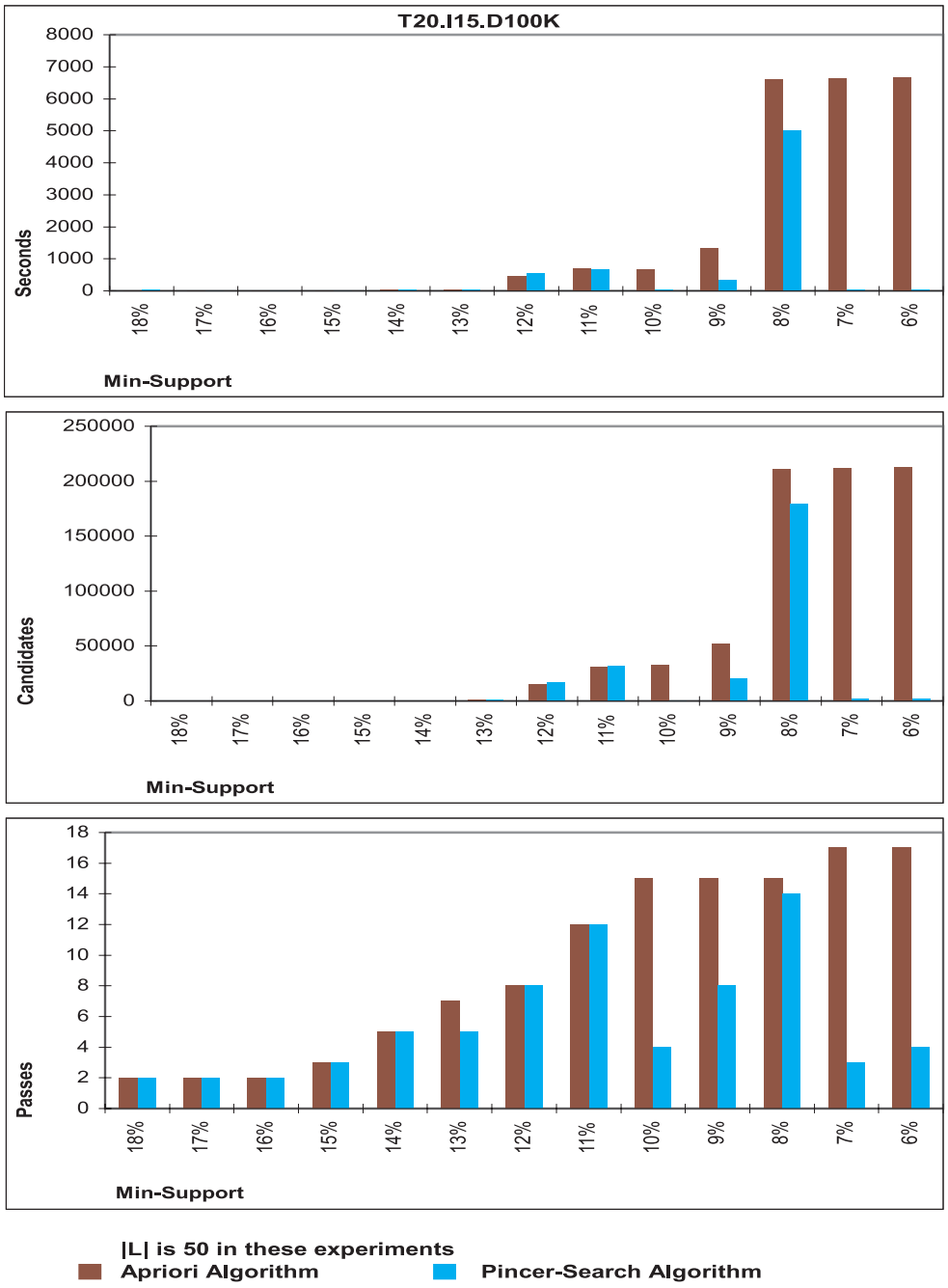

Figure 4.12: Concentrated Distribution T20.I15.D100K (Using a Hash-Tree)

4.2.4 Census Databases

A PUMS file, which contains public use microdata samples, was provided to us by Roberto Bayardo from IBM. This file contains actual census entries which constitute a five percent sample of a state that the file represents. This database is similar to the database looked at by Brin *et al.* [BMUT97]. As discussed in that paper, this PUMS database is very hard. That is because there are many more items than in the previous synthetic supermarket databases: there are about 7000 items in this PUMS database, in comparison with 1000 items in the synthetic databases. Furthermore, a number of items in the census database appear in a large fraction of the database and therefore very many frequent itemsets will be discovered. From the distribution point of view, this means that some maximal frequent itemsets are quite long and the distribution of the frequent itemsets is quite scattered.

In order to compare the performance of the two algorithms within a reasonable time, we used the same approach as they proposed in the paper: we remove all items that have 80% or more support from the database. The experimented results are shown in Fig. 4.13.

In this real-life census database, the Pincer-Search algorithm also performs well. In all cases, the Pincer-Search algorithm used less time, few candidates, and fewer passes of reading the database. The overall improvement in performance ranges

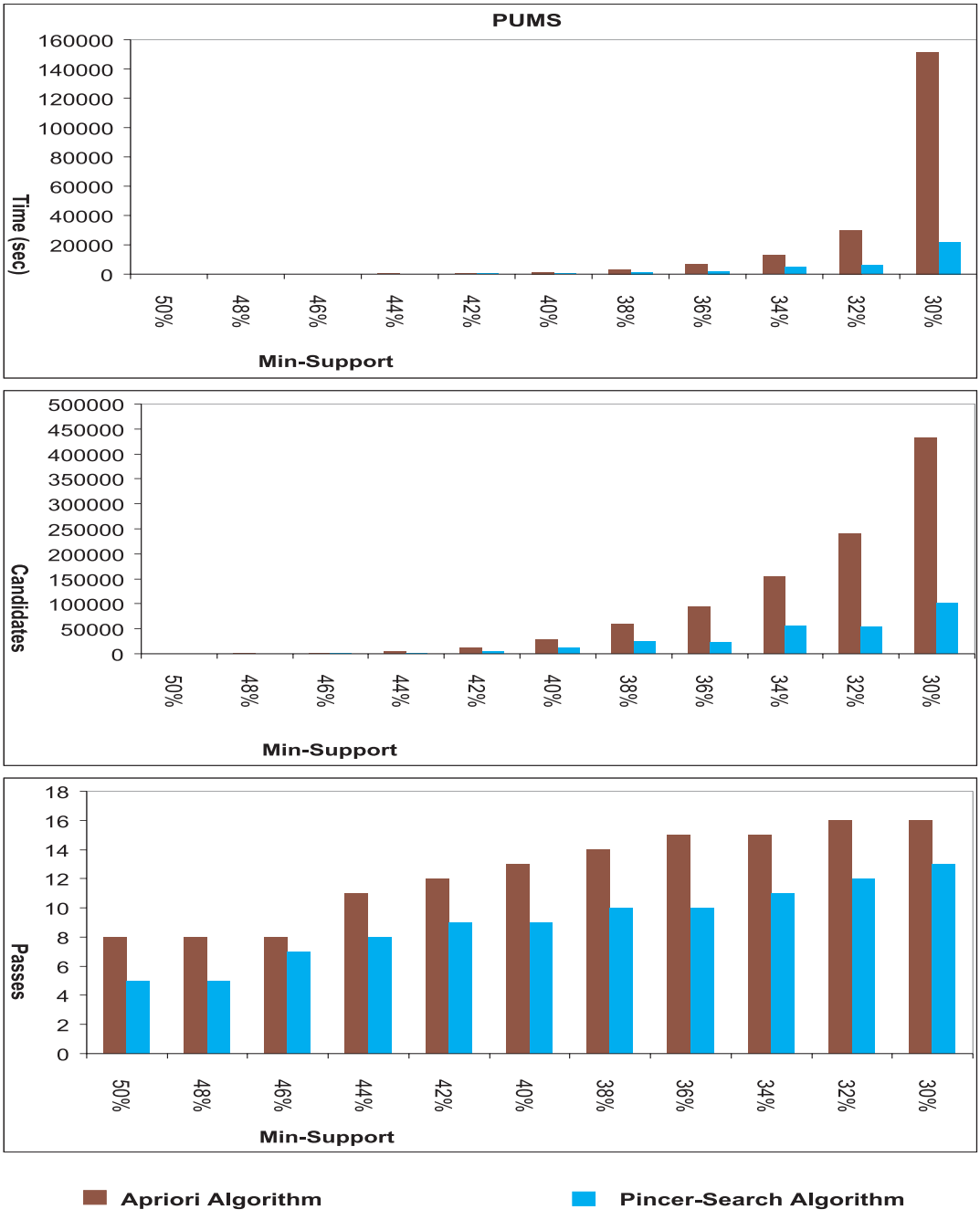

Figure 4.13: Census Database

from 10% to 650%.

4.2.5 Stock Market Databases

We also conducted some experiments on the stock market databases. We used the trading data of June 1997, collected by the New York Stock Exchange, for the experiments. The trading data contains the symbol, price, volume, and time for every trade. Around 3000 common stocks were chosen for these experiments. We are interested in discovering the price changing patterns in the database. A simplified problem can be stated as the followings:

"What are those stocks that their trading prices go up and down together 2/3 of the time in one day?"

Whether the price of a stock goes up or down is determined by looking at the first and the last trading price of the stock within an interval of time. If the last price is higher than the first price, then the price is up. If there is no trade for a stock during this period of time, then assume that the price is unchanged. Otherwise, the price is down. Collect all those stocks that their prices go up during this period of time and treat them as a transaction in our association rule mining problem. Do the same for those stocks that their prices go down during this period of time and form another transaction. Now, we can run the frequent set discovery algorithm to discover the price changing patterns.

We ran experiments on every trading day of June 1997. Here, we only show the experiments on those days that reflect extreme cases when either Pincer-Search or Apriori performed best. The experiments on June 3, June 20, and June 23 are shown in Fig. 4.14, Fig. 4.15, and Fig. 4.16 respectively. The minimum supports range from 40% to 73%. We cannot choose a fixed range of minimum support because of the data in each day are quite different from each other. The interval was set to 60 minutes.

For the experiments on June 3 data and June 23 data, Pincer-Search performed much better than the Apriori algorithm. For the experiments on June 20 data, even though the Pincer-Search algorithm used fewer passes and fewer candidates than the Apriori algorithm, the overhead of maintaining the MFCS costs too much. The Apriori algorithm performed better in this case. It is possible to adjust some of the parameters in the adaptive approach to reduce the differences. We would like to study in the future a good way of dynamically adjusting the parameters.

Because of the number of records in the database is 15, which is very small, we have only a few available values on setting the minimum support. As we can see from these figures, there is a big jump in execution time when we decrease the minimum support down to some point. For instance, see the experiment of June 3, when the minimum support is decreased from 53% (8/15) to 46% (7/15), the execution time increases significantly. When we decrease the minimum support

further, neither algorithm can complete the execution due to the limited size of the main memory. However, we suspect that Pincer-Search will do better when the minimum support is decreased (maximal frequent itemsets will be longer). In many cases, Pincer-Search did find many very long maximal frequent itemsets in very early passes. However, since the MFS is not complete, we cannot draw any conclusion.

In many of these experiments, Pincer-Search did use fewer passes. However, as the database is so small, the property of reducing the passes of reading the database for Pincer-Search algorithm did not contribute to the saving of the execution time. The improvement came purely from reducing the number of candidates.

To increase the number of records, one could reduce the interval (below 60 minutes) or consider all the trades in a month or even a year. However, by combining multiple days' data together, we might lose the opportunities to discover some distinguished patterns that occur only in some of the days. Therefore, we tried to increase the size of the database by reducing the size of the interval. The interval was set to 30 minutes and the results were shown in Fig. 4.17, Fig. 4.18, and Fig. 4.19. Pincer-Search didn't perform better than Apriori in these experiments, because the average transaction length and the average maximal frequent itemset length were short.

In all the other experiments that are not shown here, both algorithms perform

competitively, since the maximal frequent itemsets are all short. The totals of the execution time of all other experiments are 3,451,328 seconds and 3,175,642 seconds for Pincer-Search and Apriori respectively. The totals of the candidates used are 162,258 and 110,358, and the total passes used are 714 and 863 for Pincer-Search and Apriori respectively.

We will study other problems in discovering other interesting price changing patterns.

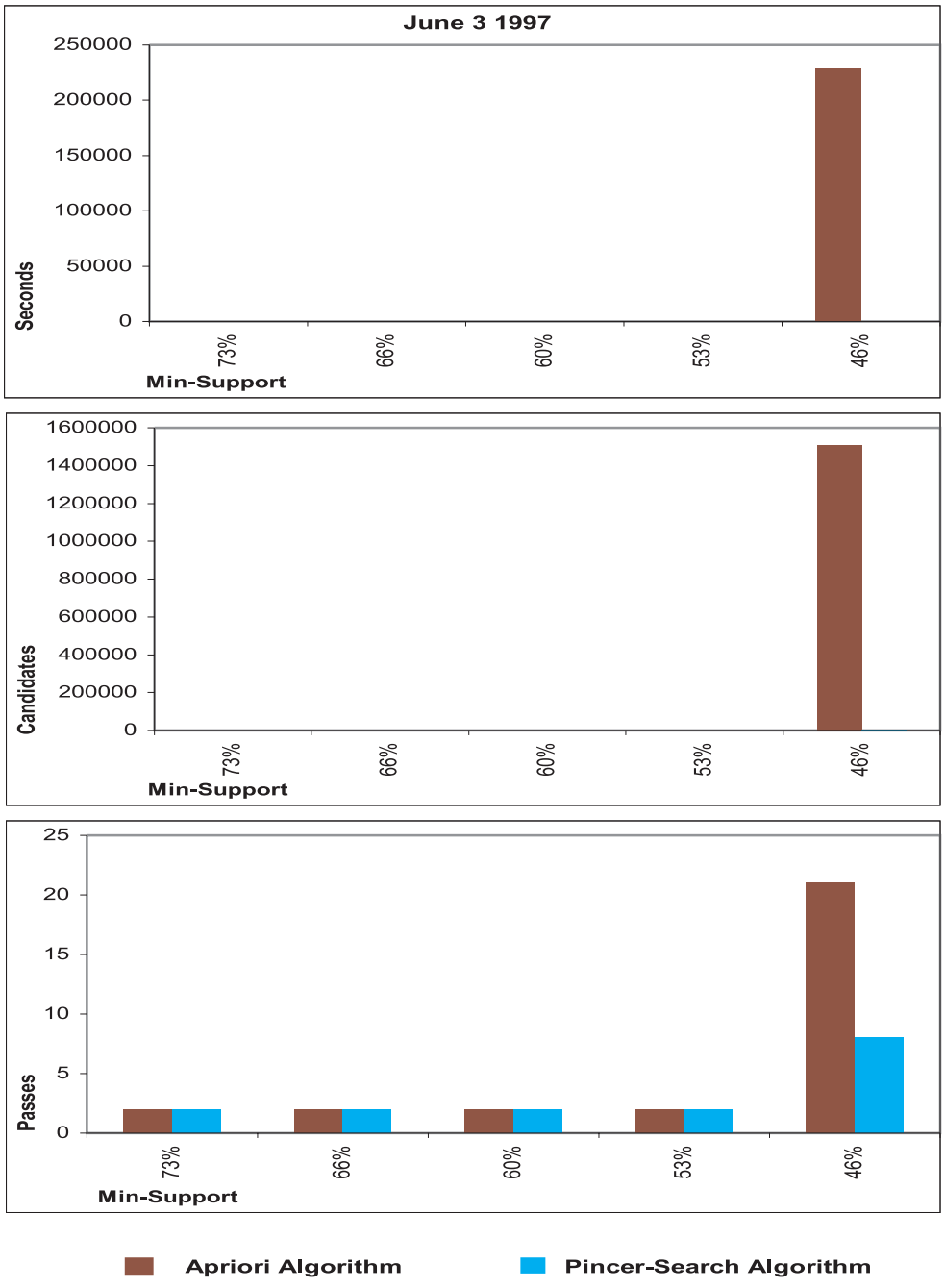

Figure 4.14: NYSE Databases June 3, 1997 (60-minute's interval)

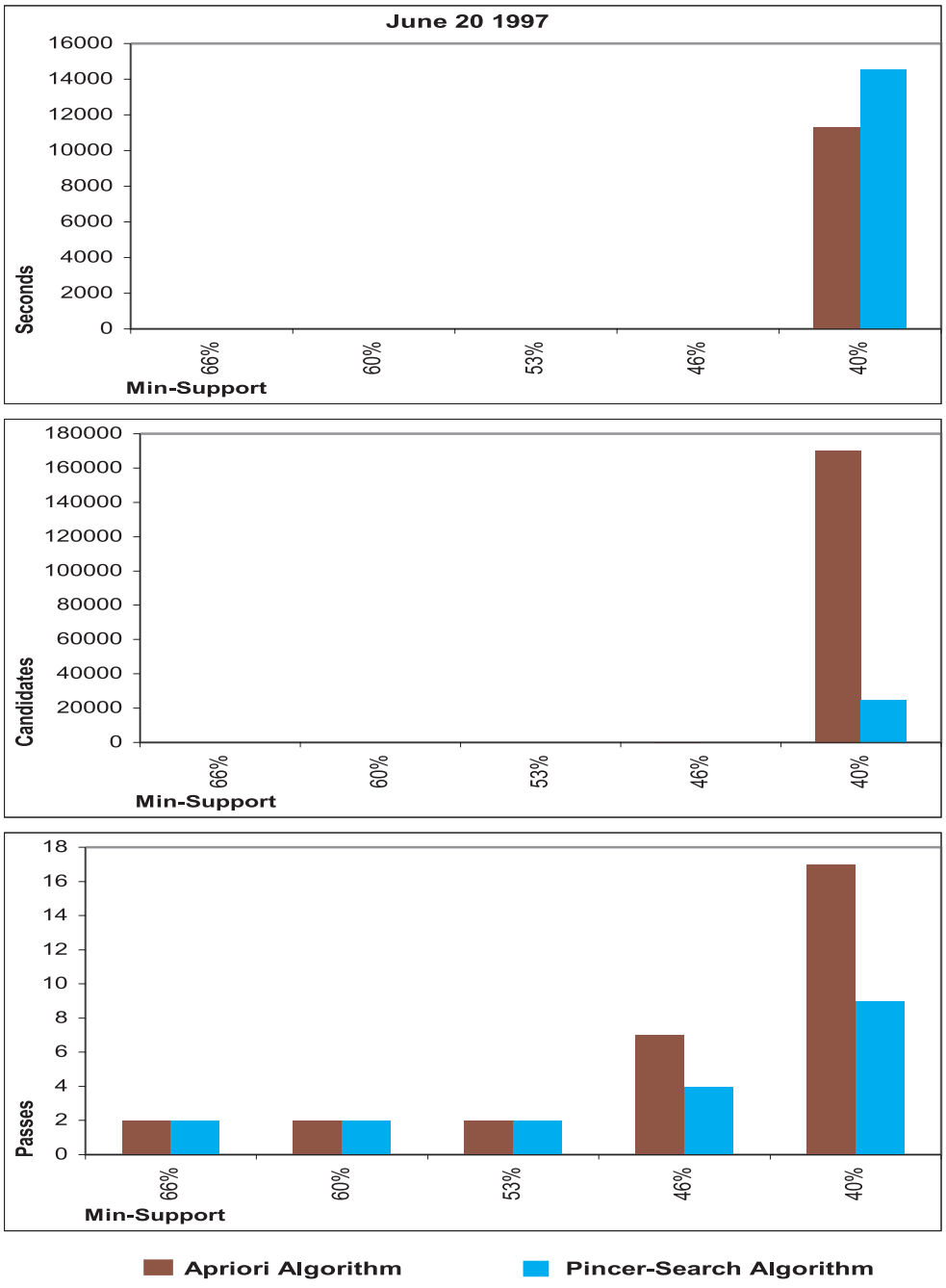

Figure 4.15: NYSE Databases June 20, 1997 (60-minute's interval)

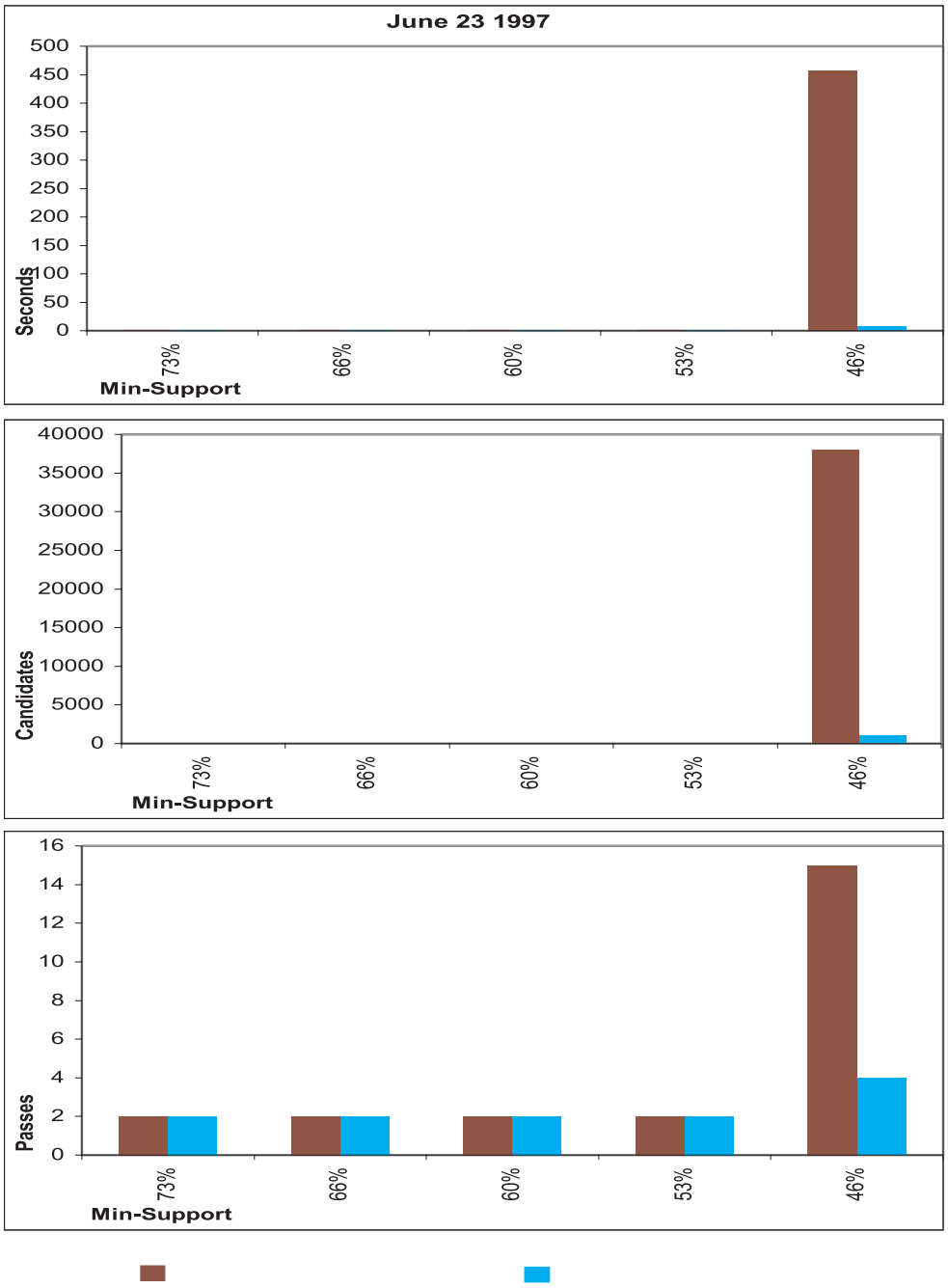

Figure 4.16: NYSE Databases June 23, 1997 (60-minute's interval)

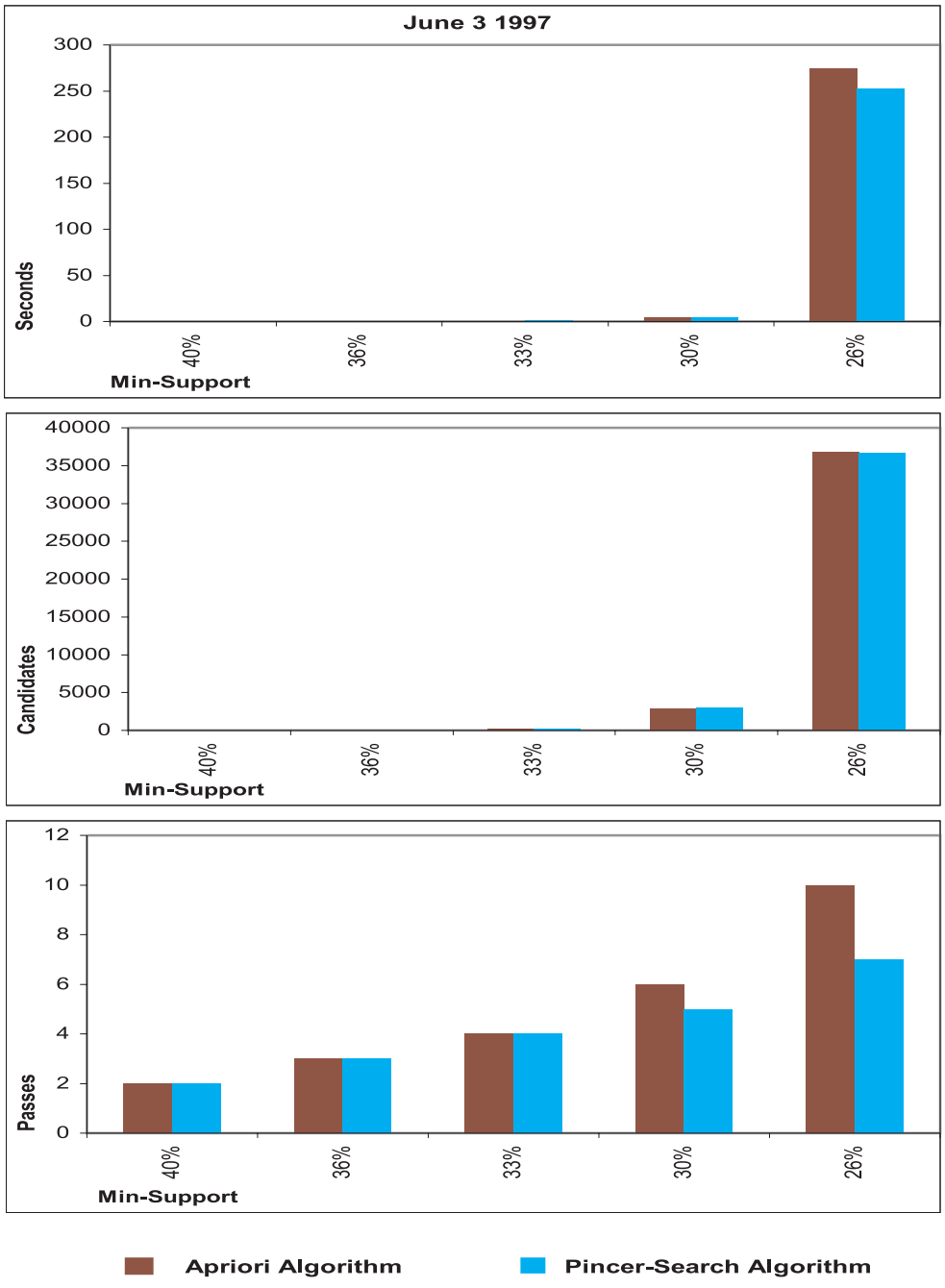

Figure 4.17: NYSE Databases June 3, 1997 (30-minute's interval)

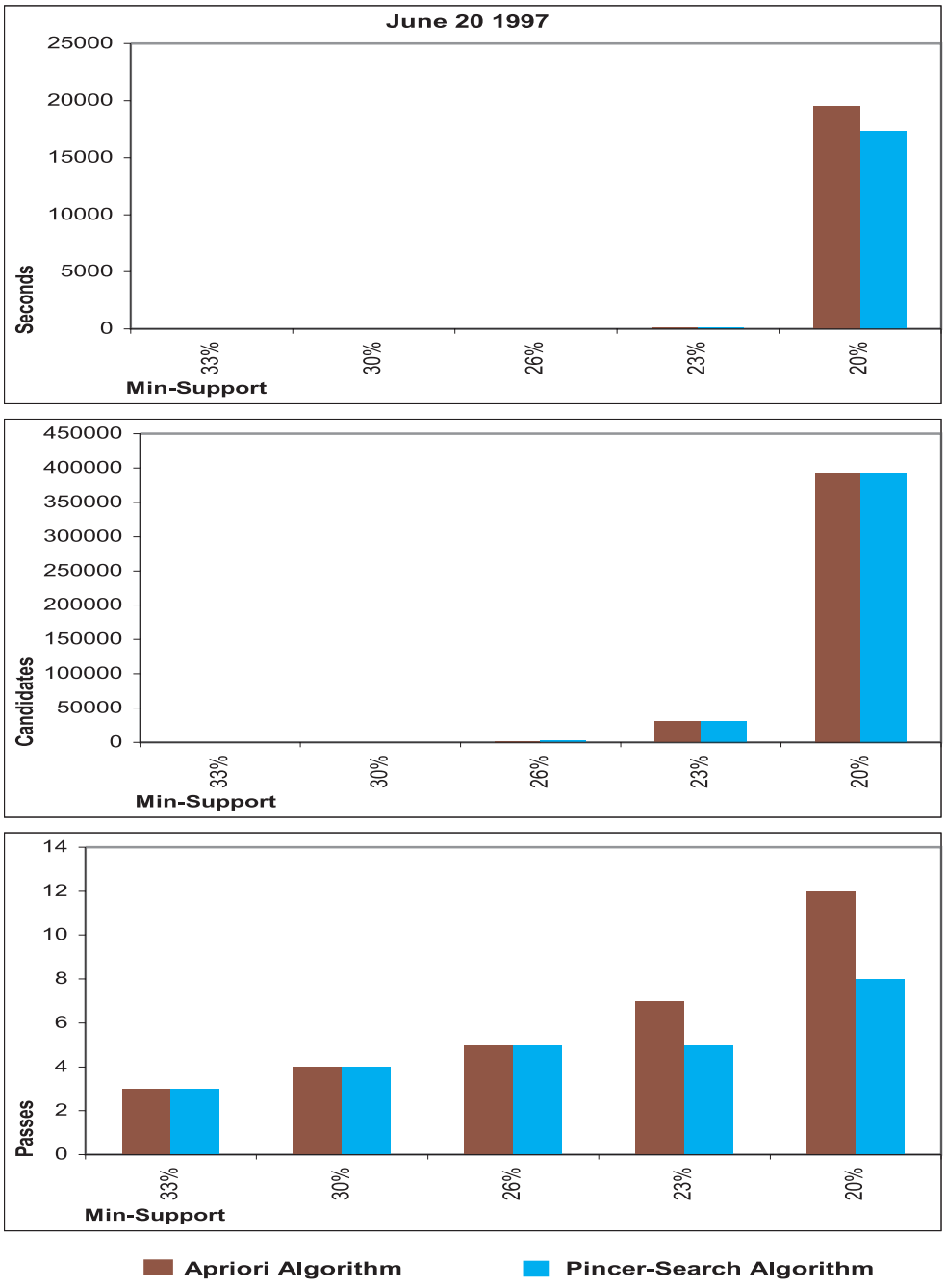

Figure 4.18: NYSE Databases June 20, 1997 (30-minute's interval)

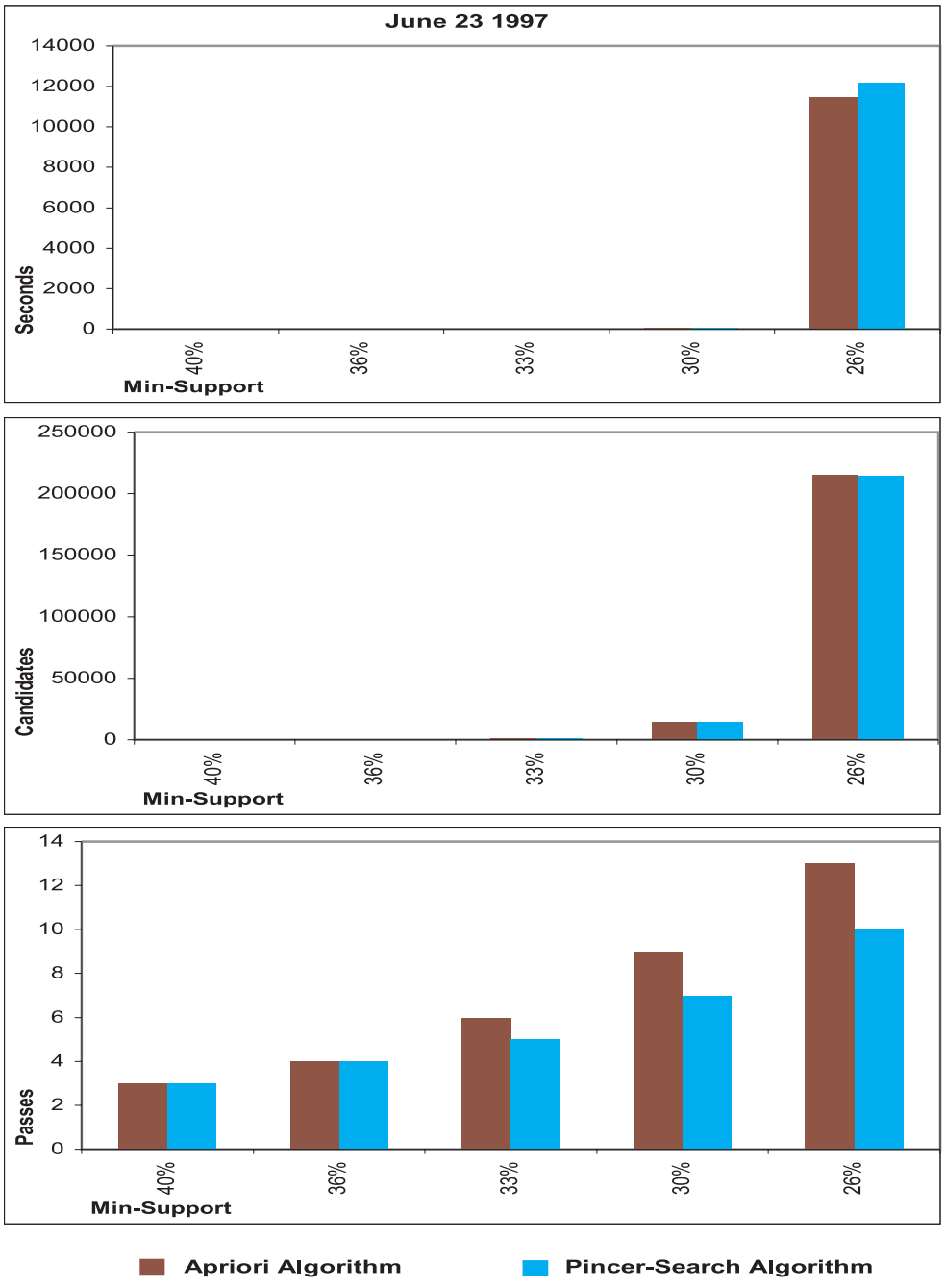

Figure 4.19: NYSE Databases June 23, 1997 (30-minute's interval)

Chapter 5

Concluding Remarks

5.1 Summary

An efficient way to discover the maximum frequent set can be very useful in various data mining problems, such as the discovery of the association rules, theories, strong rules, episodes, and minimal keys. The maximum frequent set provides a unique representation of all the frequent itemsets. In many situations, it suffices to discover the maximum frequent set, and once it is known, all the required frequent subsets can be easily generated.

In this thesis, we presented a novel algorithm that can efficiently discover the maximum frequent set. Our Pincer-Search algorithm could reduce both the number of times the database is read and the number of candidates considered. Experiments show that the improvement of using this approach can be very significant,

especially when some maximal frequent itemsets are long.

A popular assumption is that the maximal frequent itemsets are usually very short and therefore the computation of *all* (and not just maximal) frequent itemsets is feasible. Such assumption on maximal frequent itemsets does not need to be true in important applications. Consider for example the problem of discovering patters in price changes of individual stocks in a stock market. Prices of individual stocks are frequently quite correlated with each other. Therefore, the discovered patterns may contain many items (stocks) and the frequent itemsets are long. We expect our algorithm will be useful in these applications.

5.2 Future Work

The performance of the Pincer-Search algorithm in applications of discovering other price changing patterns in stock markets will be studied. The maximal frequent itemsets in many instances of such applications are likely to be long. Therefore we expect the algorithm to provide dramatic performance improvements.

Many classification problems as discussed in [B97] tend to have long patterns. We will study the performance of the Pincer-Search algorithm on these problems.

In general, if some maximal frequent itemsets are long and the maximal frequent itemsets are distributed scatteredly, then the problem of discovering the MFS can be very hard. In this case, even Pincer-Search might not be able to solve it

efficiently. Parallelizing the Pincer-Search algorithm might be a possible way to solve this hard problem. We propose to divide the candidate set in such a way that all the candidates that are subsets of an itemset in the MFCS are assigned to a same processor.

Although there might be some duplicate calculations, this way of partitioning the candidates can have the benefits that no synchronization or communication among processors is needed. Each processor can run totally independent. The duplicate calculations come from the following scenario: consider the MFCS in the kth pass, if two itemsets in the MFCS have equal or more than k same items, then there will be duplicate calculations if we assign these two itemsets (and their subsets) to two different processors.

We will study the performance of parallel Pincer-Search algorithms in the future. The way to minimize the duplicate calculations and maximize the use of available processors will be an important problem to study.

Bibliography

[AABMSS96] R. Agrawal, A. Arning, T. Bollinger, M. Mehta, J. Shafer, R. Srikant. The Quest Data Mining System. In *Proc. 2nd International Conference on Knowledge Discovery in Databases and Data Mining (KDD)*, Aug. 1996.

[AIS93a] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. SIGMOD*, May 1993.

[AIS93b] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 6, Dec. 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. 20th VLDB*, Sept. 1994.

[AS96] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Trans. on Knowledge and Data Engineering*, Jan. 1996.

[B97] R. J. Bayardo. Brute-force mining of high-confidence classification rules. In *Proc. 3rd International Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 1997.

[BMS97] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: generalizing association rules to correlations. In *Proc. SIGMOD*, May 1997.

[BMUT97] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. SIGMOD*, May 1997.

[FPS96] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. Knowledge discovery and data mining: toward a unifying framework. In *Proc. 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 1996.

[FPSU96] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthrusamy (Eds.). *Advances in Knowledge Discovery and Data Mining*. AAAI Press, Menlo Park, CA, 1996.

[GKMT97] D. Gunopulos, R. Khardon, H. Mannila, and H. Toivonen. Data mining, hypergraph traversal, and machine leaning. In *Proc. PODS*, 1997.

[GMS97] D. Gunopulos, H. Mannila, and S. Saluja. Discovering all most specific sentences by randomized algorithm. In *Proc. International Conference of Database Theory (ICDT)*, Jan. 1997.

[HCC92] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in database: an attribute-oriented approach. In *Proc. 18th VLDB*. Sept. 1992.

[HF95] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 21st VLDB*, Sept. 1995.

[HKK97] E. Han, G. Karypis, and Vipin Kumar. Scalable parallel data mining for association rules. In *Proc. SIGMOD*, May 1997.

[HKMRT96] K. Hätönen, M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen. Knowledge Discovery from Telecommunication Network Alarm Databases. In *Proc. 12th International Conference on Data Engineering (ICDE)*, Feb. 1996.

[HS93] M. Houtsma and A. Swami. Set-oriented mining of association rules. *Research Report RJ 9567*, IBM Almaden Research Center, Oct. 1993.

[KMRTB94] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Berkamo. Finding interesting rules from large sets of discovered association rules. In *Proc. 3rd International Conference on Information and Knowledge Management*, Nov. 1994.

[L96] D. Lin. Mining association rules with longest frequent candidates. Unpublished manuscript. July 1996.

[L97] D. Lin. Mining association rules for long frequent itemsets. Unpublished manuscript. March 1997.

[LK97] D. Lin and Z. Kedem. Pincer-Search: A new algorithm for discovering the maximum frequent set. *Technical Report TR1997-742*, Dept. of Computer Science, New York University, Sept. 1997.

[LK98] D. Lin and Z. Kedem. Pincer-Search: A new algorithm for discovering the maximum frequent set. In *Proc. of the 6th International Conference on Extending Database Technology (EDBT)*. Mar. 1998.

[M82] T. Mitchell. Generalization as search. *Artificial Intelligence*, Vol. 18, 1982.

[M95] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. *Technical Report No. CS-TR-3515* of CS Department, University of Maryland-College Park.

[M97] H. Mannila. Methods and problems in data mining (a tutorial). In *Proc. of International Conference on Database Theory (ICDT)*, Jan. 1997.

[MT95] H. Mannila and H. Toivonen. Discovering frequent episodes in sequences. In *Proc. 1st International Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 1995.

[MT96a] H. Mannila and H. Toivonen. On an algorithm for finding all interesting

sentences. In *13th European Meeting on Cybernetics and Systems Research*, Apr. 1996.

[MT96b] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In *Proc. 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 1996.

[MT96c] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurrences. In *Proc. 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 1996.

[MT97] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Technical Report TR C-1997-8*, Dept. of Computer Science, U. of Helsinki, Jan. 1997.

[MTV94] H. Mannila, H. Toivonen, and A. Verkamo. Improved methods for finding association rules. In *Proc. AAAI Workshop on Knowledge Discovery*, July 1994.

[NYSE97] The TAQ Database Release 1.0 in CD-ROM, New York Stock Exchange, Inc., June 1997.

[ORS98] B. Özden, S. Ramaswamy. and A. Silberschatz. Cyclic Association Rules. In *Proc. 14th International Conference on Data Engineering (ICDE)*, Feb. 1998.

[P91] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. *Knowledge Discovery in Databases*, AAAI Press, 1991.

[PBKKS97] G. Piatetsky-Shapiro, R. Brachman, T. Khabaza, W. Kloesgen, and E. Simoudis. An overview of issues in developing industrial data mining and knowledge discovery applications. In *Proc. 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 1996.

[PCY95] J. Park, M. Chen, and P. Yu. An effective hash-based algorithm for mining association rules. In *Proc. ACM-SIGMOD*, May 1995.

[S96] R. Srikant. Fast algorithm for mining association rules and sequential patterns. Ph.D. Thesis, University of Wisconsin, Madison, 1996.

[SA95a] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proc. 11th Int'l Conference on Data Engineering (ICDE)*, Mar. 1995.

[SA95b] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 21st VLDB*. Sep. 1995.

[SA96a] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the 5th International Conference on Extending Database Technology (EDBT)*. Mar. 1996.

[SA96b] R. Srikant and R. Agrawal. Mining Quantitative Association Rules in Large Relational Tables. In *Proc. SIGMOD*, June 1996.

[SON95] A. Sarasere, E. Omiecinsky, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. 21st VLDB*, Sept. 1995.

[SVA97] R. Srikant, Q. Vu, and R. Agrawal. Mining Association Rules with Item Constraints. In *Proc. of the 3rd International Conference on Knowledge Discovery in Databases and Data Mining (KDD)*, Aug. 1997.

[T96a] H. Toivonen. Discovery of frequent patterns in large data collections. *Technical Report A-1996-5* of the Department of Computer Science, University of Helsinki, Finland, 1996.

[T96b] H. Toivonen. Sampling large databases for association rules. In *Proc. 22nd VLDB*, Sept. 1996.

[Z97] M. J. Zaki. Fast mining of sequential patterns in very large databases. *Technical Report 668* of the Department of Computer Science, University of Rochester. Nov. 1997.

[ZPOL96] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel data minnig for association rules on shared-memory multi-processors. *Technical Re-*

port 618 of the Department of Computer Science, University of Rochester. May 1996.

[ZPOL97] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. 3rd International Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 1997.

Fast Algorithms for Discovering the Maximum Frequent Set

by

Dao-I Lin

Advisor: Zvi M. Kedem

Discovering frequent itemsets is a key problem in important data mining applications, such as the discovery of association rules, strong rules, episodes, and minimal keys. Typical algorithms for solving this problem operate in a bottom-up breadth-first search direction. The computation starts from frequent 1-itemsets (minimal length frequent itemsets) and continues until all maximal (length) frequent itemsets are found. During the execution, every frequent itemset is explicitly considered. Such algorithms perform reasonably well when all maximal frequent itemsets are short. However, performance drastically decreases when some of the maximal frequent itemsets are relatively long. We present a new algorithm which combines both the bottom-up and the top-down searches. The primary search direction is still bottom-up, but a restricted search is also conducted in the top-down direction. This search is used only for maintaining and updating a new data structure we designed, the maximum frequent candidate set. It is used to prune candidates in

the bottom-up search. A very important characteristic of the algorithm is that it does not require explicite examination of every frequent itemset. Therefore the algorithm performs well even when some maximal frequent itemsets are long. As its output, the algorithm produces the maximum frequent set, i.e., the set containing all maximal frequent itemsets, thus specifying immediately all frequent itemsets. We evaluate the performance of the algorithm using well-known synthetatic benchmark databases and real-life census and stock market databases. The improvement in performance can be up to several orders of magnitude, compared to the best current algorithms.