Chapter 6

Frequent Itemsets

We turn in this chapter to one of the major families of techniques for characterizing data: the discovery of frequent itemsets. This problem is often viewed as the discovery of "association rules," although the latter is a more complex characterization of data, whose discovery depends fundamentally on the discovery of frequent itemsets.

To begin, we introduce the "market-basket" model of data, which is essentially a many-many relationship between two kinds of elements, called "items" and "baskets," but with some assumptions about the shape of the data. The frequent-itemsets problem is that of finding sets of items that appear in (are related to) many of the same baskets.

The problem of finding frequent itemsets differs from the similarity search discussed in Chapter 3. Here we are interested in the absolute number of baskets that contain a particular set of items. In Chapter 3 we wanted items that have a large fraction of their baskets in common, even if the absolute number of baskets is small.

The difference leads to a new class of algorithms for finding frequent itemsets. We begin with the A-Priori Algorithm, which works by eliminating most large sets as candidates by looking first at smaller sets and recognizing that a large set cannot be frequent unless all its subsets are. We then consider various improvements to the basic A-Priori idea, concentrating on very large data sets that stress the available main memory.

Next, we consider approximate algorithms that work faster but are not guaranteed to find all frequent itemsets. Also in this class of algorithms are those that exploit parallelism, including the parallelism we can obtain through a MapReduce formulation. Finally, we discuss briefly how to find frequent itemsets in a data stream.

6.1 The Market-Basket Model

The *market-basket* model of data is used to describe a common form of many-many relationship between two kinds of objects. On the one hand, we have *items*, and on the other we have *baskets*, sometimes called "transactions." Each basket consists of a set of items (an *itemset*), and usually we assume that the number of items in a basket is small – much smaller than the total number of items. The number of baskets is usually assumed to be very large, bigger than what can fit in main memory. The data is assumed to be represented in a file consisting of a sequence of baskets. In terms of the distributed file system described in Section 2.1, the baskets are the objects of the file, and each basket is of type "set of items."

6.1.1 Definition of Frequent Itemsets

Intuitively, a set of items that appears in many baskets is said to be "frequent." To be formal, we assume there is a number s, called the *support threshold.* If I is a set of items, the *support* for I is the number of baskets for which I is a subset. We say I is *frequent* if its support is s or more.

Example 6.1: In Fig. 6.1 are sets of words. Each set is a basket, and the words are items. We took these sets by Googling `cat dog` and taking snippets from the highest-ranked pages. Do not be concerned if a word appears twice in a basket, as baskets are sets, and in principle items can appear only once. Also, ignore capitalization.

1. {Cat, and, dog, bites}
2. {Yahoo, news, claims, a, cat, mated, with, a, dog, and, produced, viable, offspring}
3. {Cat, killer, likely, is, a, big, dog}
4. {Professional, free, advice, on, dog, training, puppy, training}
5. {Cat, and, kitten, training, and, behavior}
6. {Dog, &, Cat, provides, dog, training, in, Eugene, Oregon}
7. {"Dog, and, cat", is, a, slang, term, used, by, police, officers, for, a, male–female, relationship}
8. {Shop, for, your, show, dog, grooming, and, pet, supplies}

Figure 6.1: Here are eight baskets, each consisting of items that are words

Since the empty set is a subset of any set, the support for $\emptyset$ is 8. However, we shall not generally concern ourselves with the empty set, since it tells us nothing.

Among the singleton sets, obviously {cat} and {dog} are quite frequent. "Dog" appears in all but basket (5), so its support is 7, while "cat" appears in all but (4) and (8), so its support is 6. The word "and" is also quite frequent; it appears in (1), (2), (5), (7), and (8), so its support is 5. The words "a" and "training" appear in three sets, while "for" and "is" appear in two each. No other word appears more than once.

Suppose that we set our threshold at $s = 3$. Then there are five frequent singleton itemsets: {dog}, {cat}, {and}, {a}, and {training}.

Now, let us look at the doubletons. A doubleton cannot be frequent unless both items in the set are frequent by themselves. Thus, there are only ten possible frequent doubletons. Fig. 6.2 is a table indicating which baskets contain which doubletons.

	training	a	and	cat
dog	4, 6	2, 3, 7	1, 2, 7, 8	1, 2, 3, 6, 7
cat	5, 6	2, 3, 7	1, 2, 5, 7	
and	5	2, 7		
a	none			

Figure 6.2: Occurrences of doubletons

For example, we see from the table of Fig. 6.2 that doubleton {dog, training} appears only in baskets (4) and (6). Therefore, its support is 2, and it is not frequent. There are five frequent doubletons if $s = 3$; they are

{dog, a}	{dog, and}	{dog, cat}
{cat, a}	{cat, and}	

Each appears at least three times; for instance, {dog, cat} appears five times.

Next, let us see if there are frequent triples. In order to be a frequent triple, each pair of elements in the set must be a frequent doubleton. For example, {dog, a, and} cannot be a frequent itemset, because if it were, then surely {a, and} would be frequent, but it is not. The triple {dog, cat, and} might be frequent, because each of its doubleton subsets is frequent. Unfortunately, the three words appear together only in baskets (1) and (2), so it is not a frequent triple. The triple {dog, cat, a} might be frequent, since its doubletons are all frequent. In fact, all three words do appear in baskets (2), (3), and (7), so it is a frequent triple. No other triple of words is even a candidate for being a frequent triple, since for no other triple of words are its three doubleton subsets frequent. As there is only one frequent triple, there can be no frequent quadruples or larger sets. □

On-Line versus Brick-and-Mortar Retailing

We suggested in Section 3.1.3 that an on-line retailer would use similarity measures for items to find pairs of items that, while they might not be bought by many customers, had a significant fraction of their customers in common. An on-line retailer could then advertise one item of the pair to the few customers who had bought the other item of the pair. This methodology makes no sense for a bricks-and-mortar retailer, because unless lots of people buy an item, it cannot be cost effective to advertise a sale on the item. Thus, the techniques of Chapter 3 are not often useful for brick-and-mortar retailers.

Conversely, the on-line retailer has little need for the analysis we discuss in this chapter, since it is designed to search for itemsets that appear frequently. If the on-line retailer was limited to frequent itemsets, they would miss all the opportunities that are present in the "long tail" to select advertisements for each customer individually.

6.1.2 Applications of Frequent Itemsets

The original application of the market-basket model was in the analysis of true market baskets. That is, supermarkets and chain stores record the contents of every market basket (physical shopping cart) brought to the register for checkout. Here the "items" are the different products that the store sells, and the "baskets" are the sets of items in a single market basket. A major chain might sell 100,000 different items and collect data about millions of market baskets.

By finding frequent itemsets, a retailer can learn what is commonly bought together. Especially important are pairs or larger sets of items that occur much more frequently than would be expected were the items bought independently. We shall discuss this aspect of the problem in Section 6.1.3, but for the moment let us simply consider the search for frequent itemsets. We will discover by this analysis that many people buy bread and milk together, but that is of little interest, since we already knew that these were popular items individually. We might discover that many people buy hot dogs and mustard together. That, again, should be no surprise to people who like hot dogs, but it offers the supermarket an opportunity to do some clever marketing. They can advertise a sale on hot dogs and raise the price of mustard. When people come to the store for the cheap hot dogs, they often will remember that they need mustard, and buy that too. Either they will not notice the price is high, or they reason that it is not worth the trouble to go somewhere else for cheaper mustard.

The famous example of this type is "diapers and beer." One would hardly expect these two items to be related, but through data analysis one chain store discovered that people who buy diapers are unusually likely to buy beer. The theory is that if you buy diapers, you probably have a baby at home, and if you have a baby, then you are unlikely to be drinking at a bar; hence you are more likely to bring beer home. The same sort of marketing ploy that we suggested for hot dogs and mustard could be used for diapers and beer.

However, applications of frequent-itemset analysis is not limited to market baskets. The same model can be used to mine many other kinds of data. Some examples are:

1. *Related concepts*: Let items be words, and let baskets be documents (e.g., Web pages, blogs, tweets). A basket/document contains those items/words that are present in the document. If we look for sets of words that appear together in many documents, the sets will be dominated by the most common words (stop words), as we saw in Example 6.1. There, even though the intent was to find snippets that talked about cats and dogs, the stop words "and" and "a" were prominent among the frequent itemsets. However, if we ignore all the most common words, then we would hope to find among the frequent pairs some pairs of words that represent a joint concept. For example, we would expect a pair like {Brad, Angelina} to appear with surprising frequency.

2. *Plagiarism*: Let the items be documents and the baskets be sentences. An item/document is "in" a basket/sentence if the sentence is in the document. This arrangement appears backwards, but it is exactly what we need, and we should remember that the relationship between items and baskets is an arbitrary many-many relationship. That is, "in" need not have its conventional meaning: "part of." In this application, we look for pairs of items that appear together in several baskets. If we find such a pair, then we have two documents that share several sentences in common. In practice, even one or two sentences in common is a good indicator of plagiarism.

3. *Biomarkers*: Let the items be of two types – biomarkers such as genes or blood proteins, and diseases. Each basket is the set of data about a patient: their genome and blood-chemistry analysis, as well as their medical history of disease. A frequent itemset that consists of one disease and one or more biomarkers suggests a test for the disease.

6.1.3 Association Rules

While the subject of this chapter is extracting frequent sets of items from data, this information is often presented as a collection of if–then rules, called *association rules*. The form of an association rule is $I \to j$, where I is a set of items and j is an item. The implication of this association rule is that if all of the items in I appear in some basket, then j is "likely" to appear in that basket as well.

We formalize the notion of "likely" by defining the *confidence* of the rule $I \to j$ to be the ratio of the support for $I \cup \{j\}$ to the support for I. That is, the confidence of the rule is the fraction of the baskets with all of I that also contain j.

Example 6.2 : Consider the baskets of Fig. 6.1. The confidence of the rule $\{cat,\ dog\} \to and$ is 3/5. The words "cat" and "dog" appear in five baskets: (1), (2), (3), (6), and (7). Of these, "and" appears in (1), (2), and (7), or 3/5 of the baskets.

For another illustration, the confidence of $\{cat\} \to kitten$ is 1/6. The word "cat" appears in six baskets, (1), (2), (3), (5), (6), and (7). Of these, only (5) has the word "kitten." □

Confidence alone can be useful, provided the support for the left side of the rule is fairly large. For example, we don't need to know that people are unusually likely to buy mustard when they buy hot dogs, as long as we know that many people buy hot dogs, and many people buy both hot dogs and mustard. We can still use the sale-on-hot-dogs trick discussed in Section 6.1.2. However, there is often more value to an association rule if it reflects a true relationship, where the item or items on the left somehow affect the item on the right.

Thus, we define the *interest* of an association rule $I \to j$ to be the difference between its confidence and the fraction of baskets that contain j. That is, if I has no influence on j, then we would expect that the fraction of baskets including I that contain j would be exactly the same as the fraction of all baskets that contain j. Such a rule has interest 0. However, it is interesting, in both the informal and technical sense, if a rule has either high interest, meaning that the presence of I in a basket somehow causes the presence of j, or highly negative interest, meaning that the presence of I discourages the presence of j.

Example 6.3 : The story about beer and diapers is really a claim that the association rule $\{diapers\} \to beer$ has high interest. That is, the fraction of diaper-buyers who buy beer is significantly greater than the fraction of all customers that buy beer. An example of a rule with negative interest is $\{coke\} \to pepsi$. That is, people who buy Coke are unlikely to buy Pepsi as well, even though a good fraction of all people buy Pepsi – people typically prefer one or the other, but not both. Similarly, the rule $\{pepsi\} \to coke$ can be expected to have negative interest.

For some numerical calculations, let us return to the data of Fig. 6.1. The rule $\{dog\} \to cat$ has confidence 5/7, since "dog" appears in seven baskets, of which five have "cat." However, "cat" appears in six out of the eight baskets, so we would expect that 75% of the seven baskets with "dog" would have "cat" as well. Thus, the interest of the rule is $5/7 - 3/4 = -0.036$, which is essentially 0. The rule $\{cat\} \to kitten$ has interest $1/6 - 1/8 = 0.042$. The justification is that one out of the six baskets with "cat" have "kitten" as well, while "kitten" appears in only one of the eight baskets. This interest, while positive, is close to 0 and therefore indicates the association rule is not very "interesting." □

6.1.4 Finding Association Rules with High Confidence

Identifying useful association rules is not much harder than finding frequent itemsets. We shall take up the problem of finding frequent itemsets in the balance of this chapter, but for the moment, assume it is possible to find those frequent itemsets whose support is at or above a support threshold s.

If we are looking for association rules $I \rightarrow j$ that apply to a reasonable fraction of the baskets, then the support of I must be reasonably high. In practice, such as for marketing in brick-and-mortar stores, "reasonably high" is often around 1% of the baskets. We also want the confidence of the rule to be reasonably high, perhaps 50%, or else the rule has little practical effect. As a result, the set $I \cup \{j\}$ will also have fairly high support.

Suppose we have found all itemsets that meet a threshold of support, and that we have the exact support calculated for each of these itemsets. We can find within them all the association rules that have both high support and high confidence. That is, if J is a set of n items that is found to be frequent, there are only n possible association rules involving this set of items, namely $J - \{j\} \rightarrow j$ for each j in J. If J is frequent, $J - \{j\}$ must be at least as frequent. Thus, it too is a frequent itemset, and we have already computed the support of both J and $J - \{j\}$. Their ratio is the confidence of the rule $J - \{j\} \rightarrow j$.

It must be assumed that there are not too many frequent itemsets and thus not too many candidates for high-support, high-confidence association rules. The reason is that each one found must be acted upon. If we give the store manager a million association rules that meet our thresholds for support and confidence, they cannot even read them, let alone act on them. Likewise, if we produce a million candidates for biomarkers, we cannot afford to run the experiments needed to check them out. Thus, it is normal to adjust the support threshold so that we do not get too many frequent itemsets. This assumption leads, in later sections, to important consequences about the efficiency of algorithms for finding frequent itemsets.

6.1.5 Exercises for Section 6.1

Exercise 6.1.1: Suppose there are 100 items, numbered 1 to 100, and also 100 baskets, also numbered 1 to 100. Item i is in basket b if and only if i divides b with no remainder. Thus, item 1 is in all the baskets, item 2 is in all fifty of the even-numbered baskets, and so on. Basket 12 consists of items $\{1, 2, 3, 4, 6, 12\}$, since these are all the integers that divide 12. Answer the following questions:

(a) If the support threshold is 5, which items are frequent?

! (b) If the support threshold is 5, which pairs of items are frequent?

! (c) What is the sum of the sizes of all the baskets?

! Exercise 6.1.2: For the item-basket data of Exercise 6.1.1, which basket is the largest?

Exercise 6.1.3: Suppose there are 100 items, numbered 1 to 100, and also 100 baskets, also numbered 1 to 100. Item i is in basket b if and only if b divides i with no remainder. For example, basket 12 consists of items

$$\{12, 24, 36, 48, 60, 72, 84, 96\}$$

Repeat Exercise 6.1.1 for this data.

! Exercise 6.1.4: This question involves data from which nothing interesting can be learned about frequent itemsets, because there are no sets of items that are correlated. Suppose the items are numbered 1 to 10, and each basket is constructed by including item i with probability $1/i$, each decision being made independently of all other decisions. That is, all the baskets contain item 1, half contain item 2, a third contain item 3, and so on. Assume the number of baskets is sufficiently large that the baskets collectively behave as one would expect statistically. Let the support threshold be 1% of the baskets. Find the frequent itemsets.

Exercise 6.1.5: For the data of Exercise 6.1.1, what is the confidence of the following association rules?

(a) $\{5, 7\} \rightarrow 2$.

(b) $\{2, 3, 4\} \rightarrow 5$.

Exercise 6.1.6: For the data of Exercise 6.1.3, what is the confidence of the following association rules?

(a) $\{24, 60\} \rightarrow 8$.

(b) $\{2, 3, 4\} \rightarrow 5$.

!! Exercise 6.1.7: Describe all the association rules that have 100% confidence for the market-basket data of:

(a) Exercise 6.1.1.

(b) Exercise 6.1.3.

! Exercise 6.1.8: Prove that in the data of Exercise 6.1.4 there are no interesting association rules; i.e., the interest of every association rule is 0.

6.2 Market Baskets and the A-Priori Algorithm

We shall now begin a discussion of how to find frequent itemsets or information derived from them, such as association rules with high support and confidence. The original improvement on the obvious algorithms, known as "A-Priori," from which many variants have been developed, will be covered here. The next two sections will discuss certain further improvements. Before discussing the A-priori Algorithm itself, we begin the section with an outline of the assumptions about how data is stored and manipulated when searching for frequent itemsets.

6.2.1 Representation of Market-Basket Data

As we mentioned, we assume that market-basket data is stored in a file basket-by-basket. Possibly, the data is in a distributed file system as in Section 2.1, and the baskets are the objects the file contains. Or the data may be stored in a conventional file, with a character code to represent the baskets and their items.

Example 6.4: We could imagine that such a file begins:

```
{23,456,1001}{3,18,92,145}{...
```

Here, the character { begins a basket and the character } ends it. The items in a basket are represented by integers and are separated by commas. Thus, the first basket contains items 23, 456, and 1001; the second basket contains items 3, 18, 92, and 145. □

It may be that one machine receives the entire file. Or we could be using MapReduce or a similar tool to divide the work among many processors, in which case each processor receives only a part of the file. It turns out that combining the work of parallel processors to get the exact collection of itemsets that meet a global support threshold is hard, and we shall address this question only in Section 6.4.4.

We also assume that the size of the file of baskets is sufficiently large that it does not fit in main memory. Thus, a major cost of any algorithm is the time it takes to read the baskets from disk. Once a disk block full of baskets is in main memory, we can expand it, generating all the subsets of size k. Since one of the assumptions of our model is that the average size of a basket is small, generating all the pairs in main memory should take time that is much less than the time it took to read the basket from disk. For example, if there are 20 items in a basket, then there are $\binom{20}{2} = 190$ pairs of items in the basket, and these can be generated easily in a pair of nested for-loops.

As the size of the subsets we want to generate gets larger, the time required grows larger; in fact takes approximately time $n^k/k!$ to generate all the subsets of size k for a basket with n items. Eventually, this time dominates the time needed to transfer the data from disk. However:

1. Often, we need only small frequent itemsets, so k never grows beyond 2 or 3.

2. And when we do need the itemsets for a large size k, it is usually possible to eliminate many of the items in each basket as not able to participate in a frequent itemset, so the value of n drops as k increases.

The conclusion we would like to draw is that the work of examining each of the baskets can usually be assumed proportional to the size of the file. We can thus measure the running time of a frequent-itemset algorithm by the number of times that each disk block of the data file is read.

Moreover, all the algorithms we discuss have the property that they read the basket file sequentially. Thus, algorithms can be characterized by the number of passes through the basket file that they make, and their running time is proportional to the product of the number of passes they make through the basket file times the size of that file. Since we cannot control the amount of data, only the number of passes taken by the algorithm matters, and it is that aspect of the algorithm that we shall focus upon when measuring the running time of a frequent-itemset algorithm.

6.2.2 Use of Main Memory for Itemset Counting

There is a second data-related issue that we must examine, however. All frequent-itemset algorithms require us to maintain many different counts as we make a pass through the data. For example, we might need to count the number of times that each pair of items occurs in baskets. If we do not have enough main memory to store each of the counts, then adding 1 to a random count will most likely require us to load a page from disk. In that case, the algorithm will thrash and run many orders of magnitude slower than if we were certain to find each count in main memory. The conclusion is that we cannot count anything that doesn't fit in main memory. Thus, each algorithm has a limit on how many items it can deal with.

Example 6.5: Suppose a certain algorithm has to count all pairs of items, and there are n items. We thus need space to store $\binom{n}{2}$ integers, or about $n^2/2$ integers. If integers take 4 bytes, we require $2n^2$ bytes. If our machine has 2 gigabytes, or 2^{31} bytes of main memory, then we require $n \leq 2^{15}$, or approximately $n < 33{,}000$. □

It is not trivial to store the $\binom{n}{2}$ counts in a way that makes it easy to find the count for a pair $\{i, j\}$. First, we have not assumed anything about how items are represented. They might, for instance, be strings like "bread." It is more space-efficient to represent items by consecutive integers from 1 to n, where n is the number of distinct items. Unless items are already represented this way, we need a hash table that translates items as they appear in the file to integers. That is, each time we see an item in the file, we hash it. If it is already in the hash table, we can obtain its integer code from its entry in the table. If the item is not there, we assign it the next available number (from a count of the number of distinct items seen so far) and enter the item and its code into the table.

The Triangular-Matrix Method

Even after coding items as integers, we still have the problem that we must count a pair $\{i,j\}$ in only one place. For example, we could order the pair so that $i < j$, and only use the entry $a[i,j]$ in a two-dimensional array a. That strategy would make half the array useless. A more space-efficient way is to use a one-dimensional *triangular array.* We store in $a[k]$ the count for the pair $\{i,j\}$, with $1 \leq i < j \leq n$, where

$$k = (i-1)\Big(n - \frac{i}{2}\Big) + j - i$$

The result of this layout is that the pairs are stored in lexicographic order, that is first $\{1,2\}$, $\{1,3\},\ldots,\{1,n\}$, then $\{2,3\}$, $\{2,4\},\ldots,\{2,n\}$, and so on, down to $\{n-2,n-1\}$, $\{n-2,n\}$, and finally $\{n-1,n\}$.

The Triples Method

There is another approach to storing counts that may be more appropriate, depending on the fraction of the possible pairs of items that actually appear in some basket. We can store counts as triples $[i,j,c]$, meaning that the count of pair $\{i,j\}$, with $i < j$, is c. A data structure, such as a hash table with i and j as the search key, is used so we can tell if there is a triple for a given i and j and, if so, to find it quickly. We call this approach the *triples method* of storing counts.

Unlike the triangular matrix, the triples method does not require us to store anything if the count for a pair is 0. On the other hand, the triples method requires us to store three integers, rather than one, for every pair that does appear in some basket. In addition, there is the space needed for the hash table or other data structure used to support efficient retrieval. The conclusion is that the triangular matrix will be better if at least 1/3 of the $\binom{n}{2}$ possible pairs actually appear in some basket, while if significantly fewer than 1/3 of the possible pairs occur, we should consider using the triples method.

Example 6.6: Suppose there are 100,000 items, and 10,000,000 baskets of 10 items each. Then the triangular-matrix method requires $\binom{100000}{2} = 5 \times 10^9$ (approximately) integer counts.[1] On the other hand, the total number of pairs among all the baskets is $10^7\binom{10}{2} = 4.5 \times 10^8$. Even in the extreme case that every pair of items appeared only once, there could be only 4.5×10^8 pairs with

[1]Here, and throughout the chapter, we shall use the approximation that $\binom{n}{2} = n^2/2$ for large n.

nonzero counts. If we used the triples method to store counts, we would need only three times that number of integers, or 1.35×10^9 integers. Thus, in this case the triples method will surely take much less space than the triangular matrix.

However, even if there were ten or a hundred times as many baskets, it would be normal for there to be a sufficiently uneven distribution of items that we might still be better off using the triples method. That is, some pairs would have very high counts, and the number of different pairs that occurred in one or more baskets would be much less than the theoretical maximum number of such pairs. □

6.2.3 Monotonicity of Itemsets

Much of the effectiveness of the algorithms we shall discuss is driven by a single observation, called *monotonicity* for itemsets:

- If a set I of items is frequent, then so is every subset of I.

The reason is simple. Let $J \subseteq I$. Then every basket that contains all the items in I surely contains all the items in J. Thus, the count for J must be at least as great as the count for I, and if the count for I is at least s, then the count for J is at least s. Since J may be contained in some baskets that are missing one or more elements of $I - J$, it is entirely possible that the count for J is strictly greater than the count for I.

In addition to making the A-Priori Algorithm work, monotonicity offers us a way to compact the information about frequent itemsets. If we are given a support threshold s, then we say an itemset is *maximal* if no superset is frequent. If we list only the maximal itemsets, then we know that all subsets of a maximal itemset are frequent, and no set that is not a subset of some maximal itemset can be frequent.

Example 6.7 : Let us reconsider the data of Example 6.1 with support threshold $s = 3$. We found that there were five frequent singletons, those with words "cat," "dog," "a," "and," and "training." Each of these is contained in a frequent doubleton, except for "training," so one maximal frequent itemset is {training}. There are also five frequent doubletons with $s = 3$, namely

{dog, a}	{dog, and}	{dog, cat}
{cat, a}	{cat, and}	

We also found one frequent triple, {dog, cat, a}, and there are no larger frequent itemsets. Thus, this triple is maximal, but the three frequent doubletons it contains are *not* maximal. The other frequent doubletons, {dog, and} and {cat, and}, are maximal. Notice that we can deduce from the frequent doubletons that singletons like {dog} are frequent. □

6.2.4 Tyranny of Counting Pairs

As you may have noticed, we have focused on the matter of counting pairs in the discussion so far. There is a good reason to do so: in practice the most main memory is required for determining the frequent pairs. The number of items, while possibly very large, is rarely so large we cannot count all the singleton sets in main memory at the same time.

What about larger sets – triples, quadruples, and so on? Recall that in order for frequent-itemset analysis to make sense, the result has to be a small number of sets, or we cannot even *read* them all, let alone consider their significance. Thus, in practice the support threshold is set high enough that it is only a rare set that is frequent. Monotonicity tells us that if there is a frequent triple, then there are three frequent pairs contained within it. And of course there may be frequent pairs contained in no frequent triple as well. Thus, we expect to find more frequent pairs than frequent triples, more frequent triples than frequent quadruples, and so on.

That argument would not be enough were it impossible to avoid counting all the triples, since there are many more triples than pairs. It is the job of the A-Priori Algorithm and related algorithms to avoid counting many triples or larger sets, and they are, as we shall see, effective in doing so. Thus, in what follows, we concentrate on algorithms for computing frequent pairs.

6.2.5 The A-Priori Algorithm

For the moment, let us concentrate on finding the frequent pairs only. If we have enough main memory to count all pairs, using either of the methods discussed in Section 6.2.2 (triangular matrix or triples), then it is a simple matter to read the file of baskets in a single pass. For each basket, we use a double loop to generate all the pairs. Each time we generate a pair, we add 1 to its count. At the end, we examine all pairs to see which have counts that are equal to or greater than the support threshold s; these are the frequent pairs.

However, this simple approach fails if there are too many pairs of items to count them all in main memory. The *A-Priori* Algorithm is designed to reduce the number of pairs that must be counted, at the expense of performing two passes over data, rather than one pass.

The First Pass of A-Priori

In the first pass, we create two tables. The first table, if necessary, translates item names into integers from 1 to n, as described in Section 6.2.2. The other table is an array of counts; the ith array element counts the occurrences of the item numbered i. Initially, the counts for all the items are 0.

As we read baskets, we look at each item in the basket and translate its name into an integer. Next, we use that integer to index into the array of counts, and we add 1 to the integer found there.

Between the Passes of A-Priori

After the first pass, we examine the counts of the items to determine which of them are frequent as singletons. It might appear surprising that many singletons are not frequent. But remember that we set the threshold s sufficiently high that we do not get too many frequent sets; a typical s would be 1% of the baskets. If we think about our own visits to a supermarket, we surely buy certain things more than 1% of the time: perhaps milk, bread, Coke or Pepsi, and so on. We can even believe that 1% of the customers buy diapers, even though we may not do so. However, many of the items on the shelves are surely not bought by 1% of the customers: Creamy Caesar Salad Dressing for example.

For the second pass of A-Priori, we create a new numbering from 1 to m for just the frequent items. This table is an array indexed 1 to n, and the entry for i is either 0, if item i is not frequent, or a unique integer in the range 1 to m if item i is frequent. We shall refer to this table as the *frequent-items table.*

The Second Pass of A-Priori

During the second pass, we count all the pairs that consist of two frequent items. Recall from Section 6.2.3 that a pair cannot be frequent unless both its members are frequent. Thus, we miss no frequent pairs. The space required on the second pass is $2m^2$ bytes, rather than $2n^2$ bytes, if we use the triangular-matrix method for counting. Notice that the renumbering of just the frequent items is necessary if we are to use a triangular matrix of the right size. The complete set of main-memory structures used in the first and second passes is shown in Fig. 6.3.

Also notice that the benefit of eliminating infrequent items is amplified; if only half the items are frequent we need one quarter of the space to count. Likewise, if we use the triples method, we need to count only those pairs of two frequent items that occur in at least one basket.

The mechanics of the second pass are as follows.

1. For each basket, look in the frequent-items table to see which of its items are frequent.

2. In a double loop, generate all pairs of frequent items in that basket.

3. For each such pair, add one to its count in the data structure used to store counts.

Finally, at the end of the second pass, examine the structure of counts to determine which pairs are frequent.

6.2.6 A-Priori for All Frequent Itemsets

The same technique used for finding frequent pairs without counting all pairs lets us find larger frequent itemsets without an exhaustive count of all sets. In

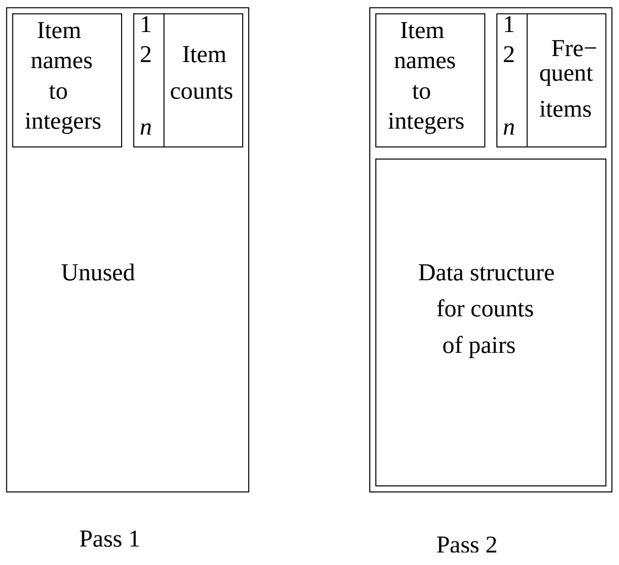

Figure 6.3: Schematic of main-memory use during the two passes of the A-Priori Algorithm

the A-Priori Algorithm, one pass is taken for each set-size k. If no frequent itemsets of a certain size are found, then monotonicity tells us there can be no larger frequent itemsets, so we can stop.

The pattern of moving from one size k to the next size $k+1$ can be summarized as follows. For each size k, there are two sets of itemsets:

1. C_k is the set of *candidate* itemsets of size k – the itemsets that we must count in order to determine whether they are in fact frequent.

2. L_k is the set of truly frequent itemsets of size k.

The pattern of moving from one set to the next and one size to the next is suggested by Fig. 6.4.

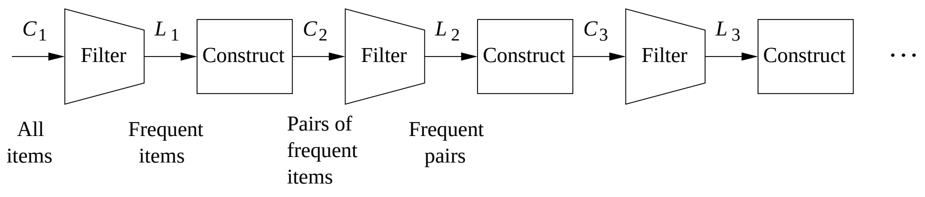

Figure 6.4: The A-Priori Algorithm alternates between constructing candidate sets and filtering to find those that are truly frequent

We start with C_1, which is all singleton itemsets, i.e., the items themselves. That is, before we examine the data, any item could be frequent as far as we know. The first filter step is to count all items, and those whose counts are at least the support threshold s form the set L_1 of frequent items.

The set C_2 of candidate pairs is the set of pairs both of whose items are in L_1; that is, they are frequent items. Note that we do not construct C_2 explicitly. Rather we use the definition of C_2, and we test membership in C_2 by testing whether both of its members are in L_1. The second pass of the A-Priori Algorithm counts all the candidate pairs and determines which appear at least s times. These pairs form L_2, the frequent pairs.

We can follow this pattern as far as we wish. The set C_3 of candidate triples is constructed (implicitly) as the set of triples, any two of which is a pair in L_2. Our assumption about the sparsity of frequent itemsets, outlined in Section 6.2.4 implies that there will not be too many frequent pairs, so they can be listed in a main-memory table. Likewise, there will not be too many candidate triples, so these can all be counted by a generalization of the triples method. That is, while triples are used to count pairs, we would use quadruples, consisting of the three item codes and the associated count, when we want to count triples. Similarly, we can count sets of size k using tuples with $k + 1$ components, the last of which is the count, and the first k of which are the item codes, in sorted order.

To find L_3 we make a third pass through the basket file. For each basket, we need only look at those items that are in L_1. From these items, we can examine each pair and determine whether or not that pair is in L_2. Any item of the basket that does not appear in at least two frequent pairs, both of which consist of items in the basket, cannot be part of a frequent triple that the basket contains. Thus, we have a fairly limited search for triples that are both contained in the basket and are candidates in C_3. Any such triples found have 1 added to their count.

Example 6.8 : Suppose our basket consists of items 1 through 10. Of these, 1 through 5 have been found to be frequent items, and the following pairs have been found frequent: $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, and $\{4, 5\}$. At first, we eliminate the nonfrequent items, leaving only 1 through 5. However, 1 and 5 appear in only one frequent pair in the itemset, and therefore cannot contribute to a frequent triple contained in the basket. Thus, we must consider adding to the count of triples that are contained in $\{2, 3, 4\}$. There is only one such triple, of course. However, we shall not find it in C_3, because $\{2, 4\}$ evidently is not frequent. □

The construction of the collections of larger frequent itemsets and candidates proceeds in essentially the same manner, until at some pass we find no new frequent itemsets and stop. That is:

1. Define C_k to be all those itemsets of size k, every $k - 1$ of which is an itemset in L_{k-1}.

2. Find L_k by making a pass through the baskets and counting all and only the itemsets of size k that are in C_k. Those itemsets that have count at least s are in L_k.

6.2.7 Exercises for Section 6.2

Exercise 6.2.1 : If we use a triangular matrix to count pairs, and n, the number of items, is 20, what pair's count is in $a[100]$?

! **Exercise 6.2.2 :** In our description of the triangular-matrix method in Section 6.2.2, the formula for k involves dividing an arbitrary integer i by 2. Yet we need to have k always be an integer. Prove that k will, in fact, be an integer.

! **Exercise 6.2.3 :** Let there be I items in a market-basket data set of B baskets. Suppose that every basket contains exactly K items. As a function of I, B, and K:

(a) How much space does the triangular-matrix method take to store the counts of all pairs of items, assuming four bytes per array element?

(b) What is the largest possible number of pairs with a nonzero count?

(c) Under what circumstances can we be certain that the triples method will use less space than the triangular array?

!! **Exercise 6.2.4 :** How would you count all itemsets of size 3 by a generalization of the triangular-matrix method? That is, arrange that in a one-dimensional array there is exactly one element for each set of three items.

! **Exercise 6.2.5 :** Suppose the support threshold is 5. Find the maximal frequent itemsets for the data of:

(a) Exercise 6.1.1.

(b) Exercise 6.1.3.

Exercise 6.2.6 : Apply the A-Priori Algorithm with support threshold 5 to the data of:

(a) Exercise 6.1.1.

(b) Exercise 6.1.3.

! **Exercise 6.2.7 :** Suppose we have market baskets that satisfy the following assumptions:

1. The support threshold is 10,000.

2. There are one million items, represented by the integers $0, 1, \ldots, 999999$.

3. There are N frequent items, that is, items that occur 10,000 times or more.

4. There are one million pairs that occur 10,000 times or more.

5. There are $2M$ pairs that occur exactly once. Of these pairs, M consist of two frequent items; the other M each have at least one nonfrequent item.

6. No other pairs occur at all.

7. Integers are always represented by 4 bytes.

Suppose we run the A-Priori Algorithm and can choose on the second pass between the triangular-matrix method for counting candidate pairs and a hash table of item-item-count triples. Neglect in the first case the space needed to translate between original item numbers and numbers for the frequent items, and in the second case neglect the space needed for the hash table. As a function of N and M, what is the minimum number of bytes of main memory needed to execute the A-Priori Algorithm on this data?

6.3 Handling Larger Datasets in Main Memory

The A-Priori Algorithm is fine as long as the step with the greatest requirement for main memory – typically the counting of the candidate pairs C_2 – has enough memory that it can be accomplished without thrashing (repeated moving of data between disk and main memory). Several algorithms have been proposed to cut down on the size of candidate set C_2. Here, we consider the PCY Algorithm, which takes advantage of the fact that in the first pass of A-Priori there is typically lots of main memory not needed for the counting of single items. Then we look at the Multistage Algorithm, which uses the PCY trick and also inserts extra passes to further reduce the size of C_2.

6.3.1 The Algorithm of Park, Chen, and Yu

This algorithm, which we call *PCY* after its authors, exploits the observation that there may be much unused space in main memory on the first pass. If there are a million items and gigabytes of main memory, we do not need more than 10% of the main memory for the two tables suggested in Fig. 6.3 – a translation table from item names to small integers and an array to count those integers. The PCY Algorithm uses that space for an array of integers that generalizes the idea of a Bloom filter (see Section 4.3). The idea is shown schematically in Fig. 6.5.

Think of this array as a hash table, whose buckets hold integers rather than sets of keys (as in an ordinary hash table) or bits (as in a Bloom filter). Pairs of items are hashed to buckets of this hash table. As we examine a basket during the first pass, we not only add 1 to the count for each item in the basket, but

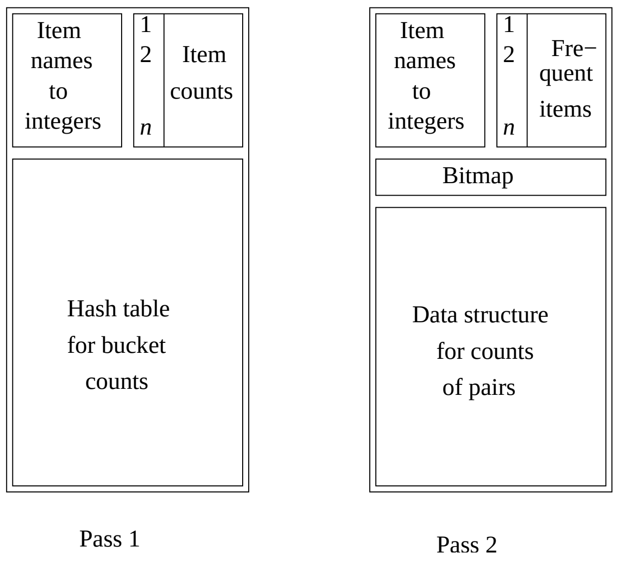

Figure 6.5: Organization of main memory for the first two passes of the PCY Algorithm

we generate all the pairs, using a double loop. We hash each pair, and we add 1 to the bucket into which that pair hashes. Note that the pair itself doesn't go into the bucket; the pair only affects the single integer in the bucket.

At the end of the first pass, each bucket has a count, which is the sum of the counts of all the pairs that hash to that bucket. If the count of a bucket is at least as great as the support threshold s, it is called a *frequent bucket.* We can say nothing about the pairs that hash to a frequent bucket; they could all be frequent pairs from the information available to us. But if the count of the bucket is less than s (an *infrequent bucket*), we know no pair that hashes to this bucket can be frequent, even if the pair consists of two frequent items. That fact gives us an advantage on the second pass. We can define the set of candidate pairs C_2 to be those pairs $\{i, j\}$ such that:

1. i and j are frequent items.

2. $\{i, j\}$ hashes to a frequent bucket.

It is the second condition that distinguishes PCY from A-Priori.

Example 6.9 : Depending on the data and the amount of available main memory, there may or may not be a benefit to using the hash table on pass 1. In the worst case, all buckets are frequent, and the PCY Algorithm counts exactly the same pairs as A-Priori does on the second pass. However, sometimes, we can expect most of the buckets to be infrequent. In that case, PCY reduces the memory requirements of the second pass.

Suppose we have a gigabyte of main memory available for the hash table on the first pass. Suppose also that the data file has a billion baskets, each with ten items. A bucket is an integer, typically 4 bytes, so we can maintain a quarter of a billion buckets. The number of pairs in all the baskets is $10^9 \times \binom{10}{2}$ or 4.5×10^{10} pairs; this number is also the sum of the counts in the buckets. Thus, the average count is $4.5 \times 10^{10}/2.5 \times 10^8$, or 180. If the support threshold s is around 180 or less, we might expect few buckets to be infrequent. However, if s is much larger, say 1000, then it must be that the great majority of the buckets are infrequent. The greatest possible number of frequent buckets is $4.5 \times 10^{10}/1000$, or 45 million out of the 250 million buckets. □

Between the passes of PCY, the hash table is summarized as a *bitmap*, with one bit for each bucket. The bit is 1 if the bucket is frequent and 0 if not. Thus integers of 32 bits are replaced by single bits, and the bitmap shown in the second pass in Fig. 6.5 takes up only 1/32 of the space that would otherwise be available to store counts. However, if most buckets are infrequent, we expect that the number of pairs being counted on the second pass will be much smaller than the total number of pairs of frequent items. Thus, PCY can handle some data sets without thrashing during the second pass, while A-Priori would run out of main memory and thrash.

There is another subtlety regarding the second pass of PCY that affects the amount of space needed. While we were able to use the triangular-matrix method on the second pass of A-Priori if we wished, because the frequent items could be renumbered from 1 to some m, we cannot do so for PCY. The reason is that the pairs of frequent items that PCY lets us avoid counting are placed randomly within the triangular matrix; they are the pairs that happen to hash to an infrequent bucket on the first pass. There is no known way of compacting the matrix to avoid leaving space for the uncounted pairs.

Consequently, we are forced to use the triples method in PCY. That restriction may not matter if the fraction of pairs of frequent items that actually appear in buckets were small; we would then want to use triples for A-Priori anyway. However, if most pairs of frequent items appear together in at least one bucket, then we are forced in PCY to use triples, while A-Priori can use a triangular matrix. Thus, unless PCY lets us avoid counting at least 2/3 of the pairs of frequent items, we cannot gain by using PCY instead of A-Priori.

While the discovery of frequent pairs by PCY differs significantly from A-Priori, the later stages, where we find frequent triples and larger sets if desired, are essentially the same as A-Priori. This statement holds as well for each of the improvements to A-Priori that we cover in this section. As a result, we shall cover only the construction of the frequent pairs from here on.

6.3.2 The Multistage Algorithm

The *Multistage Algorithm* improves upon PCY by using several successive hash tables to reduce further the number of candidate pairs. The tradeoff is that

Multistage takes more than two passes to find the frequent pairs. An outline of the Multistage Algorithm is shown in Fig. 6.6.

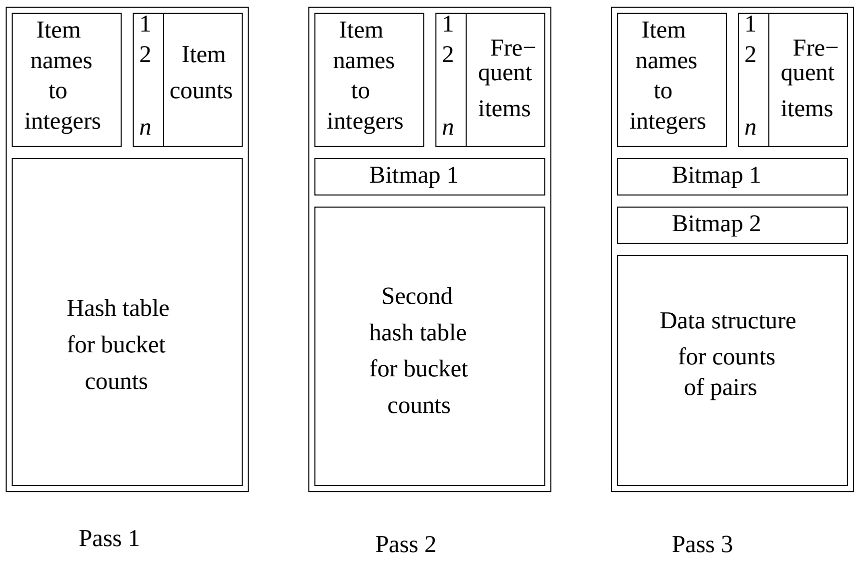

Figure 6.6: The Multistage Algorithm uses additional hash tables to reduce the number of candidate pairs

The first pass of Multistage is the same as the first pass of PCY. After that pass, the frequent buckets are identified and summarized by a bitmap, again the same as in PCY. But the second pass of Multistage does not count the candidate pairs. Rather, it uses the available main memory for another hash table, using another hash function. Since the bitmap from the first hash table takes up 1/32 of the available main memory, the second hash table has almost as many buckets as the first.

On the second pass of Multistage, we again go through the file of baskets. There is no need to count the items again, since we have those counts from the first pass. However, we must retain the information about which items are frequent, since we need it on both the second and third passes. During the second pass, we hash certain pairs of items to buckets of the second hash table. A pair is hashed only if it meets the two criteria for being counted in the second pass of PCY; that is, we hash $\{i, j\}$ if and only if i and j are both frequent, and the pair hashed to a frequent bucket on the first pass. As a result, the sum of the counts in the second hash table should be significantly less than the sum for the first pass. The result is that, even though the second hash table has only 31/32 of the number of buckets that the first table has, we expect there to be many fewer frequent buckets in the second hash table than in the first.

After the second pass, the second hash table is also summarized as a bitmap, and that bitmap is stored in main memory. The two bitmaps together take up

A Subtle Error in Multistage

Occasionally, an implementation tries to eliminate the second requirement for $\{i, j\}$ to be a candidate – that it hashes to a frequent bucket on the first pass. The (false) reasoning is that if it didn't hash to a frequent bucket on the first pass, it wouldn't have been hashed at all on the second pass, and thus would not contribute to the count of its bucket on the second pass. While it is true that the pair is not counted on the second pass, that doesn't mean it wouldn't have hashed to a frequent bucket had it been hashed. Thus, it is entirely possible that $\{i, j\}$ consists of two frequent items and hashes to a frequent bucket on the second pass, yet it did not hash to a frequent bucket on the first pass. Therefore, all three conditions must be checked on the counting pass of Multistage.

slightly less than 1/16th of the available main memory, so there is still plenty of space to count the candidate pairs on the third pass. A pair $\{i, j\}$ is in C_2 if and only if:

1. i and j are both frequent items.
2. $\{i, j\}$ hashed to a frequent bucket in the first hash table.
3. $\{i, j\}$ hashed to a frequent bucket in the second hash table.

The third condition is the distinction between Multistage and PCY.

It might be obvious that it is possible to insert any number of passes between the first and last in the multistage Algorithm. There is a limiting factor that each pass must store the bitmaps from each of the previous passes. Eventually, there is not enough space left in main memory to do the counts. No matter how many passes we use, the truly frequent pairs will always hash to a frequent bucket, so there is no way to avoid counting them.

6.3.3 The Multihash Algorithm

Sometimes, we can get most of the benefit of the extra passes of the Multistage Algorithm in a single pass. This variation of PCY is called the *Multihash Algorithm.* Instead of using two different hash tables on two successive passes, use two hash functions and two separate hash tables that share main memory on the first pass, as suggested by Fig. 6.7.

The danger of using two hash tables on one pass is that each hash table has half as many buckets as the one large hash table of PCY. As long as the average count of a bucket for PCY is much lower than the support threshold, we can operate two half-sized hash tables and still expect most of the buckets of both hash tables to be infrequent. Thus, in this situation we might well choose the multihash approach.

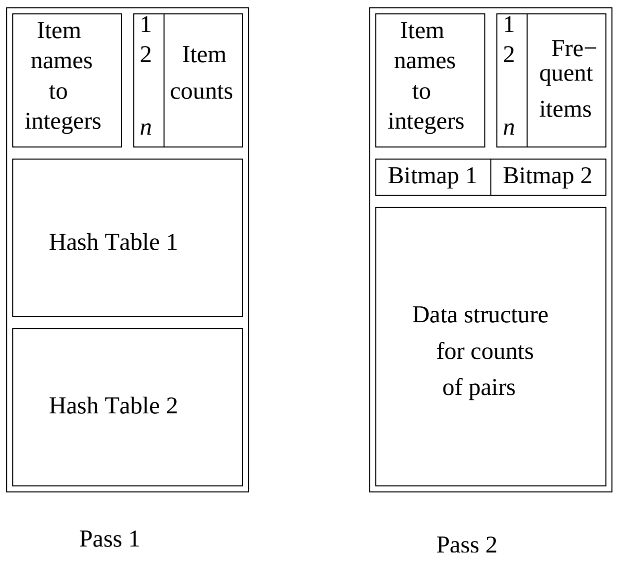

Figure 6.7: The Multihash Algorithm uses several hash tables in one pass

Example 6.10 : Suppose that if we run PCY, the average bucket will have a count $s/10$, where s is the support threshold. Then if we used the Multihash approach with two half-sized hash tables, the average count would be $s/5$. As a result, at most 1/5th of the buckets in either hash table could be frequent, and a random infrequent pair has at most probability $(1/5)^2 = 0.04$ of being in a frequent bucket in both hash tables.

By the same reasoning, the upper bound on the infrequent pair being in a frequent bucket in the one PCY hash table is at most 1/10. That is, we might expect to have to count 2.5 times as many infrequent pairs in PCY as in the version of Multihash suggested above. We would therefore expect Multihash to have a smaller memory requirement for the second pass than would PCY.

But these upper bounds do not tell the complete story. There may be many fewer frequent buckets than the maximum for either algorithm, since the presence of some very frequent pairs will skew the distribution of bucket counts. However, this analysis is suggestive of the possibility that for some data and support thresholds, we can do better by running several hash functions in main memory at once. □

For the second pass of Multihash, each hash table is converted to a bitmap, as usual. Note that the two bitmaps for the two hash functions in Fig. 6.7 occupy exactly as much space as a single bitmap would for the second pass of the PCY Algorithm. The conditions for a pair $\{i, j\}$ to be in C_2, and thus to require a count on the second pass, are the same as for the third pass of Multistage: i and j must both be frequent, and the pair must have hashed to a frequent bucket according to both hash tables.

Just as Multistage is not limited to two hash tables, we can divide the available main memory into as many hash tables as we like on the first pass of Multihash. The risk is that should we use too many hash tables, the average count for a bucket will exceed the support threshold. At that point, there may be very few infrequent buckets in any of the hash tables. Even though a pair must hash to a frequent bucket in every hash table to be counted, we may find that the probability an infrequent pair will be a candidate rises, rather than falls, if we add another hash table.

6.3.4 Exercises for Section 6.3

Exercise 6.3.1 : Here is a collection of twelve baskets. Each contains three of the six items 1 through 6.

$$\begin{array}{cccc}
\{1,2,3\} & \{2,3,4\} & \{3,4,5\} & \{4,5,6\} \\
\{1,3,5\} & \{2,4,6\} & \{1,3,4\} & \{2,4,5\} \\
\{3,5,6\} & \{1,2,4\} & \{2,3,5\} & \{3,4,6\}
\end{array}$$

Suppose the support threshold is 4. On the first pass of the PCY Algorithm we use a hash table with 11 buckets, and the set $\{i,j\}$ is hashed to bucket $i \times j$ mod 11.

(a) By any method, compute the support for each item and each pair of items.

(b) Which pairs hash to which buckets?

(c) Which buckets are frequent?

(d) Which pairs are counted on the second pass of the PCY Algorithm?

Exercise 6.3.2 : Suppose we run the Multistage Algorithm on the data of Exercise 6.3.1, with the same support threshold of 4. The first pass is the same as in that exercise, and for the second pass, we hash pairs to nine buckets, using the hash function that hashes $\{i,j\}$ to bucket $i + j \mod 9$. Determine the counts of the buckets on the second pass. Does the second pass reduce the set of candidate pairs? Note that all items are frequent, so the only reason a pair would not be hashed on the second pass is if it hashed to an infrequent bucket on the first pass.

Exercise 6.3.3 : Suppose we run the Multihash Algorithm on the data of Exercise 6.3.1. We shall use two hash tables with five buckets each. For one, the set $\{i,j\}$, is hashed to bucket $2i+3j+4 \mod 5$, and for the other, the set is hashed to $i+4j \mod 5$. Since these hash functions are not symmetric in i and j, order the items so that $i < j$ when evaluating each hash function. Determine the counts of each of the 10 buckets. How large does the support threshold have to be for the Multistage Algorithm to eliminate more pairs than the PCY Algorithm would, using the hash table and function described in Exercise 6.3.1?

! **Exercise 6.3.4:** Suppose we perform the PCY Algorithm to find frequent pairs, with market-basket data meeting the following specifications:

1. The support threshold is 10,000.
2. There are one million items, represented by the integers $0, 1, \ldots, 999999$.
3. There are 250,000 frequent items, that is, items that occur 10,000 times or more.
4. There are one million pairs that occur 10,000 times or more.
5. There are P pairs that occur exactly once and consist of two frequent items.
6. No other pairs occur at all.
7. Integers are always represented by 4 bytes.
8. When we hash pairs, they distribute among buckets randomly, but as evenly as possible; i.e., you may assume that each bucket gets exactly its fair share of the P pairs that occur once.

Suppose there are S bytes of main memory. In order to run the PCY Algorithm successfully, the number of buckets must be sufficiently large that most buckets are not frequent. In addition, on the second pass, there must be enough room to count all the candidate pairs. As a function of S, what is the largest value of P for which we can successfully run the PCY Algorithm on this data?

! **Exercise 6.3.5:** Under the assumptions given in Exercise 6.3.4, will the Multihash Algorithm reduce the main-memory requirements for the second pass? As a function of S and P, what is the optimum number of hash tables to use on the first pass?

! **Exercise 6.3.6:** Suppose we perform the 3-pass Multistage Algorithm to find frequent pairs, with market-basket data meeting the following specifications:

1. The support threshold is 10,000.
2. There are one million items, represented by the integers $0, 1, \ldots, 999999$. All items are frequent; that is, they occur at least 10,000 times.
3. There are one million pairs that occur 10,000 times or more.
4. There are P pairs that occur exactly once.
5. No other pairs occur at all.
6. Integers are always represented by 4 bytes.

7. When we hash pairs, they distribute among buckets randomly, but as evenly as possible; i.e., you may assume that each bucket gets exactly its fair share of the P pairs that occur once.

8. The hash functions used on the first two passes are completely independent.

Suppose there are S bytes of main memory. As a function of S and P, what is the expected number of candidate pairs on the third pass of the Multistage Algorithm?

6.4 Limited-Pass Algorithms

The algorithms for frequent itemsets discussed so far use one pass for each size of itemset we investigate. If main memory is too small to hold the data and the space needed to count frequent itemsets of one size, there does not seem to be any way to avoid k passes to compute the exact collection of frequent itemsets. However, there are many applications where it is not essential to discover every frequent itemset. For instance, if we are looking for items purchased together at a supermarket, we are not going to run a sale based on every frequent itemset we find, so it is quite sufficient to find most but not all of the frequent itemsets.

In this section we explore some algorithms that have been proposed to find all or most frequent itemsets using at most two passes. We begin with the obvious approach of using a sample of the data rather than the entire dataset. An algorithm called SON uses two passes, gets the exact answer, and lends itself to implementation by MapReduce or another parallel computing regime. Finally, Toivonen's Algorithm uses two passes on average, gets an exact answer, but may, rarely, not terminate in any given amount of time.

6.4.1 The Simple, Randomized Algorithm

Instead of using the entire file of baskets, we could pick a random subset of the baskets and pretend it is the entire dataset. We must adjust the support threshold to reflect the smaller number of baskets. For instance, if the support threshold for the full dataset is s, and we choose a sample of 1% of the baskets, then we should examine the sample for itemsets that appear in at least $s/100$ of the baskets.

The safest way to pick the sample is to read the entire dataset, and for each basket, select that basket for the sample with some fixed probability p. Suppose there are m baskets in the entire file. At the end, we shall have a sample whose size is very close to pm baskets. However, if we have reason to believe that the baskets appear in random order in the file already, then we do not even have to read the entire file. We can select the first pm baskets for our sample. Or, if the file is part of a distributed file system, we can pick some chunks at random to serve as the sample.

Why Not Just Pick the First Part of the File?

The risk in selecting a sample from one portion of a large file is that the data is not uniformly distributed in the file. For example, suppose the file were a list of true market-basket contents at a department store, organized by date of sale. If you took only the first baskets in the file, you would have old data. For example, there would be no iPods in these baskets, even though iPods might have become a popular item later.

As another example, consider a file of medical tests performed at different hospitals. If each chunk comes from a different hospital, then picking chunks at random will give us a sample drawn from only a small subset of the hospitals. If hospitals perform different tests or perform them in different ways, the data may be highly biased.

Having selected our sample of the baskets, we use part of main memory to store these baskets. The balance of the main memory is used to execute one of the algorithms we have discussed, such as A-Priori, PCY, Multistage, or Multihash. However, the algorithm must run passes over the main-memory sample for each itemset size, until we find a size with no frequent items. There are no disk accesses needed to read the sample, since it resides in main memory. As frequent itemsets of each size are discovered, they can be written out to disk; this operation and the initial reading of the sample from disk are the only disk I/O's the algorithm does.

Of course the algorithm will fail if whichever method from Section 6.2 or 6.3 we choose cannot be run in the amount of main memory left after storing the sample. If we need more main memory, then an option is to read the sample from disk for each pass. Since the sample is much smaller than the full dataset, we still avoid most of the disk I/O's that the algorithms discussed previously would use.

6.4.2 Avoiding Errors in Sampling Algorithms

We should be mindful of the problem with the simple algorithm of Section 6.4.1: it cannot be relied upon either to produce all the itemsets that are frequent in the whole dataset, nor will it produce only itemsets that are frequent in the whole. An itemset that is frequent in the whole but not in the sample is a *false negative*, while an itemset that is frequent in the sample but not the whole is a *false positive*.

If the sample is large enough, there are unlikely to be serious errors. That is, an itemset whose support is much larger than the threshold will almost surely be identified from a random sample, and an itemset whose support is much less than the threshold is unlikely to appear frequent in the sample. However, an itemset whose support in the whole is very close to the threshold is as likely to be frequent in the sample as not.

We can eliminate false positives by making a pass through the full dataset and counting all the itemsets that were identified as frequent in the sample. Retain as frequent only those itemsets that were frequent in the sample and also frequent in the whole. Note that this improvement will eliminate all false positives, but a false negative is not counted and therefore remains undiscovered.

To accomplish this task in a single pass, we need to be able to count all frequent itemsets of all sizes at once, within main memory. If we were able to run the simple algorithm successfully with the main memory available, then there is a good chance we shall be able to count all the frequent itemsets at once, because:

(a) The frequent singletons and pairs are likely to dominate the collection of all frequent itemsets, and we had to count them all in one pass already.

(b) We now have all of main memory available, since we do not have to store the sample in main memory.

We cannot eliminate false negatives completely, but we can reduce their number if the amount of main memory allows it. We have assumed that if s is the support threshold, and the sample is fraction p of the entire dataset, then we use ps as the support threshold for the sample. However, we can use something smaller than that as the threshold for the sample, such a $0.9ps$. Having a lower threshold means that more itemsets of each size will have to be counted, so the main-memory requirement rises. On the other hand, if there is enough main memory, then we shall identify as having support at least $0.9ps$ in the sample almost all those itemsets that have support at least s is the whole. If we then make a complete pass to eliminate those itemsets that were identified as frequent in the sample but are not frequent in the whole, we have no false positives and hopefully have none or very few false negatives.

6.4.3 The Algorithm of Savasere, Omiecinski, and Navathe

Our next improvement avoids both false negatives and false positives, at the cost of making two full passes. It is called the *SON* Algorithm after the authors. The idea is to divide the input file into chunks (which may be "chunks" in the sense of a distributed file system, or simply a piece of the file). Treat each chunk as a sample, and run the algorithm of Section 6.4.1 on that chunk. We use ps as the threshold, if each chunk is fraction p of the whole file, and s is the support threshold. Store on disk all the frequent itemsets found for each chunk.

Once all the chunks have been processed in that way, take the union of all the itemsets that have been found frequent for one or more chunks. These are the *candidate* itemsets. Notice that if an itemset is not frequent in any chunk, then its support is less than ps in each chunk. Since the number of chunks is $1/p$, we conclude that the total support for that itemset is less than $(1/p)ps = s$. Thus, every itemset that is frequent in the whole is frequent in at least one chunk, and we can be sure that all the truly frequent itemsets are among the candidates; i.e., there are no false negatives.

We have made a total of one pass through the data as we read each chunk and processed it. In a second pass, we count all the candidate itemsets and select those that have support at least s as the frequent itemsets.

6.4.4 The SON Algorithm and MapReduce

The SON algorithm lends itself well to a parallel-computing environment. Each of the chunks can be processed in parallel, and the frequent itemsets from each chunk combined to form the candidates. We can distribute the candidates to many processors, have each processor count the support for each candidate in a subset of the baskets, and finally sum those supports to get the support for each candidate itemset in the whole dataset. This process does not have to be implemented in MapReduce, but there is a natural way of expressing each of the two passes as a MapReduce operation. We shall summarize this MapReduce-MapReduce sequence below.

First Map Function: Take the assigned subset of the baskets and find the itemsets frequent in the subset using the algorithm of Section 6.4.1. As described there, lower the support threshold from s to ps if each Map task gets fraction p of the total input file. The output is a set of key-value pairs $(F, 1)$, where F is a frequent itemset from the sample. The value is always 1 and is irrelevant.

First Reduce Function: Each Reduce task is assigned a set of keys, which are itemsets. The value is ignored, and the Reduce task simply produces those keys (itemsets) that appear one or more times. Thus, the output of the first Reduce function is the candidate itemsets.

Second Map Function: The Map tasks for the second Map function take all the output from the first Reduce Function (the candidate itemsets) and a portion of the input data file. Each Map task counts the number of occurrences of each of the candidate itemsets among the baskets in the portion of the dataset that it was assigned. The output is a set of key-value pairs (C, v), where C is one of the candidate sets and v is the support for that itemset among the baskets that were input to this Map task.

Second Reduce Function: The Reduce tasks take the itemsets they are given as keys and sum the associated values. The result is the total support for each of the itemsets that the Reduce task was assigned to handle. Those itemsets whose sum of values is at least s are frequent in the whole dataset, so the Reduce task outputs these itemsets with their counts. Itemsets that do not have total support at least s are not transmitted to the output of the Reduce task.[2]

6.4.5 Toivonen's Algorithm

This algorithm uses randomness in a different way from the simple sampling algorithm of Section 6.4.1. Toivonen's Algorithm, given sufficient main memory, will use one pass over a small sample and one full pass over the data. It will give neither false negatives nor positives, but there is a small yet nonzero probability that it will fail to produce any answer at all. In that case it needs to be repeated until it gives an answer. However, the average number of passes needed before it produces all and only the frequent itemsets is a small constant.

Toivonen's algorithm begins by selecting a small sample of the input dataset, and finding from it the candidate frequent itemsets. The process is exactly that of Section 6.4.1, except that it is essential the threshold be set to something less than its proportional value. That is, if the support threshold for the whole dataset is s, and the sample size is fraction p, then when looking for frequent itemsets in the sample, use a threshold such as $0.9ps$ or $0.8ps$. The smaller we make the threshold, the more main memory we need for computing all itemsets that are frequent in the sample, but the more likely we are to avoid the situation where the algorithm fails to provide an answer.

Having constructed the collection of frequent itemsets for the sample, we next construct the *negative border*. This is the collection of itemsets that are not frequent in the sample, but all of their *immediate subsets* (subsets constructed by deleting exactly one item) are frequent in the sample.

Example 6.11: Suppose the items are $\{A, B, C, D, E\}$ and we have found the following itemsets to be frequent in the sample: $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, $\{B, C\}$, $\{C, D\}$. Note that $\emptyset$ is also frequent, as long as there are at least as many baskets as the support threshold, although technically the algorithms we have described omit this obvious fact. First, $\{E\}$ is in the negative border, because it is not frequent in the sample, but its only immediate subset, $\emptyset$, is frequent.

The sets $\{A, B\}$, $\{A, C\}$, $\{A, D\}$ and $\{B, D\}$ are in the negative border. None of these sets are frequent, and each has two immediate subsets, both of which are frequent. For instance, $\{A, B\}$ has immediate subsets $\{A\}$ and $\{B\}$. Of the other six doubletons, none are in the negative border. The sets $\{B, C\}$ and $\{C, D\}$ are not in the negative border, because they are frequent. The remaining four pairs are each E together with another item, and those are not in the negative border because they have an immediate subset $\{E\}$ that is not frequent.

None of the triples or larger sets are in the negative border. For instance, $\{B, C, D\}$ is not in the negative border because it has an immediate subset

[2]Strictly speaking, the Reduce function has to produce a value for each key. It can produce 1 as the value for itemsets found frequent and 0 for those not frequent.

$\{B, D\}$ that is not frequent. Thus, the negative border consists of five sets: $\{E\}$, $\{A, B\}$, $\{A, C\}$, $\{A, D\}$ and $\{B, D\}$. $\square$

To complete Toivonen's algorithm, we make a pass through the entire dataset, counting all the itemsets that are frequent in the sample or are in the negative border. There are two possible outcomes.

1. No member of the negative border is frequent in the whole dataset. In this case, the correct set of frequent itemsets is exactly those itemsets from the sample that were found to be frequent in the whole.

2. Some member of the negative border is frequent in the whole. Then we cannot be sure that there are not some even larger sets, in neither the negative border nor the collection of frequent itemsets for the sample, that are also frequent in the whole. Thus, we can give no answer at this time and must repeat the algorithm with a new random sample.

6.4.6 Why Toivonen's Algorithm Works

Clearly Toivonen's algorithm never produces a false positive, since it only reports as frequent those itemsets that have been counted and found to be frequent in the whole. To argue that it never produces a false negative, we must show that when no member of the negative border is frequent in the whole, then there can be no itemset whatsoever that is:

1. Frequent in the whole, but

2. In neither the negative border nor the collection of frequent itemsets for the sample.

Suppose the contrary. That is, there is a set S that is frequent in the whole, but not in the negative border and not frequent in the sample. Also, this round of Toivonen's Algorithm produced an answer, which would certainly not include S among the frequent itemsets. By monotonicity, all subsets of S are also frequent in the whole. Let T be a subset of S that is of the smallest possible size among all subsets of S that are not frequent in the sample.

We claim that T must be in the negative border. Surely T meets one of the conditions for being in the negative border: it is not frequent in the sample. It also meets the other condition for being in the negative border: each of its immediate subsets is frequent in the sample. For if some immediate subset of T were not frequent in the sample, then there would be a subset of S that is smaller than T and not frequent in the sample, contradicting our selection of T as a subset of S that was not frequent in the sample, yet as small as any such set.

Now we see that T is both in the negative border and frequent in the whole dataset. Consequently, this round of Toivonen's algorithm did not produce an answer.

6.4.7 Exercises for Section 6.4

Exercise 6.4.1: Suppose there are eight items, $A, B, \ldots, H$, and the following are the maximal frequent itemsets: $\{A, B\}$, $\{B, C\}$, $\{A, C\}$, $\{A, D\}$, $\{E\}$, and $\{F\}$. Find the negative border.

Exercise 6.4.2: Apply Toivonen's Algorithm to the data of Exercise 6.3.1, with a support threshold of 4. Take as the sample the first row of baskets: $\{1, 2, 3\}$, $\{2, 3, 4\}$, $\{3, 4, 5\}$, and $\{4, 5, 6\}$, i.e., one-third of the file. Our scaled-down support theshold will be 1.

(a) What are the itemsets frequent in the sample?

(b) What is the negative border?

(c) What is the outcome of the pass through the full dataset? Are any of the itemsets in the negative border frequent in the whole?

!! **Exercise 6.4.3:** Suppose item i appears exactly s times in a file of n baskets, where s is the support threshold. If we take a sample of $n/100$ baskets, and lower the support threshold for the sample to $s/100$, what is the probability that i will be found to be frequent? You may assume that both s and n are divisible by 100.

6.5 Counting Frequent Items in a Stream

Suppose that instead of a file of baskets we have a stream of baskets, from which we want to mine the frequent itemsets. Recall from Chapter 4 that the difference between a stream and a data file is that stream elements are only available when they arrive, and typically the arrival rate is so great that we cannot store the entire stream in a way that allows easy querying. Further, it is common that streams evolve over time, so the itemsets that are frequent in today's stream may not be frequent tomorrow.

A clear distinction between streams and files, when frequent itemsets are considered, is that there is no end to a stream, so eventually an itemset is going to exceed the support threshold, as long as it appears repeatedly in the stream. As a result, for streams, we must think of the support threshold s as a fraction of the baskets in which an itemset must appear in order to be considered frequent. Even with this adjustment, we still have several options regarding the portion of the stream over which that fraction is measured.

In this section, we shall discuss several ways that we might extract frequent itemsets from a stream. First, we consider ways to use the sampling techniques of the previous section. Then, we consider the decaying-window model from Section 4.7, and extend the method described in Section 4.7.3 for finding "popular" items.

6.5.1 Sampling Methods for Streams

In what follows, we shall assume that stream elements are baskets of items. Perhaps the simplest approach to maintaining a current estimate of the frequent itemsets in a stream is to collect some number of baskets and store it as a file. Run one of the frequent-itemset algorithms discussed in this chapter, meanwhile ignoring the stream elements that arrive, or storing them as another file to be analyzed later. When the frequent-itemsets algorithm finishes, we have an estimate of the frequent itemsets in the stream. We then have several options.

1. We can use this collection of frequent itemsets for whatever application is at hand, but start running another iteration of the chosen frequent-itemset algorithm immediately. This algorithm can either:

   (a) Use the file that was collected while the first iteration of the algorithm was running. At the same time, collect yet another file to be used at another iteration of the algorithm, when this current iteration finishes.

   (b) Start collecting another file of baskets now, and run the algorithm when an adequate number of baskets has been collected.

2. We can continue to count the numbers of occurrences of each of these frequent itemsets, along with the total number of baskets seen in the stream, since the counting started. If any itemset is discovered to occur in a fraction of the baskets that is significantly below the threshold fraction s, then this set can be dropped from the collection of frequent itemsets. When computing the fraction, it is important to include the occurrences from the original file of baskets, from which the frequent itemsets were derived. If not, we run the risk that we shall encounter a short period in which a truly frequent itemset does not appear sufficiently frequently and throw it out. We should also allow some way for new frequent itemsets to be added to the current collection. Possibilities include:

   (a) Periodically gather a new segment of the baskets in the stream and use it as the data file for another iteration of the chosen frequent-itemsets algorithm. The new collection of frequent items is formed from the result of this iteration and the frequent itemsets from the previous collection that have survived the possibility of having been deleted for becoming infrequent.

   (b) Add some random itemsets to the current collection, and count their fraction of occurrences for a while, until one has a good idea of whether or not they are currently frequent. Rather than choosing new itemsets completely at random, one might focus on sets with items that appear in many itemsets already known to be frequent. For example, a good choice is to pick new itemsets from the negative border (Section 6.4.5) of the current set of frequent itemsets.

6.5.2 Frequent Itemsets in Decaying Windows

Recall from Section 4.7 that a decaying window on a stream is formed by picking a small constant c and giving the ith element prior to the most recent element the weight $(1-c)^i$, or approximately e^{-ci}. Section 4.7.3 actually presented a method for computing the frequent items, provided the support threshold is defined in a somewhat different way. That is, we considered, for each item, a stream that had 1 if the item appeared at a certain stream element and 0 if not. We defined the "score" for that item to be the sum of the weights where its stream element was 1. We were constrained to record all items whose score was at least 1/2. We can not use a score threshold above 1, because we do not initiate a count for an item until the item appears in the stream, and the first time it appears, its score is only 1 (since 1, or $(1-c)^0$, is the weight of the current item).

If we wish to adapt this method to streams of baskets, there are two modifications we must make. The first is simple. Stream elements are baskets rather than individual items, so many items may appear at a given stream element. Treat each of those items as if they were the "current" item and add 1 to their score after multiplying all current scores by $1-c$, as described in Section 4.7.3. If some items in a basket have no current score, initialize the scores of those items to 1.

The second modification is trickier. We want to find all frequent itemsets, not just singleton itemsets. If we were to initialize a count for an itemset whenever we saw it, we would have too many counts. For example, one basket of 20 items has over a million subsets, and all of these would have to be initiated for one basket. On the other hand, as we mentioned, if we use a requirement above 1 for initiating the scoring of an itemset, then we would never get any itemsets started, and the method would not work.

A way of dealing with this problem is to start scoring certain itemsets as soon as we see one instance, but be conservative about which itemsets we start. We may borrow from the A-Priori trick, and only start an itemset I if all its immediate proper subsets are already being scored. The consequence of this restriction is that if I is truly frequent, eventually we shall begin to count it, but we never start an itemset unless it would at least be a candidate in the sense used in the A-Priori Algorithm.

Example 6.12: Suppose I is a large itemset, but it appears in the stream periodically, once every $1/2c$ baskets. Then its score, and that of its subsets, never falls below $e^{-1/2}$, which is greater than $1/2$. Thus, once a score is created for some subset of I, that subset will continue to be scored forever. The first time I appears, only its singleton subsets will have scores created for them. However, the next time I appears, each of its doubleton subsets will commence scoring, since each of the immediate subsets of those doubletons is already being scored. Likewise, the kth time I appears, its subsets of size $k-1$ are all being scored, so we initiate scores for each of its subsets of size k. Eventually, we reach the size $|I|$, at which time we start scoring I itself. □

6.5.3 Hybrid Methods

The approach of Section 6.5.2 offers certain advantages. It requires a limited amount of work each time a stream element arrives, and it always provides an up-to-date picture of what is frequent in the decaying window. Its big disadvantage is that it requires us to maintain scores for each itemset with a score of at least 1/2. We can limit the number of itemsets being scored by increasing the value of the parameter c. But the larger c is, the smaller the decaying window is. Thus, we could be forced to accept information that tracks the local fluctuations in frequency too closely, rather than integrating over a long period.

We can combine the ideas from Sections 6.5.1 and 6.5.2. For example, we could run a standard algorithm for frequent itemsets on a sample of the stream, with a conventional threshold for support. The itemsets that are found frequent by this algorithm will be treated as if they all arrived at the current time. That is, they each get a score equal to a fixed fraction of their count.

More precisely, suppose the initial sample has b baskets, c is the decay constant for the decaying window, and the minimum score we wish to accept for a frequent itemset in the decaying window is s. Then the support threshold for the initial run of the frequent-itemset algorithm is bcs. If an itemset I is found to have support t in the sample, then it is initially given a score of $t/(bc)$.

Example 6.13: Suppose $c = 10^{-6}$ and the minimum score we wish to accept in the decaying window is 10. Suppose also we take a sample of 10^8 baskets from the stream. Then when analyzing that sample, we use a support threshold of $10^8 \times 10^{-6} \times 10 = 1000$.

Consider an itemset I that has support 2000 in the sample. Then the initial score we use for I is $2000/(10^8 \times 10^{-6}) = 20$. After this initiation step, each time a basket arrives in the stream, the current score will be multiplied by $1 - c = 0.999999$. If I is a subset of the current basket, then add 1 to the score. If the score for I goes below 10, then it is considered to be no longer frequent, so it is dropped from the collection of frequent itemsets. □

We do not, sadly, have a reasonable way of initiating the scoring of new itemsets. If we have no score for itemset I, and 10 is the minimum score we want to maintain, there is no way that a single basket can jump its score from 0 to anything more than 1. The best strategy for adding new sets is to run a new frequent-itemsets calculation on a sample from the stream, and add to the collection of itemsets being scored any that meet the threshold for that sample but were not previously being scored.

6.5.4 Exercises for Section 6.5

!! **Exercise 6.5.1:** Suppose we are counting frequent itemsets in a decaying window with a decay constant c. Suppose also that with probability p, a given stream element (basket) contains both items i and j. Additionally, with probability p the basket contains i but not j, and with probability p it contains j but not i. As a function of c and p, what is the fraction of time we shall be scoring the pair $\{i, j\}$?

6.6 Summary of Chapter 6

- ✦ *Market-Basket Data*: This model of data assumes there are two kinds of entities: items and baskets. There is a many–many relationship between items and baskets. Typically, baskets are related to small sets of items, while items may be related to many baskets.

- ✦ *Frequent Itemsets*: The support for a set of items is the number of baskets containing all those items. Itemsets with support that is at least some threshold are called frequent itemsets.

- ✦ *Association Rules*: These are implications that if a basket contains a certain set of items I, then it is likely to contain another particular item j as well. The probability that j is also in a basket containing I is called the confidence of the rule. The interest of the rule is the amount by which the confidence deviates from the fraction of all baskets that contain j.

- ✦ *The Pair-Counting Bottleneck*: To find frequent itemsets, we need to examine all baskets and count the number of occurrences of sets of a certain size. For typical data, with a goal of producing a small number of itemsets that are the most frequent of all, the part that often takes the most main memory is the counting of pairs of items. Thus, methods for finding frequent itemsets typically concentrate on how to minimize the main memory needed to count pairs.

- ✦ *Triangular Matrices*: While one could use a two-dimensional array to count pairs, doing so wastes half the space, because there is no need to count pair $\{i, j\}$ in both the i-j and j-i array elements. By arranging the pairs (i, j) for which $i < j$ in lexicographic order, we can store only the needed counts in a one-dimensional array with no wasted space, and yet be able to access the count for any pair efficiently.

- ✦ *Storage of Pair Counts as Triples*: If fewer than 1/3 of the possible pairs actually occur in baskets, then it is more space-efficient to store counts of pairs as triples (i, j, c), where c is the count of the pair $\{i, j\}$, and $i < j$. An index structure such as a hash table allows us to find the triple for (i, j) efficiently.

- ✦ *Monotonicity of Frequent Itemsets*: An important property of itemsets is that if a set of items is frequent, then so are all its subsets. We exploit this property to eliminate the need to count certain itemsets by using its contrapositive: if an itemset is not frequent, then neither are its supersets.

✦ *The A-Priori Algorithm for Pairs*: We can find all frequent pairs by making two passes over the baskets. On the first pass, we count the items themselves, and then determine which items are frequent. On the second pass, we count only the pairs of items both of which are found frequent on the first pass. Monotonicity justifies our ignoring other pairs.

✦ *Finding Larger Frequent Itemsets*: A-Priori and many other algorithms allow us to find frequent itemsets larger than pairs, if we make one pass over the baskets for each size itemset, up to some limit. To find the frequent itemsets of size k, monotonicity lets us restrict our attention to only those itemsets such that all their subsets of size $k - 1$ have already been found frequent.

✦ *The PCY Algorithm*: This algorithm improves on A-Priori by creating a hash table on the first pass, using all main-memory space that is not needed to count the items. Pairs of items are hashed, and the hash-table buckets are used as integer counts of the number of times a pair has hashed to that bucket. Then, on the second pass, we only have to count pairs of frequent items that hashed to a frequent bucket (one whose count is at least the support threshold) on the first pass.

✦ *The Multistage Algorithm*: We can insert additional passes between the first and second pass of the PCY Algorithm to hash pairs to other, independent hash tables. At each intermediate pass, we only have to hash pairs of frequent items that have hashed to frequent buckets on all previous passes.

✦ *The Multihash Algorithm*: We can modify the first pass of the PCY Algorithm to divide available main memory into several hash tables. On the second pass, we only have to count a pair of frequent items if they hashed to frequent buckets in all hash tables.

✦ *Randomized Algorithms*: Instead of making passes through all the data, we may choose a random sample of the baskets, small enough that it is possible to store both the sample and the needed counts of itemsets in main memory. The support threshold must be scaled down in proportion. We can then find the frequent itemsets for the sample, and hope that it is a good representation of the data as whole. While this method uses at most one pass through the whole dataset, it is subject to false positives (itemsets that are frequent in the sample but not the whole) and false negatives (itemsets that are frequent in the whole but not the sample).

✦ *The SON Algorithm*: An improvement on the simple randomized algorithm is to divide the entire file of baskets into segments small enough that all frequent itemsets for the segment can be found in main memory. Candidate itemsets are those found frequent for at least one segment. A second pass allows us to count all the candidates and find the exact collection of frequent itemsets. This algorithm is especially appropriate in a MapReduce setting.

✦ *Toivonen's Algorithm*: This algorithm starts by finding frequent itemsets in a sample, but with the threshold lowered so there is little chance of missing an itemset that is frequent in the whole. Next, we examine the entire file of baskets, counting not only the itemsets that are frequent in the sample, but also, the negative border – itemsets that have not been found frequent, but all their immediate subsets are. If no member of the negative border is found frequent in the whole, then the answer is exact. But if a member of the negative border is found frequent, then the whole process has to repeat with another sample.

✦ *Frequent Itemsets in Streams*: If we use a decaying window with constant c, then we can start counting an item whenever we see it in a basket. We start counting an itemset if we see it contained within the current basket, and all its immediate proper subsets already are being counted. As the window is decaying, we multiply all counts by $1 - c$ and eliminate those that are less than $1/2$.

6.7 References for Chapter 6

The market-basket data model, including association rules and the A-Priori Algorithm, are from [1] and [2].

The PCY Algorithm is from [4]. The Multistage and Multihash Algorithms are found in [3].

The SON Algorithm is from [5]. Toivonen's Algorithm appears in [6].

1. R. Agrawal, T. Imielinski, and A. Swami, "Mining associations between sets of items in massive databases," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 207–216, 1993.

2. R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," *Intl. Conf. on Very Large Databases*, pp. 487–499, 1994.

3. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J.D. Ullman, "Computing iceberg queries efficiently," *Intl. Conf. on Very Large Databases*, pp. 299-310, 1998.

4. J.S. Park, M.-S. Chen, and P.S. Yu, "An effective hash-based algorithm for mining association rules," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 175–186, 1995.

5. A. Savasere, E. Omiecinski, and S.B. Navathe, "An efficient algorithm for mining association rules in large databases," *Intl. Conf. on Very Large Databases*, pp. 432–444, 1995.

6. H. Toivonen, "Sampling large databases for association rules," *Intl. Conf. on Very Large Databases*, pp. 134–145, 1996.