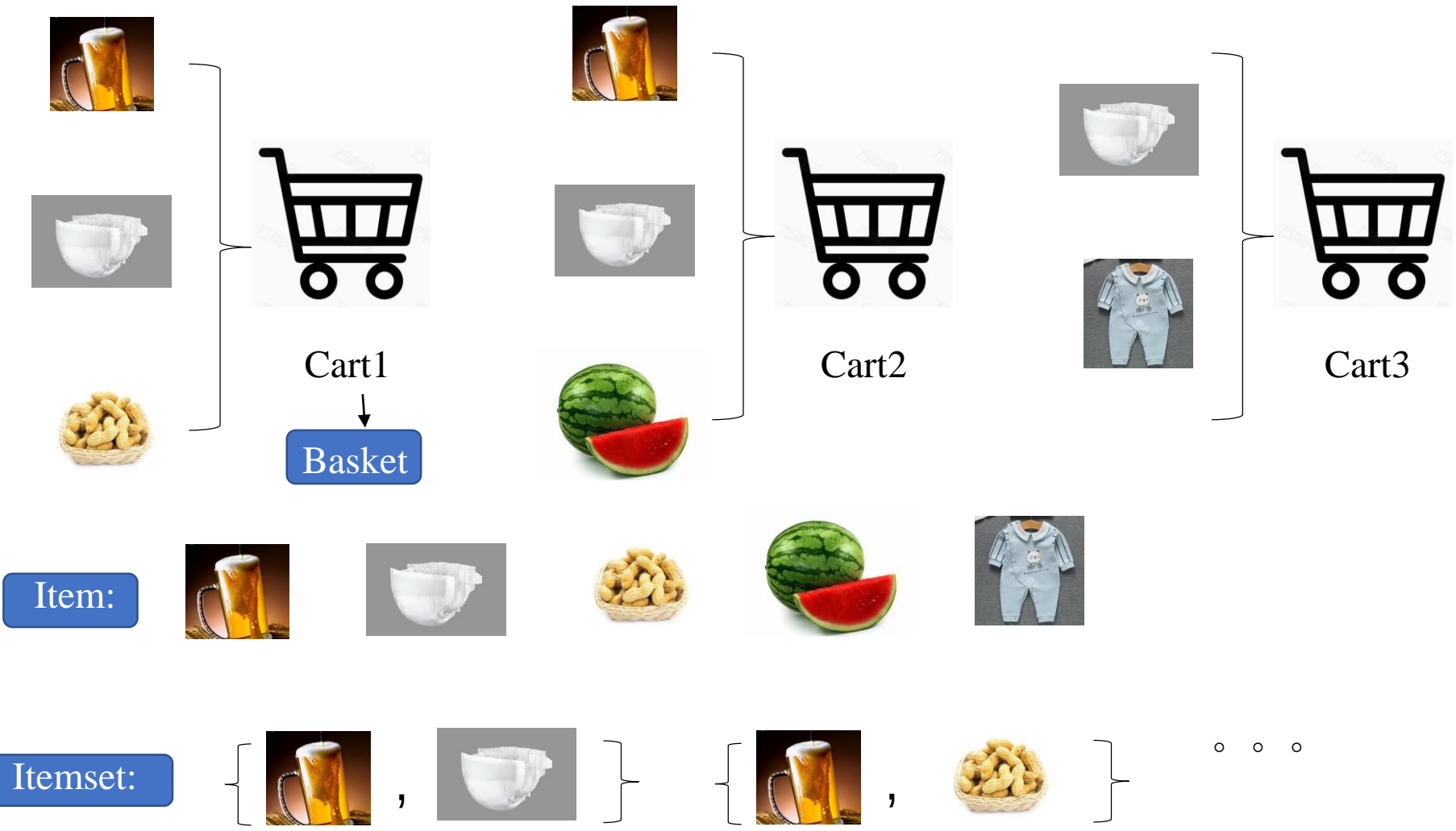


Chapter 6 Frequent Itemsets

Tianlong Zhou

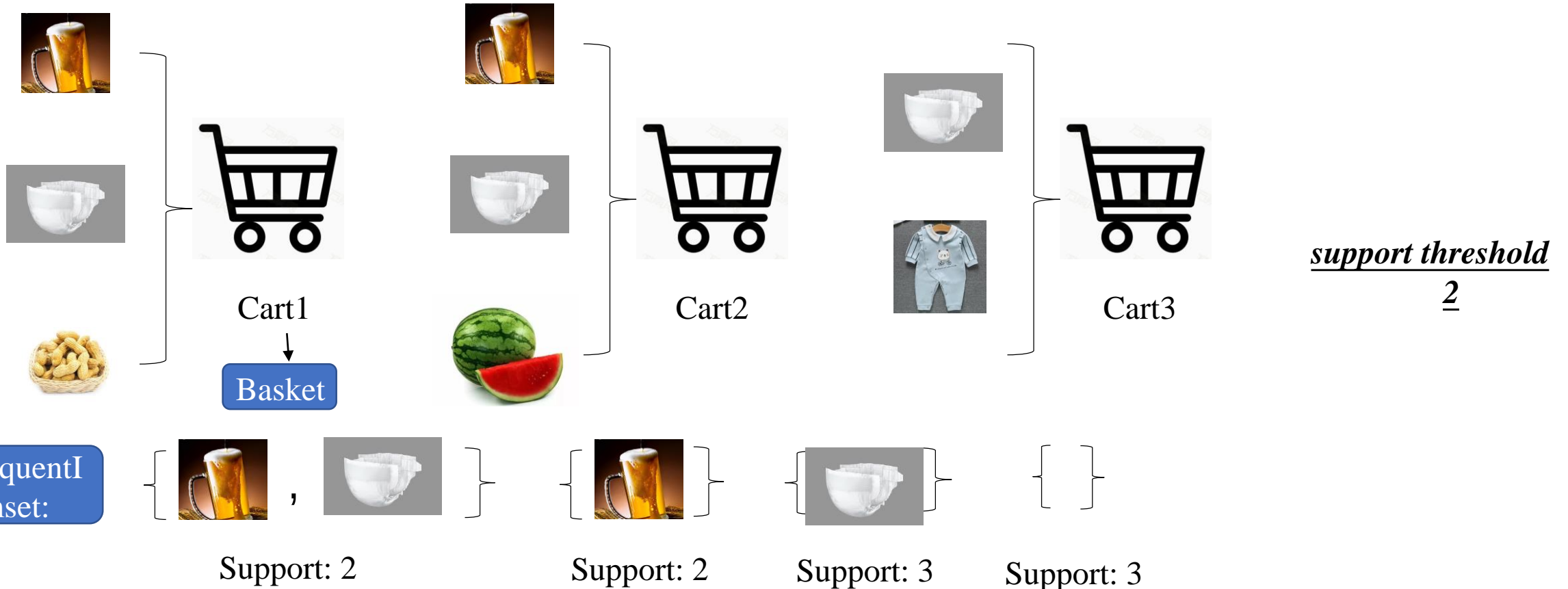
2021\08\23

Definition of Frequent Itemsets



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.



Applications of Frequent Itemsets

1. *Related concepts:*

Let items be words, and let baskets be documents. 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. 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.

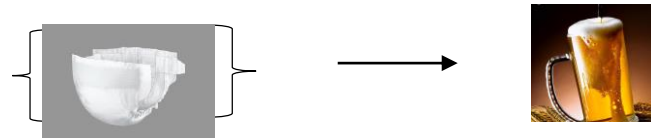
Association Rules

The form of an association rule is $I \rightarrow j$, 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:

the ratio of the support for $I \cup \{j\}$ to the support for I .



Support: 3

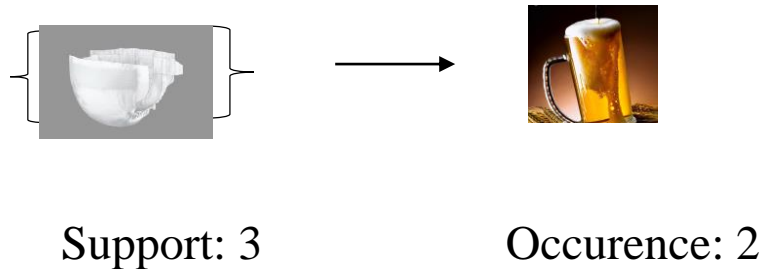
Occurence: 2

confidence = $2 / 3$

Association Rules

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.

We define the interest of an association rule $I \rightarrow j$ to be the difference between its confidence and the fraction of baskets that contain j .



$$\text{interest} = 2 / 3 - 2 / 4 = 1 / 6$$

Market Baskets and the A-Priori Algorithm

We assume that market-basket data is stored in a file basket-by-basket.

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

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.

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.

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.

Market Baskets and the A-Priori Algorithm

We assume that market-basket data is stored in a file basket-by-basket.

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

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.

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.

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.

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:

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

(即严格上三角矩阵的上三角区域进行标号，化二维为一维。)

The Triples Method

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.

$[i, j, c]$




$H(i, j) = c$

Monotonicity of Itemsets

Monotonicity:

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

Frequent Itemset: {  ,  }



Frequent Itemset: {  } {  } { }

The A-Priori Algorithm

For the moment, let us concentrate on finding the frequent pairs only. 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 .

The other table is an array of counts; the i th array element counts the occurrences of the item numbered i . Initially, the counts for all the items are 0.

The A-Priori Algorithm

For the moment, let us concentrate on finding the frequent pairs only. 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.

The other table is an array of counts; the i th array element counts the occurrences of the item numbered i . Initially, the counts for all the items are 0.

Item	beer
Number	1

Table 1

Number	1
Count	0

Table 2

The A-Priori Algorithm

Between the Passes of A-Priori

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*.

Number	1	2	3	4
Count	0	2	1	3

Table 2



Number	1	2	3	4
F	0	1	0	2

Frequent-items table

support threshold
2

The A-Priori Algorithm

The Second Pass of A-Priori

During the second pass, we count all the pairs that consist of two frequent items.

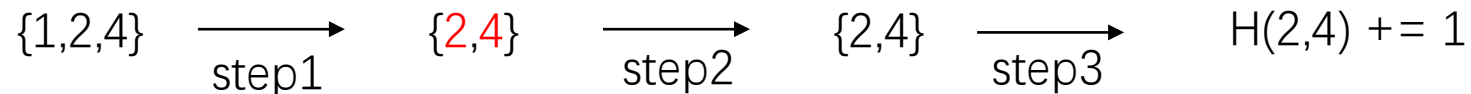
A pair cannot be frequent unless both its members are frequent. (**Monotonicity**)

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.

Number	1	2	3	4
F	0	1	0	2

Frequent-items table



A-Priori for All Frequent Itemsets

In 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. (**step2**)
2. L_k is the set of truly frequent itemsets of size k . (**step3**)

The pattern of moving from one set to the next and one size to the next is suggested by:

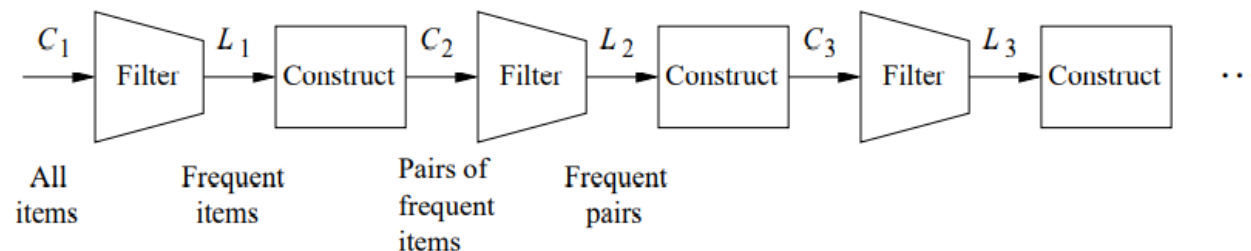
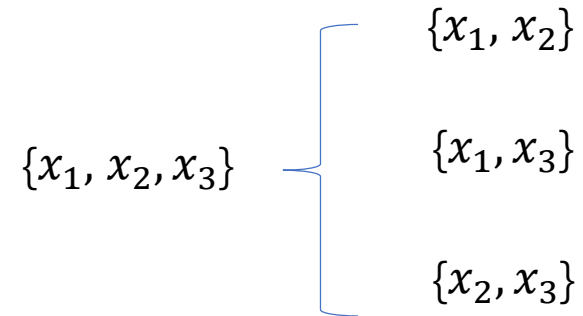


Figure 6.4: The A-Priori Algorithm alternates between constructing candidate sets and filtering to find those that are truly frequent

A-Priori for All Frequent Itemsets

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



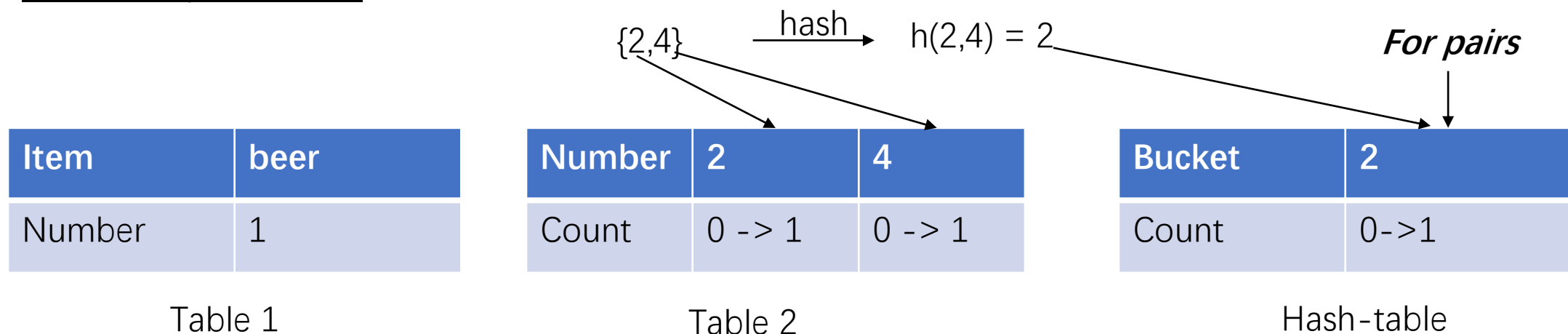
1. Find L_k by making a pass through the baskets and counting all and only the itemsets of size k that are in C_k . Those itemsets that have count at least s are in L_k .

The Algorithm of Park, Chen, and Yu(PCY)

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 .

The First Pass of PCY

As we examine a basket during the first pass, we not only add 1 to the count for each item in the basket, but 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.



The Algorithm of Park, Chen, and Yu(PCY)

The Second Pass of PCY

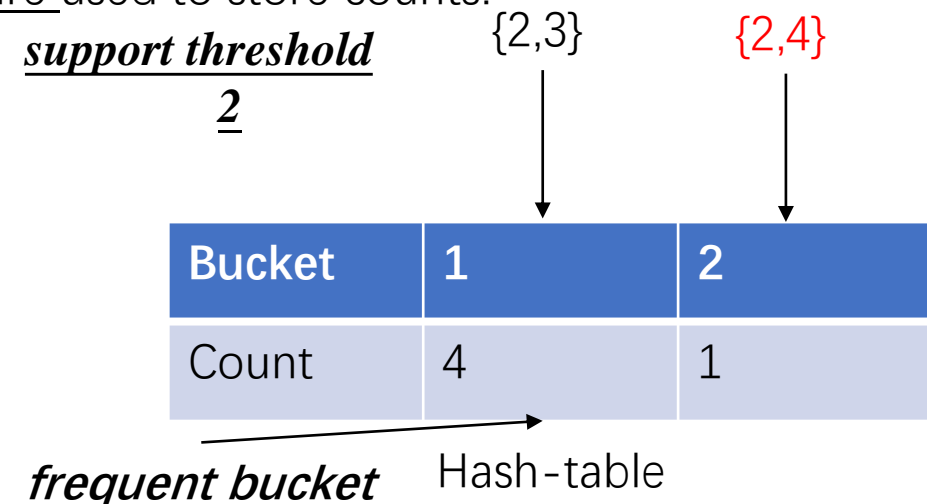
We can define the set of candidate pairs C_2 to be those pairs $\{i, j\}$ such that:

1. i and j are frequent items.
2. $\{i, j\}$ hashes to a frequent bucket(the count is not smaller than support threshold)[**distinguishes PCY from A-Priori**]

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

Number	1	2	3	4
F	0	2	2	3

Frequent-items table



The Algorithm of Park, Chen, and Yu(PCY)

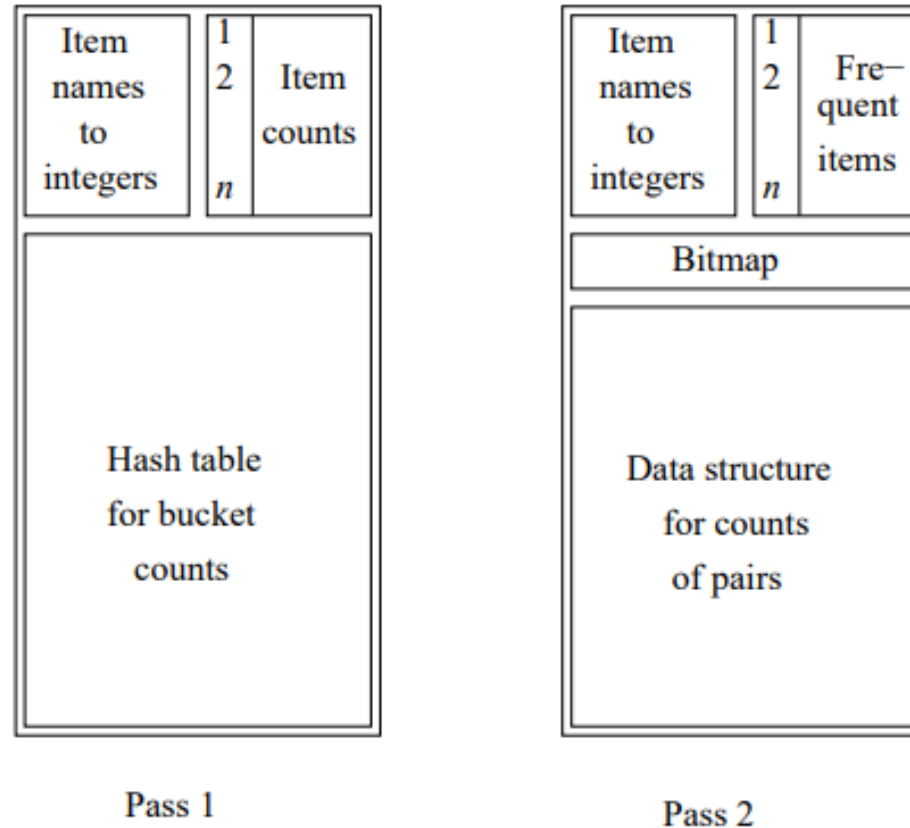


Figure 6.5: Organization of main memory for the first two passes of the PCY Algorithm

The Multistage Algorithm

Pass 1:

The first pass of Multistage is the same as the first pass of PCY.

Pass 2:

During the second pass, we hash certain pairs of items to buckets of *the second hash table*.

Pass 3:

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*.

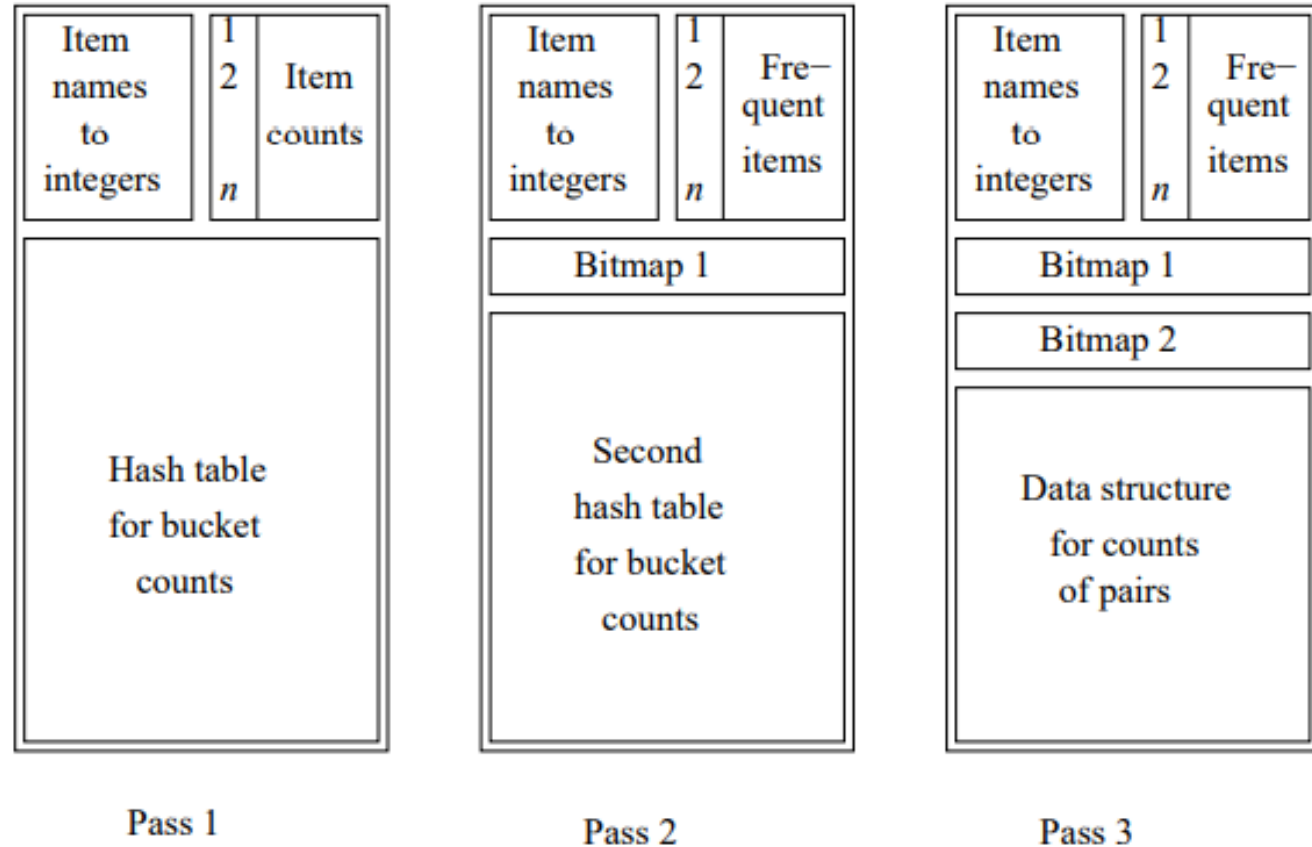


Figure 6.6: The Multistage Algorithm uses additional hash tables to reduce the number of candidate pairs

The Multihash Algorithm

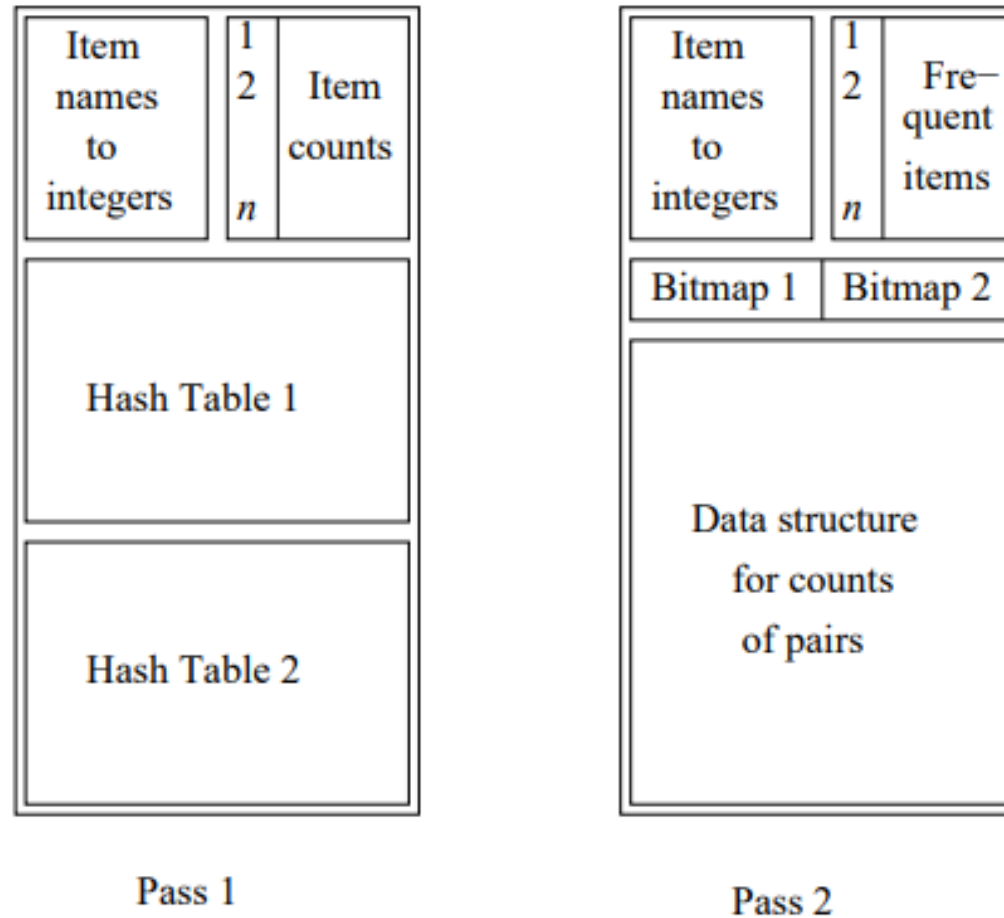


Figure 6.7: The Multihash Algorithm uses several hash tables in one pass

Limited-Pass Algorithms

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.

In this section we explore some algorithms that have been proposed to find all or most frequent itemsets using at most two passes.

The Simple, Randomized Algorithm(SRA)

Key:

Instead of using the entire file of baskets, we **could pick a random subset** of the baskets and pretend it is the entire dataset.

Sampling Methods:

1. 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.
2. 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.
3. If the file is part of a distributed file system, we can pick some chunks at random to serve as the sample

The Simple, Randomized Algorithm

Calculate Frequent Itemsets :

Threshold: p_s

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.

Avoiding Errors in Sampling Algorithms

False Negative:

An itemset that is frequent in the whole but not in the sample is a *false negative*

Solution:

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 sss. But a false negative is not counted and therefore remains undiscovered.

False Positive:

an itemset that is frequent in the sample but not the whole is a *false positive*.

Solution:

We cannot eliminate false negatives completely, but we can reduce their number if the amount of main memory allows it by reducing threshold for the sample.

The Algorithm of Savasere, Omiecinski, and Navathe(SON)

SON Algorithm avoids both false negatives and false positives, at the cost of making two full passes.

- Pass1**
- Step1:**
divide the input file into chunks.
 - Step2:**
Treat each chunk as a sample, and run the SRA algorithm 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.
 - Step3:**
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. ***we can be sure that all the truly frequent itemsets are among the candidates; i.e., there are no false negatives.***
- Pass2**
- Step4:**
In a second pass, we count all the candidate itemsets and select those that have support at least **s** as the frequent itemsets.

Toivonen's Algorithm

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*.

Step1:

select a small sample of the input dataset, and finding from it the candidate frequent itemsets.

Step2:

construct the negative border.

Negative Border:

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:

Suppose the items are {A, B, C, D, E}. Frequent Itemsets: {A}, {B}, {C}, {D}, {B, C}, {C, D}.

Then {E} and {A,B} are in the negative border. {A, E} is not in the negative border, because {E} is not a frequent itemset.

Toivonen's Algorithm

Step3:

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.

Why?

Hint: 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.

Counting Frequent Items in a Stream

In what follows, we shall assume that stream elements are baskets of items.

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.

Sampling Methods for Streams

the simplest approach to maintaining a current estimate of the frequent itemsets in a stream is to **(step1)** collect some number of baskets and store it as a file. **(step2)** 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.

(Step3)Then(Optional):

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.
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. **(Drop)** 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.

Sampling Methods for Streams

(For Step3.2)

We should also allow some way for new frequent itemsets to be added to the current collection.

1. 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.
2. 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.

Sampling Methods for Streams

(For Step3.2)

We should also allow some way for new frequent itemsets to be added to the current collection.

1. 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.
2. 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.