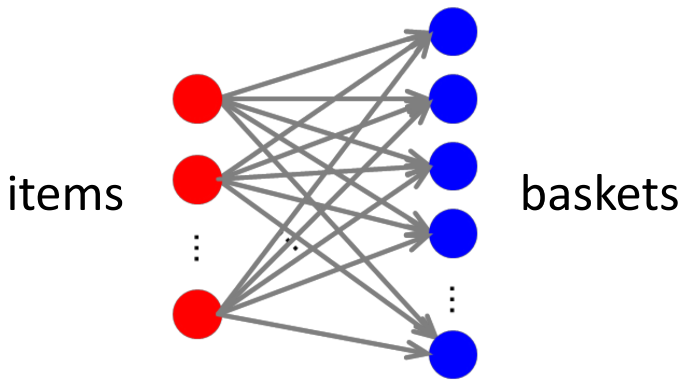
Chapter 6

Frequent Itemsets

Market-Basket Model

The Discovery of Frequent Itemsets

- Often viewed as the discovery of "***association rules***"
  - Although the latter is a more complex characterization of data
  - Because it depends fundamentally on the discovery of frequent itemsets
- The "***market-basket***" model of data
  - Essentially a <u>many-to-many relationship</u> between "items" and "baskets"



- The "***frequent-itemsets***" problem
  - Find sets of items that appear in (are related to) many of the same baskets

Algorithms for Finding Frequent Itemsets

- A-Priori algorithm
  - Eliminates 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***
- Various improvements to the basic A-Priori idea
  - Concentrating on ***very large data sets*** that stress the available main memory
- Approximate algorithms
  - Work faster but are not guaranteed to find all frequent itemsets
  - Also in this class are those that exploit parallelism (e.g., MapReduce)

The Market-Basket Model

- Used to describe a common form of many-to-many relationship between two kinds of objects
  - ***Items*** (e.g., things sold in a supermarket)
  - ***Baskets*** (sometimes called "transactions")
  - Each basket consists of a set of items (an ***itemset***)
    - Usually we assume that the number of times in a basket is ***small***
  - The number of baskets is very ***large***
    - Bigger than what can fit in main memory

TID	*Items*
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Input

Association rules discovered:

{Milk} → {Coke}
{Diaper, Milk} → {Beer}

Output

Definition of Frequent Itemsets

- Frequent itemsets
  - A set of items that appears in ***many*** baskets

- Formal definition
  - The ***support*** for an itemset I: the number of baskets for which I is a subset
  - We are given a support threshold s
  - We say I is ***frequent*** if its support $\geq s$

TID	*Items*
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

⇨ The support for $\{\text{Beer, Bread}\} = 2$

(Ex) Frequent Itemsets

- Items = {milk, coke, pepsi, beer, juice}

- Support threshold = 3

$$B_1 = \{\text{m, c, b}\} \qquad B_2 = \{\text{m, p, j}\}$$
$$B_3 = \{\text{m, b}\} \qquad B_4 = \{\text{c, j}\}$$
$$B_5 = \{\text{m, p, b}\} \qquad B_6 = \{\text{m, c, b, j}\}$$
$$B_7 = \{\text{c, b, j}\} \qquad B_8 = \{\text{b, c}\}$$

- Frequent itemsets (i.e., support ≥ 3)

$$\{\text{m}\}, \{\text{c}\}, \{\text{b}\}, \{\text{j}\}, \{\text{m, b}\}, \{\text{b, c}\}, \{\text{c, j}\}$$

Applications of Frequent Itemsets (1/3)

- The original application: analysis of true market baskets
  - **Items**: the different products that the store sells
  - **Baskets**: the sets of items in a single market basket
  - **Application**
    - A major chain might sell 10^5 different items and have millions of baskets
    - By finding frequent itemsets, a retailer can learn what is commonly bought together
    - Especially important are the ***unexpected*** sets of items bought together
      - (ex) $\{\text{Bread, Milk}\}$ (X), $\{\text{Diaper, Beer}\}$ (O)
    - It offers the supermarket an opportunity to do some clever marketing
      - (ex) Advertise a sale on diapers and raise the price of beer
      - (ex) Place diapers and beer close together

Applications of Frequent Itemsets (2/3)

- Related concepts
  - **Items:** words
  - **Baskets:** documents (e.g., Web pages, blogs, tweets)
  - **Application**
    - Words appearing together in many documents may be a joint concept
      - (ex) {Brad, Angelina}
- Plagiarism
  - **items:** documents
  - **Baskets:** sentences
    - Note that an item is “in” a basket if the sentence is in the document
  - **Application**
    - Documents appearing together in many baskets share many sentences in common → plagiarism!

Applications of Frequent Itemsets (3/3)

- Drug side effects
  - **Items:** drugs and side-effects
  - **Baskets:** patients
  - **Application**
    - Can detect combinations of drugs that result in particular side effects
    - But requires extension!
      - ***Absence*** of an item needs to be observed as well as presence

Association Rules

- Represented as if-then rules about the contents of baskets

- The form of an association rule

$$\boxed{I \rightarrow j}$$

  - $I = \{i_1, i_2, \ldots, i_k\}$: a set of times (i.e., itemset)
  - j: an item

- Implication
  - If all of the items in I appear in some basket, then j is "likely" to appear in that basket as well

The Confidence of an Association Rule

- Formalizes the notion of "likely"

- Definition
  - Let $\mathrm{conf}(I \rightarrow j)$ be the confidence of the association rule $I \rightarrow j$
    - $I = \{i_1, i_2, \ldots, i_k\}$: an itemset
  - Let $\mathrm{support}(I)$ be the support for an itemset I
  - Then, $\mathrm{conf}(I \rightarrow j)$ is defined as follows:

$$\boxed{\mathrm{conf}(I \rightarrow j) = \frac{\mathrm{support}(I \cup \{j\})}{\mathrm{support}(I)}}$$

  - That is, the confidence of $I \rightarrow j$ is the probability of j given I

The Interest of an Association Rule

- Not all high-confidence rules are interesting
  - If everyone purchases *milk*, $X \rightarrow milk$ will have 100% confidence for any X
  - A rule $I \rightarrow j$ is more valuable if it reflects a true relationship
    - I somehow affects j

- Definition
  - The interest of a rule $I \rightarrow j$, $\text{interest}(I \rightarrow j)$, is defined as follows:

$$\boxed{\text{interest}(I \rightarrow j) = \text{conf}(I \rightarrow j) - \frac{\text{support}(\{j\})}{\#\ of\ baskets}}$$

  - $\text{interest}(I \rightarrow j) = 0 \rightarrow$ I has no influence on j
  - $\text{interest}(I \rightarrow j) > 0 \rightarrow$ the presence of I causes the presence of j
  - $\text{interest}(I \rightarrow j) < 0 \rightarrow$ the presence of I discourages the presence of j

(Ex) Confidence and Interest

$$B_1 = \{\mathrm{m, c, b}\} \qquad B_2 = \{\mathrm{m, p, j}\}$$
$$B_3 = \{\mathrm{m, b}\} \qquad B_4 = \{\mathrm{c, j}\}$$
$$B_5 = \{\mathrm{m, p, b}\} \qquad B_6 = \{\mathrm{m, c, b, j}\}$$
$$B_7 = \{\mathrm{c, b, j}\} \qquad B_8 = \{\mathrm{b, c}\}$$

- Consider the association rule $\{\mathrm{m, b}\} \rightarrow \mathrm{c}$
  - $\mathrm{conf}(\{\mathrm{m, b}\} \rightarrow \mathrm{c}) = 2/4 = 0.5$
  - $\mathrm{interest}(\{\mathrm{m, b}\} \rightarrow \mathrm{c}) = 0.5 - 5/8 = 0.5 - 0.625 = -0.125$
    - Item c appears in 5/8 of the baskets
    - Thus, the rule is not very interesting!

Finding Association Rules (1/2)

- Problem definition
  - Find all association rules $I \rightarrow j$ with $\text{support}(I \cup \{j\}) \geq s$ and $\text{conf}(I \rightarrow j) \geq c$
    - s must be reasonably high (in practice, often around 1% of the baskets)
    - c must also be reasonably high (perhaps 50%)

- Suppose we have found all itemsets with support $\geq s$
  - We also have the exact support for each of these itemsets

- We then can ***easily*** find within them all the association rules that have both high support and high confidence

Finding Association Rules (2/2)

- Let J be a set of n items that is found to be frequent
  - Then there are only n possible rules, namely $J - \{j\} \rightarrow j$ for each j in J

- For every element j of J, generate a rule $J - \{j\} \rightarrow j$
  - Since $\text{support}(J) \geq s$, it is obvious that $\text{support}(J - \{j\} \cup \{j\}) \geq s$
  - Thus, we only need to check if $\text{conf}(J - \{j\} \rightarrow j) \geq c$
  - We can easily compute $\text{conf}(J - \{j\} \rightarrow j) = \text{support}(J)/\text{support}(J - \{j\})$
    - Because J is frequent, $J - \{j\}$ must be at least as frequent
    - Thus we already have $\text{support}(J)$ and $\text{support}(J - \{j\})$
  - Output the rule $J - \{j\} \rightarrow j$ if $\text{conf}(J - \{j\} \rightarrow j) \geq c$

- The ***hard*** part is finding all frequent itemsets! → explained next

(Ex) Finding Association Rules

$B_1 = \{\text{m, c, b}\}$ $B_2 = \{\text{m, p, j}\}$

$B_3 = \{\text{m, b}\}$ $B_4 = \{\text{c, j}\}$

$B_5 = \{\text{m, p, b}\}$ $B_6 = \{\text{m, c, b, j}\}$

$B_7 = \{\text{c, b, j}\}$ $B_8 = \{\text{b, c}\}$

- Let support threshold $s = 3$ and confidence threshold $c = 0.75$

1. Find frequent itemsets:
   - {b, m}, {b, c}, {c, m}, {c, j}, {m, c, b}
2. Find association rules:

~~{b} → m: $c = 4/6$~~
{m} → b: $c = 4/5$

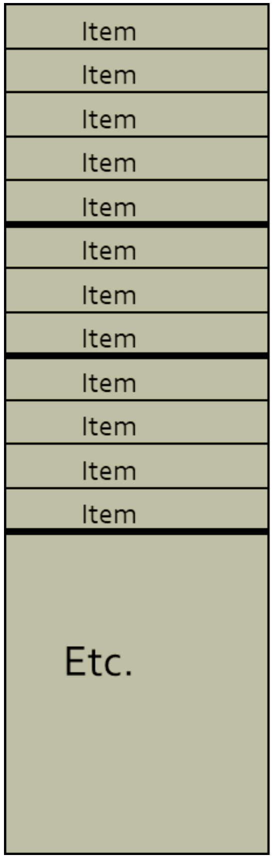
{b} → c: $c = 5/6$
~~{c} → b: $c = 3/5$~~

~~{b, c} → m: $c = 3/5$~~
{b, m} → c: $c = 3/4$
…

A-Priori Algorithm

A-Priori Algorithm

- The original algorithm to find association rules with high support and confidence
  - From which many variants have been developed
- Assumption: representation of market-basket data
  - Market-basket data is stored in a file basket-by-basket
  - Each item is represented by an ***integer*** code
    - (ex) `{23,456,1001}{3,18,92,145}{...`
  - The average size of a basket is small
  - However, the size of the file of baskets is very large
    - Thus, it does not fit in main memory



The Cost of An Association Rule Algorithm

- A major cost of an association rule algorithm is the time it takes to ***read the baskets from disk***
  - Because the file of baskets is too large to fit in main memory

- Once a basket is in main memory, generating all the subsets of size k in main memory should take time much less than the time it took to read the basket from disk
  - (ex) if there are 20 items in a basket, there are ${}_{20}C_2$ = 190 subsets of size 2
  - As k gets larger, the time required to generates all the subsets of size k for a basket with n items grows larger
  - However, it is usually possible to eliminate many of the items in each basket as not able to participate in a frequent itemset, so n drops as k increases

The Cost of An Association Rule Algorithm

- Consequently, the work of examining each of the baskets can usually be assumed proportional to the size of the file

- In practice, association rule algorithms read the data in ***passes***
  - All baskets read in turn (sequentially)

- Thus, the running time of an association rule algorithm is proportional to:
  - ***The number of passes*** taken by the algorithm × the size of the file

- Thus, what does matter is ***the number of passes***

Main Memory Bottleneck

- For many association rule algorithms, ***main memory*** is the critical resource
  - For example, we need to ***count*** the occurrence of each pair of items
    - (ex) {Bread,Milk}:10, {Milk,Beer}:8, {Coke,Beer}:4, …
  - However, the number of pairs of items can be very large
    - For n items, the number of all possible pairs is ${}_nC_2$
  - If we do not have enough main memory to store each of the counts, then the algorithm will ***thrash***
    - Because adding 1 to a random count will most likely require us to load a page from disk
- Thus, each algorithm has a ***limit*** on how many items it can deal with

Finding Frequent Pairs

- The hardest problem often turns out to be finding the frequent ***pairs*** of items $\{i_1, i_2\}$
  - Why?
    - Frequent pairs are common, frequent triples are rare
    - Probability of being frequent drops exponentially with size
- Thus, let's first concentrate on pairs, then extend to larger sets
- The approach:
  - We always need to generate all the itemsets
  - But we would only like to count (keep track) of those itemsets that in the end turn out to be frequent

Counting Pairs in Memory

- For n items, the number of pairs is ${}_nC_2 = n(n-1)/2$
  - Suppose $n = 10^5$ and counts are 4 byte integers
  - Naively, $4{\times}10^5(10^5 - 1)/2 \approx 2{\times}10^{10}$ bytes $=$ 20 GB of memory needed

- How can we reduce the amount of memory required to count?
  - Triangular matrix method
    - Store counts in a one-dimensional triangular array
  - Triple method
    - Store counts as triples $[i, j, c]$, meaning the count of pair $\{i, j\}$ $(i < j)$ is c

Triangular Matrix Method

- We could use a two-dimensional array a
  - We store the count of a pair $\{i, j\}$ in $a[i, j]$
  - However, this strategy makes half the array useless because $\{i, j\} = \{j, i\}$

- A more space-efficient way
  - Use a ***one-dimensional triangular array***
  - We store in $a[k]$ the count of a pair $\{i, j\}$ $(1 \leq i < j \leq n)$, where

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

  - In this layout, the pairs are stored in lexicographic order
    - $\{1, 2\}, \{1, 3\}, \ldots, \{1, n\}, \{2, 3\}, \{2, 4\}, \ldots, \{2, n\}, \{3, 4\}, \{3, 5\}, \ldots, \{3, n\}, \ldots, \{n-2, n-1\}, \{n-2, n\}, \ldots, \{n-1, n\}$

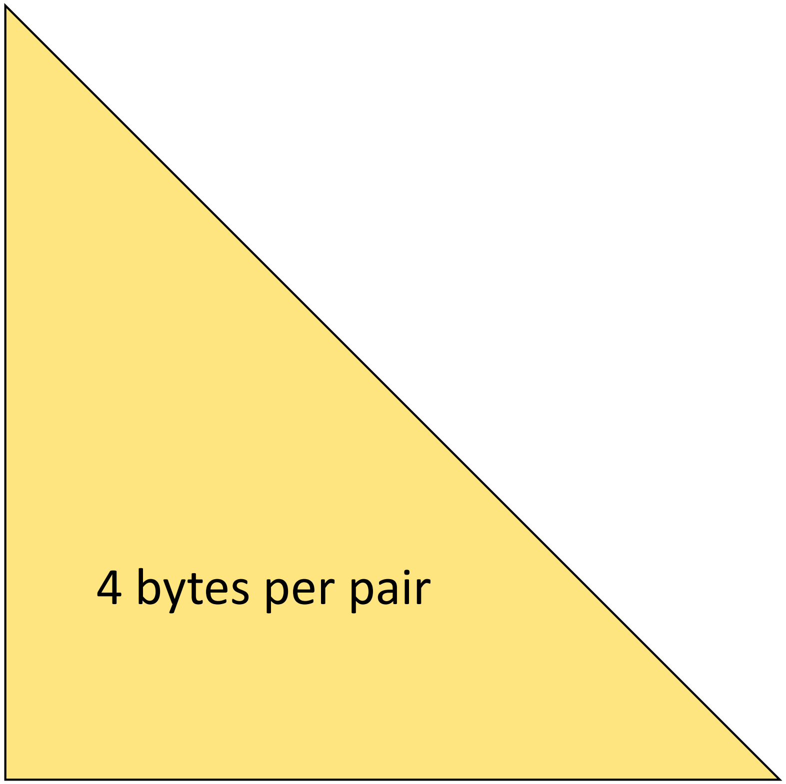
Triple Method

- The triangular matrix method requires us to store the count of a pair even when the count is 0
  - Inefficient when the counts of most pairs are 0

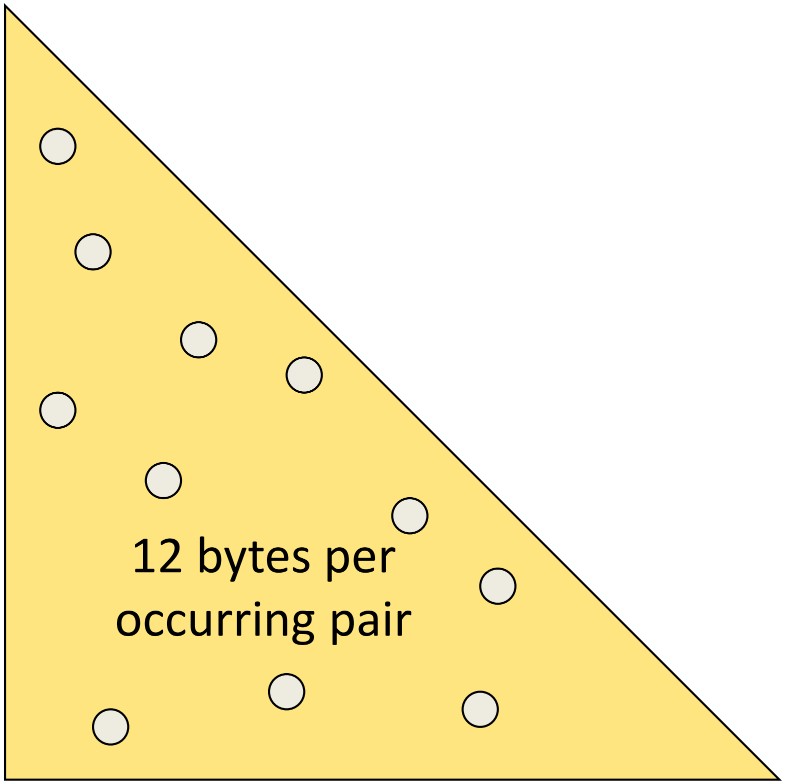
- Thus, we store triples $[i, j, c]$ only for pairs $\{i, j\}$ $(i < j)$ with $c > 0$
  - Also, we also use a hash table with i and j as the search key to find a triple $[i, j, c]$ quickly

- Comparisons
  - Triangular matrix method: uses **4 bytes** per pair (but for all pairs)
  - Triple method: uses **4+4+4=12 bytes** per pair (but only for pairs with $c > 0$)
  - Therefore, the triangular matrix will be better if at least 1/3 of ${}_nC_2$ possible pairs actually appear in the basket data

Comparing the Two Approaches



Triangular Matrix



Triples

Monotonicity of Itemsets

- If we have too many items so the pairs do not fit into memory, can we do better?

- The key observation driving much of the effectiveness of the algorithms we shall discuss

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

- Proof
  - For $J \subseteq I$, 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
  - Consequently, if the count for I is at least s, then count for J is at least s

Maximal Frequent Itemsets

- Monotonicity offers us a way to compact the information about frequent itemsets

- We say an itemset is ***maximal*** if no superset is frequent
  - (ex) $\text{support}(\{a\}) > s$ and $\text{support}(\{a, b\}) > s$ → $\{a\}$ is ***not*** maximal

- Why useful?
  - If we list only the maximal itemsets, then we know that ***all*** of their subsets are frequent
  - No set that is not a subset of some maximal itemset can be frequent
  - Thus, we can reduce the number of output itemsets
    - (ex) $\{a\}, \{b\}, \{a, b\}$ → $\{a, b\}$

(Ex) Maximal Frequent Itemsets

	Support	Maximal (s=3)
{a}	4	No
{b}	5	No
{c}	3	No
{a, b}	4	Yes
{a, c}	2	No
{b, c}	3	Yes
{a, b, c}	2	No

Frequent, but superset {b, c} also frequent

Frequent, and its only superset, {a, b, c} is not frequent

Tyranny of Counting Pairs

- In practice, the most main memory is required for determining the frequent ***pairs***
  - The number of items n is rarely so large so we can count them in memory
  - The number of pairs can be very large
  - The number of ***frequent*** triples, quadruples, and larger sets are ***rare***
    - # of frequent pairs > # of frequent triples > # of frequent quadruples > …
- Thus, we first concentrate on algorithms for computing ***frequent pairs***, then extend to larger sets
  - 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 effective in doing so

The A-Priori Algorithm

- Naïve approach (if we have enough memory)
  - Read the file of baskets in a single pass
  - For each basket of n items, generate all the ${}_nC_2$ pairs by a double loop
  - Each time we generate a pair, add 1 to its count
  - At the end, examine all pairs and output those pairs whose count > s
  - However, this approach fails if there are too many pairs of items to count them all in main memory

- A-Priori algorithm
  - Designed to reduce the number of pairs that must be counted
  - To do so, it performs ***two*** passes over data, rather than one pass

The First Pass of A-Priori

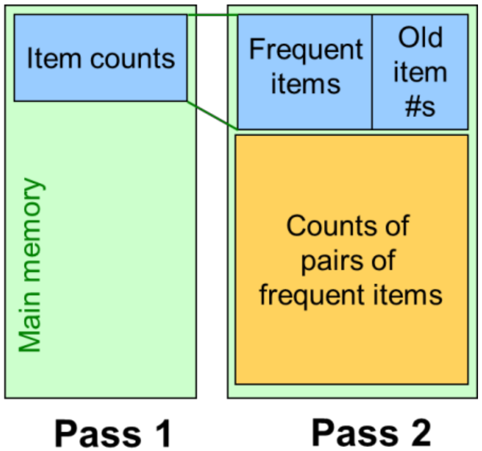
- We create two tables
  - The first table: translates item names into integers from 1 to n
  - The other table: an array of counts
    - The ith array element counts the occurrences of the item numbered i
    - Initially, the counts for all the items are 0
- As we read baskets
  - We look at each item in the basket and translate its name into an integer
  - Next, we use that integer to index into the array of counts
  - We add 1 to the integer found there

✓ Requires only memory proportional to the number of items

Between the Passes of A-Priori

- We examine the counts of items to determine which of them are ***frequent***

- 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
  - 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-item table***



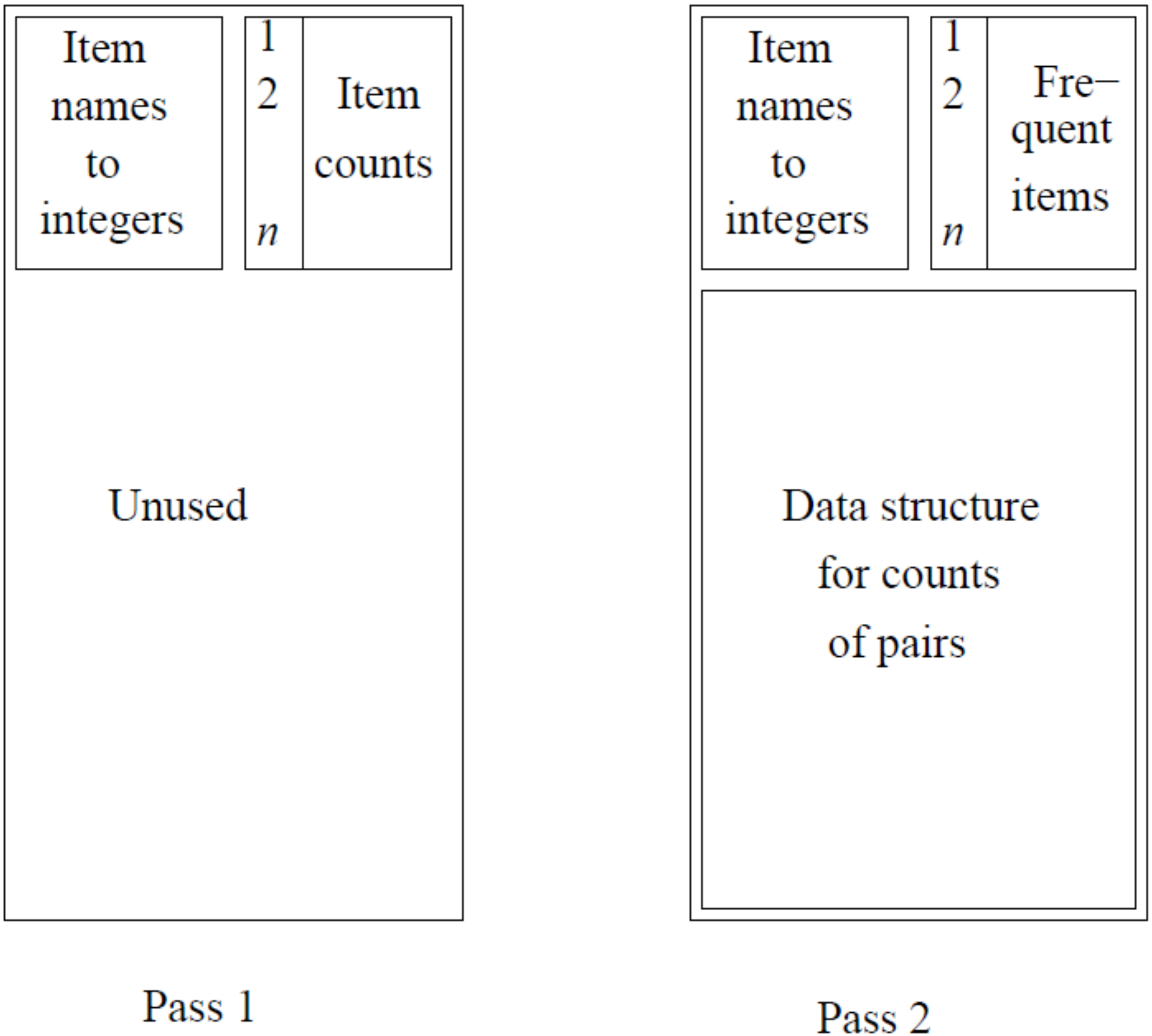
The Second Pass of A-Priori

- We count all the pairs that consists of two ***frequent*** items
  - Recall that a pair cannot be frequent unless both its items are frequent
  - If we use the triangular-matrix method
    - The space required is $2m^2$ bytes, rather than $2n^2$ bytes
  - If we use the triples method
    - The renumbering of just the frequent items is necessary

- The mechanics
  - For each basket, look in the frequent-items table to find frequent items
  - In a double loop, generate all pairs of frequent items in that basket
  - For each such pair, add 1 to its count
  - Finally, examine the counts to determine which pairs are frequent

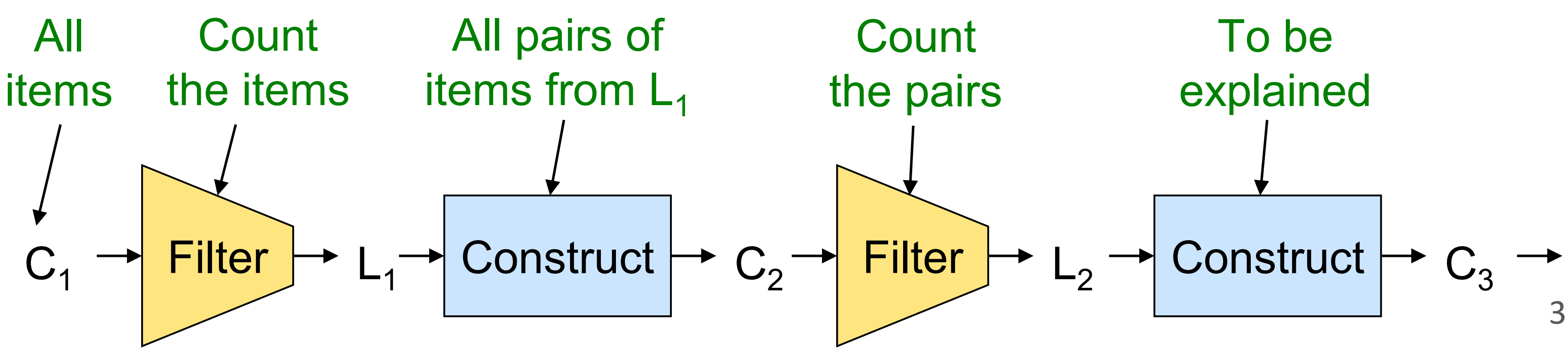
✓ Requires memory proportional to square of ***frequent*** items only

Main Memory Use of A-Priori Algorithm



A-Priori for All Frequent Itemsets

- We can use the ***same*** technique to find larger frequent itemsets without an exhaustive count of all sets
- For each size $k = 1, 2, \ldots,$ we construct two sets of itemsets
  - C_k = the set of candidate itemsets of size k
    - The itemsets we must count in order to determine whether they are frequent
  - L_k = the set of truly frequent itemsets of size k
  - If $L_k = \varnothing$, then we stop
    - By monotonicity, there can be no larger frequent itemsets



Constructing C_k from $L_{k\text{-}1}$

- Step 1: Join
  - Generate C_k by merging a pair of frequent itemsets in $L_{k\text{-}1}$ if their $(k-2)$ items are identical
- Step 2: Prune
  - Delete all itemsets $I \in C_k$ if some $(k-1)$-subset of I is not in $L_{k\text{-}1}$
    - Because all $(k-1)$-subsets of I must be frequent if I is frequent
    - In other words, C_k is all those itemsets of size k, every $k-1$ of which is an itemset in $L_{k\text{-}1}$
- Example
  - Let $L_3 = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{1, 3, 5\}, \{2, 3, 4\}\}$
  - After the join step, $C_4 = \{\{1, 2, 3, 4\}, \{1, 3, 4, 5\}\}$
  - After the prune step, $C_4 = \{\{1, 2, 3, 4\}\}$ ($\because \{1, 4, 5\}$ is not in L_3)

Constructing L_k from C_k

- Make 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
- If we find $L_k = \emptyset$, then we stop

(Ex) A-Priori for All Frequent Itemsets

① Let $C_1 = \{\{\text{b}\}, \{\text{c}\}, \{\text{j}\}, \{\text{m}\}, \{\text{n}\}, \{\text{p}\}\}$

② Count the support of itemsets in C_1

③ Prune non-frequent: $L_1 = \{\{\text{b}\}, \{\text{c}\}, \{\text{j}\}, \{\text{m}\}\}$

④ Construct $C_2 = \{\{\text{b, c}\}, \{\text{b, j}\}, \{\text{b, m}\}, \{\text{c, j}\}, \{\text{c, m}\}, \{\text{j, m}\}\}$

⑤ Count the support of itemsets in C_2

⑥ Prune non-frequent: $L_1 = \{\{\text{b, m}\}, \{\text{b, c}\}, \{\text{c, m}\}, \{\text{c, j}\}\}$

⑦ Construct $C_3 = \{\{\text{b, m, c}\}, \cancel{\{\text{b, c, j}\}}, \cancel{\{\text{b, m, j}\}}, \cancel{\{\text{c, m, j}\}}\}$

⑧ Count the support of itemsets in C_3

⑨ Prune non-frequent: $L_3 = \{\{\text{b, m, c}\}\}$

Handling Larger Datasets

Handling Larger Datasets in Main Memory

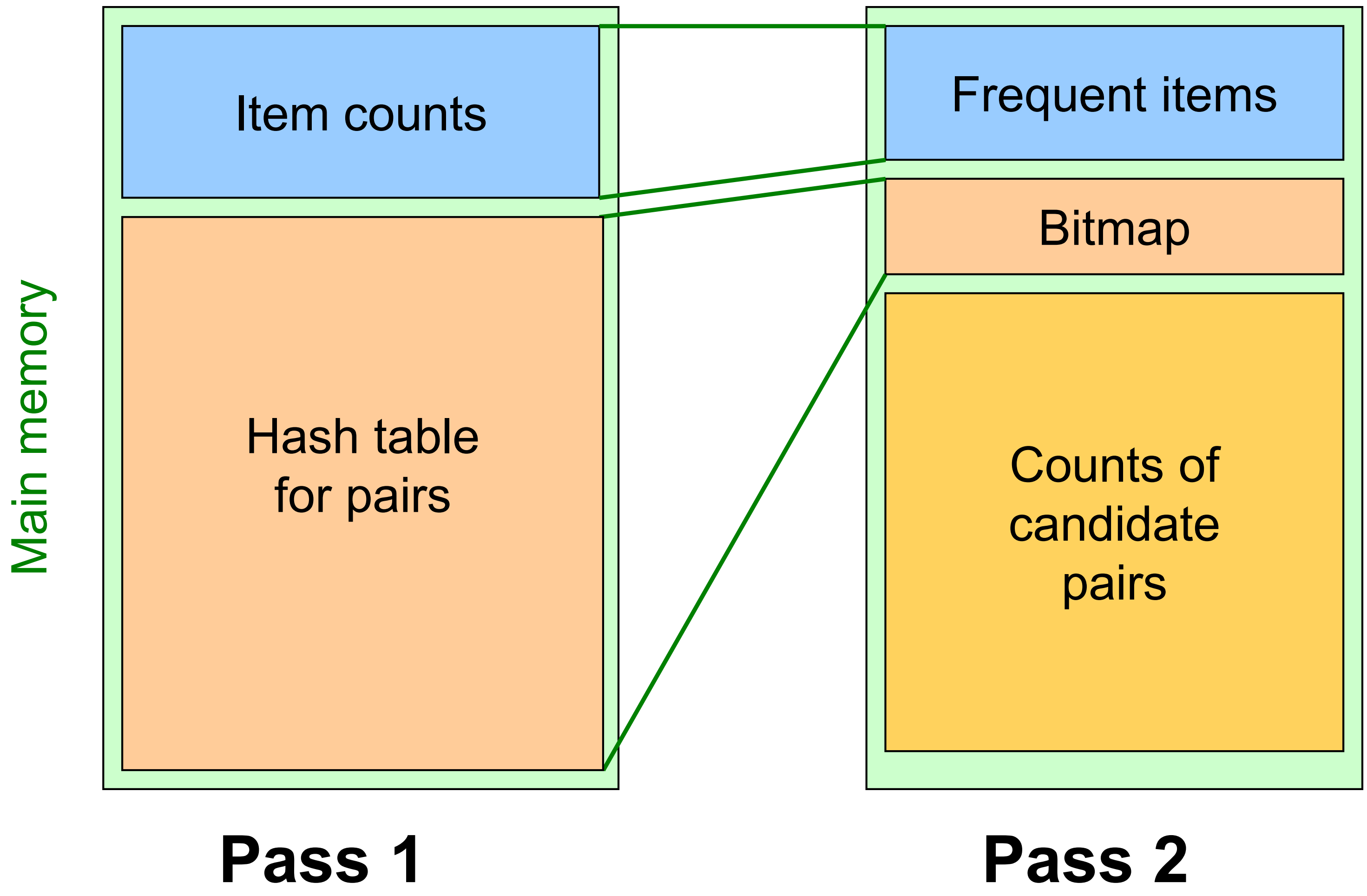
- The A-Priori algorithm is fine ***as long as*** the counting of the candidate pairs C_2 can be accomplished in memory
  - Otherwise, ***thrashing*** occurs
    - i.e., repeated moving of data between disk and main memory

- Several algorithms have been proposed to ***cut down*** on the size of candidate set C_2
  - PCY algorithm
  - Multistage algorithm
  - Multihash algorithm

PCY (Park-Chen-Yu) Algorithm

- Observation
  - There may be much ***unused*** space in main memory on the first pass
    - We do not need more than the main memory for the two tables
    - A translation table from items to integers and an array to count those integers
    - (ex) If there are 10^6 items, only 12×10^6 bytes = 12 MB is used for the counts
- Can we use the unused memory to ***reduce*** memory required in the second pass?
- The idea of PCY
  - Use the unused space to store some information that can ***filter*** infrequent pairs in the second pass
  - Similar to the idea of a Bloom filter

Main Memory Use of PCY



The First Pass of PCY

- In addition to item counts, maintain a ***hash table*** with as many buckets as fit in memory
  - As we examine a basket during the first pass, we not only count each item in the basket, but generate all pairs, using a double loop
  - We hash each pair and add 1 to the bucket into which that pair hashes
  - Note that we just keep the count for each bucket, ***not*** the actual pairs

```
 FOR (each basket) :
      FOR (each item in the basket) :
            add 1 to item's count;
New  { FOR (each pair of items) :
 in  {      hash the pair to a bucket;
PCY  {      add 1 to the count for that bucket;
```

Observation about Buckets

- At the end of the first pass, each bucket has a count
  - The sum of counts of all the pairs that hash to that bucket
- If the count of a bucket $\geq s$ (i.e., the support threshold)
  - The bucket is called a ***frequent*** bucket
  - The pairs that hash to a frequent bucket ***may or may not*** be frequent
- However, if the count of a bucket $< s$
  - The bucket is called an ***infrequent*** bucket
  - ***no*** pair that hashes to an infrequent bucket can be frequent
- Thus, we ***only*** count pairs that hash to ***frequent*** buckets in the second pass!

Between The Passes of PCY

- The hash table is replaced by a bitmap, with 1 bit for each bucket
  - The bit is 1, if the bucket is frequent and 0, if not
- Space reduction
  - 4 byte integers for each bucket are replaced by a single bit
  - Thus, the bitmap takes only **1/32** of the space
- If most buckets are infrequent (this is what we expect)
  - The number of pairs being counted on the second pass will 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

The Second Pass of PCY

- Count only those pairs $\{i, j\}$ that meet the following conditions for being a candidate pair in C_2:
  ① Both i and j are frequent items
  ② The pair $\{i, j\}$ hashes to a bucket whose bit in the bitmap is 1
    - That is, $\{i, j\}$ hashes to a ***frequent*** bucket

- Both conditions are necessary for the pair to have a chance of being frequent

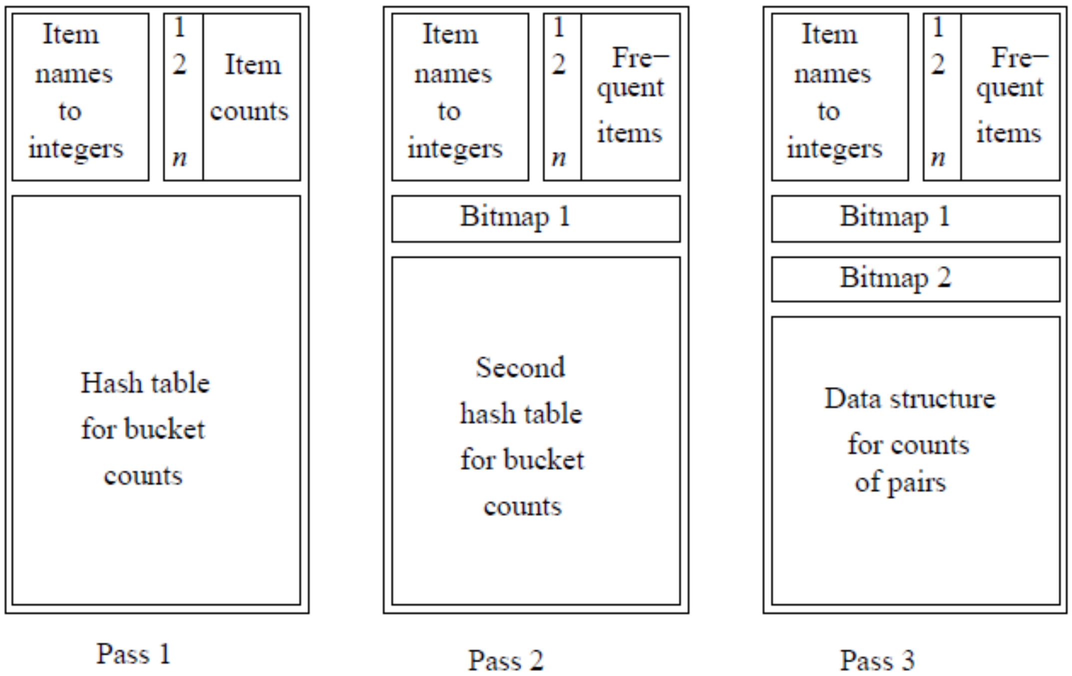
- It is the ***second condition*** that distinguishes PCY from A-Priori

Counting Pairs in PCY

- However, on the second pass of PCY, we ***cannot*** use the triangular matrix method to count pairs
  - Because the pairs of frequent items are placed ***randomly*** within the triangular matrix, we cannot compact the matrix
- Consequently, we are forced to use the ***triples methods*** in PCY
  - Note that A-Priori can use the both methods
  - That restrict may not matter if the fraction of pairs of frequent items that actually appear in buckets were small, but …
- 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
  - Triangular matrix: 4 bytes per pair
  - Triple method: 12 bytes per pair

Multistage Algorithm

- Improves PCY to reduce ***further*** the number of candidate pairs
  - By using ***several*** successive hash tables
  - However, the tradeoff is that it requires more than two passes
- Takes ***more than*** two passes to find the frequent pairs
  - The first pass → the second pass → the third pass



The First and Second Passes

- The first pass
  - The same as the first pass of PCY
  - After that pass, the frequent buckets are summarized by a bitmap

- The second pass
  - Again go through the file of baskets
  - Using ***another*** hash function, rehash only those pairs $\{i, j\}$ that meet the following two criteria:
    1. Both i and j are frequent items
    2. The pair $\{i, j\}$ hashed to a frequent bucket on the ***first*** pass
  - Summarize the second hash table to a bitmap
    - As result, we expect there to be many ***fewer*** frequent buckets in the second hash table than in the first

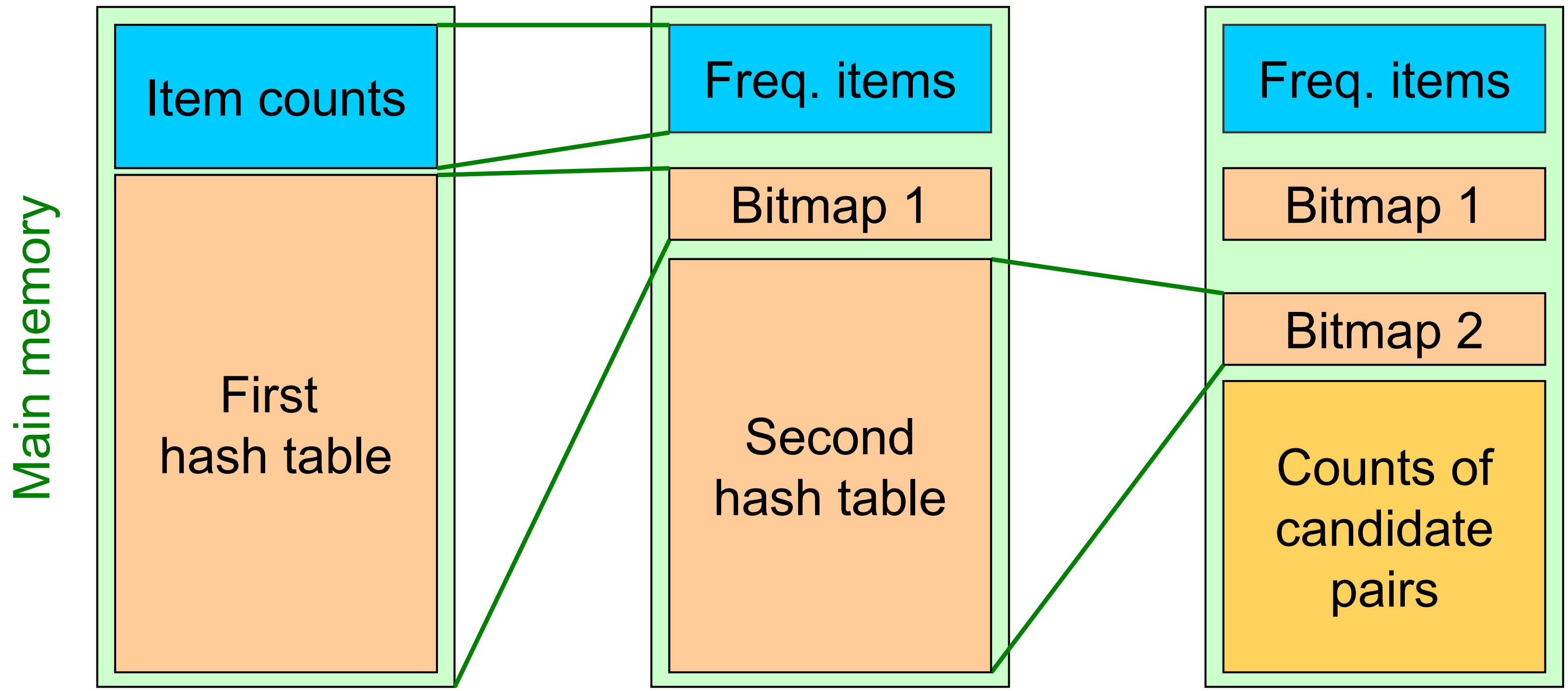
The Third Pass

- Count only those pairs $\{i, j\}$ that meet the following conditions for being a candidate pair in C_2:
  ① Both i and j are frequent items
  ② $\{i, j\}$ hashes to a frequent bucket in the ***first*** hash table
  ③ $\{i, j\}$ hashes to a frequent bucket in the ***second*** hash table

- The ***third condition*** is the distinction between Multistage and PCY

- Important points
  - The two hash functions have to be ***independent***
  - We need to check ***both*** hashes on the third pass
    - If not, we would end up counting pairs of frequent items that hashed first to an infrequent bucket but happened to hash second to a frequent bucket

Main Memory Use of Multistage



Pass 1

Count items
Hash pairs {i,j}

Pass 2

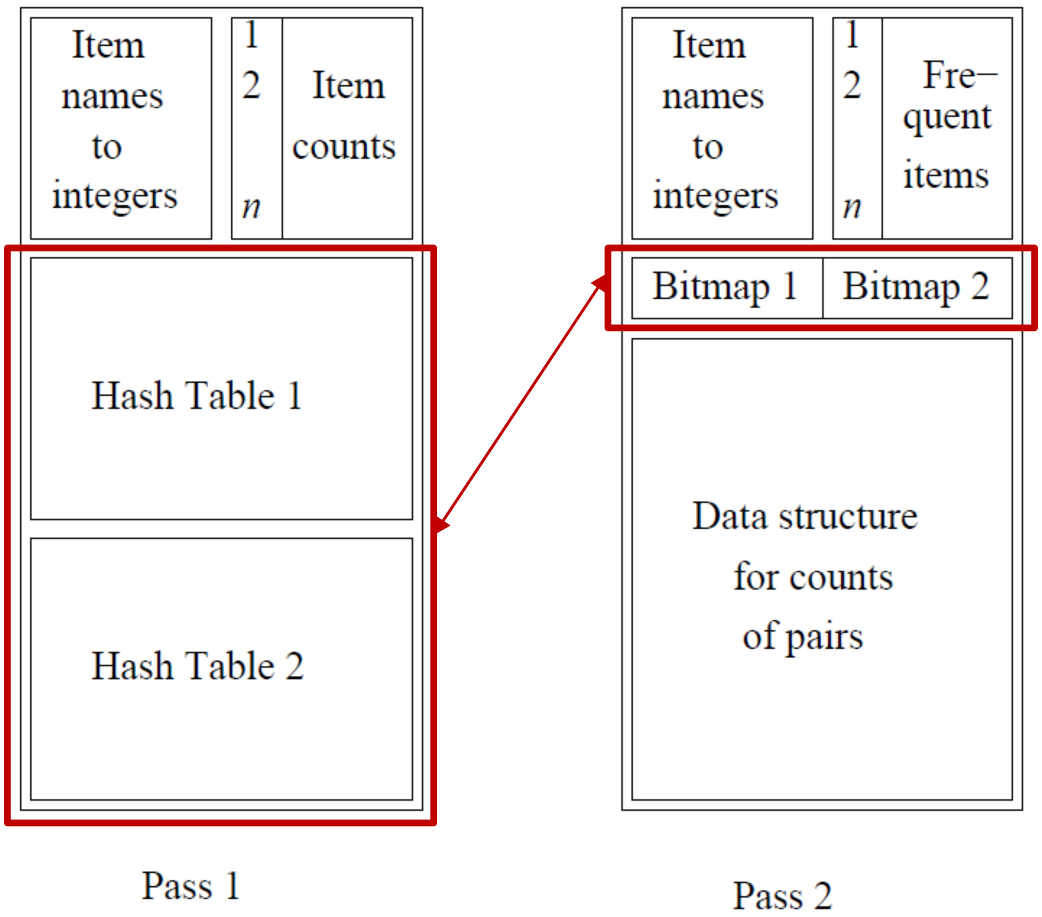
Hash pairs {i,j} into Hash2 iff:
i,j are frequent,
{i,j} hashes to freq. bucket in B1

Pass 3

Count pairs {i,j} iff:
i,j are frequent,
{i,j} hashes to freq. bucket in B1
{i,j} hashes to freq. bucket in B2

Multihash Algorithm

- Key idea
  - Get the benefit of the extra passes of Multistage in a ***single*** pass
  - Use ***several independent*** hash tables on the first pass



The Danger of Using Two Hash Tables

- Each hash table has ***half*** as many buckets as the one large hash table of PCY
  - Thus, the average count of a bucket ***doubles***
  - We have to sure most buckets will still not reach count s

1	1	1	1	1	1	1	1

A single large hash table

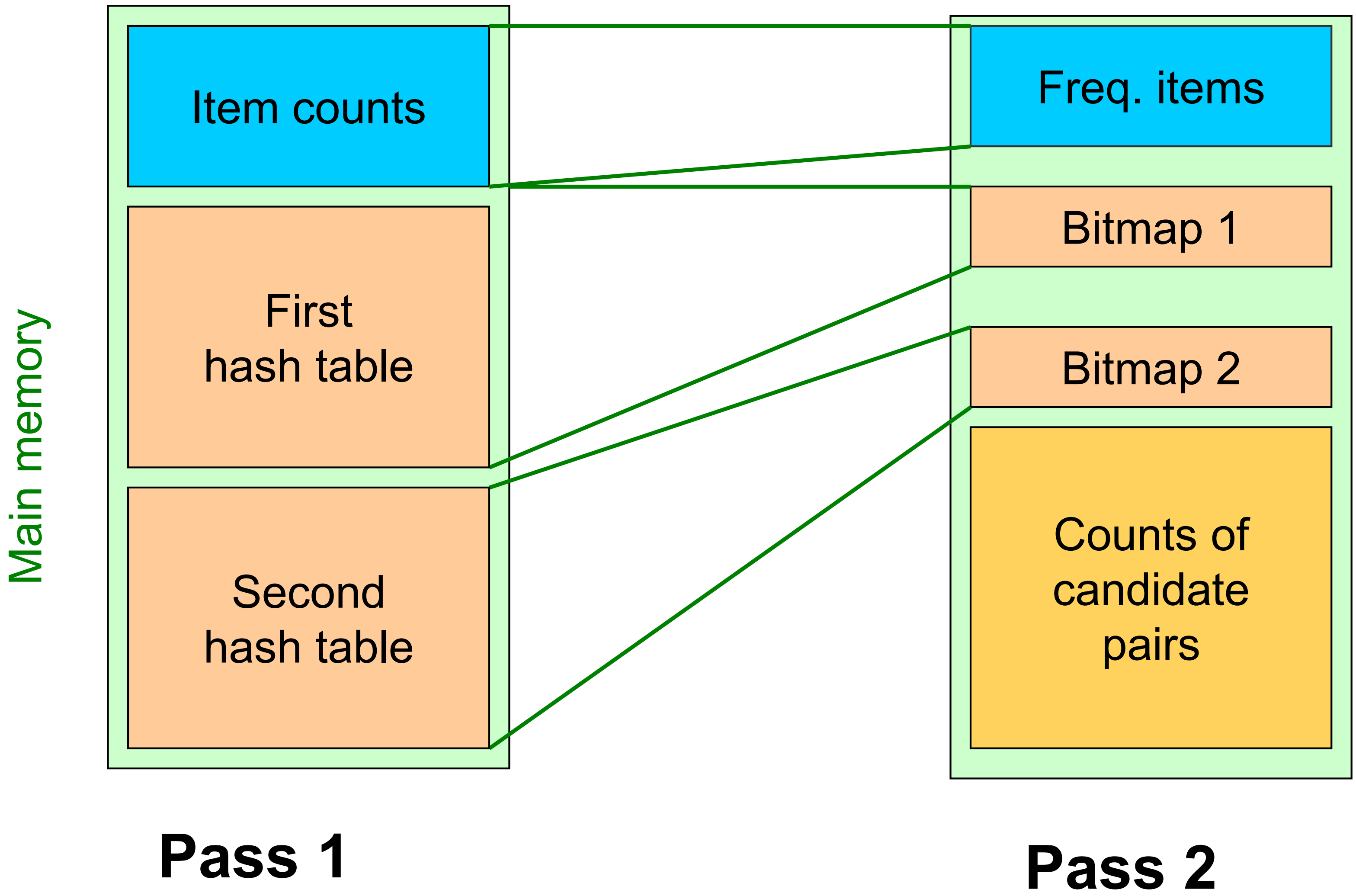
2	2	2	2

2	2	2	2

Two small hash tables

- If the average count of a bucket for PCY is much lower than s
  - 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
- If so, we can get a benefit like multistage, but in only ***two*** passes

Main Memory Use of Multihash



The Second Pass of Multihash

- Each hash table is converted to a bitmap, as usual
- Note
  - The two bitmaps for the two hash functions occupy exactly as much space as a singe bitmap would for the second pass of PCY
- 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
  ① Both i and j are frequent items
  ② $\{i, j\}$ hashes to a frequent bucket in the ***first*** hash table
  ③ $\{i, j\}$ hashes to a frequent bucket in the ***second*** hash table

Extensions of PCY

- Either Multistage or Multihash can use ***more than*** two hash functions

- Multistage (with more than two hash functions)
  - There is a point of diminishing returns, since the bit-vectors eventually consume all of main memory

- Multihash (with more than two hash functions)
  - The bit-vectors occupy exactly what one PCY bitmap does, but too many hash functions makes all counts $\geq s$

Limited-Pass Algorithm

Limited-Pass Algorithms

- The algorithms for frequent itemsets discussed so far use one pass for ***each*** size of itemset
  - i.e., a total of k passes to find frequent itemsets of size 1, 2, …, k
  - Can we use fewer passes?

- However, there are many applications where it is ***not*** essential to discover every frequent itemset
  - In many cases, it is quit sufficient to find most but not all of the frequent itemsets (e.g., a supermarket application)

- We explore some algorithms that uses ***at most two*** passes
  - But we may miss some frequent itemsets
  - (ex) Random sampling, SON (Savasere, Omiecinski, and Navathe)

Random Sampling (1/2)

- Key idea
  - Instead of using the entire file of baskets, pick a random subset of the baskets and pretend it is the entire dataset
- We ***must*** adjust the support threshold s to $p \cdot s$ to reflect the smaller number of baskets
  - p: the sampling rate
  - (ex) if we choose a sample of 1% of the baskets, we adjust s to $s/100$
- How can we pick the sample?
  - For each basket, select that basket with some fixed probability p
  - (ex) if there are m baskets in the entire file, we shall sample $p \cdot m$ baskets

Random Sampling (2/2)

- Having selected a sample, we store these baskets in main memory and execute one of the algorithms ***in main memory***
  - A-Prioir, PCY, Multistage, or Multihash

- Although the algorithm must run passes over the main-memory sample, there are ***no*** disk accesses needed to read the sample
  - Since it resides in main memory
  - Therefore, ***only two*** passes are required

- Of course the algorithm will fail if it can't be run in main memory
  - An option is to read the sample from disk for each pass
  - Sine the sample is much smaller than the full dataset, we still avoid most of the disk I/O

Avoiding Errors in Sampling Algorithms

- In random sampling, there can be ***errors***
  - False negative
    - An itemset that is frequent in the whole but ***not*** in the sample
  - False positive
    - An itemset that is frequent in the sample but ***not*** in the whole

- If the sample is large enough, there are unlikely to be serious errors
  - An itemset whose support is much ***larger*** than s will almost surely be identified from a random sample
  - An itemset whose support is much ***less*** than s is unlikely to appear frequent in the sample

How Can We Reduce Such Errors?

- We can eliminate ***false positives***
  - Make a pass through the full dataset and count 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
  - But requires more passes

- We cannot eliminate ***false negatives*** but can reduce their number
  - Use a support threshold ***smaller*** than $p{\cdot}s$ (e.g., $0.9p{\cdot}s$) to catch more truly frequent itemsets
  - We shall identify, as having support at least $0.9p{\cdot}s$ in the sample, almost all those itemsets that have support at least s in the whole
  - But requires more space

SON Algorithm

- Improves the random sampling to avoid ***both*** false negatives and false positives, at the cost of making ***two*** passes

- First pass
  - Divide the input file into ***chunks***
  - Treat each chunk as a sample, and find all frequent items in that chunk
    - We use $p \cdot s$ as the threshold, if each chunk is fraction p of the whole file
  - Store on disk all the frequent itemsets found for each chunk
  - These frequent itemsets become the ***candidate*** itemsets

- Second pass
  - Count all the candidate itemsets and select those that have support at least s as the frequent itemsets

SON Algorithm: Key Idea

- In the SON algorithm, an itemset becomes a candidate if it is frequent in ***any*** one or more chunks

- Key "monotonicity" idea
  - An itemset ***cannot*** be frequent in the whole unless it is frequent in at least one chunk
  - Proof
    - If an itemset is not frequent in any chunk, then its support is less than $p{\cdot}s$ in each chunk
    - Since the number of chunks is $1/p$, the total support of that items is less than $(1/p)p{\cdot}s = s$

- Therefore, there are ***no*** false negatives
  - Every itemset frequent in the whole is frequent in at least one chuck

SON Algorithm and MapReduce

- SON algorithm lends itself ***well*** to a parallel-computing environment

- Distributed version
  ① Distribute baskets among many nodes
  ② Compute frequent itemsets at each node
  ③ Distribute the candidates to all nodes
  ④ Count the support for each candidate at each node
  ⑤ Finally sum those supports to get the support for each candidate

- SON algorithm can be easily expressed as ***MapReduce*** jobs
  – Of course, this process does not have to be implemented in MapReduce

SON: MapReduce Version (1/2)

- **Phase 1**: Find candidates itemsets

- Map task
  - Input: a subset of the baskets
  - Output: a set of key-value pairs $(F, 1)$
    - F: a frequent itemset from the subset

- Reduce task
  - Input: a set of key-values pairs $(F, [1, 1, \ldots])$
    - F: a frequent itemset identified from some subset
  - Output: a set of key-value pairs $(F, 1)$
    - F: a candidate itemset

SON: MapReduce Version (2/2)

- **Phase 2**: Find true frequent itemsets

- Map task
  - Input: a subset of the candidate itemsets, a subset of the baskets
  - Output: a set of key-value pairs (C, v)
    - C: a candidate itemset
    - v: the count for C among the subset of the baskets

- Reduce task
  - Input: a set of key-values pairs $(C, [v_1, v_2, \ldots])$
    - C: a candidate itemset
  - Output: a set of key-value pairs (C, sum)
    - C: a frequent itemset whose sum is at least s
    - sum: the sum of values $v_1, v_2, \ldots$