Frequent Itemsets

The Market-Basket Model
Association Rules
A-Priori Algorithm
Other Algorithms

Jeffrey D. Ullman
Stanford University

The Market-Basket Model

- A large set of *items*, e.g., things sold in a supermarket.
- A large set of *baskets*, each of which is a small set of the items, e.g., the things one customer buys on one day. (Transactions)
- *A-priori Algorithm and A-priori property*
- *Candidate set generation*

Support

- Simplest question: find sets of items that appear "frequently" in the baskets.
- *Support* for itemset *I* = the number of baskets containing all items in *I*.
  - Sometimes given as a percentage of the baskets.
- Given a *support threshold* *s*, a set of items appearing in at least *s* baskets is called a *frequent itemset*.

Example: Frequent Itemsets

- Items={milk, coke, pepsi, beer, juice}.
- Support = 3 baskets.

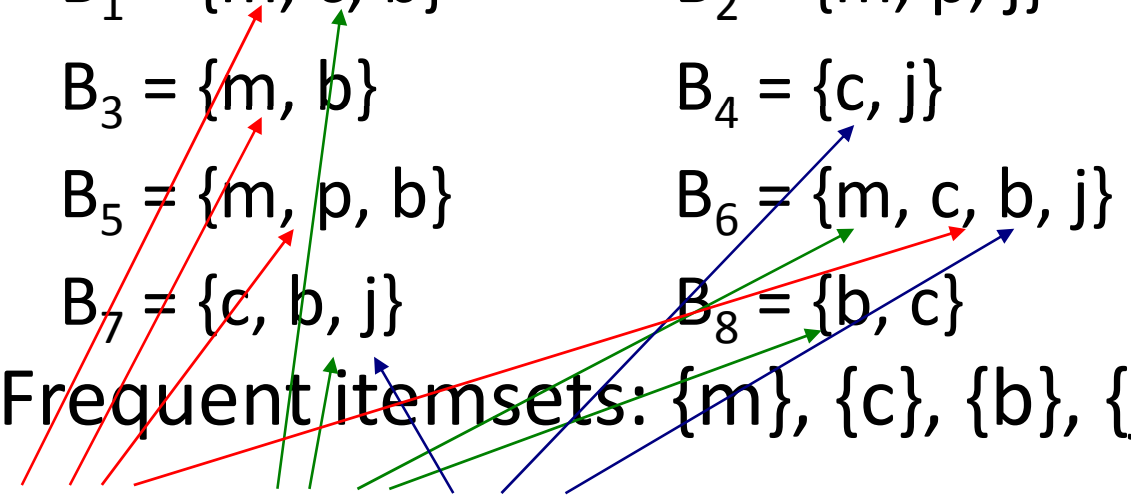
B_1 = {m, c, b} B_2 = {m, p, j}

B_3 = {m, b} B_4 = {c, j}

B_5 = {m, p, b} B_6 = {m, c, b, j}

B_7 = {c, b, j} B_8 = {b, c}

- Frequent itemsets: {m}, {c}, {b}, {j}, {m,b}, {b,c}, {c,j}.



Applications

- "Classic" application was analyzing what people bought together in a brick-and-mortar store.
  - Apocryphal story of "diapers and beer" discovery.
  - Used to position potato chips between diapers and beer to enhance sales of potato chips.
- Many other applications, including plagiarism detection; see MMDS.
- Related concepts identification
- Biomarkers – genes, proteins, diseases

Association Rules

- If-then rules about the contents of baskets.
- $\{i_1, i_2, \ldots, i_k\} \rightarrow j$ means: "if a basket contains all of $i_1, \ldots, i_k$ then it is *likely* to contain j."
- *Confidence* of this association rule is the probability of j given $i_1, \ldots, i_k$.
  - That is, the fraction of the baskets with $i_1, \ldots, i_k$ that also contain j.
- Generally want both high confidence and high support for the set of items involved.
- We'll worry about support and itemsets, not association rules.

Example: Confidence

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

- An association rule: {m, b} → c.
  - Confidence = 2/4 = 50%. Interest = 2/4 – 5/8 = -1/8
- Confidence of I -> j = support of I U j / support of I
- Interest of an association rule I → j is the difference between its confidence and the fraction of baskets that contain j.
- If interest =0, then I has no influence on j
- If interest >0 then presence of I has some influence in j
  {bread} -> {milk} – High Interest
- If interest < 0 then presence of I discourages the presence of j
  {pepsi} -> {coke} – Negative Interest

problem

- Trace the results of using the Apriori algorithm on the grocery store example with support threshold *s=33.34% and confidence threshold c=60%. Show the candidate and frequent itemsets for each database scan. Enumerate all the final frequent itemsets. Also indicate the association rules that are generated and highlight the strong ones, sort them by confidence*

Transaction ID	Items
T1	HotDogs, Buns, Ketchup
T2	HotDogs, Buns
T3	HotDogs, Coke, Chips
T4	Chips, Coke
T5	Chips, Ketchup
T6	HotDogs, Coke, Chips

- All Frequent Itemsets: {HotDogs}, {Buns}, {Ketchup}, {Coke}, {Chips}, {HotDogs, Buns}, {HotDogs, Coke}, {HotDogs, Chips}, {Coke, Chips}, {HotDogs, Coke, Chips}.

Computation Model

- Typically, data is kept in flat files.
- Stored on disk. Can be distributed
- Stored basket-by-basket.
- {23,456,1001}{3,18,92,145}{...
- Each { } is a basket
- Expand baskets into pairs, triples, etc. as you read baskets.
  - Use *k* nested loops to generate all sets of size *k*.
  - For example, if there are 20 items in a basket, then there are $20C_2 = 190$ pairs of items in the basket, and these can be generated easily in a pair of nested for-loops.

Computation Model – (2)

- As the size of the subsets (k) 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.
- The true cost of mining disk-resident data is usually the number of disk I/O's.
- In practice, algorithms for finding frequent itemsets read the data in *passes* – all baskets read in turn.
- Thus, we measure the cost by the number of passes an algorithm takes.

Main-Memory Bottleneck

- For many frequent-itemset algorithms, main memory is the critical resource.
- As we read baskets, we need to count something, e.g., occurrences of pairs of items.
- The number of different things we can count is limited by main memory.
- Swapping counts in/out is a disaster.
- Items can be represented as an integer or string (bread, milk). We can represent the items from 1 to n using hash table.

Finding Frequent Pairs

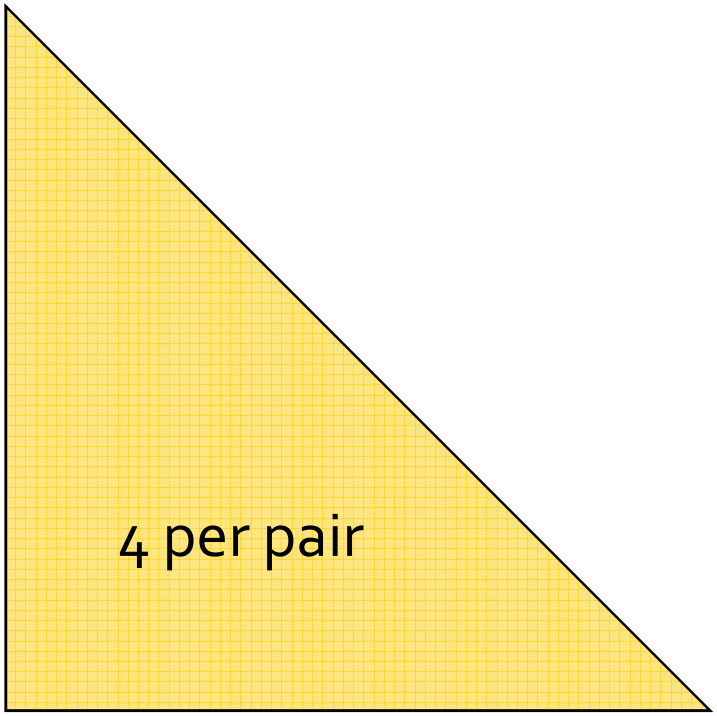
- The hardest problem often turns out to be finding the frequent pairs.
  - Why? Often frequent pairs are common, frequent triples are rare.
    - Why? Support threshold is usually set high enough that you don't get too many frequent itemsets.
- We'll concentrate on pairs, then extend to larger sets.

Naïve Algorithm

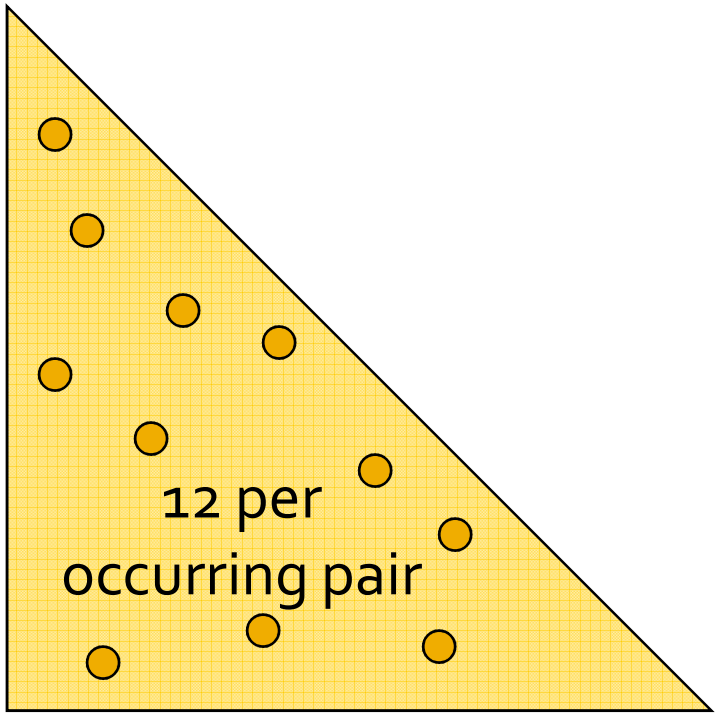
- Read file once, counting in main memory the occurrences of each pair.
  - From each basket of n items, generate its $n(n-1)/2$ pairs by two nested loops.
- Fails if $(\#\text{items})^2$ exceeds main memory.
  - Example: Walmart sells 100K items, so probably OK.
  - Example: Web has 100B pages, so definitely not OK.

Details of Main-Memory Counting

- Two approaches:
  1. Count all pairs, using a triangular matrix method.
  2. Triples method- Keep a table of triples [*i*, *j*, *c*] = "the count of the pair of items {*i*, *j*} is *c*." hash table with i and j as the search key
- (1) requires only 4 bytes/pair.
  - Note: always assume integers are 4 bytes.
- (2) requires 12 bytes, but only for those pairs with count > 0. Doesnot store anything if the count for the pair is 0.



Triangular matrix



Tabular method

Triangular-Matrix Approach

- Number items 1, 2,…, n.
  - Requires table of size O(n) to convert item names to consecutive integers.
- Count {i, j} only if $i < j$ *in 2D array*. Half array is free.
- Efficient way is 1D triangular array.
- Keep pairs in the order {1,2}, {1,3},…, {1,n}, {2,3}, {2,4},…, {2,n}, {3,4},…, {3,n},…, {n -1,n}.
- Find pair {i, j}, where i<j, at the position k:

$$k = (i - 1)(n - i/2) + j - i$$

- 1<=i < j<=n
- Total number of pairs $n(n-1)/2$; total bytes about $2n^2$.

Details of Tabular Approach / triples

- Total bytes used is about 12*p*, where *p* is the number of pairs that actually occur.
  - Beats triangular matrix if at most 1/3 of possible pairs actually occur.
- May require extra space for retrieval structure, storage of i,j,c e.g., a hash table.

The A-Priori Algorithm

Monotonicity of "Frequent"
Candidate Pairs
Extension to Larger Itemsets

A-Priori Algorithm

- A two-pass approach called *a-priori* limits the need for main memory.
- Key idea: *monotonicity*:
  - If a set I of items is frequent, then so is every subset of I.
  - if a set of items appears at least *s* times, so does every subset of the set.
- Contrapositive for pairs: if item *i* does not appear in *s* baskets, then no pair including *i* can appear in *s* baskets.
- If we are given a support threshold s, then we say an itemset is maximal if no superset is frequent.

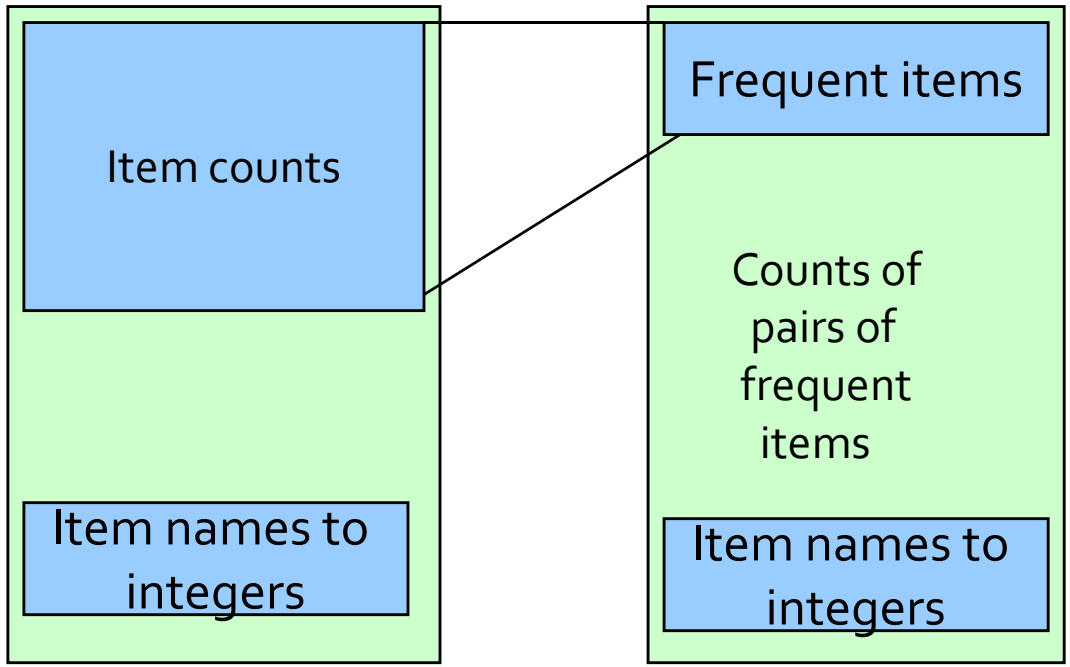
A-Priori Algorithm – (2)

- Base approach: For each basket use 2 loops to generate the all possible pairs. Each time we find their occurrence then add 1 to it. At end check for threshold and select frequent items.
- 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.
- Pass 1: Read baskets and translates item names into integers (hash table) and a table to count in main memory the occurrences of each item.
  - Requires only memory proportional to #items.
- Items that appear at least *s* times are the *frequent items*.

A-Priori Algorithm – (3)

- Pass 2: Read baskets again and count in main memory only those pairs both of which were found in Pass 1 to be frequent.
- Requires memory proportional to square of *frequent* items only (for counts), plus a list of the frequent items (so you know what must be counted). Uses triangular method.

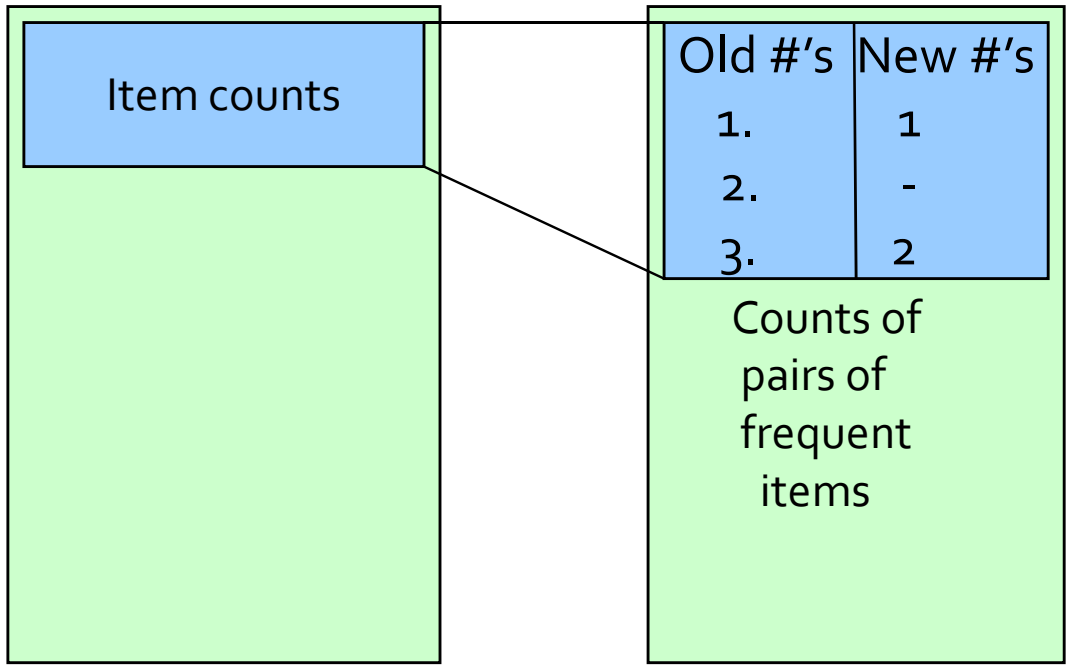
Picture of A-Priori



Detail for A-Priori

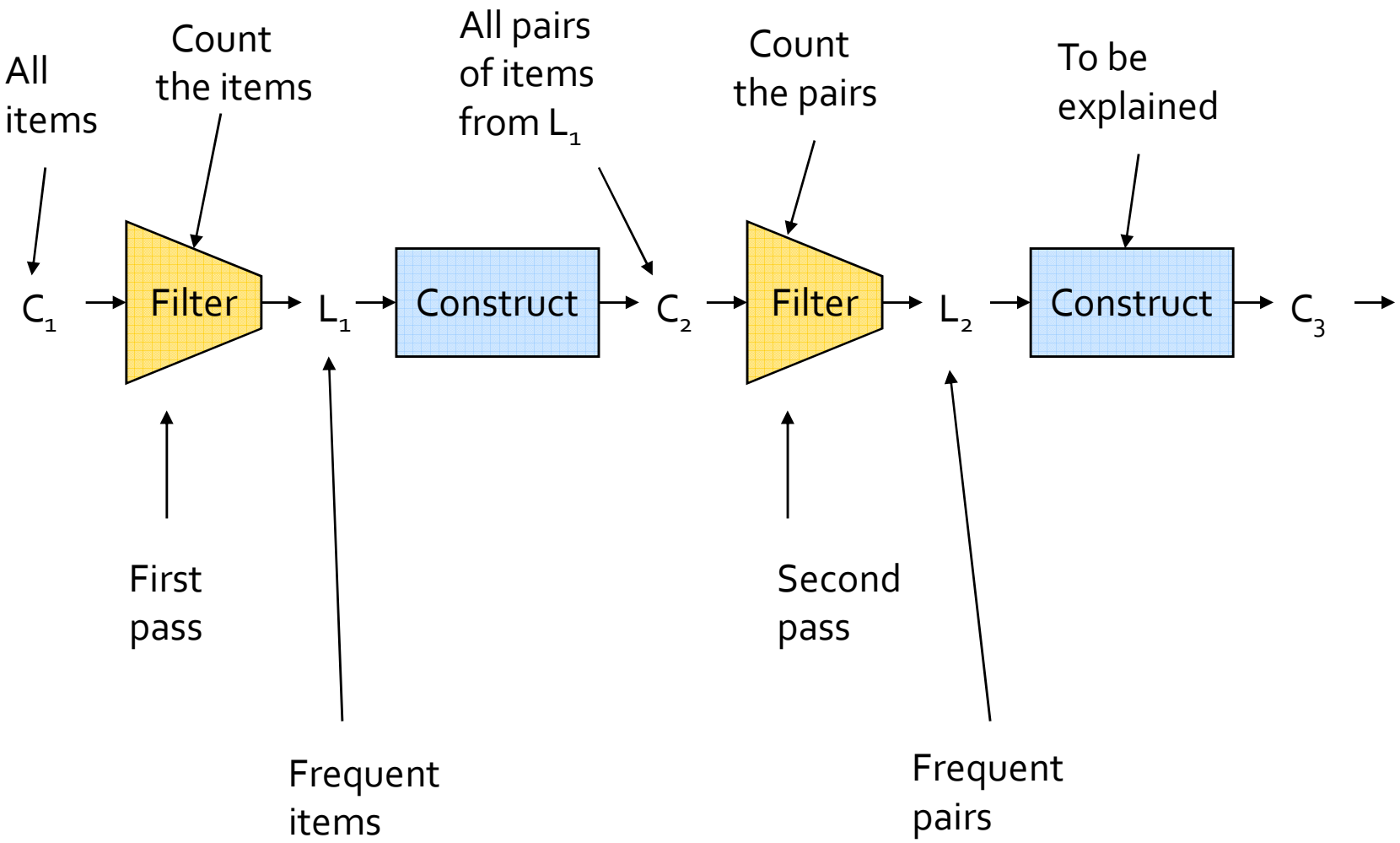
- You can use the triangular matrix method with *n* = number of frequent items.
  - May save space compared with storing triples.
- Trick: number frequent items 1, 2,… and keep a table relating new numbers to original item numbers.

A-Priori Using Triangular Matrix



Frequent Triples, Etc.

- For each size of itemsets k, we construct two sets of *k-sets* (sets of size k):
  - C_k = *candidate* k-sets = those that might be frequent sets (support $\geq s$) based on information from the pass for itemsets of size $k - 1$.
  - L_k = the set of truly frequent k-sets.


All items
Count the items
All pairs of items from L_1
Count the pairs
To be explained
C_1
Filter
L_1
Construct
C_2
Filter
L_2
Construct
C_3
First pass
Second pass
Frequent items
Frequent pairs

Passes Beyond Two

- C_1 = all items
- In general, L_k = members of C_k with support $\geq s$.
  - Requires one pass.
- C_{k+1} = (k+1)-sets, each k of which is in L_k .

Memory Requirements

- At the k^{th} pass, you need space to count each member of C_k.
- In realistic cases, because you need fairly high support, the number of candidates of each size drops, once you get beyond pairs.

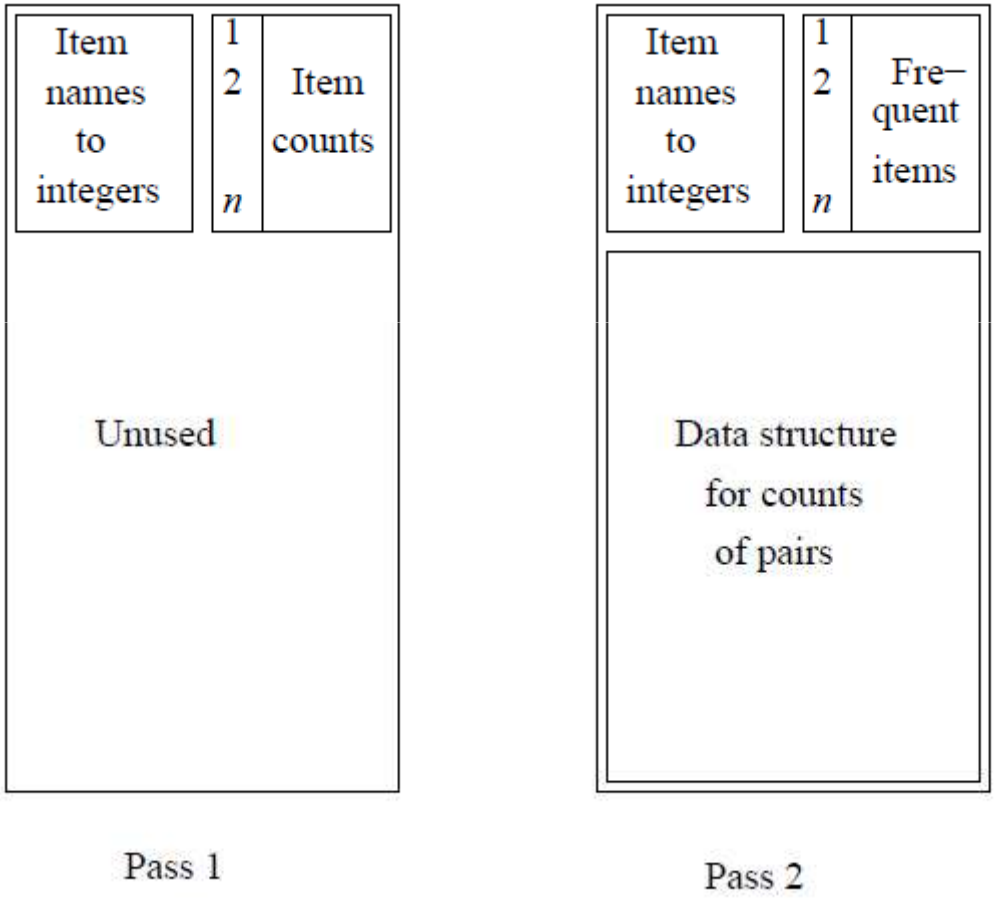
The PCY (Park-Chen-Yu) Algorithm

- Improvement to A-Priori
- Exploits Empty Memory on First Pass
- Frequent Buckets

PCY Algorithm

- Several algorithms are proposed to cut down the size of candidate set C_2
- During Pass 1 of A-priori, most memory is idle.
- In first pass of A-Priori there is typically lots of main memory not needed for the counting of single items.
- Main memory has 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 (hash table store integers or bits not keys)
- Use that memory to keep counts of buckets into which pairs of items are hashed.
  - Just the count, not the pairs themselves.
- For each basket, enumerate all its pairs, hash them, and increment the resulting bucket count by 1.

A-priori – main memory



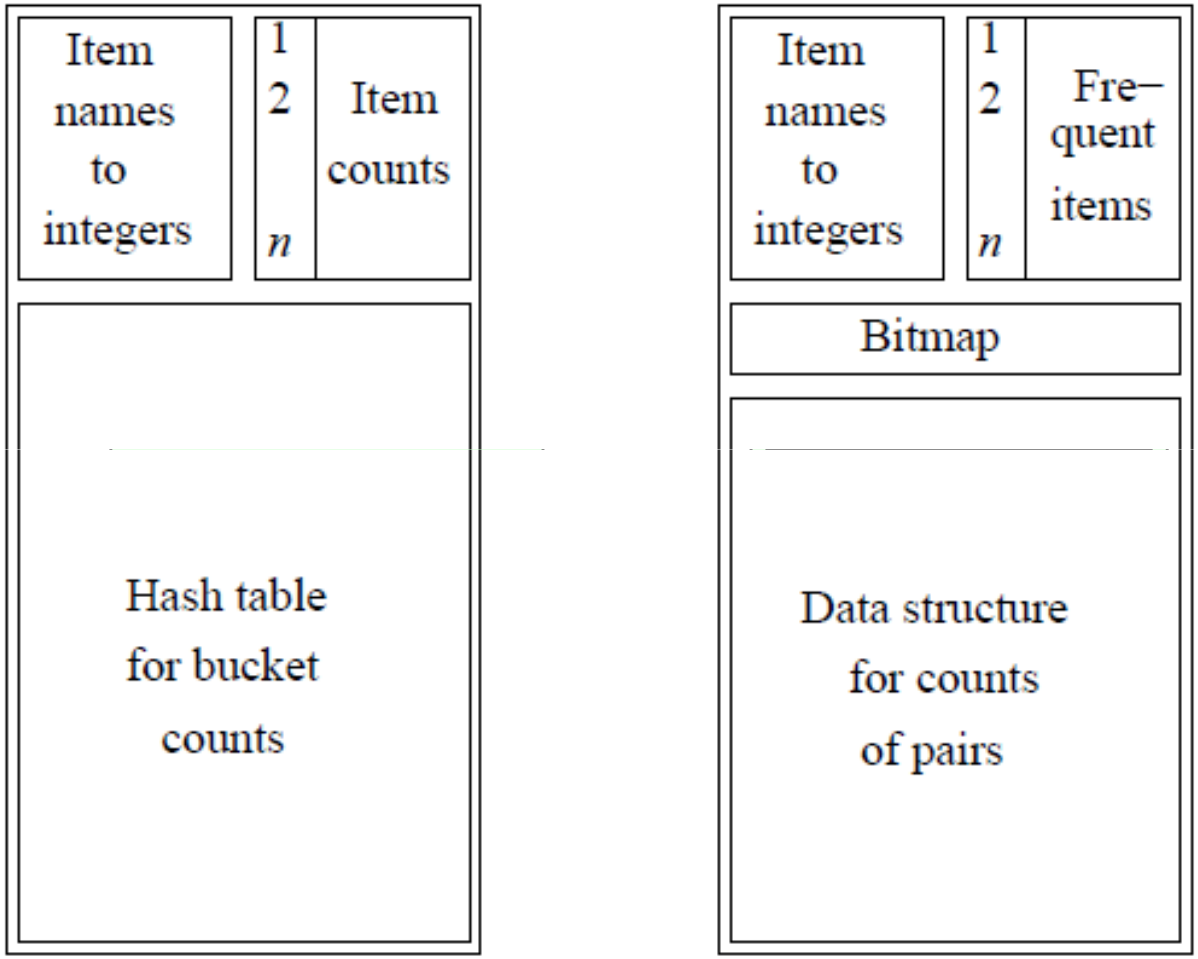
PCY Algorithm – (2)

- 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
- A bucket is *frequent* if its count is at least the support threshold.
- If a bucket is not frequent, no pair that hashes to that bucket could possibly be a frequent pair.
- On Pass 2, we only count pairs of frequent items that also hash to a frequent bucket.

PCY algorithm

- 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.
- 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. (i.e., 1/32 of the space used on Pass 1).

Picture of PCY



Pass 1: Memory Organization

- Space to count each item.
  - One (typically) 4-byte integer per item.
- Use the rest of the space for as many integers, representing buckets using hashtable, as we can.

PCY Algorithm – Pass 1

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

Observations About Buckets

1. A bucket that a frequent pair hashes to is surely frequent.
   - We cannot use the hash table to eliminate any member of this bucket.
2. Even without any frequent pair, a bucket can be frequent.
   - Again, nothing in the bucket can be eliminated.

Observations – (2)

3. But in the best case, the count for a bucket is less than the support *s*.
   - Now, all pairs that hash to this bucket can be eliminated as candidates, even if the pair consists of two frequent items.

PCY Algorithm – Between Passes

- Replace the buckets by a bit-vector (the "bitmap"):
  - 1 means the bucket is frequent; 0 means it is not.
- Also, decide which items are frequent and list them for the second pass.

PCY Algorithm – Pass 2

- Count all pairs {*i*, *j*} that meet the conditions for being a candidate pair:
  1. Both *i* and *j* are frequent items.
  2. The pair {*i*, *j*}, hashes to a bucket number whose bit in the bit vector is 1.

Memory Details

- Buckets require a few bytes each.
  - Note: we don't have to count past *s*.
  - # buckets is O(main-memory size).
- On second pass, a table of (item, item, count) triples is essential. (Triples or Tabular method)
  - Thus, hash table on Pass 1 must eliminate 2/3 of the candidate pairs for PCY to beat a-priori.

More Extensions to A-Priori

- The MMDS book covers several other extensions beyond the PCY idea
- "Multistage" and
- "Multihash"

Limited Pass Algorithms
All (Or Most) Frequent Itemsets In ≤ 2 Passes

1. Simple Randomized Algorithm
2. Savasere-Omiecinski- Navathe (SON) Algorithm
3. Toivonen's Algorithm

Simple Randomized Algorithm

- Not essential to find all frequent itemsets. (most is sufficient)
- Take a random sample of the market baskets.
- Use as your support threshold a suitable, scaled-back number.
  - Example: if your sample is 1% (1/100) of the baskets, use *s*/100 as your support threshold instead of *s*.
- select that basket for the sample with some fixed probability *'p'*
- *'m'* baskets in file. Then sample size is close to *'pm'* baskets

Simple Algorithm – Option

- Have sample in main memory.
- Run a-priori or one of its improvements (PCY, multistage, multihash) on samples (for sets of all sizes, not just pairs) in main memory, so you don't pay for disk I/O each time you increase the size of itemsets.

Errors in sampling algorithms

- An itemset that is frequent in the whole but not in the sample is a false negative
- An itemset that is frequent in the sample but not the whole is a false positive.
- Optionally, verify that your guesses are truly frequent in the entire data set by a second pass.
- But you don't catch sets frequent in the whole but not in the sample.
  - Smaller threshold, e.g., $s/125$ instead of $s/100$, helps catch more truly frequent itemsets.
    - But requires more space.

SON Algorithm – pass 1
Savasere-Omiecinski- Navathe

- Avoids both false negatives and false positives, at the cost of making two full passes.
- Partition the baskets into small subsets or chunks.
- Read each subset into main memory and perform the first pass of the simple algorithm on each subset.
  - Parallel processing of the subsets a good option.
- An itemset becomes a candidate if it is found to be frequent (with support threshold suitably scaled down) in *any* one or more subsets of the baskets.
  - *'ps'* - if each chunk is fraction *'p'* of the whole file, and *'s'* is the support threshold

SON Algorithm – Pass 2

- Store all frequent itemsets in disk ('C').
- On a second pass, count all the candidate itemsets and determine which are frequent in the entire set.
- If an itemset is not frequent in any chunk, then its support is less than ps in each chunk. So it is not frequent in whole set.
- Key "monotonicity" idea: an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset.
- no false negatives

SON and MapReduce

- Chunks can be processed in parallel
- First Map function
  - From the sample (chunks) baskets find frequent itemsets using simple randomized algorithm by threshold *'ps'*
  - *Input – fraction of input file*
  - *Output – key-value pairs (F,1), F is frequent itemset*
- First Reduce
  - Key –itemsets
  - Reduce task produces keys (itemsets) that appear one or more times
  - Output – candidate itemsets

SON and MapReduce

- Second Map function
  - Input – candidate sets and a portion of input data file
  - 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
  - Output -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
  - Sum the associated values – total support for each of the itemsets
  - itemsets whose sum of values is at least ‘s’ are frequent in the whole dataset

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
- No FP, FN, but a small probability of failure to produce any answer. Reiterate
- Start as in the simple algorithm, but lower the threshold slightly for the sample.
  - Example: if the sample is 1% of the baskets, use $s/125$ as the support threshold rather than $s/100$.
  - Goal is to avoid missing any itemset that is frequent in the full set of baskets but memory requirement is high

Toivonen's Algorithm – (2)

- Add to the itemsets that are frequent in the sample the *negative border* of these itemsets.
- An itemset is in the negative border if it is not deemed frequent in the sample, but *all* its immediate subsets are.
  - *Immediate* = "delete exactly one element."

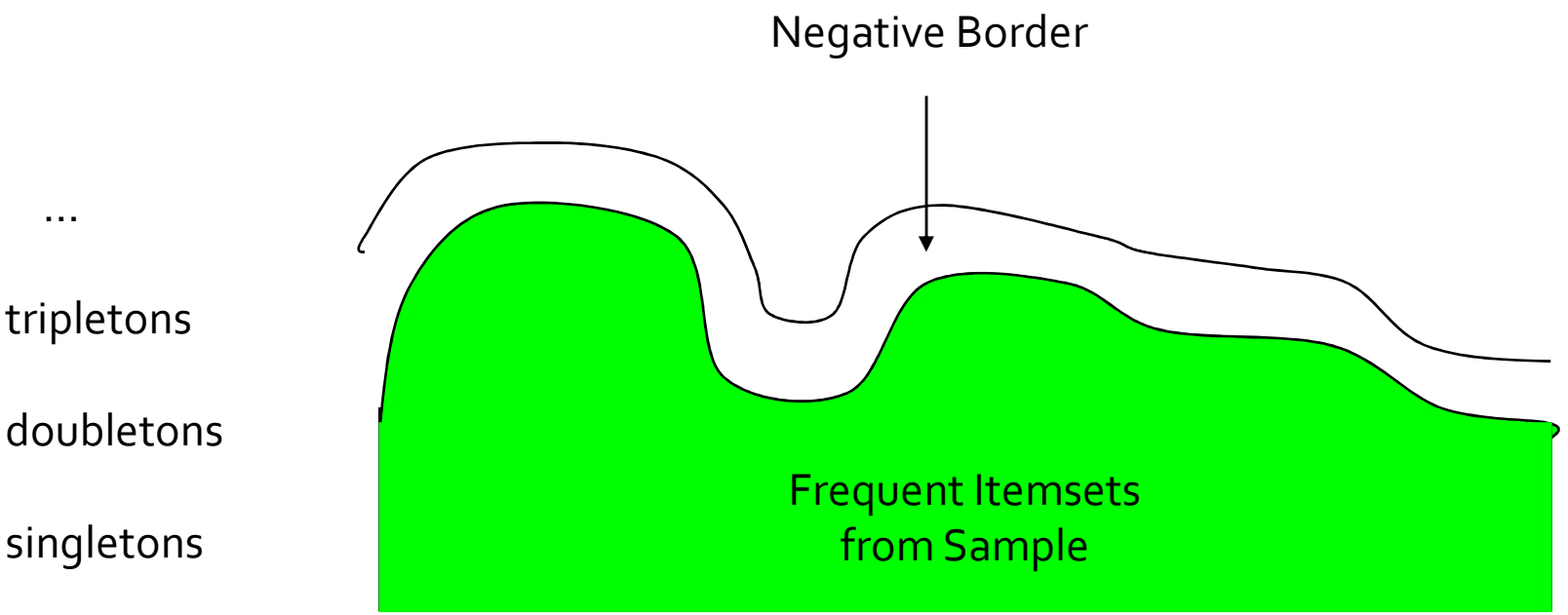
Example: Negative Border

- Items are {A,B,C,D,E}
- Frequent itemsets - {A}, {B}, {C}, {D}, {B,C},{C,D}
- Negative Border
  - {E}, {A,B}, {A,C}, {A,D} and {B,D}
  - {A,E} is not in negative border since {E} is not frequent
  - None of the tripletons are in negative border {B,C,D} is not in negative border since {B,D} is not frequent. Similarly {A,B,C} is not since {A,B} is not frequent

Example: Negative Border

- {*A*,*B*,*C*,*D*} is in the negative border if and only if:
  1. It is not frequent in the sample, but
  2. All of {*A*,*B*,*C*}, {*B*,*C*,*D*}, {*A*,*C*,*D*}, and {*A*,*B*,*D*} are.
- {*A*} is in the negative border if and only if it is not frequent in the sample.
  - Because the empty set is always frequent.

Picture of Negative Border



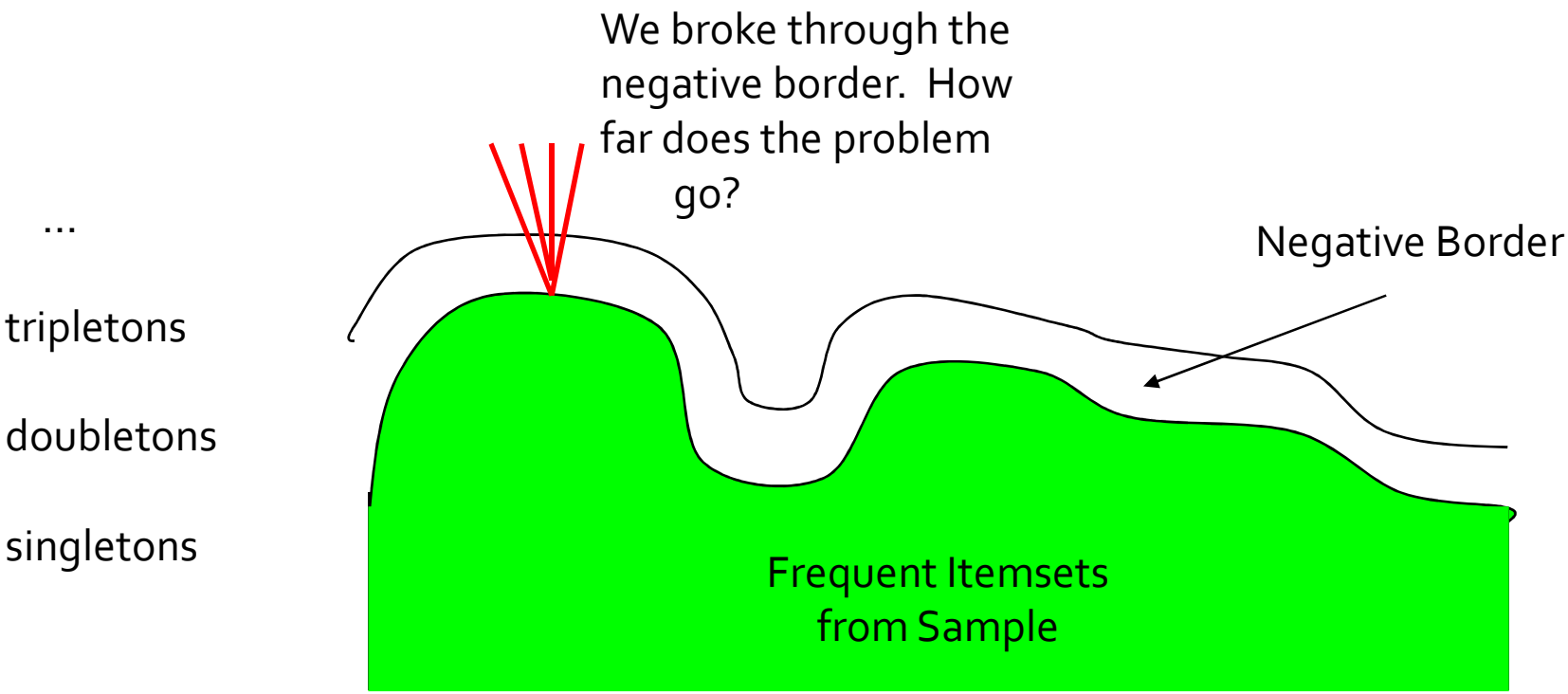
Toivonen's Algorithm – (3)

- In a second pass, count all candidate frequent itemsets from the first pass, and also count their negative border.
- If no itemset from the negative border turns out to be frequent, then the candidates found to be frequent in the whole data are *exactly* the frequent itemsets.

Toivonen's Algorithm – (4)

- What if we find that something in the negative border is actually frequent?
- We must start over again with another sample!
- Try to choose the support threshold so the probability of failure is low, while the number of itemsets checked on the second pass fits in main-memory.

If Something in the Negative Border Is Frequent . . .



Theorem:

- If there is an itemset that is frequent in the whole, but not frequent in the sample, then there is a member of the negative border for the sample that is frequent in the whole.

Proof:

- Suppose not; i.e.;
  1. There is an itemset *S* frequent in the whole but not frequent in the sample, and
  2. Nothing in the negative border is frequent in the whole.
- Let *T* be a smallest subset of *S* that is not frequent in the sample.
- *T* is frequent in the whole (*S* is frequent + monotonicity).
- *T* is in the negative border (else not "smallest").