Frequent Itemsets

The Market-Basket Model
Association Rules
A-Priori Algorithm

Mining of Massive Datasets

Leskovec, Rajaraman, and 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.

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.
- Given a *support threshold s*, sets of items that appear in at least *s* baskets are called *frequent itemsets*.

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

- Items = products; baskets = sets of products someone bought in one trip to the store.
- Example application: given that many people buy beer and diapers together:
  - Run a sale on diapers; raise price of beer.
- Only useful if many buy diapers & beer.
  - Essential for brick-and-mortar stores, not on-line stores.

Applications – (2)

- Baskets = sentences; items = documents containing those sentences.
- Items that appear together too often could represent plagiarism.
- Notice items do not have to be “in” baskets.
  - But it is better if baskets have small numbers of items, while items can be in large numbers of baskets.

Applications – (3)

- Baskets = documents; items = words.
- Unusual words appearing together in a large number of documents, e.g., “Brad” and “Angelina,” may indicate an interesting relationship.

Scale of the Problem

- WalMart sells 100,000 items and can store billions of baskets.
- The Web has billions of words and many billions of pages.

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.

Example: Confidence

$+ \quad B_1 = \{m, c, b\}$ $\qquad B_2 = \{m, p, j\}$

$- \quad B_3 = \{m, b\}$ $\qquad B_4 = \{c, j\}$

$- \quad B_5 = \{m, p, b\}$ $\qquad + \; B_6 = \{m, c, b, j\}$

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

- An association rule: $\{m, b\} \rightarrow c$.
  - Confidence = 2/4 = 50%.

Finding Association Rules

- Question: "find all association rules with support $\geq s$ and confidence $\geq c$."
  - Note: "support" of an association rule is the support of the set of items on the left.
- Hard part: finding the frequent itemsets.
  - Note: if $\{i_1, i_2, \ldots, i_k\} \to j$ has high support and confidence, then both $\{i_1, i_2, \ldots, i_k\}$ and $\{i_1, i_2, \ldots, i_k, j\}$ will be "frequent."

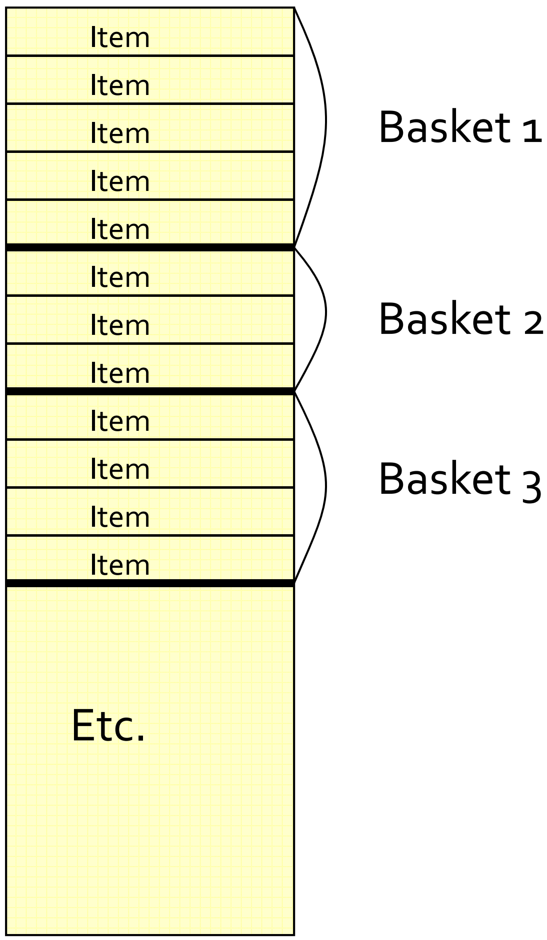
Finding Association Rules – (2)

1. Find all sets with support at least cs.
2. Find all sets with support at least s.
3. If $\{i_1, i_2, \ldots, i_k, j\}$ has support at least cs, see which subsets missing one element have support at least s.
   - Take j to be the missing element.
4. $\{i_1, i_2, \ldots, i_k\} \rightarrow j$ is an acceptable association rule if $\{i_1, i_2, \ldots, i_k\}$ has support $s_1 \geq s$, $\{i_1, i_2, \ldots, i_k, j\}$ has support $s_2 \geq cs$, and s_2/s_1, the confidence of the rule, is at least c.

Computation Model

- Typically, data is kept in flat files.
- Stored on disk.
- Stored basket-by-basket.
- Expand baskets into pairs, triples, etc. as you read baskets.
  - Use k nested loops to generate all sets of size k.

File Organization



Example: items are positive integers, and boundaries between baskets are –1.

Computation Model – (2)

- 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.
- The number of different things we can count is limited by main memory.
- Swapping counts in/out is a disaster.

Finding Frequent Pairs

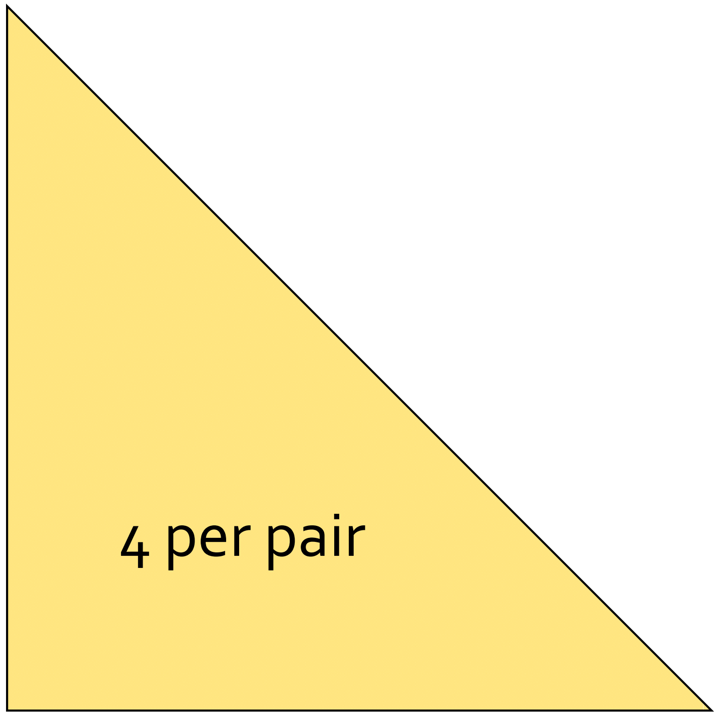
- The hardest problem often turns out to be finding the frequent pairs.
  - Why? Often frequent pairs are common, frequent triples are rare.
    - Why? Support threshold is usually set high enough that you don't get too many frequent itemsets.
- We'll concentrate on pairs, then extend to larger sets.

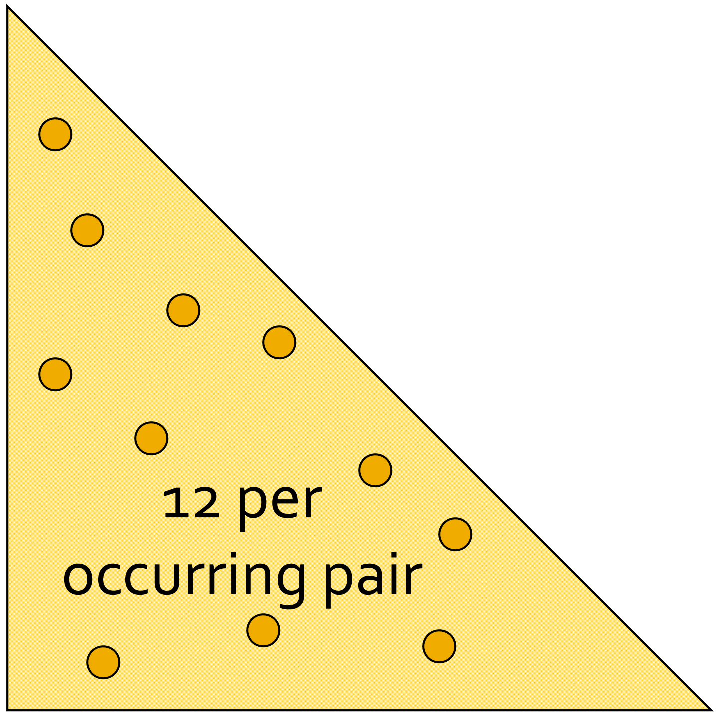
Naïve Algorithm

- 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 $(\#items)^2$ exceeds main memory.
  - Remember: #items can be 100K (Wal-Mart) or 100B (Web pages).

Details of Main-Memory Counting

- Two approaches:
  1. Count all pairs, using a triangular matrix.
  2. Keep a table of triples [*i*, *j*, *c*] = "the count of the pair of items {*i*, *j*} is *c*."
- (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.





Triangular matrix

Tabular method

Triangular-Matrix Approach

- Number items 1, 2,…
  - Requires table of size O(n) to convert item names to consecutive integers.
- Count $\{i, j\}$ only if $i < j$.
- Keep pairs in the order $\{1,2\}$, $\{1,3\}$,…, $\{1,n\}$, $\{2,3\}$, $\{2,4\}$,…,$\{2,n\}$, $\{3,4\}$,…, $\{3,n\}$,…$\{n-1,n\}$.

Triangular-Matrix Approach – (2)

- Find pair $\{i, j\}$, where i<j, at the position:

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

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

Details of Tabular Approach

- 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, 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 of items appears at least *s* times, so does every subset of *s*.
- Contrapositive for pairs: if item *i* does not appear in *s* baskets, then no pair including *i* can appear in *s* baskets.

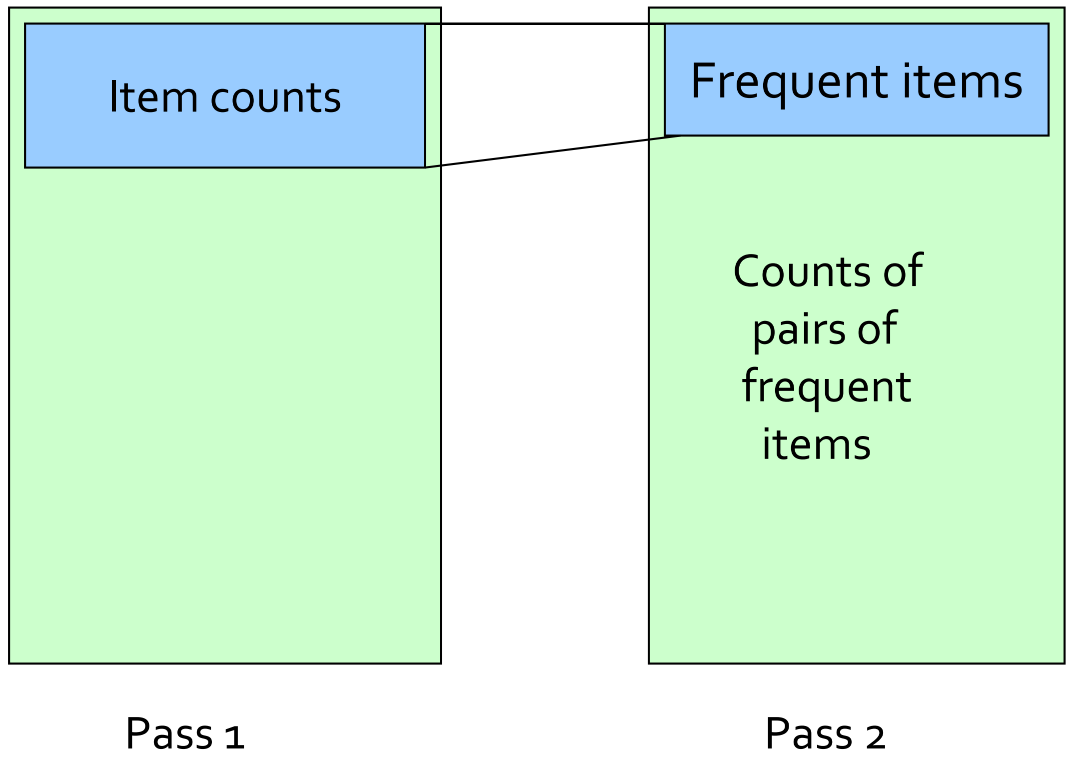
A-Priori Algorithm – (2)

- Pass 1: Read baskets and 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).

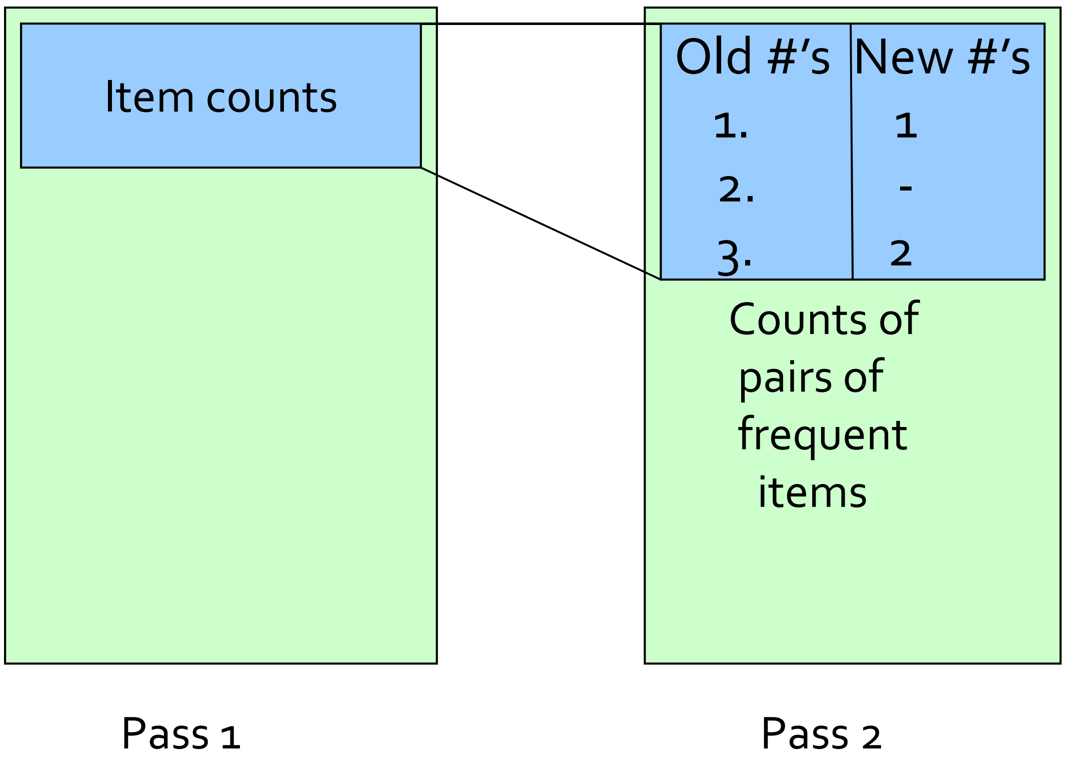
Picture of A-Priori



Detail for A-Priori

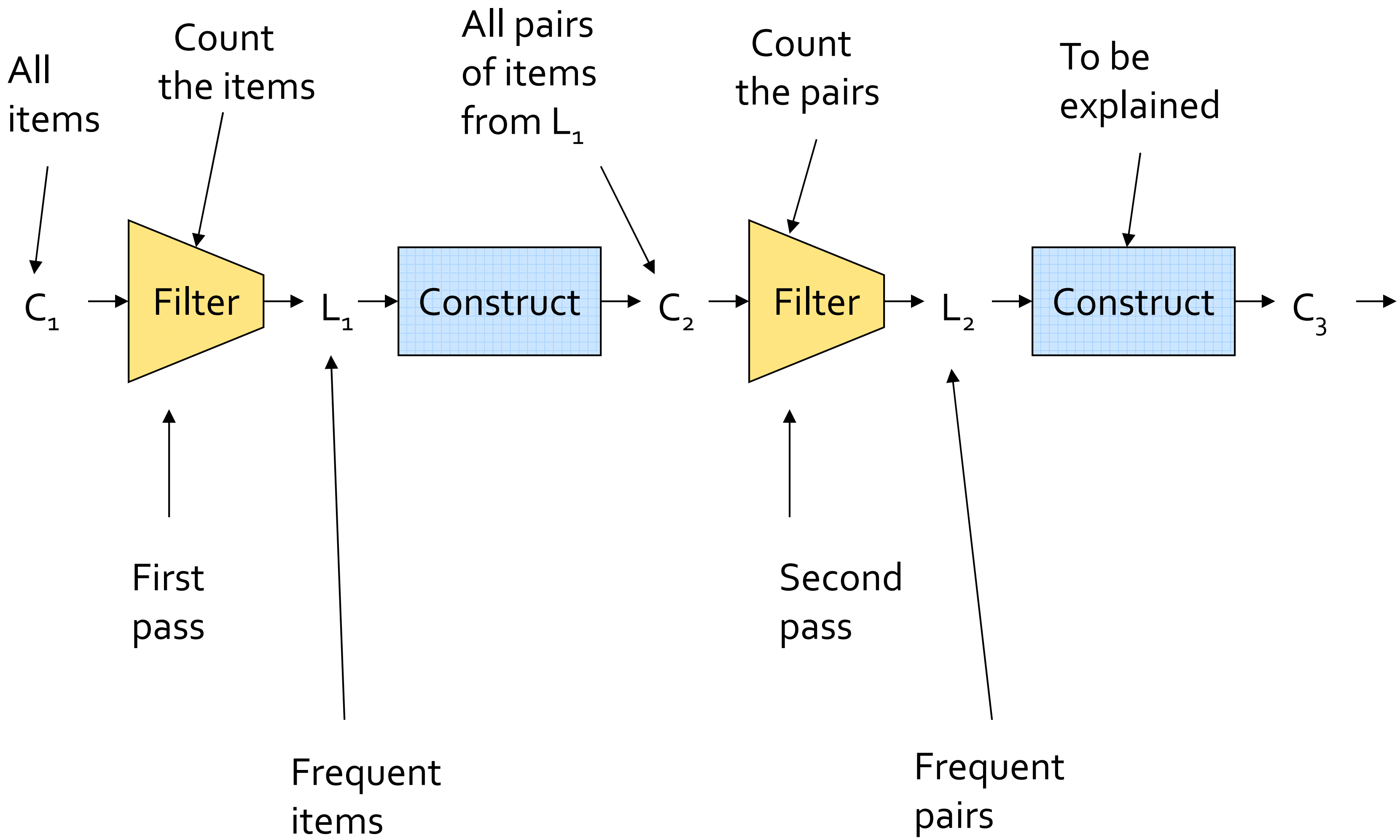
- You can use the triangular matrix method with n = number of frequent items.
  - May save space compared with storing triples.
- Trick: number frequent items 1,2,... and keep a table relating new numbers to original item numbers.

A-Priori Using Triangular Matrix



Frequent Triples, Etc.

- For each 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 $k - 1$.
  - 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
Frequent items
Second pass
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.

Improvements to A-Priori

Park-Chen-Yu Algorithm
Multistage and Multihash
Single-Pass Approximate Algorithms

Mining of Massive Datasets

Leskovec, Rajaraman, and Ullman

Stanford University

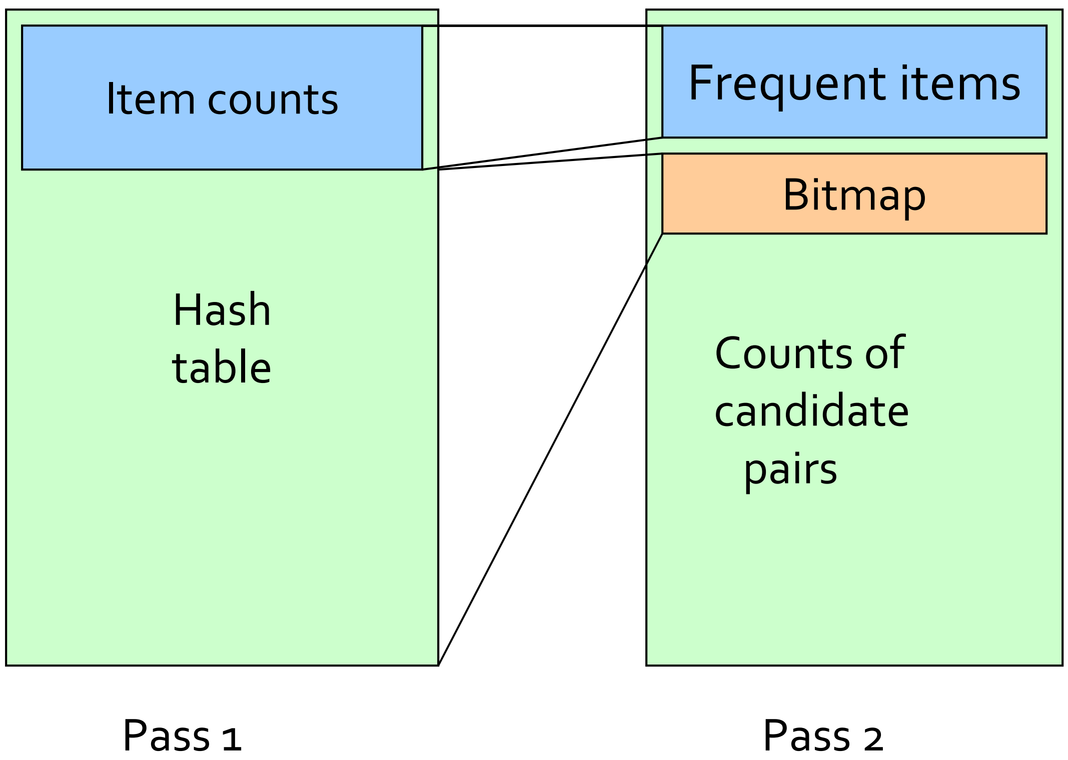
PCY Algorithm

- During Pass 1 of A-priori, most memory is idle.
- 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.

PCY Algorithm – (2)

- 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 that hash to frequent buckets.

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, 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.
- 4-byte integers are replaced by bits, so the bit-vector requires 1/32 of memory.
- 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.

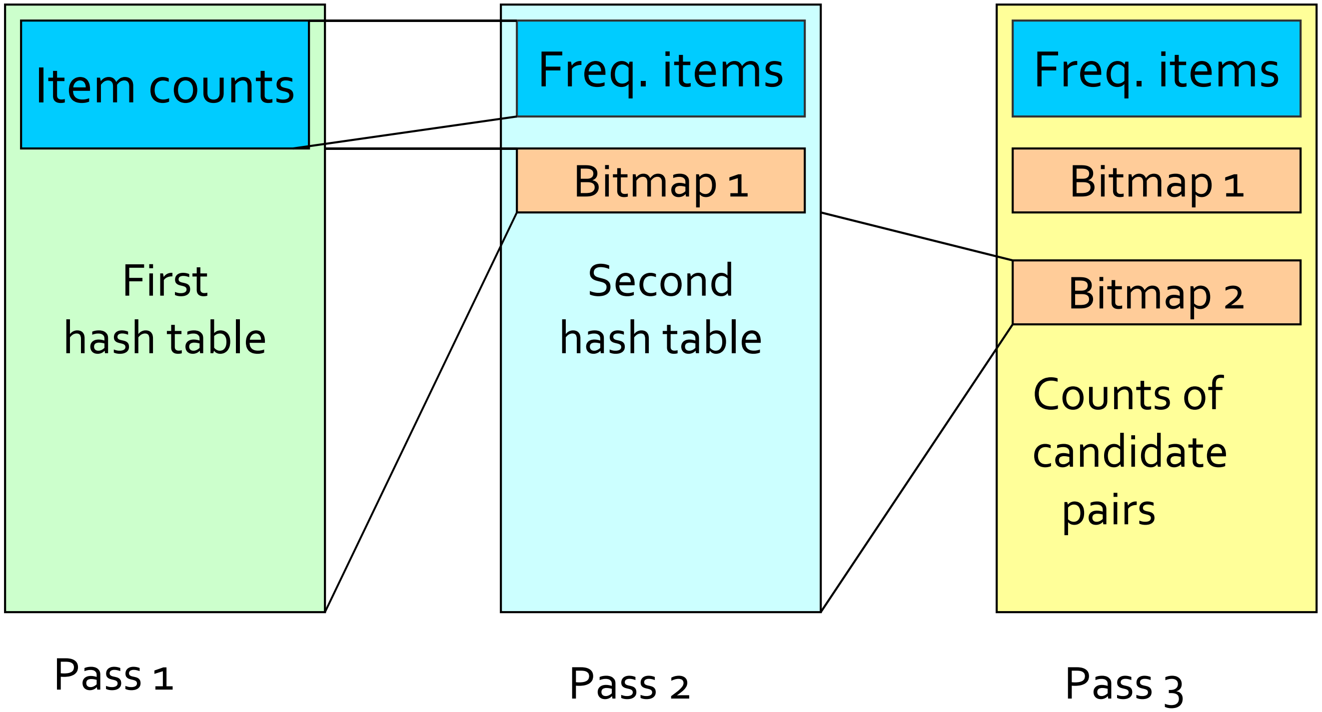
Memory Details

- 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.
  - Thus, hash table must eliminate 2/3 of the candidate pairs for PCY to beat a-priori.

Multistage Algorithm

- Key idea: After Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY.
- On middle pass, fewer pairs contribute to buckets, so fewer *false positives* – frequent buckets with no frequent pair.

Multistage Picture



Multistage – Pass 3

- Count only those pairs {*i*, *j*} that satisfy these candidate pair conditions:
  1. Both *i* and *j* are frequent items.
  2. Using the first hash function, the pair hashes to a bucket whose bit in the first bit-vector is 1.
  3. Using the second hash function, the pair hashes to a bucket whose bit in the second bit-vector is 1.

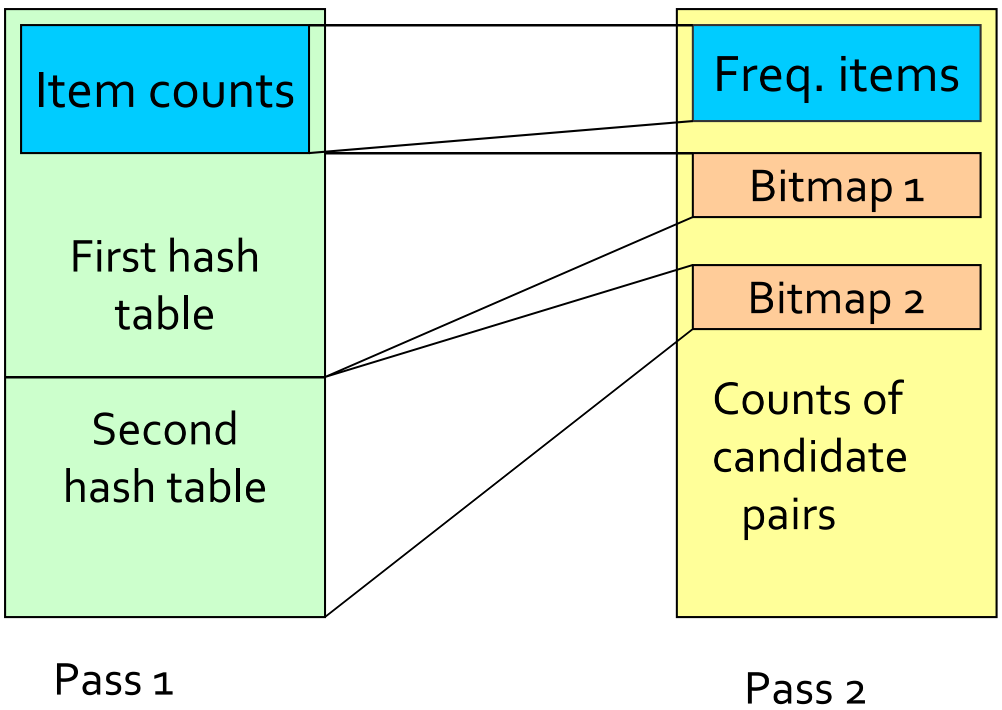
Important Points

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

Multihash

- Key idea: use several independent hash tables on the first pass.
- Risk: halving the number of buckets doubles the average count. We have to be sure most buckets will still not reach count *s*.
- If so, we can get a benefit like multistage, but in only 2 passes.

Multihash Picture



All (Or Most) Frequent Itemsets In $\leq$ 2 Passes

Simple Algorithm

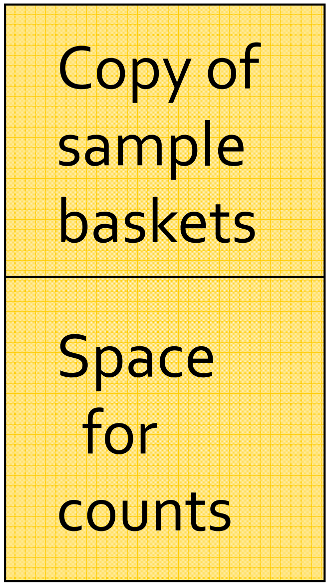
Savasere-Omiecinski- Navathe (SON) Algorithm

Toivonen's Algorithm

Simple Algorithm

- Take a random sample of the market baskets.
- Run a-priori or one of its improvements (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.
- Use as your support threshold a suitable, scaled-back number.
  - Example: if your sample is 1/100 of the baskets, use $s/100$ as your support threshold instead of s.

Main-Memory Picture



Simple Algorithm – Option

- 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

- Repeatedly read small subsets of the baskets into main memory and perform the first pass of the simple algorithm on each subset.
- An itemset becomes a candidate if it is found to be frequent in *any* one or more subsets of the baskets.

SON Algorithm – Pass 2

- On a second pass, count all the candidate itemsets and determine which are frequent in the entire set.
- Key "monotonicity" idea: an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset.

SON Algorithm – Distributed Version

- This idea lends itself to distributed data mining.
- If baskets are distributed among many nodes, compute *local* frequent itemsets at each node, then distribute the candidates from each node.
- Each node counts all the candidate itemsets.
- Finally, accumulate the counts of all candidates.

Toivonen's Algorithm

- 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.

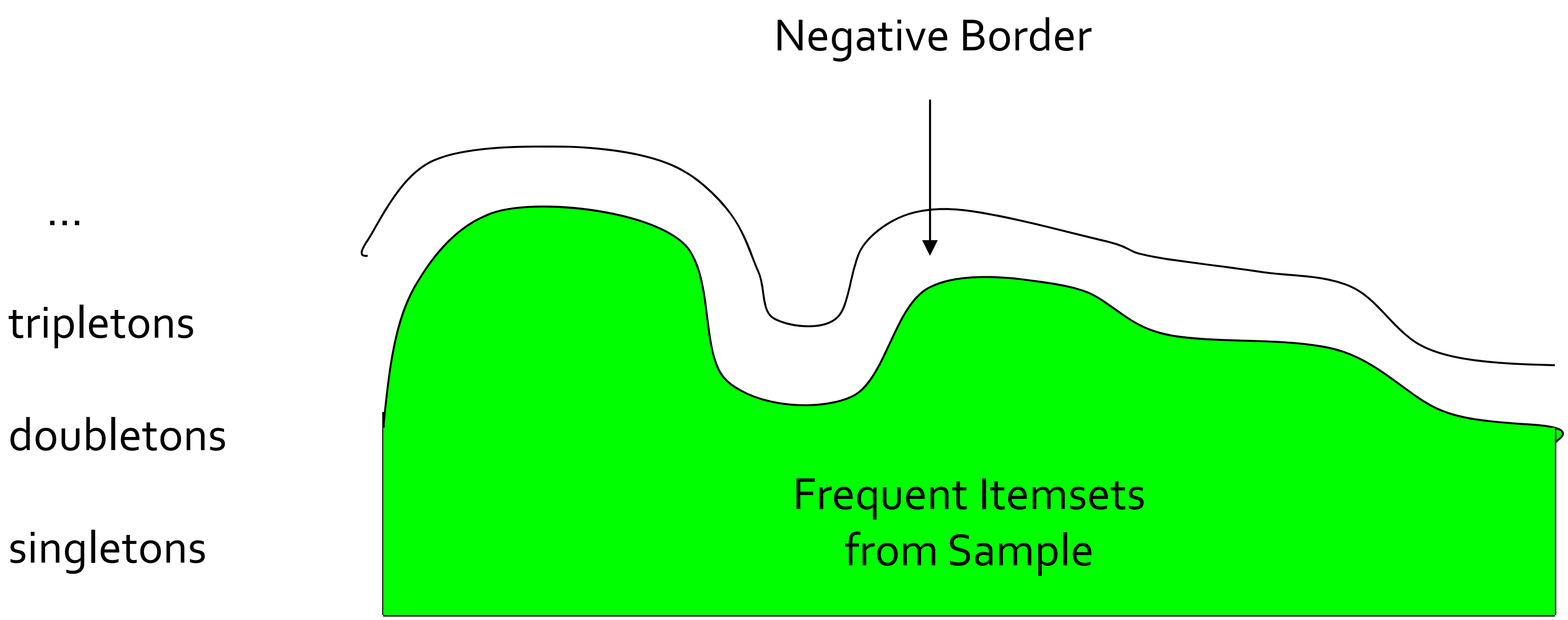
Toivonen's Algorithm – (2)

- 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.

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.
    - Unless there are fewer baskets than the support threshold (silly case).

Picture of Negative Border



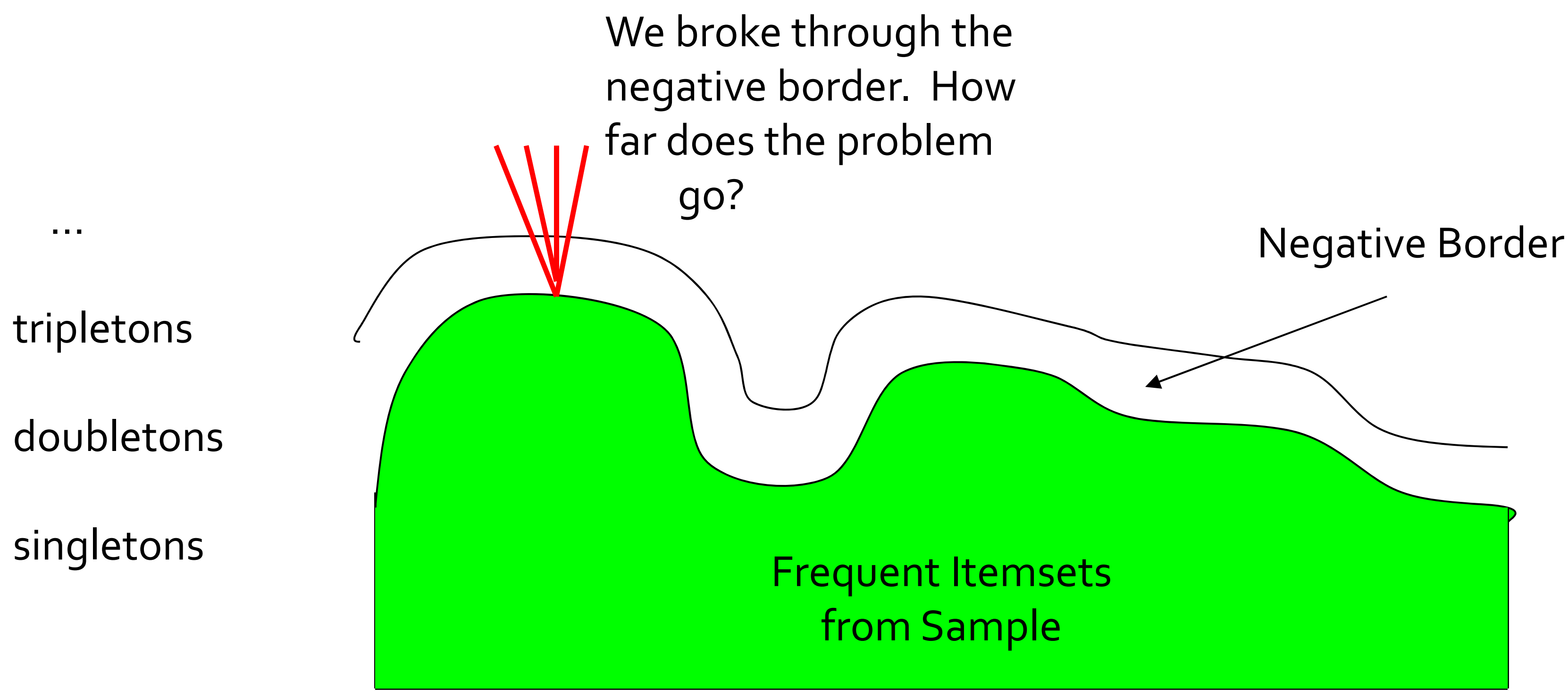
Toivonen's Algorithm – (3)

- 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").