Association Rules

CS345a: Data Mining
Jure Leskovec and Anand Rajaraman
Stanford University

Slides adapted from lectures by Jeff Ullman

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
- Can be used to model any many-many relationship, not just in the retail setting
- Find "interesting" connections between items

Frequent Itemsets

- Simplest question: Find sets of items that appear together "frequently" in baskets
- *Support* for itemset *I* = the number of baskets containing all items in *I*
  - Often expressed as a fraction of the total number of baskets
- Given a *supportthreshold**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 – (1)

- Items = products; baskets = sets of products someone bought in one trip to the store

- Suppose many people buy beer and diapers together
  - Run a sale on diapers; raise price of beer

- Only useful if many buy diapers &beer

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
- Baskets = Web pages; items = words.
  - Co-occurrence of relatively rare words, e.g., “Brad” and “Angelina,” may indicate an interesting relationship

Applications – (3)

- Baskets = patients; items = drugs and side-effects
- Baskets = movies; items = Oscar nominations and wins in different categories
  - Does being nominated in certain categories boost win likelihood in other categories?
  - See my blog post on the “Oscar Halo” (http://datawocky.com/2009/02/oscar-halo-academy-awards-and-the-matthew-effect.html)

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 $I = \{i_1, \ldots, i_k\}$

Interesting Association Rules

- Not all high-confidence rules are interesting
  - The rule X → milk may have high confidence for many itemsets X, because milk is just purchased very often (independent of X)

- Interesting rules are those with high positive or negative interest values

Example: Confidence and Interest

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

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

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

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

- Association rule: $\{m, b\} \rightarrow c$.
  - Confidence = 2/4 = 0.5
  - Interest = 0.5 – 4/8 = 0

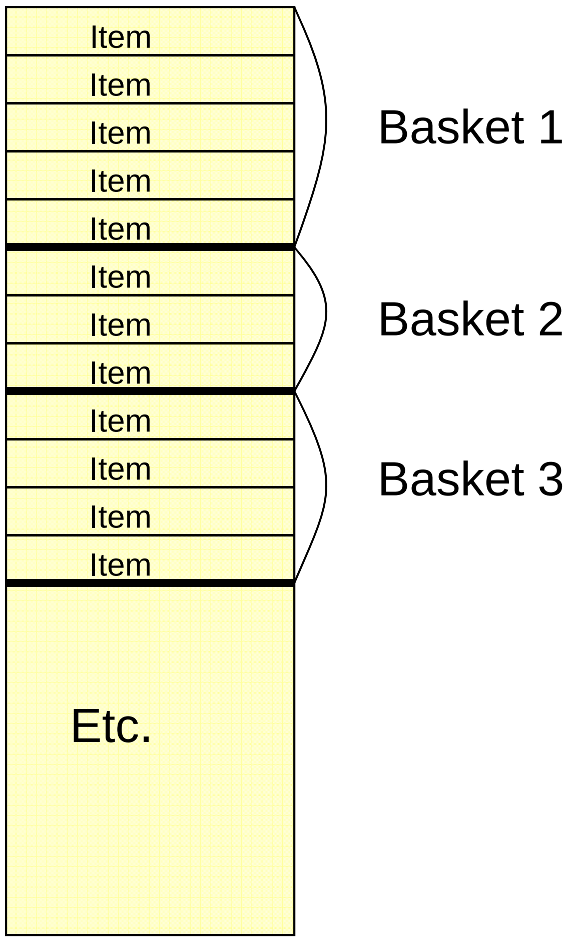
Finding Association Rules

- Problem: 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.
  - If $\{i_1, i_2,\ldots,i_k\} \rightarrow j$ 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."

Computation Model

- Typically, data is kept in flat files rather than in a database system.
  - 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, association-rule algorithms read the data in *passes* – all baskets read in turn
- 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 (why?)

Finding Frequent Pairs

- The hardest problem often turns out to be finding the frequent pairs.
  - Often frequent pairs are common, frequent triples are rare.
  - Probability of being frequent drops exponentially with size; number of sets grows more slowly with size.
- 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.
  - #items can be 100K (Wal-Mart) or 10B (Web pages).

Counting pairs in memory

- Approach 1: Store triples [i, j, c] where count(i , j) = c
  - If integers and and item ids are 4 bytes, needs approximately 12 bytes for pairs with count > 0
  - Plus some additional overhead for a hashtable

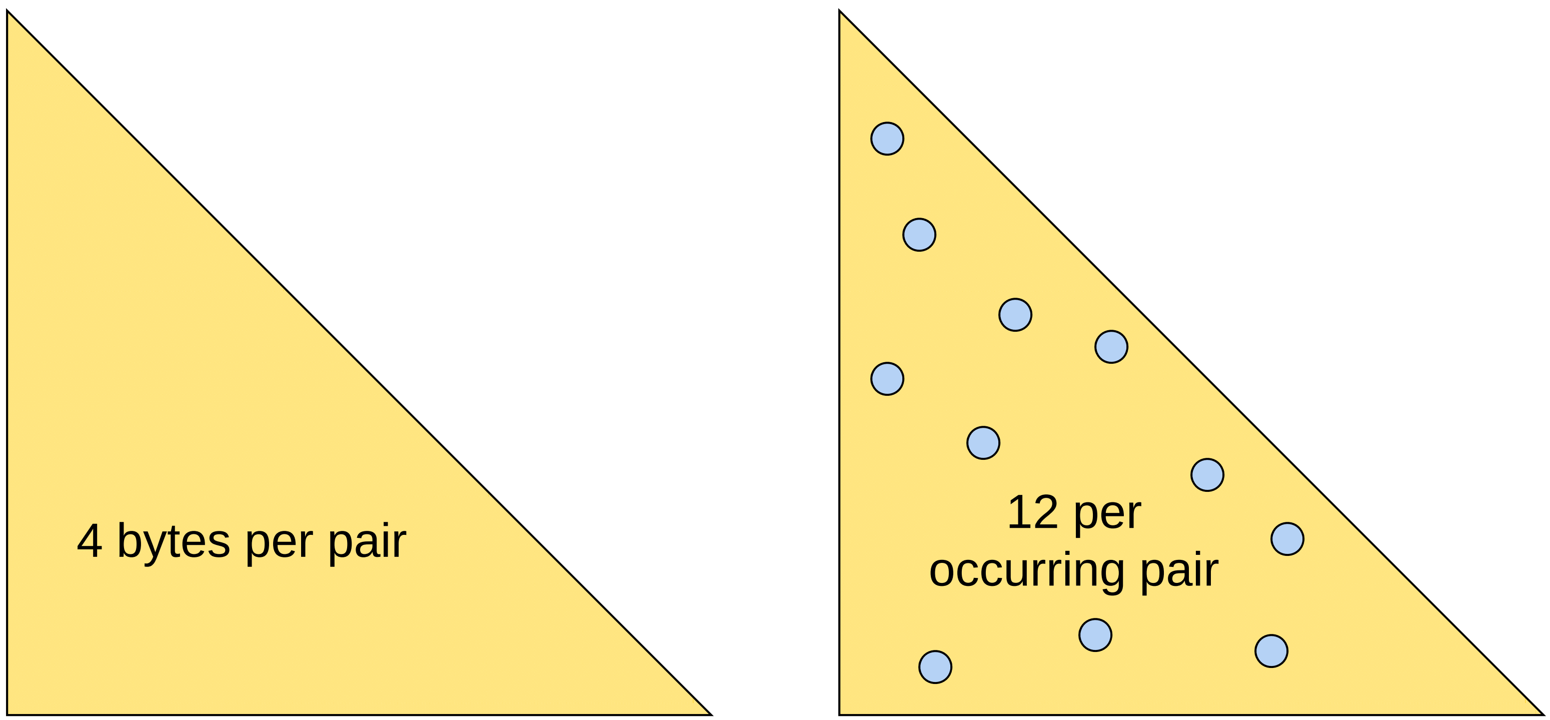
- What if most pairs occur, even if infrequently?

Triangular Matrix Approach

- Approach 2: Count all pairs
  - Number items 1,2,3,…
  - Count $\{i, j\}$ only if $i<j$.
- Keep pair counts in lexicographic order:
  - $\{1,2\}, \{1,3\},\ldots, \{1,n\}, \{2,3\}, \{2,4\},\ldots,\{2,n\}, \{3,4\},\ldots$
- Pair $\{i, j\}$ is at position $(i-1)(n-i/2) + j-i$

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

Comparing approaches



Triangular Matrix

Triples

A-Priori Algorithm – (1)

- 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.
- Contrapositive for pairs: If item i does not appear in s baskets, then no pair including i can appear in s baskets.

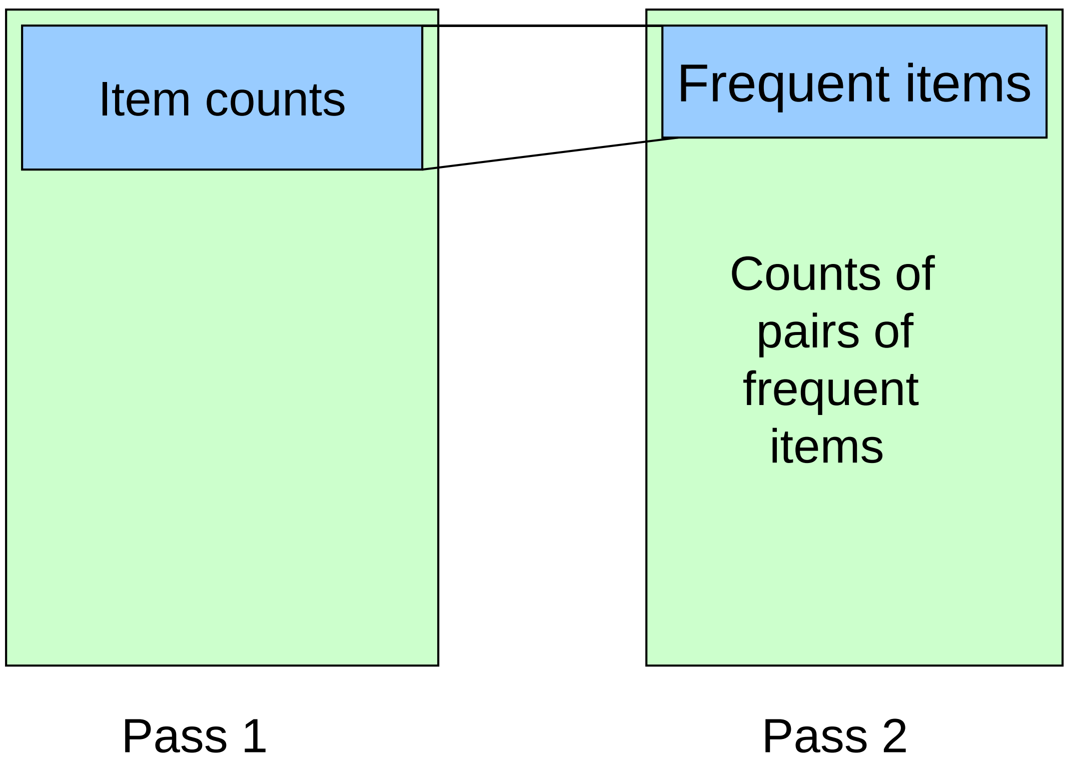
A-Priori Algorithm – (2)

- 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



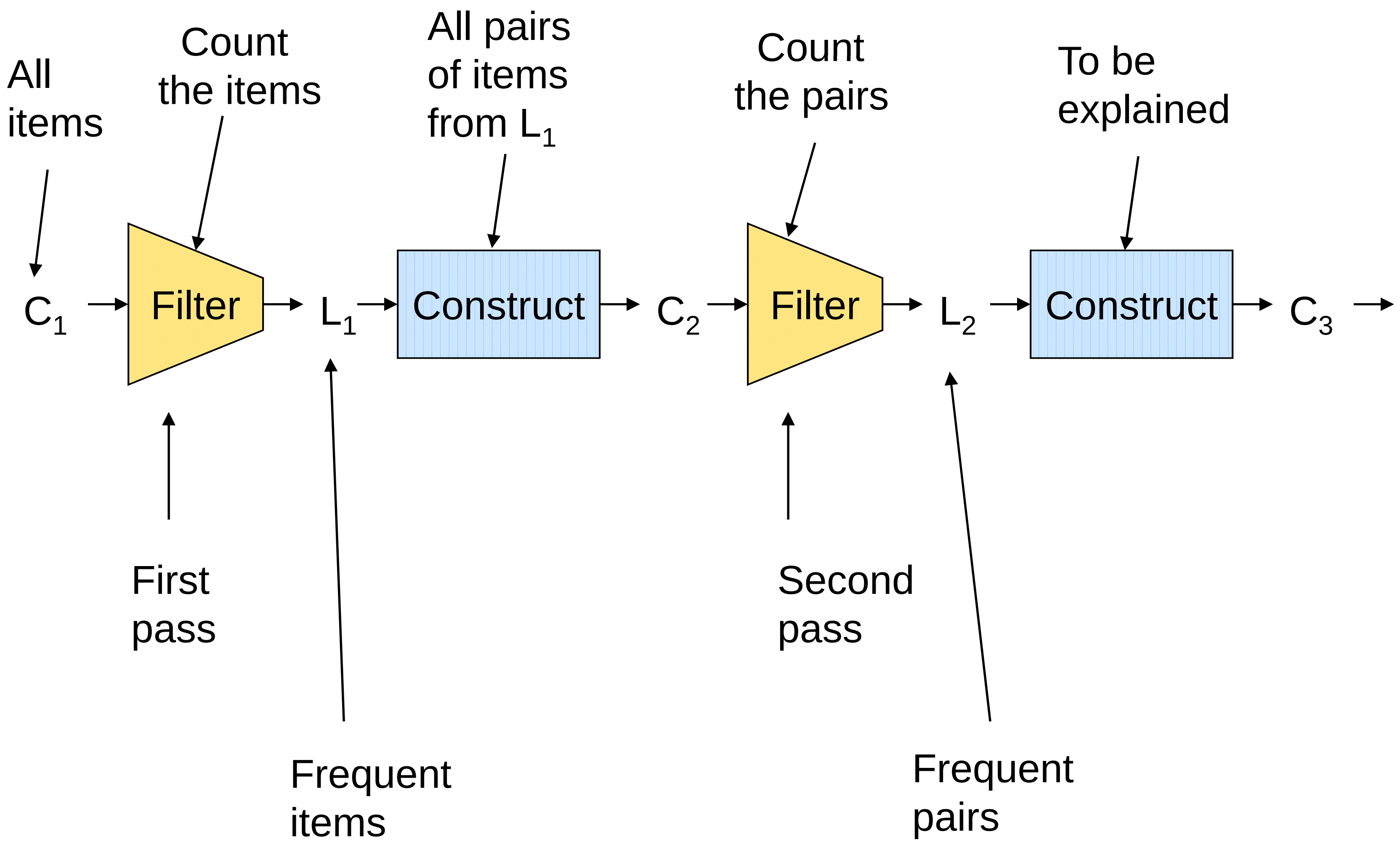
Detail for A-Priori

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

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 based on information from the pass for k –1.
  - L_k = the set of truly frequent k -sets.

Frequent Itemsets – (1)



Frequent Itemsets – (2)

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

A-Priori for All Frequent Itemsets

- One pass for each *k*
- Needs room in main memory to count each candidate *k*–set
- For typical market-basket data and reasonable support (e.g., 1%), *k* = 2 requires the most memory.

PCY Algorithm

- Observation: In pass 1 of a-priori, most memory is idle
  - We store only individual item counts
  - Can we use the idle memory to reduce memory required in pass 2?
- Pass 1 of PCY: In addition to item counts, maintain a hash table with as many buckets as will fit in memory

PCY Algorithm – First Pass

```
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. For a bucket with total count less than *s*, none of its pairs can be frequent
2. A bucket that a frequent pair hashes to is surely frequent
3. Even without any frequent pair, a bucket can be frequent

We can surely eliminate all pairs that hash into buckets of Type (1)

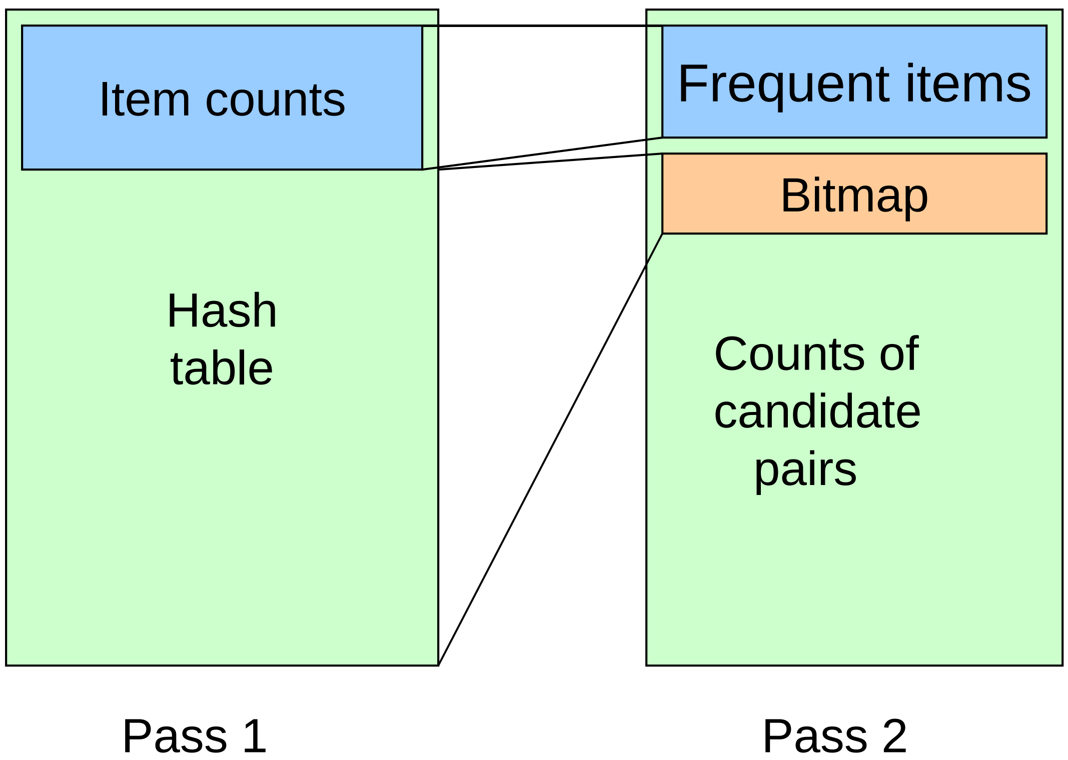
PCY Algorithm – Between Passes

- Replace the buckets by a bit-vector:
  - 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.

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.
- Notice all these conditions are necessary for the pair to have a chance of being frequent.

Picture of PCY



Memory Details

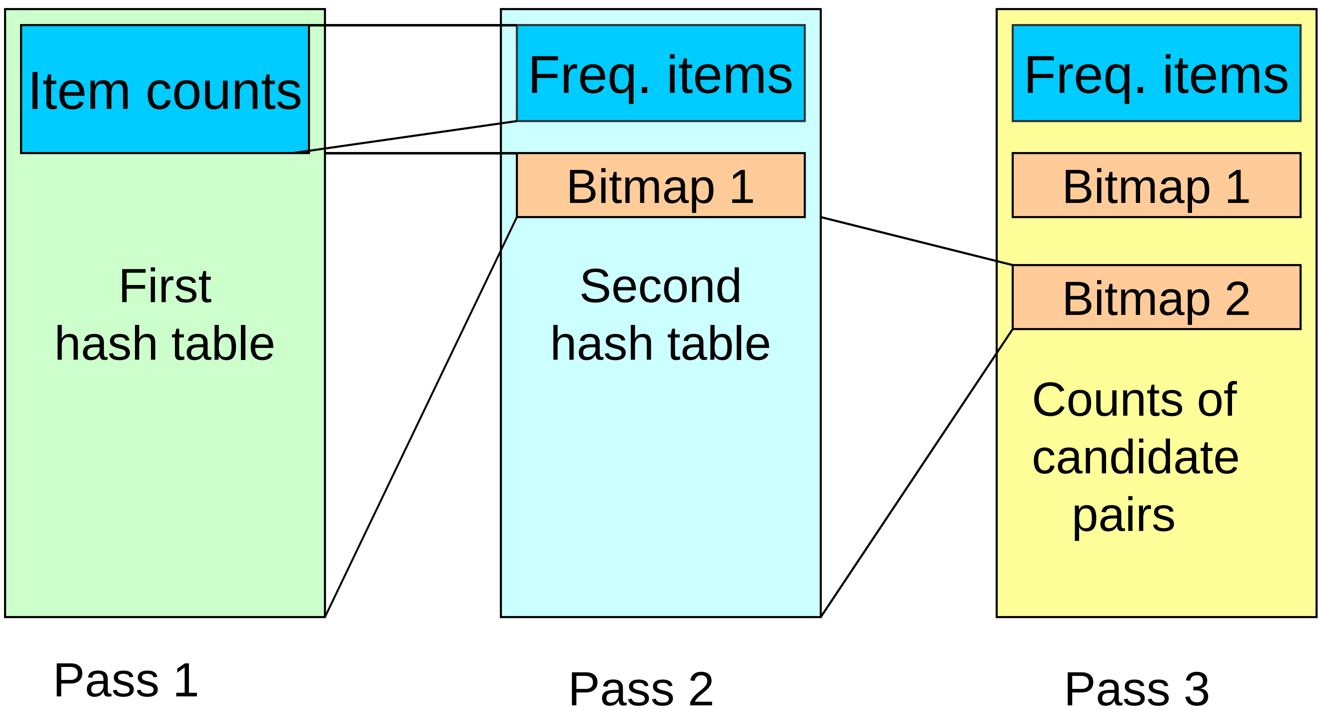
- Buckets require a few bytes each.
  - Note: we don't have to count past *s*.
  - # buckets is O(main-memory size)
- On second pass, a table of (item, item, count) triples is essential (why?)
  - Hash table must eliminate approx. 2/3 of the candidate pairs for PCY to beat a-priori.

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



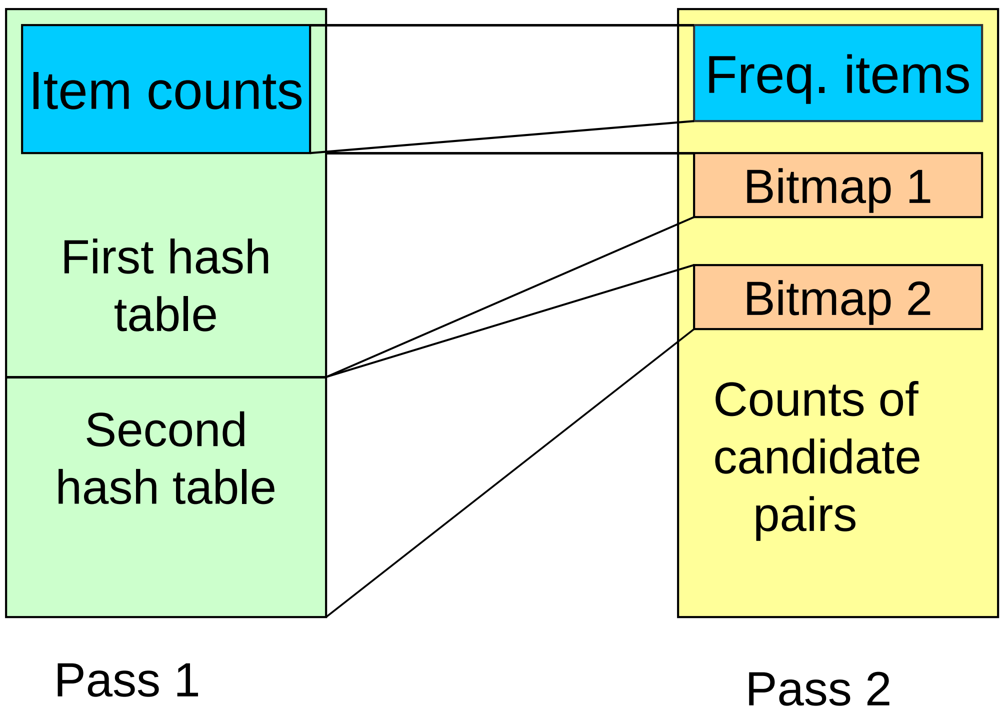
Multistage – Pass 3

- 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.

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



Frequent Itemsets In $\leq$ 2 Passes

- A-Priori, PCY, etc., take *k* passes to find frequent itemsets of size *k*.
- Other techniques use 2 or fewer passes for all sizes, but may miss some frequent itemsets
  - Random sampling
  - SON (Savasere, Omiecinski, and Navathe)
  - Toivonen (see textbook)

Random Sampling (1)

- Take a random sample of the market baskets
- Run a-priori or one of its improvementsin main memory
  - So we don't pay for disk I/O each time we increase the size of itemsets
  - Reduce support threshold proportionally to match sample size

Random Sampling (2)

- Optionally, verify that the candidate pairs are truly frequent in the entire data set by a second pass (avoid false positives)

- May miss some frequent itemsets
  - Smaller threshold helps catch more truly frequent itemsets.

SON Algorithm – (1)

- Repeatedly read small subsets of the baskets into main memory and run an in-memory algorithm to find all frequent itemsets
  - Note: we are not sampling, but processing the entire file in memory-sized chunks

- An itemset becomes a candidate if it is found to be frequent in *any* one or more subsets of the baskets.

SON Algorithm – (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

- SON lends itself to distributed data mining

- Baskets distributed among many nodes
  - Compute frequent itemsets at each node
  - Distribute candidates to all nodes
  - Accumulate the counts of all candidates.

SON: Map/Reduce

- Phase 1: Find candidate itemsets
  - Map?
  - Reduce?

- Phase 2: Find true frequent itemsets
  - Map?
  - Reduce?

Compacting the Output

1. *Maximal Frequent itemsets*: no immediate superset is frequent

2. *Closed itemsets*: no immediate superset has the same count (> 0).
   - Stores not only frequent information, but exact counts.

Example: Maximal/Closed

	Count	Maximal (s=3)	Closed
A	4	No	No
B	5	No	Yes
C	3	No	No
AB	4	Yes	Yes
AC	2	No	No
BC	3	Yes	Yes
ABC	2	No	Yes

Frequent, but superset BC also frequent.

Frequent, and its only superset, ABC, not freq.

Superset BC has same count.

Its only super-set, ABC, has smaller count.