Lecture #2
Frequent Pattern Mining: Itemsets, Association Rules, Algorithms, and Applications*

* Slides partly based on Jiawei Han, and Jeff Ullman

Frequent Pattern Mining: Overview

- **Frequent pattern**
  - a pattern (a set of items, subsequences, substructures, etc.) that occurs frequently in a data set
- **Motivation:** Finding inherent regularities in data
  - What products were often purchased together in supermarket?
  - What are the subsequent purchases after buying a PC?
  - What kinds of DNA are sensitive to this new drug?

Frequent Pattern Mining: Overview

- **Applications**
  - Market-Basket analysis to improve sales
  - Web log (click stream) analysis
  - DNA sequence analysis
  - Plagiarism detection
  - …
- First proposed by Agrawal, Imielinski, and Swami [1993] in the context of frequent itemsets and association rule mining

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
  - baskets are also referred to as transactions in the literature

Frequent Itemsets: Concept

- **Simple computational question:** find sets of items that appear "frequently" in the baskets.

- Support for itemset *I*
  - *Absolute:* the number of baskets containing all items in *I*
  - *Relative:* the fraction of baskets that contain items in *I* (i.e., the probability that a basket contains items in *I*)

- Frequent Itemset
  - Given a *support threshold* *s*, a set of items appearing in at least *s* baskets is called a *frequent itemset*

Frequent Itemsets: Example

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

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}

Frequent Itemsets: 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
  - Items = documents; baskets = sentences
  - Basket/sentence contains all the items/documents that have that sentence
  - MOSS software: https://theory.stanford.edu/~aiken/moss/

Frequent Patterns: Combinatorial Explosion and ``Closed'' Patterns

- A long pattern contains a combinatorial number of sub-patterns, e.g., $\{a_1, \ldots, a_{100}\}$ contains $\binom{100}{1} + \binom{100}{2} + \ldots + \binom{100}{100} = 2^{100} - 1 = 1.27 \times 10^{30}$ sub-patterns!
- <u>Solution:</u> Mine ``closed'' patterns
- An itemset I is <u>closed</u> if I is *frequent* and there exists *no super-set* $J \supset I$, *with the same support* as I

- Closed pattern is a lossless compression of frequent patterns: Reducing the # of patterns and rules

Association Rules: Concept

- 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"
  - Example: {bread, peanut-butter} → jelly
- *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

Subtle point: "probability" implies there is a process generating random baskets. Really we're just computing the fraction of baskets, because we're computer scientists, not statisticians.

Association Rules: Example

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

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

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

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

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

Computation Model

- Typically, data is a file consisting of a list of baskets
- The true cost of mining disk-resident data is usually the number of disk I/O's
- In practice, we 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 mining 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

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
  - Support threshold is usually set high enough that you don't get too many frequent itemsets

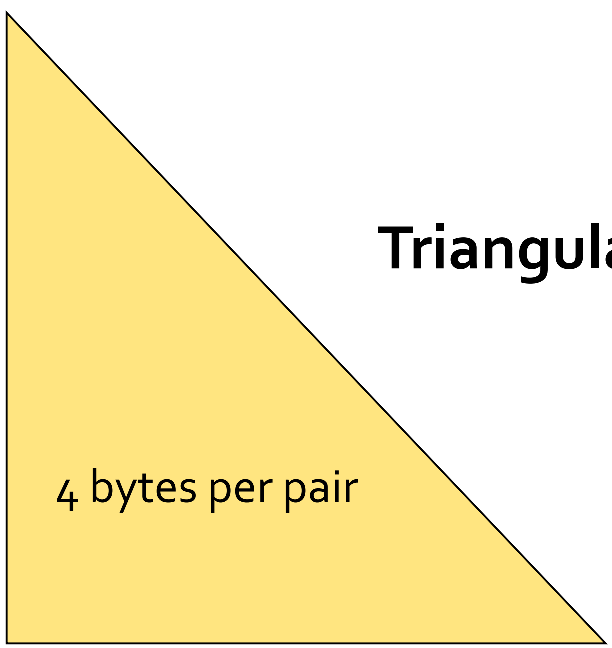
- We'll first concentrate on computing frequent pairs, and then extend to larger sets.

Finding Frequent Pairs: 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
  - Example 1: Walmart sells 100K items, so probably OK.
  - Example 2: Web has 100B pages, so definitely not OK.

Counting in Main Memory: Two Approaches

- Approach 1: Count all pairs using a triangular matrix
  - Count[i,j] in row i, column j, provided i < j
  - use a “ragged array,” so the empty triangle is not there (more details later)

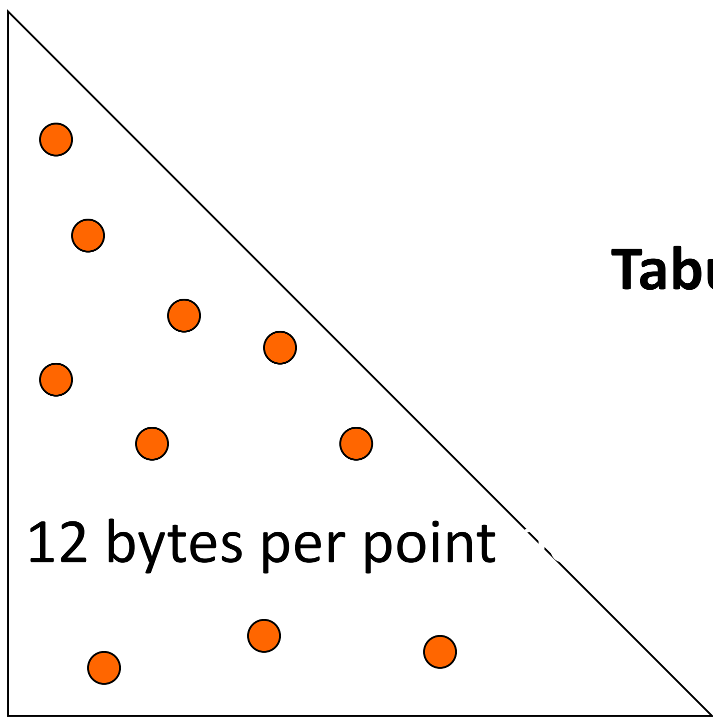


Triangular matrix

Note: assume integers are 4 bytes

Counting in Main Memory: Two Approaches

- Approach 2: Store a table of pairs with count > 0
  - Keep a table of triples [i, j, c] = "the count of the pair of items {i, j} is c"



Tabular method

Note: assume integers are 4 bytes

Triangular Matrix vs. Tabular Method

- **Triangular matrix**
  - Requires 4 bytes per pair
- **Tabular method**
  - Requires 12 bytes per pair, but only for those pairs with count > 0

- **Comparison:** tabular approach beats triangular matrix only when at most 1/3 of all pairs have a nonzero count

Triangular Matrix as One-Dimensional Array

- Number items $1, 2, \ldots, n$
  - 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 $\{1,2\}, \{1,3\}, \ldots, \{1,n\}, \{2,3\}, \{2,4\}, \ldots, \{2,n\}, \{3,4\}, \ldots, \{3,n\}, \ldots, \{n-1,n\}$
- Find pair $\{i, j\}$, where $i<j$, at the position:

$$\boxed{(i-1)(n-i/2)+j-i}$$

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

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 the set

- 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

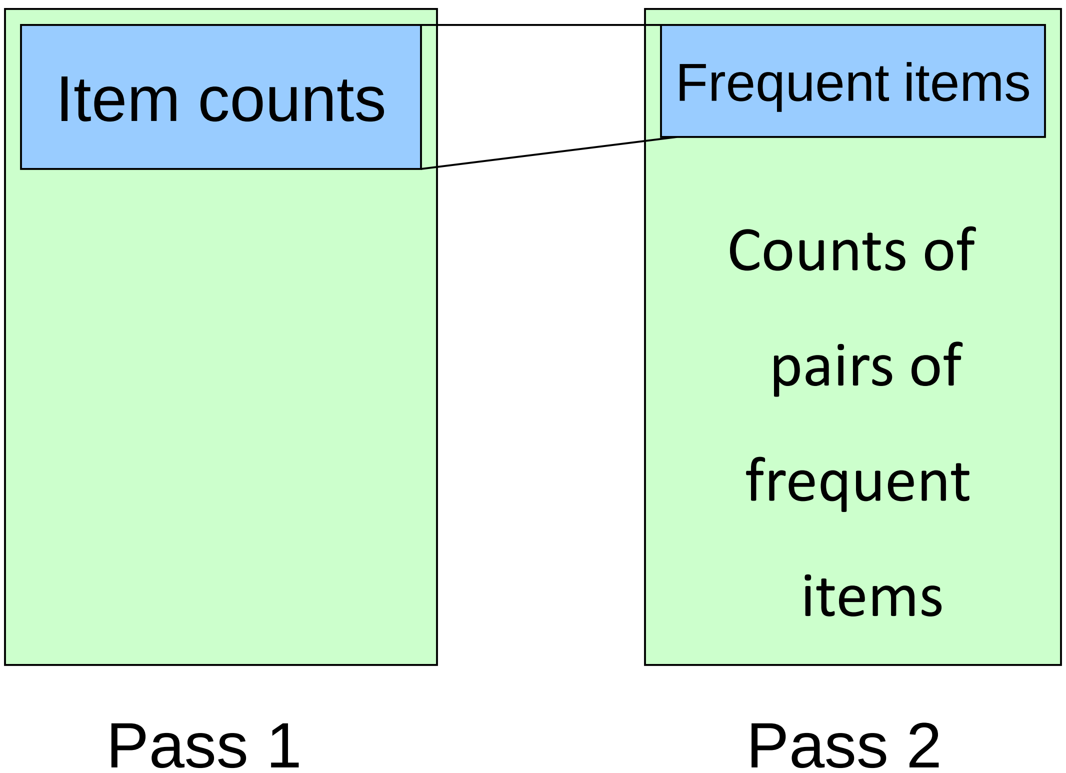
- Items that appear at least *s* times are the *frequent items*

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 table of the frequent items (so you know what must be counted).

Picture of A-Priori



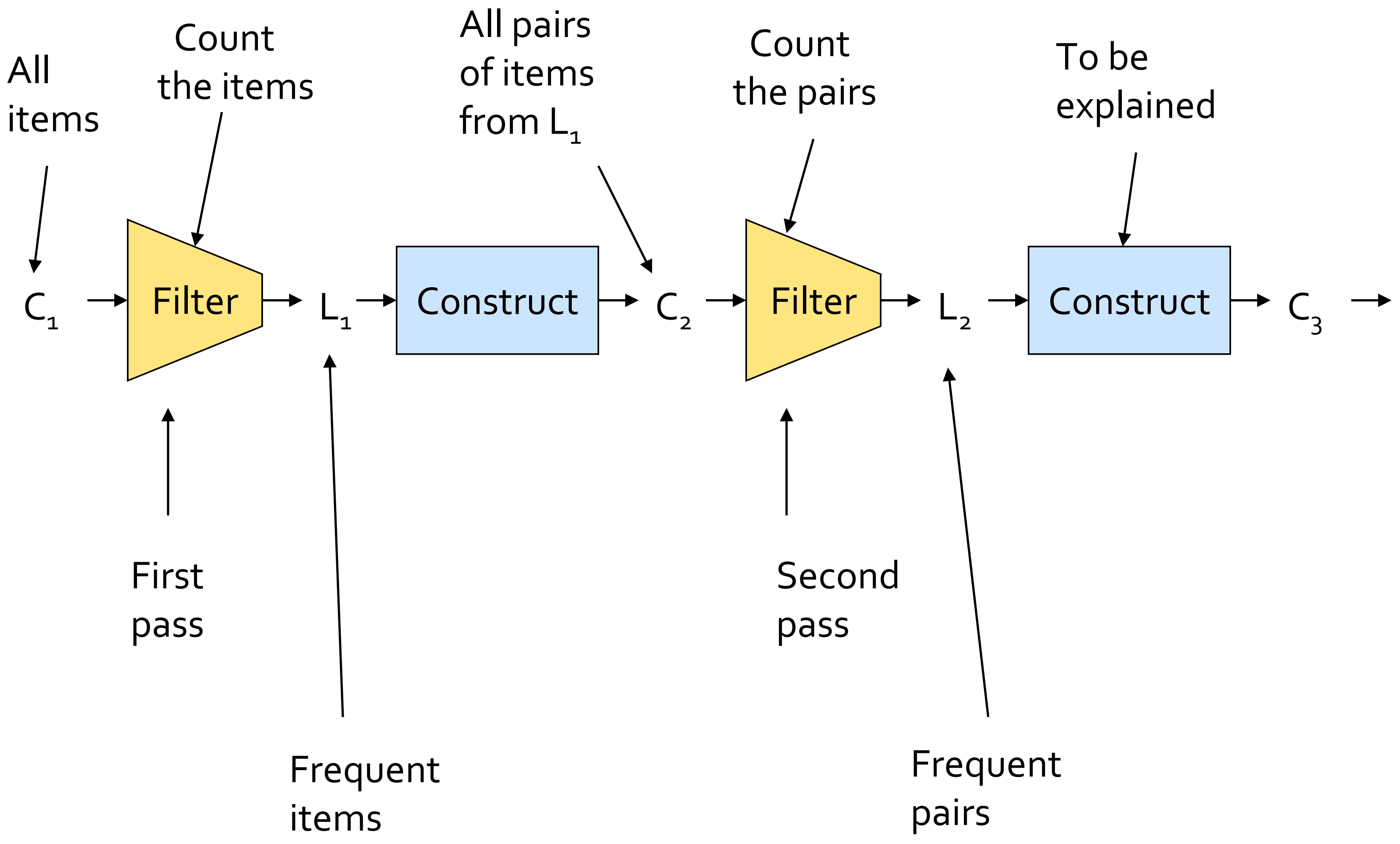
Detail for A-Priori

- 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

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

A-Priori Algorithm: Pictorial Illustration



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
- How would you generate C_{k+1} from L_k?
  - Enumerating all sets of size $k+1$ and testing each seems really dumb

A-Priori Algorithm: Example

$Sup_{min} = 2$

Database TDB

Bid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

1st scan →

C_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

→ L_1

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

C_2

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

← 2nd scan

C_2

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

← L_2

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

C_3

Itemset
{B, C, E}

3rd scan →

L_3

Itemset	sup
{B, C, E}	2

A-Priori Algorithm: Pseudocode

C_k: Candidate itemset of size k
L_k : frequent itemset of size k

L_1 = {frequent items};
for (k = 1; L_k !=$\varnothing$; k++) **do begin**
    C_{k+1} = candidates generated from L_k;
    for each basket t in database do
        increment the count of all candidates in C_{k+1} that
         are contained in t
    L_{k+1} = candidates in C_{k+1} with min_support
    end
return $\cup_k L_k$;

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 over A-Priori Algorithm

- **The PCY (Park-Chen-Yu) Algorithm**
  - Improvement to A-Priori
  - Exploits empty memory on first Pass
  - Frequent buckets

PCY Algorithm

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

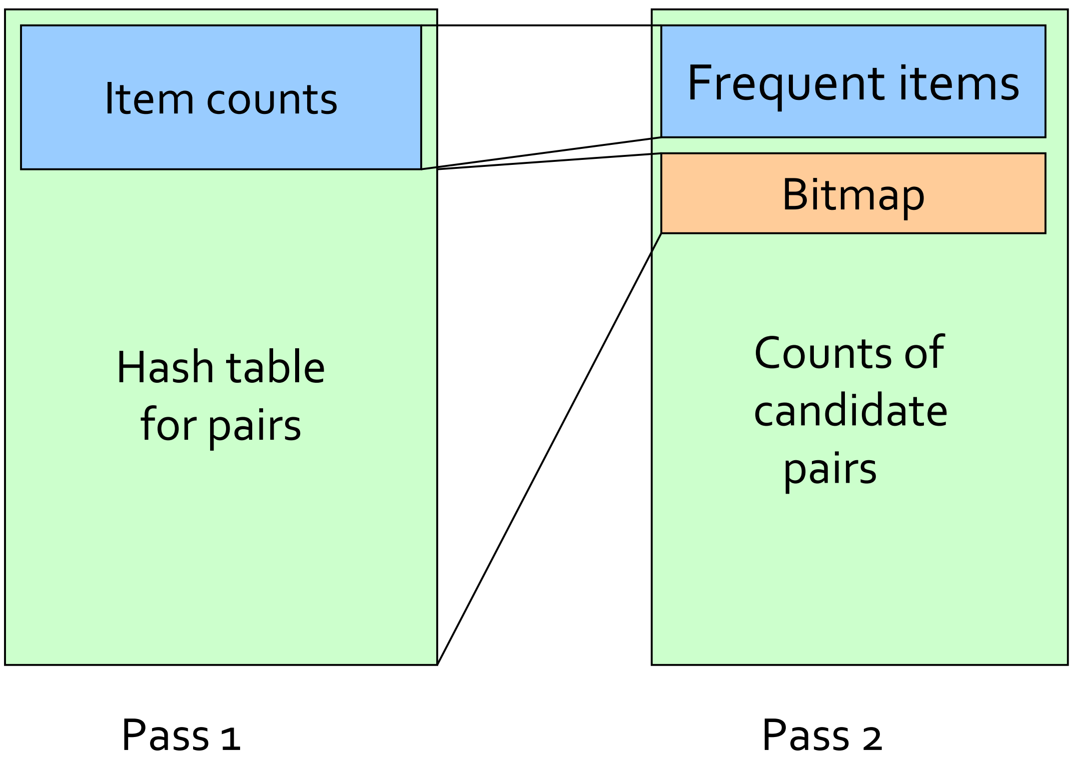
- Use that memory to keep counts of buckets into which pairs of items are hashed
  - Just the count, not the pairs themselves

- 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 of frequent items that <u>also</u> hash to a frequent bucket.
- A *bitmap* tells which buckets are frequent, using only one bit per bucket (i.e., 1/32 of the space used on Pass 1).

Picture of PCY



Pass 1 of PCY: 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 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
3. But if the count for a bucket is less than the support *s*, all pairs that hash to this bucket can be eliminated, 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

PCY Algorithm: Memory Details

- Buckets require a few bytes each
  - Note: we don't have to count past *s*
    - If $s < 2^{16}$, 2 bytes/bucket will do
  - # buckets is O(main-memory size)

- On second pass, a table of (item, item, count) triples is essential
  - Thus, hash table on Pass 1 must eliminate 2/3 of the candidate pairs for PCY to beat a-priori

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
  - Do not sneer; "random sample" is often a cure for the problem of having too large a dataset
- 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.

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 (requires more space)

SON Algorithm: Pass 1

- Partition the baskets into small subsets

- 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 is a candidate if it is frequent (with support threshold suitably scaled down) in *at least one* subset.

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

Toivonen's Algorithm

- Start as in the simple algorithm, but lower the threshold slightly for the sample
  - <u>Example: </u>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)

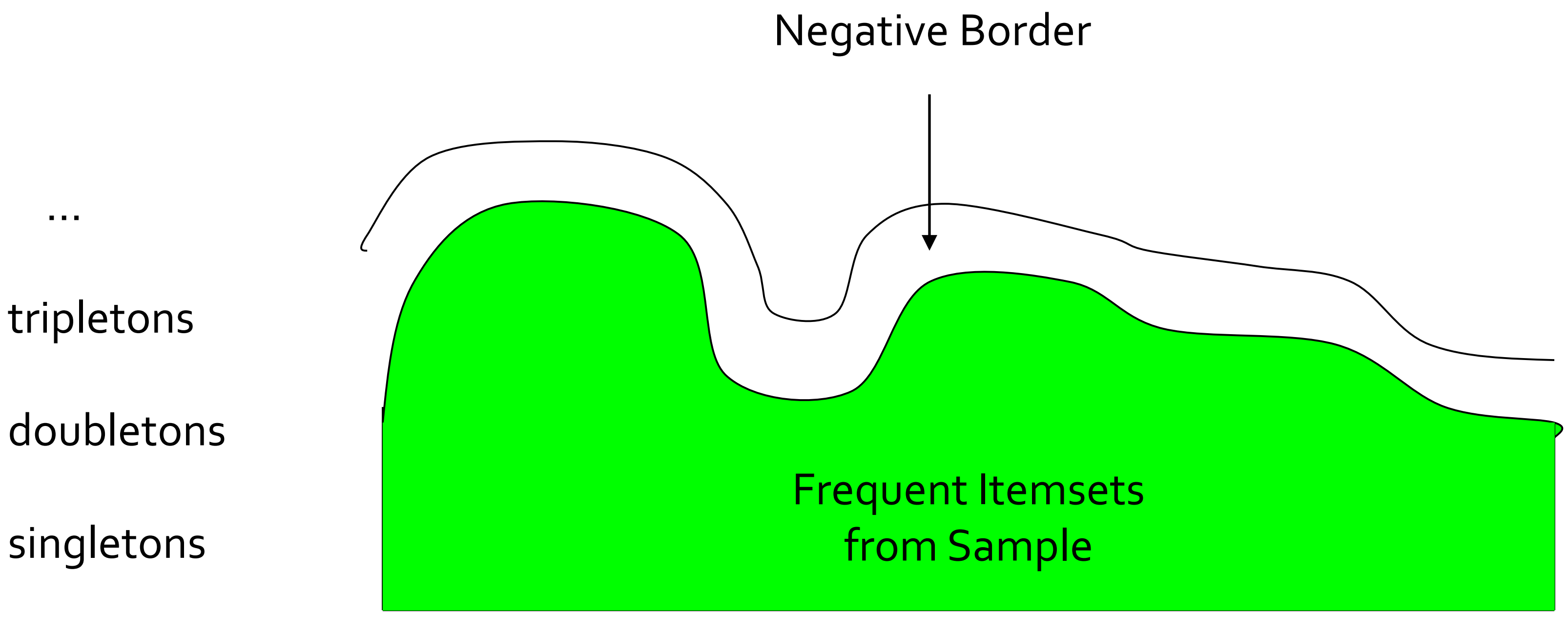
- Add to the itemsets that are frequent in the sample the *negative border* of these itemsets

- An itemset is in the negative border if it is not deemed frequent in the sample, but *all* its immediate subsets are
  - *Immediate subset* = "delete exactly one element"

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)
- Useful trick: When processing the sample by A-Priori, each member of C_k is either in L_k or in the negative border, never both

Picture of Negative Border



Toivonen's Algorithm (3)

- In a second pass, count all candidate frequent itemsets from the first pass, and also count sets in their negative border

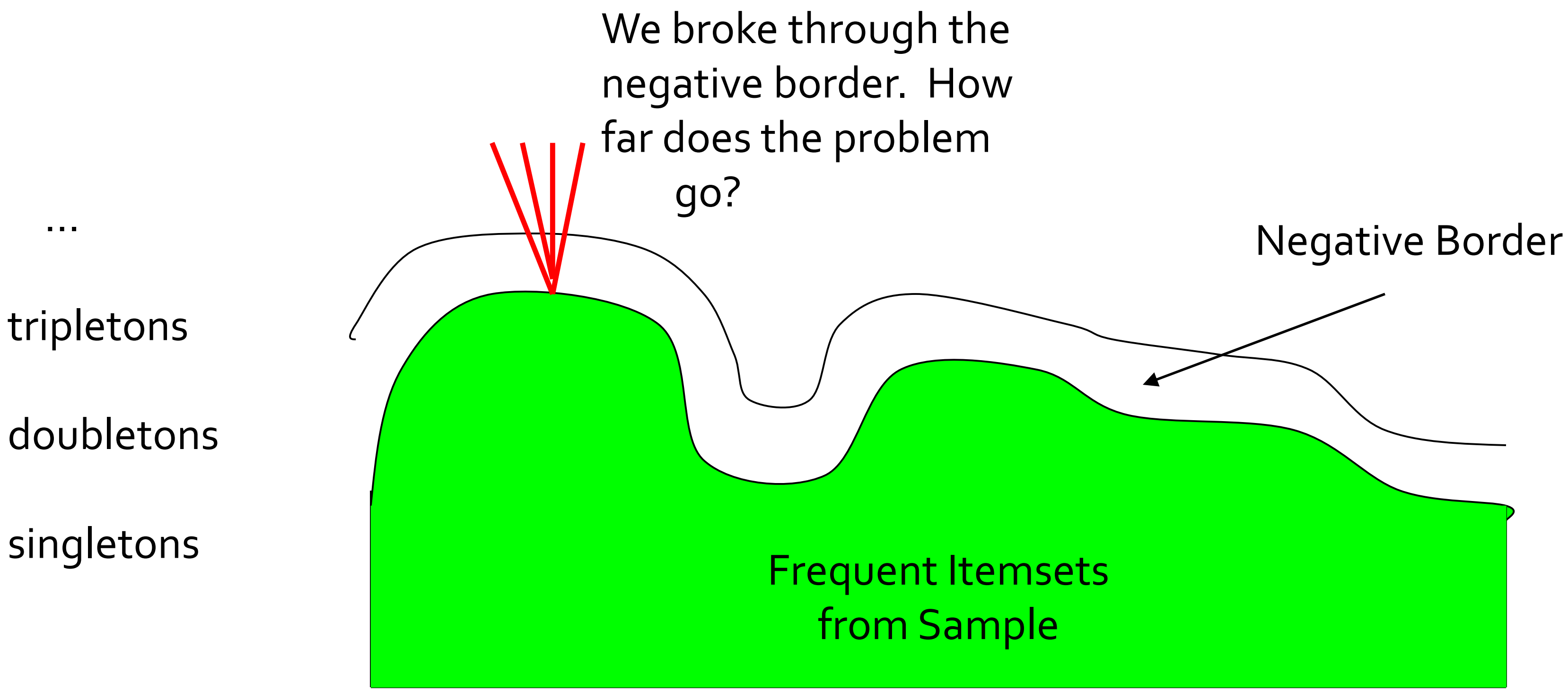
- If no itemset from the negative border turns out to be frequent, then the candidates found to be frequent in the whole data are *exactly* the frequent itemsets

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")