Frequent Itemset Mining & Association Rules

CS246: Mining Massive Datasets
Jure Leskovec, Stanford University
http://cs246.stanford.edu

Announcements

Final Exam on Gradescope; open notes
- Mar 11 (Mon) 10AM PT – Mar 12 (Tue) 10AM
- 3hr in 24hr window

We will be releasing HW1 today
- It is due in 2 weeks (1/25 at 11:59 PM)
- **Please start early.** The homework is long

  - Requires proving theorems as well as coding

We will also be releasing Colab 0 and Colab 1
- They are due in 1 week (1/18 at 11:59 PM)

Send OAE letters to course email by 4/12
- cs246-win2324-staff@lists.stanford.edu

All HW/Colab links will be posted on Ed

Announcements

Spark OH
- Saturday, 01/13, 9:30 AM - 10:30 AM PT
- Zoom: https://stanford.zoom.us/j/93824616839?pwd=Ky9MQjU5cmZjMDg1ZkEzbzF0SGlZUT09

Linear Algebra Recitation
- Thursday, 01/18, 5:30 PM - 7:30 PM PT
- Room: 200-305
- Zoom: https://stanford.zoom.us/j/98848225994?pwd=WnZQc1BiVXFQdWxVRUZGdVZ5dDF6Zz09

Proofs and Probability Recitation
- Friday, 01/19, from 2:30 PM - 4:30 PM PT
- Room: Hewlett 201
- Zoom: TBD (we will try to livestream the session)

Association Rule Discovery

Supermarket shelf management – Market-basket model:

- **Goal:** Identify items that are bought together by sufficiently many customers
- **Approach:** Process the sales data collected with barcode scanners to find dependencies among items
- **A classic rule:**
  - If someone buys diaper and milk, then he/she is likely to buy beer
  - Don't be surprised if you find six-packs next to diapers!

The Market-Basket Model

- A **large set** of **items**
  - e.g., things sold in a supermarket
- A **large set** of **baskets**
  - Each basket is a **small subset of items**
    - e.g., the things one customer buys on one day
- **Discover** **association rules:**

People who bought {x,y,z} tend to buy {v,w}
  - Example application: Amazon

Input:

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

Output:

Rules Discovered:
{Milk} --> {Coke}
{Diaper, Milk} --> {Beer}

More generally

- **A general many-to-many mapping (association) between two kinds of things**
  - But we are interested in connections among "items", not "baskets"
- **Items and baskets are abstract:**
  - **For example:**
    - Items/baskets can be products/shopping basket
    - Items/baskets can be words/documents
    - Items/baskets can be base-pairs/genes
    - Items/baskets can be drugs/patients

Applications – (1)

- **Items** = products; **Baskets** = sets of products someone bought in one trip to the store
- **Real market baskets:** Chain stores keep TBs of data about what customers buy together
  - Tells how typical customers navigate stores, lets them position tempting items together:
    - Apocryphal story of "diapers and beer" discovery
    - Used to position potato chips between diapers and beer to enhance sales of potato chips
- **Amazon's 'people who bought *X* also bought *Y*'**

Applications – (2)

- **Baskets** = sentences; **Items** = documents in which those sentences appear
  - Items that appear together too often could represent plagiarism
  - Notice items do not have to be "in" baskets

- **Baskets** = patients; **Items** = drugs & side-effects
  - Has been used to 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

Outline

First: Define

Frequent itemsets

Association rules:

Confidence, Support, Interestingness

Then: Algorithms for finding frequent itemsets

Finding frequent pairs

A-Priori algorithm

PCY algorithm

Frequent Itemsets

- **Simplest question:** Find sets of items that appear together "frequently" in baskets
- *Support* for itemset I: Number of baskets containing all items in I
    - (Often expressed as a fraction of the total number of baskets)
- Given a *support threshold s*, then sets of items that appear in at least s baskets are called *frequent itemsets*

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

Support of
{Beer, Bread} = 2

Example: Frequent Itemsets

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

  B₁ = {m, c, b}   **B₂** = {m, p, j}

  B₃ = {m, b}   **B₄** = {c, j}

  B₅ = {m, p, b}   **B₆** = {m, c, b, j}

  B₇ = {c, b, j}   **B₈** = {b, c}

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

Define: Association Rules

- **Define: 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"
- **In practice there are many rules, want to find significant/interesting ones!**
- *Confidence* of association rule is the probability of j given $I = \{i_1,\ldots,i_k\}$

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

$\text{conf}(I \rightarrow j) =$
$= P(j|I) = \frac{P(I,j)}{P(I)}$

Interesting Association Rules

- **Not all high-confidence rules are interesting**

  - The rule $X \rightarrow \boldsymbol{milk}$ may have high confidence for many itemsets X, because milk is just purchased very often (independent of X)

- **Interest** of an association rule $I \rightarrow j$:
  abs. difference between its confidence and
  the fraction of baskets that contain j

$$Interest(I \rightarrow j) = |conf(I \rightarrow j) - P[j]| = |P(j|I) - P(j)|$$

  - Interesting rules: those with high interest values (usually above 0.5)

  - Why absolute value? Want to capture both *positive* and *negative* associations between itemsets and items

Example: Confidence and Interest

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

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

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

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

- **Association rule: $\{m, b\} \rightarrow c$**

  - **Support** = 2

  - **Confidence** = 2/4 = 0.5

  - **Interest** = |0.5 − 5/8| = 1/8

    - Item **c** appears in 5/8 of the baskets

    - The rule is not very interesting!

Association Rule Mining

- **Problem: Find all association rules with support ≥ s and confidence ≥ c**

  - **Note:** Support of an association rule is the support of the entire set of items in the rule (left side + right side)

- **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"

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

Mining Association Rules

$$\text{conf}(I \to j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}$$

- **Step 1:** Find all frequent itemsets I
  - (we will explain this next)
- **Step 2: Rule generation**
  - For every subset A of I, generate a rule $A \to I \setminus A$
    - Since I is frequent, A is also frequent
    - **Variant 1:** Single pass to compute the rule confidence
      - confidence(**A,B→C,D**) = support(**A,B,C,D**) / support(**A,B**)
    - **Variant 2:**
      - **Observation:** If **A,B,C→D** is below confidence, then so is **A,B→C,D**
      - Can generate "bigger" rules from smaller ones!
  - **Output the rules above the confidence threshold**

Example

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

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

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

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

- **Support threshold $s = 3$, confidence $c = 0.75$**
- **Step 1) Find frequent itemsets:**
  - **{b,m} {b,c} {c,m} {c,j} {m,c,b}**
- **Step 2) Generate rules:**
  - ~~**b** ⟶ **m**: c=4/6~~   **b** ⟶ **c**: c=5/6   ~~**b,c** ⟶ **m**: c=3/5~~
  - **m** ⟶ **b**: c=4/5   …   **b,m** ⟶ **c**: c=3/4
  - ~~**b** ⟶ **c,m**: c=3/6~~

Compacting the Output

- **To reduce the number of rules, we can post-process them and only output:**

  - **Maximal frequent itemsets:**
    No immediate superset is frequent
    - Gives more pruning

  or

  - **Closed itemsets:**
    No immediate superset has the same support (> 0)
   - Stores not only frequent information, but exact supports/counts

Example: Maximal/Closed

	Support	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 support.

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

Step 2:
Finding Frequent Itemsets

Itemsets: Computation Model

- **Back to finding frequent itemsets**
- Typically, data is kept in flat files rather than in a database system:
  - Stored on disk
  - Stored basket-by-basket
  - Baskets are **small** but we have many baskets and many items
    - Expand baskets into pairs, triples, etc. as you read baskets
    - Use k nested loops to generate all sets of size k

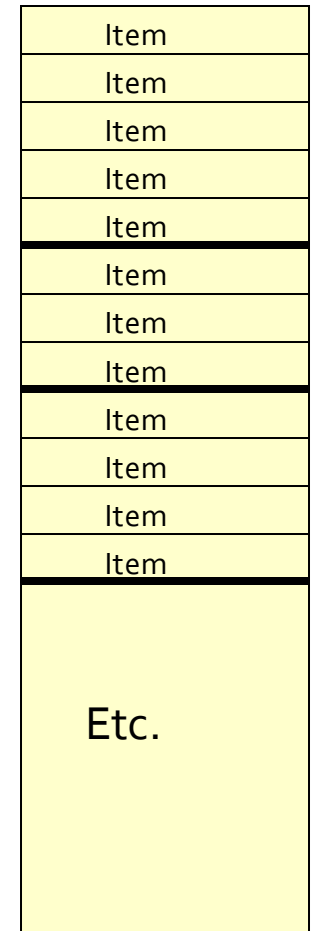| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Etc. |
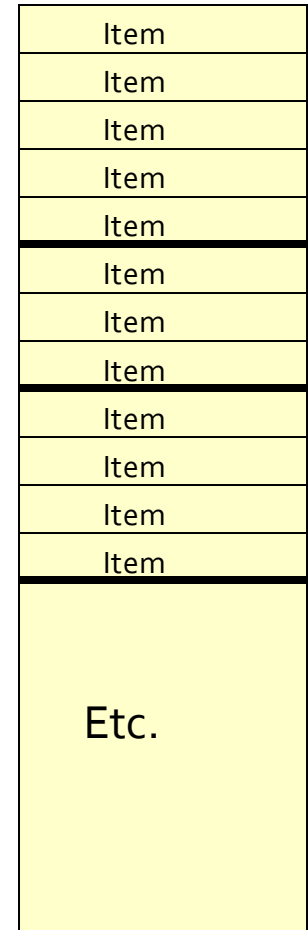
Items are positive integers, and boundaries between baskets are −1.

Note: We want to find frequent itemsets. To find them, we have to count them. To count them, we have to enumerate them.

Computation Model

- The true cost of mining disk-resident data is usually the **number of disk I/Os**

- 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 makes over the data

| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Item |
| Etc. |

Items are positive integers, and boundaries between baskets are –1.

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 of main-memory is a bad idea

    - Q: Why?

Finding Frequent Pairs

- **The hardest problem often turns out to be finding the frequent pairs of items $\{i_1, i_2\}$**
  - **Why?** Freq. pairs are common, freq. triples are rare
    - **Why?** Probability of being frequent drops exponentially with size; number of sets grows more slowly with size

- **Let's first concentrate on pairs, then extend to larger sets**

Finding Frequent Pairs

- **The approach:**
  - We always need to "generate" all the itemsets
  - But we would only like to count (keep track of) only those itemsets that in the end turn out to be frequent
- **Scenario:**
  - Imagine we aim to identify frequent pairs
  - We will need to enumerate all pairs of items
    - For every basket, enumerate all pairs of items in that basket
  - **But,** rather than keeping a count for every pair, we hope to discard a lot of pairs and only keep track of the ones that will in the end turn out to be frequent

Naïve Algorithm

- **Naïve approach to finding frequent pairs**
- Read file once, counting in main memory the occurrences of each pair:

  - From each basket **b** of n_b items, generate its $n_b(n_b-1)/2$ pairs by two nested loops
  - A data structure then keeps count of every pair

- **Fails if (#items)^2 exceeds main memory**

  - **Remember:** #items can be 100K (Wal-Mart) or 10B (Web pages)
    - Suppose 10^5 items, counts are 4-byte integers
    - Number of pairs of items: $10^5(10^5-1)/2 \approx 5*10^9$
    - Therefore, $2*10^{10}$ (20 gigabytes) of memory is needed

# Counting Pairs in Memory

Goal: Count the number of occurrences of each pair of items *(i,j)*:

- **Approach 1:** Count all pairs using a matrix

- **Approach 2:** Keep a table of triples [*i, j, c*] = "the count of the pair of items {*i, j*} is *c*."
  - If integers and item ids are 4 bytes, we need approximately 12 bytes for pairs with count > 0
  - Plus some additional overhead for the hashtable
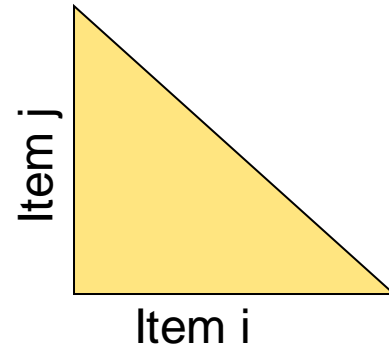
Comparing the Two Approaches

Item j

4 bytes per pair

Item i

Triangular Matrix

12 bytes per occurring pair
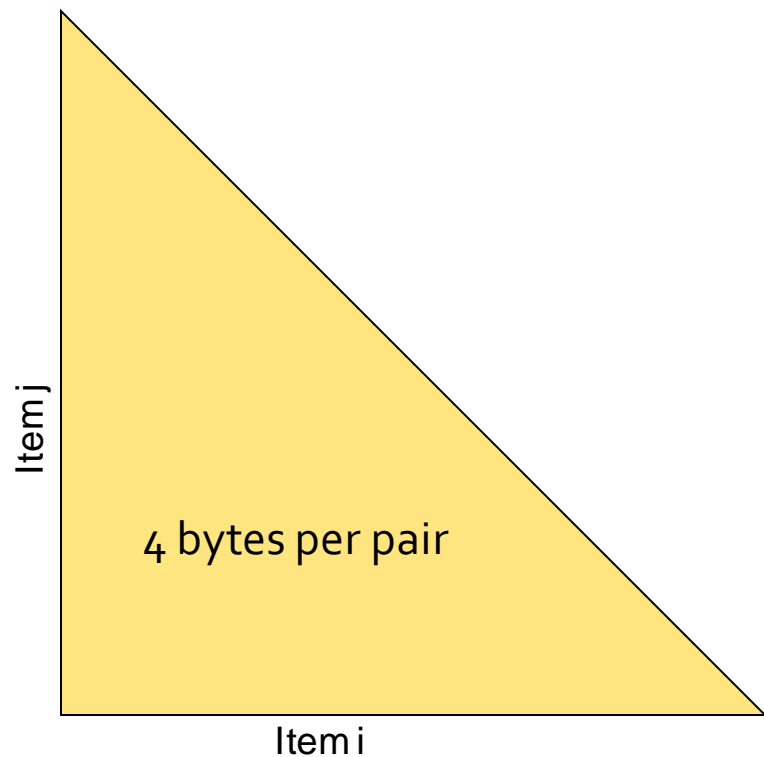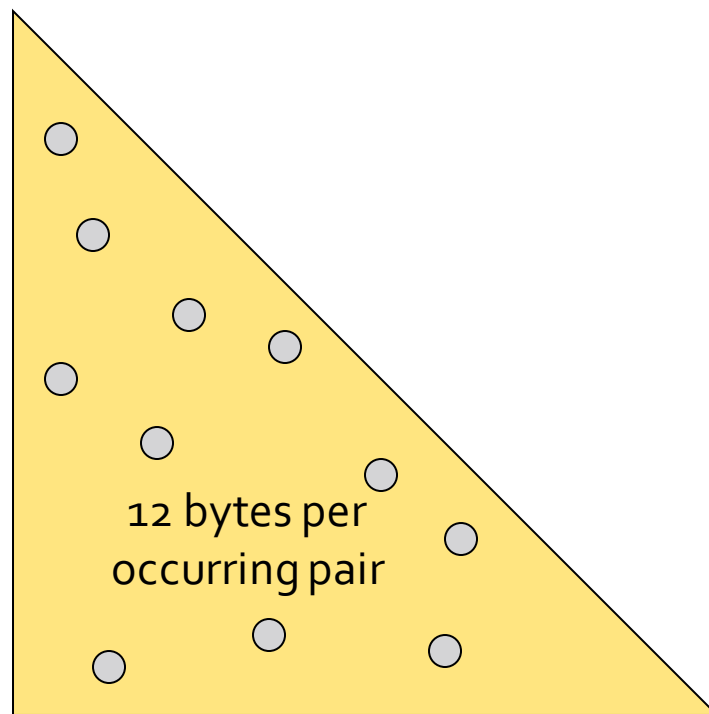
Triples (item i, item j, count)

Comparing the Two Approaches

- **Approach 1: Triangular Matrix**
    - **n** = total number items
    - Count pair of items $\{i, j\}$ 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: **[n(n - 1) - (n - i)(n - i + 1)]/2 + (j - i)**
    - Total number of pairs $n(n-1)/2$; total bytes= O(n^2)
    - **Triangular Matrix** requires 4 bytes per pair

- **Approach 2** uses **12 *bytes*** per occurring pair *(but only for pairs with count > 0)*

- Approach 2 beats Approach 1 if less than **1/3** of possible pairs actually occur

- **Approach 1: Triangular Matrix**
  - **n** = total number items
  - C...
  - K...

  - P... 2 + (j - i)
  - T...
  - T...

Problem is when we have too many items so all the pairs do not fit into memory.

Can we do better?

- **Ap**...
  (bu...

- Approach 2 beats Approach 1 if less than **1/3** of possible pairs actually occur
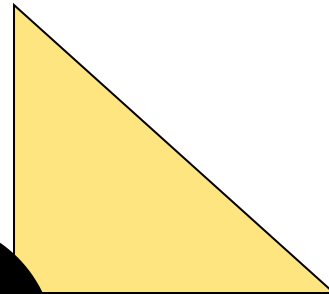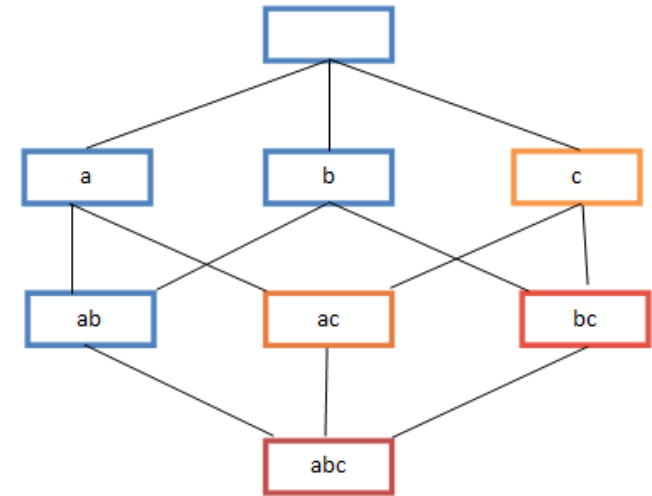
A-Priori Algorithm

Key concepts:

- **Monotonicity of "Frequent"**
- **Notion of Candidate Pairs**
- **Extension to Larger Itemsets**

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 I appears at least s times, so does every **subset J of I**

- **Contrapositive for pairs:**

  If item i does not appear in s baskets, then no pair including i can appear in s baskets

- **So, how does A-Priori find freq. pairs?**

A-Priori Algorithm – (2)

- **Pass 1:** Read baskets and count in main memory the # of occurrences of each **individual item**
  - Requires only memory proportional to #items

- **Items that appear $\geq s$ times are the <u>frequent items</u>**

- **Pass 2:** Read baskets again and keep track of the count of <u>only</u> those pairs where both elements are frequent (from Pass 1)
  - Requires memory (for counts) proportional to square of the number of **frequent** items (not the square of total # of items)
 - Plus a list of the frequent items (so you know what must be counted)

Main-Memory: Picture of A-Priori

Pass 1

Pass 2

Main memory

Item counts

Frequent items

Counts of pairs of frequent items (candidate pairs)
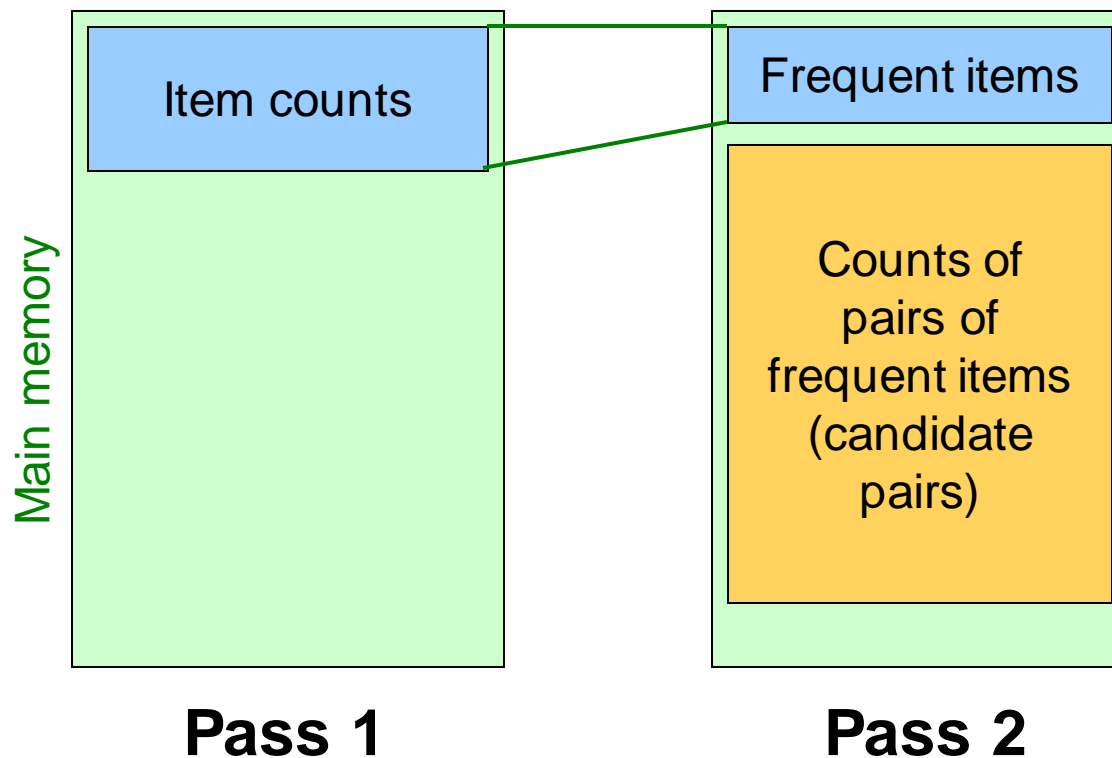
Green box represents the amount of available main memory. Smaller boxes represent how the memory is used.
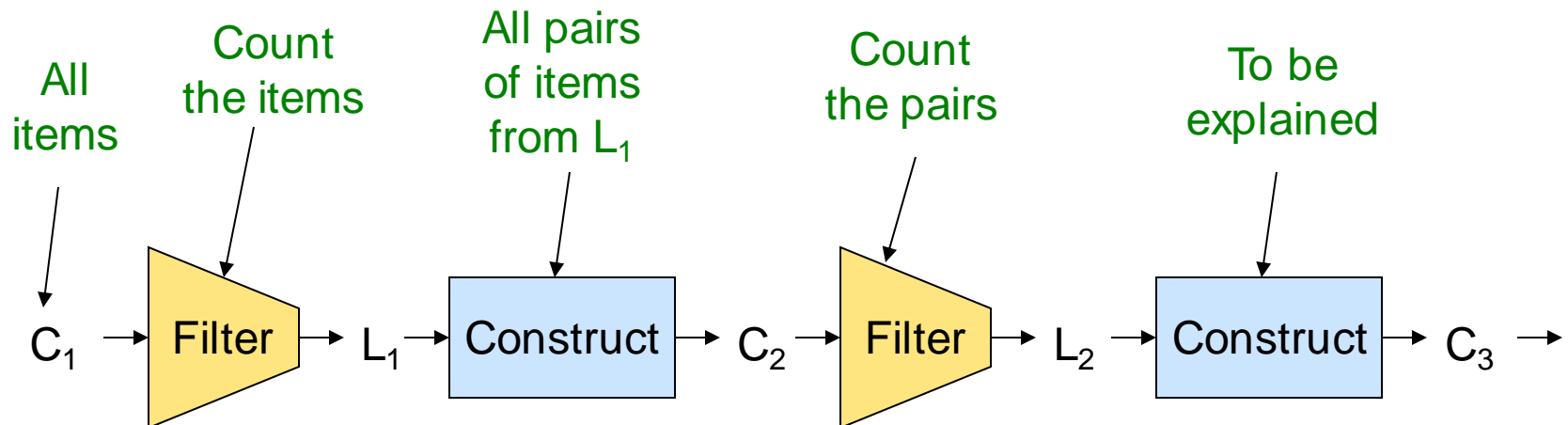
Detail for A-Priori

- 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

Item counts	Frequent items / Old item IDs
Main memory	Counts of pairs of frequent items
Pass 1	**Pass 2**
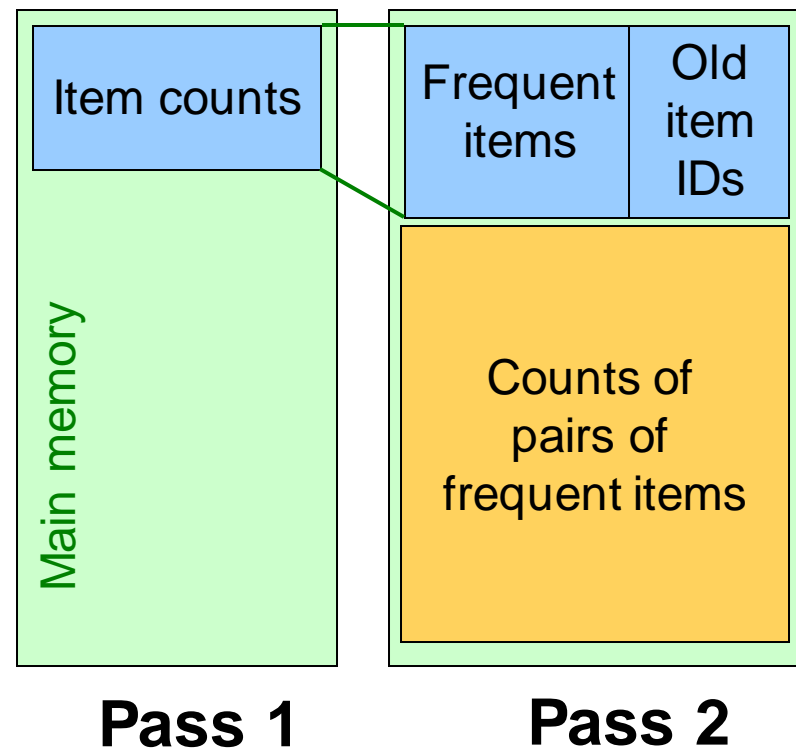
Frequent Triples, Etc.

- **For each k, we construct two sets of k-tuples (sets of size k):**

  - **C_k = candidate k-tuples = 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-tuples**

All items   Count the items   All pairs of items from L_1 Count the pairs   To be explained

$C_1 \rightarrow$ Filter $\rightarrow L_1 \rightarrow$ Construct $\rightarrow C_2 \rightarrow$ Filter $\rightarrow L_2 \rightarrow$ Construct $\rightarrow C_3 \rightarrow$

Example

- **Hypothetical steps of the A-Priori algorithm**
  - C_1 = { {b} {c} {j} {m} {n} {p} }
  - Count the support of itemsets in C_1
  - Prune non-frequent. We get: L_1 = { b, c, j, m }
  - Generate C_2 = { {b,c} {b,j} {b,m} {c,j} {c,m} {j,m} }
  - Count the support of itemsets in C_2
  - Prune non-frequent. L_2 = { {b,m} {b,c} {c,m} {c,j} }
  - Generate C_3 = { {b,c,m} {b,c,j} {b,m,j} {c,m,j} } **
  - Count the support of itemsets in C_3
  - Prune non-frequent. L_3 = { {b,c,m} }

A-Priori for All Frequent Itemsets

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

- **Many possible extensions:**
  - Association rules with intervals:
    - For example: Men over 65 have 2 cars
  - Association rules when items are in a taxonomy
    - Bread, Butter $\rightarrow$ FruitJam
    - BakedGoods, MilkProduct $\rightarrow$ PreservedGoods
  - Lower the support s as itemset gets bigger

PCY (Park-Chen-Yu) Algorithm

Key concepts:

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

PCY (Park-Chen-Yu) 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/elements** as fit in memory

  Note:
  Bucket≠Basket

  - Keep a **count** for each bucket into which **pairs** of items are **hashed**

    - **For each bucket just keep the count, not the actual pairs that hash to the bucket!**

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

New in PCY

- **Few things to note:**

  - Pairs of items need to be generated from the input file; they are not present in the file

  - We are not just interested in the presence of a pair, but we need to see whether it is present at least *s* (support) times

Observations about Buckets

- **Observation: If a bucket contains a frequent pair, then the bucket is surely frequent**
- However, even without any frequent pair, a bucket can still be frequent ☹
    - So, we cannot use the hash to eliminate any member (pair) of a "frequent" bucket
- **But, for a bucket with total count less than s, none of its pairs can be frequent ☺**
    - Pairs that hash to this bucket can be eliminated as candidates (even if the pair consists of 2 frequent items)

- **Pass 2:**
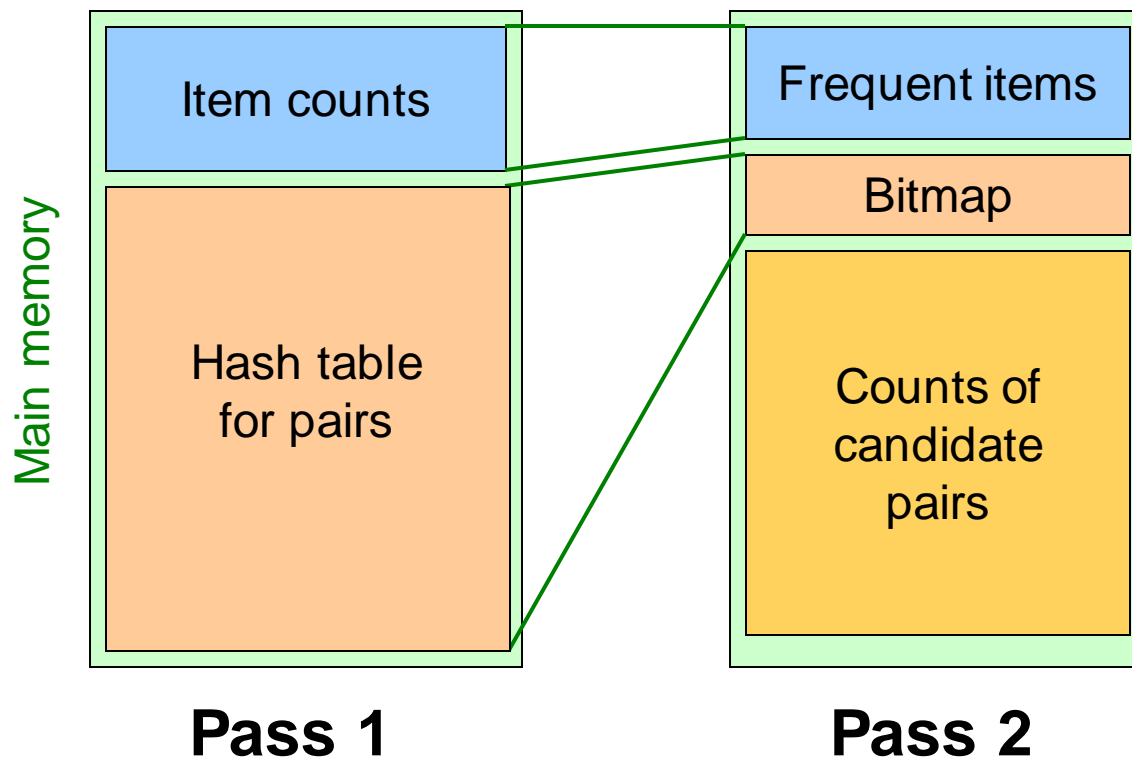 Only count pairs that hash to frequent buckets

PCY Algorithm – Between Passes

- **Replace the buckets by a bit-vector:**
    - **1** means the bucket count exceeded the support s (call it a **frequent bucket**); **0** means it did not

- **4-byte integer counts 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 whose bit in the bit vector is **1** (i.e., a **frequent bucket**)

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

Main-Memory: Picture of PCY

Main memory

Pass 1
- Item counts
- Hash table for pairs

Pass 2
- Frequent items
- Bitmap
- Counts of candidate pairs

Main-Memory Details

- **Buckets require a few bytes each:**
    - **Note:** we do not have to count past s
    - #buckets is *O(main-memory size)*

- On second pass, a table of (item, item, count) triples is essential (we cannot use triangular matrix approach)
    - Thus, hash table must eliminate approx. 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**"

- For reading on your own, Sect. 6.4 of MMDS

- **Recommended video** (starting about 10:10): https://www.youtube.com/watch?v=AGAkNiQnbjY

Frequent Itemsets in ≤ 2 Passes

Key concepts:

- **Random Sampling Algorithm**
- **Savasere-Omiecinski-Navathe (SON) Algorithm**
- **Toivonen's Algorithm**

Frequent Itemsets in ≤ 2 Passes

- A-Priori, PCY, etc., take *k* passes to find frequent itemsets of size *k*
- **Can we use fewer passes?**

- **Use 2 or fewer passes for all sizes, but may miss some frequent itemsets**
- **3 different approaches:**
  - Random sampling:
    - Do not sneer; "random sample" is often a cure for the problem of having too large a dataset.
  - SON (Savasere, Omiecinski, and Navathe)
  - Toivonen

Random Sampling (1)

- **Take a random sample of the market baskets**

- **Run a-priori or one of its improvements in 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 the sample size

    - Example: if your sample is 1/100 of the baskets, use s/100 as your support threshold instead of s.

Main memory

Copy of sample baskets
Space for counts

Random Sampling (2)

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

- **But you don't catch sets that are frequent in the whole data but not in the sample**
  - Smaller threshold, e.g., $s/125$, helps catch more truly frequent itemsets
    - But requires more space

- **SON algorithm tries to deal with this (next)**

SON Algorithm – 2-pass algorithm

- **SON Algorithm:** 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 one or more subsets of the baskets.

- 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 dataset unless it is frequent in at least one subset.

  THE PIGEONHOLE PRINCIPLE

  - Pigeonhole principle
- However, even with SON algorithm we still don't know whether we found **all** frequent itemsets
  - An itemset may be infrequent in all subsets but frequent overall
- **Toivonen's algorithm solves this (next)**
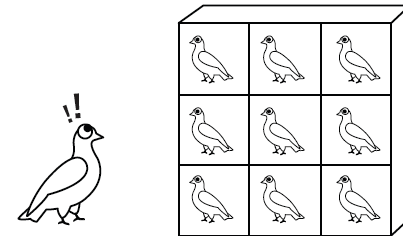
Pass 1:

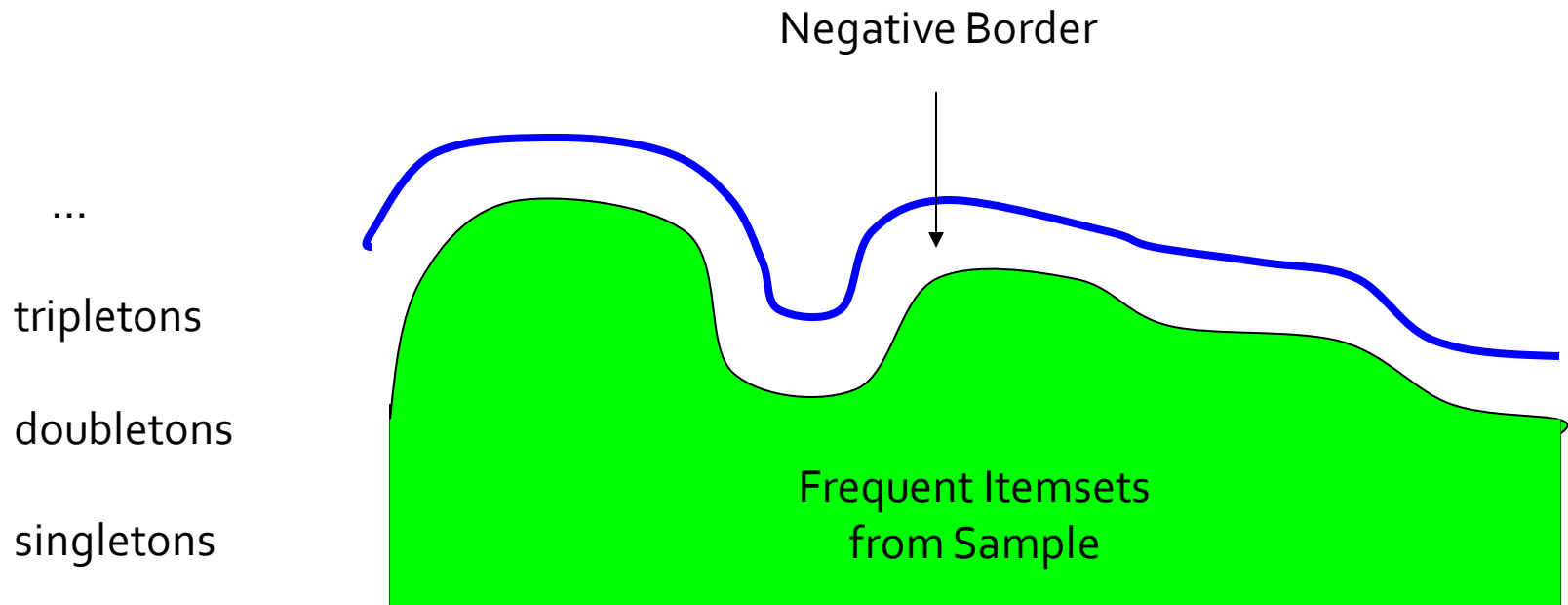- **Start with a random sample, but lower the threshold slightly for the sample:**

  - **Example**: If the sample is 1% of the baskets, use $s/125$ as the support threshold rather than $s/100$

- Find frequent itemsets in the sample

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

  - **Negative border:** An itemset is in the negative border if it is not frequent in the sample, but *all* its immediate subsets are

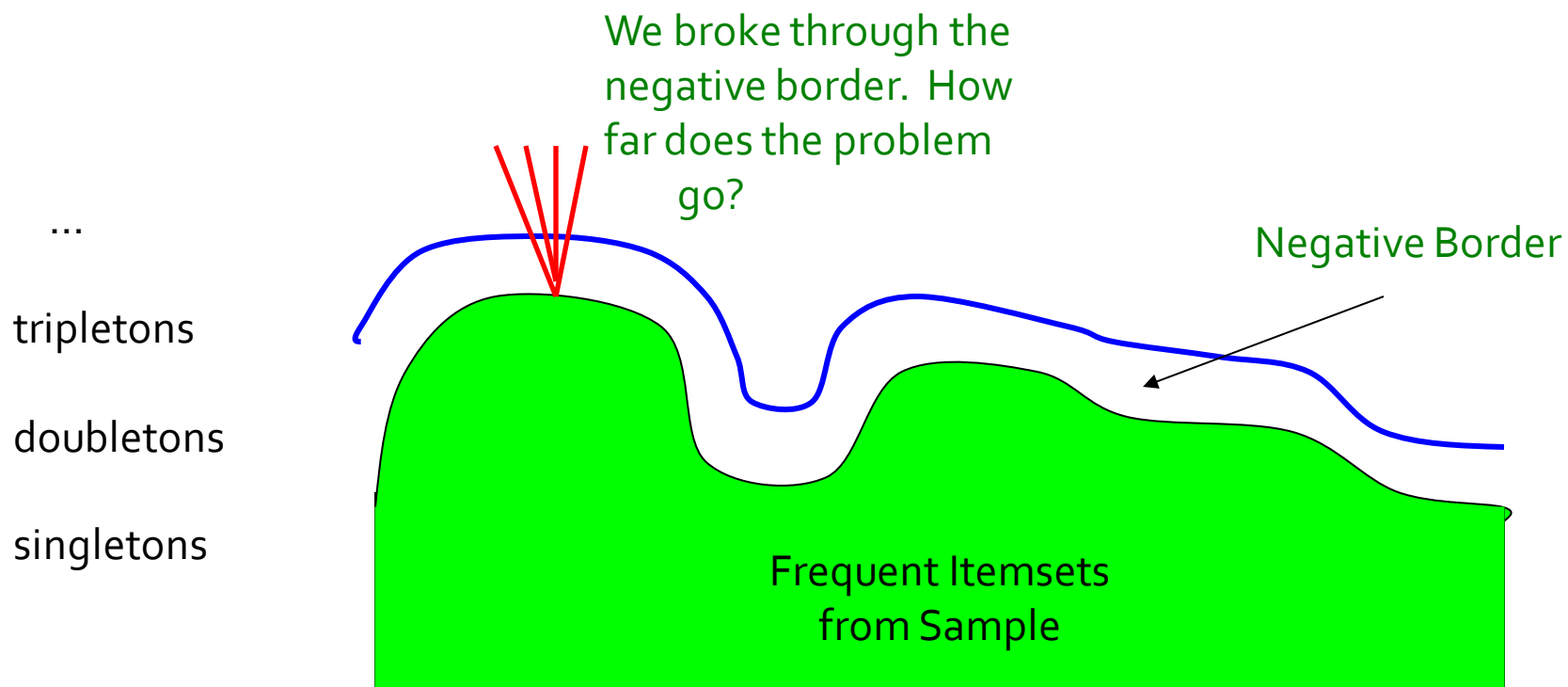   - Immediate subset = "delete exactly one element"

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.

Negative Border

...

tripletons

doubletons

singletons

Frequent Itemsets
from Sample

Toivonen's Algorithm

- **Pass 1:**
  - Start with the random sample, but lower the threshold slightly for the subset
  - To the itemsets that are frequent in the sample, add the **negative border** of these itemsets
- **Pass 2:**
  - 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 we found *all* the frequent itemsets.
  - What if we find that something in the negative border is 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.

We broke through the negative border. How far does the problem go?

Negative Border

...

tripletons

doubletons

singletons

Frequent Itemsets from Sample

Theorem:

- If there is an itemset *S* that is frequent in full data, but not frequent in the sample, then the negative border contains at least one itemset that is frequent in the full data.

Proof by contradiction:

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