Frequent Itemset Mining & Association Rules

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}
    - 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 ask about 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 basepairs/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:
    - 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

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}.

Association Rules

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

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

Interesting Association Rules

- **Not all high-confidence rules are interesting**
  - The rule $X \rightarrow \textit{milk}$ may have high confidence for many itemsets X, because milk is just purchased very often (independent of X) and the confidence will be high
- **Interest** of an association rule $I \rightarrow j$: difference between its confidence and the fraction of baskets that contain j

$$\text{Interest}(I \rightarrow j) = |\,\text{conf}(I \rightarrow j) - \Pr[j]\,|$$

  - Interesting rules are those with high positive or negative interest values (usually above 0.5)

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)| = 0.8
    - Item ***c*** appears in 5/8 of the baskets
    - Rule is not very interesting!

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 in the rule (left and 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

- **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, 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 $c = 0.75$**
- **1) Frequent itemsets:**
  - {b,m} {b,c} {c,m} {c,j} {m,c,b}
- **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.

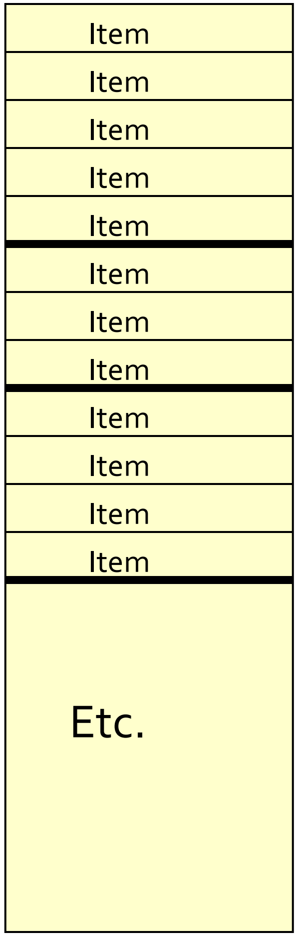
Superset BC has same support.

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

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***

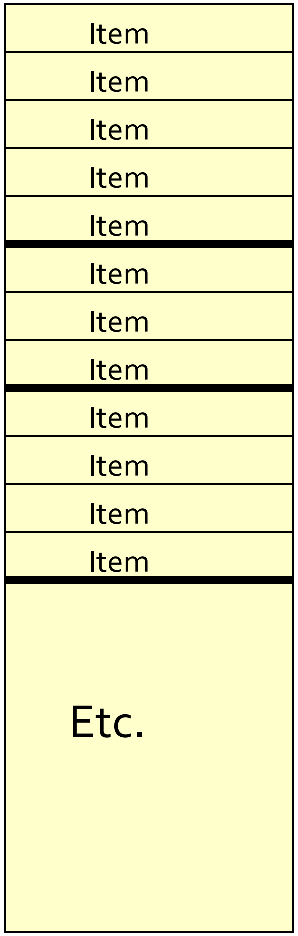


Items are positive integers, and boundaries between baskets are –1.

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



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 is a disaster

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**
- **The approach:**
  - We always need to generate all the itemsets
  - But we would only like to count (keep track) of those itemsets that in the end turn out to be frequent

Naïve Algorithm

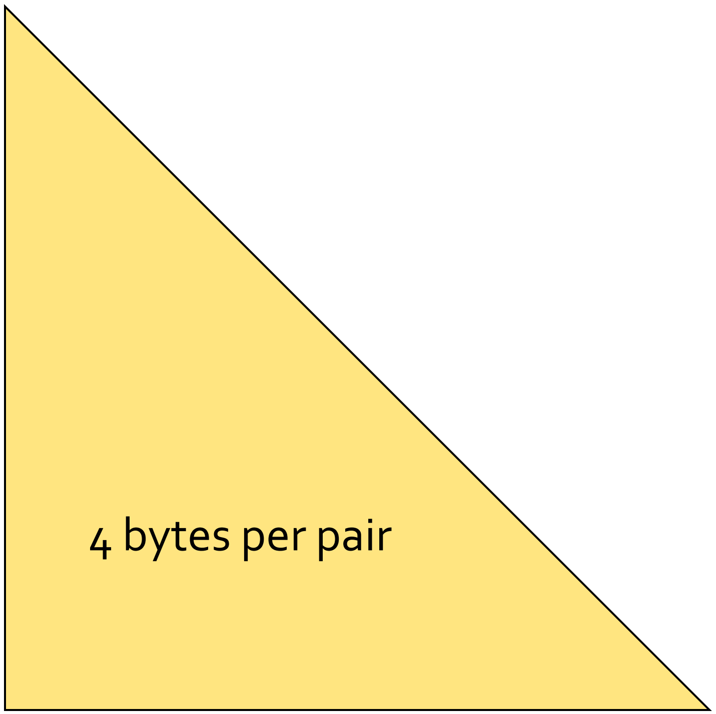
- **Naïve approach to finding frequent pairs**
- Read file once, counting in main memory the occurrences of each pair:
  - From each basket of ***n*** items, generate its ***n(n-1)/2*** pairs by two nested loops
- **Fails if (#items)^2 exceeds main memory**
  - **Remember:** #items can be 100K (Wal-Mart) or 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 needed

Counting Pairs in Memory

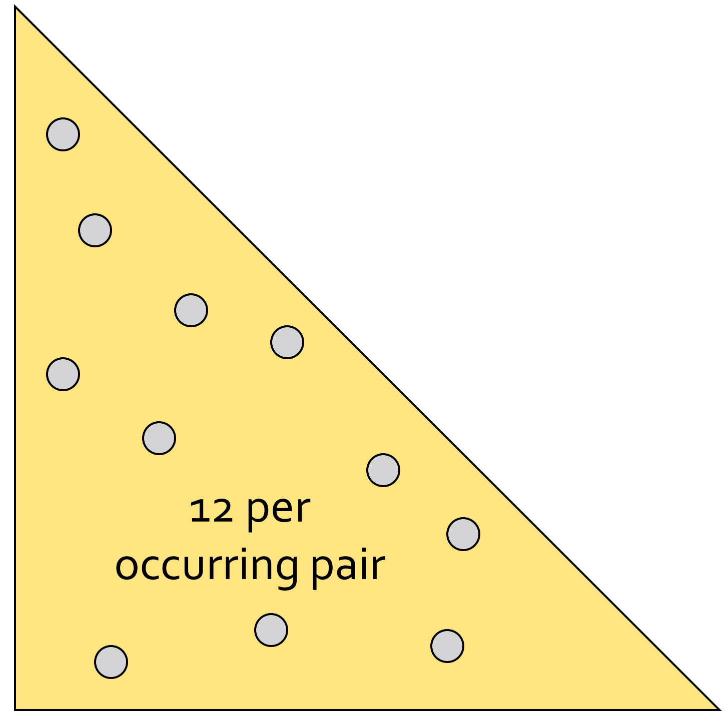
Two approaches:

- **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 2 Approaches

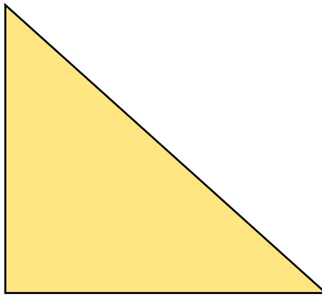


Triangular Matrix



Triples

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: $\boldsymbol{(i-1)(j-i)+j-1}$
  - Total number of pairs $\boldsymbol{n(n-1)/2}$; total bytes= $O(\boldsymbol{n^2})$
  - **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

Comparing the two approaches

- **Approach 1: Triangular Matrix**
  - **n** = total number items
  - Co
  - K
  - F
  - T … $2n^2$
  - T

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

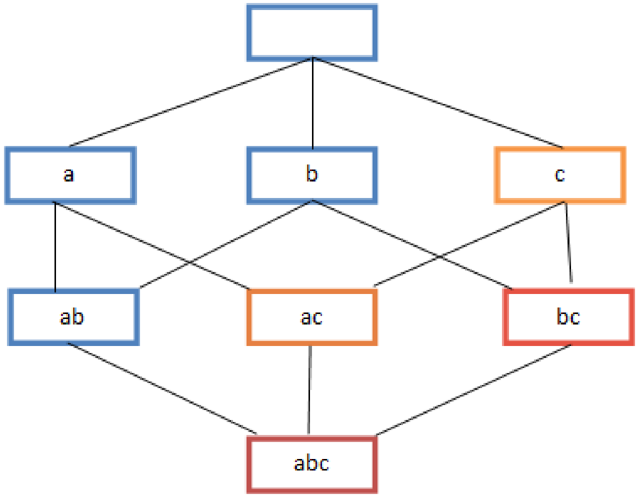
Can we do better?

- **Ap** … pair *(bu*
- Approach 2 beats Approach 1 if less than **1/3** of possible pairs actually occur

A-Priori Algorithm

- Monotonicity of “Frequent”
- Notion of Candidate Pairs
- Extension to Larger Itemsets

A-Priori Algorithm – (1)



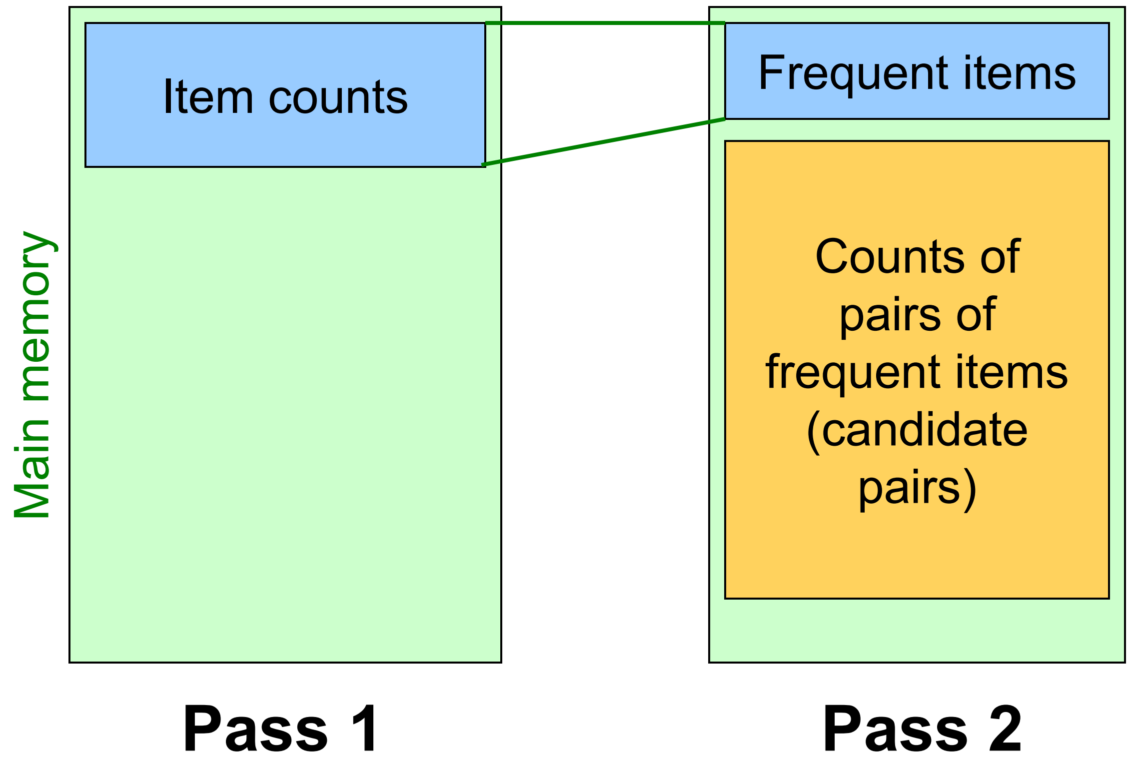
- A **two-pass** approach called ***A-Priori*** limits the need for main memory
- **Key idea: *monotonicity***
  - If a set of items I appears at least s times, so does every **subset** J of I
- **Contrapositive for pairs:**
  If item i does not appear in s baskets, then no pair including i can appear 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 occurrences of each **individual item**
  - Requires only memory proportional to #items
- **Items that appear $\geq s$ times are the frequent items**
- **Pass 2:** Read baskets again and count in main memory only those pairs where both elements are frequent (from Pass 1)
  - 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)

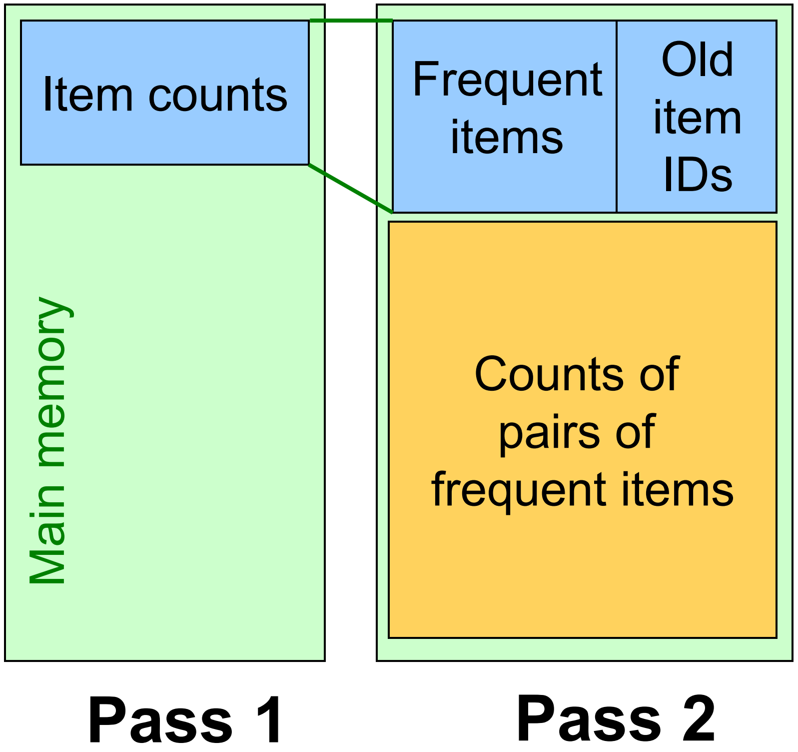
Main-Memory: Picture of A-Priori



Green box represents the amount of available main memory. Smaller boxes represent how the memory is used.

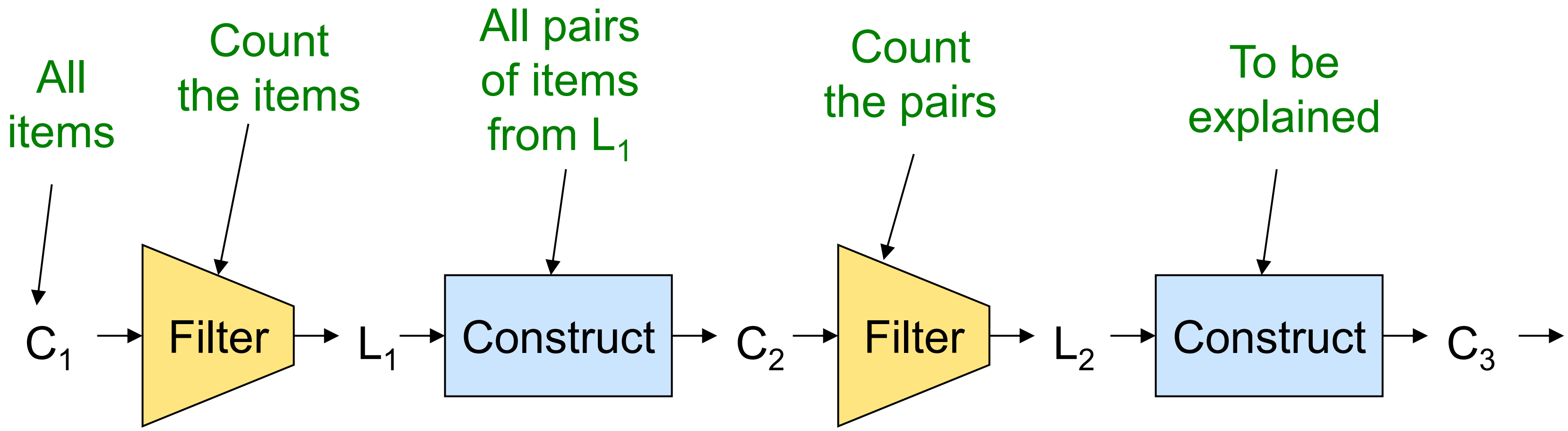
Detail for A-Priori

- 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



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



Example

** Note here we generate new candidates by generating C_k from L_{k-1} and L_1.
But that one can be more careful with candidate generation. For example, in C_3 we know {b,m,j} cannot be frequent since {m,j} is not frequent

- **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: 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 $\longrightarrow$ FruitJam
    - BakedGoods, MilkProduct $\longrightarrow$ PreservedGoods
  - Lower the support ***s*** as itemset gets bigger

PCY (Park-Chen-Yu) Algorithm

- 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 as fit in memory
  - 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!**

PCY Algorithm – First Pass

New in PCY

```
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;
```

- **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

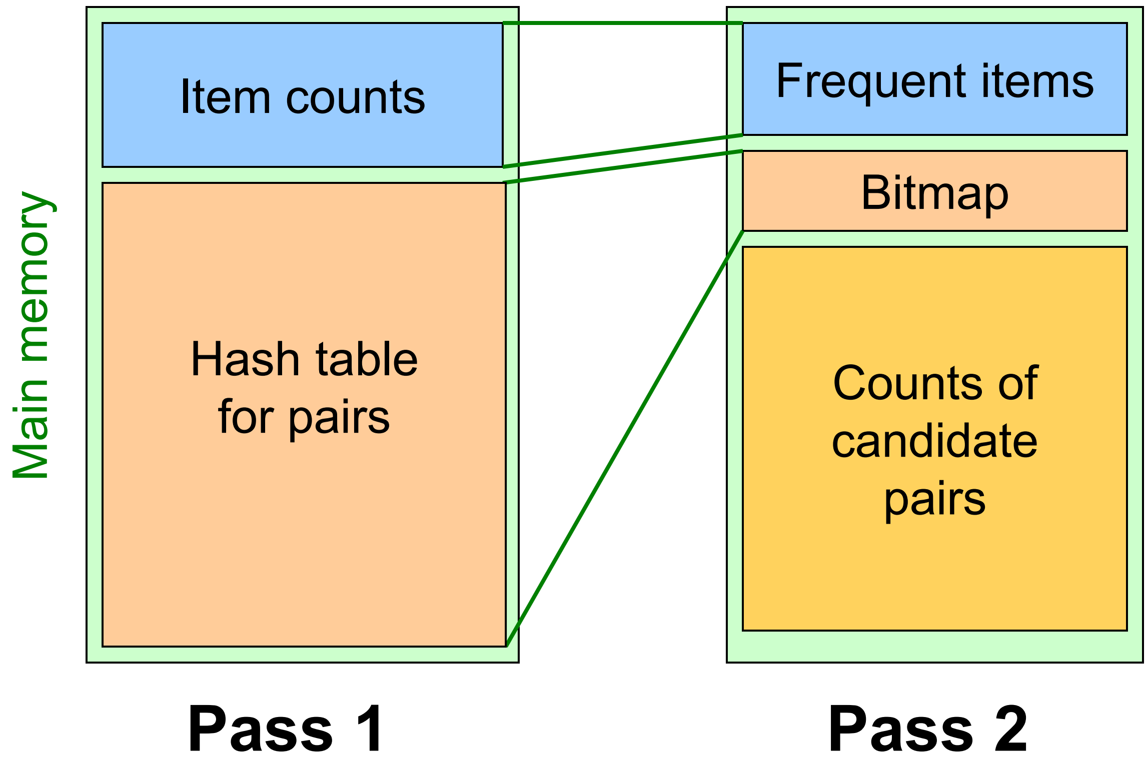
- **Replace the buckets by a bit-vector:**
  - **1** means the bucket count exceeded the support s (call it a **frequent bucket**); **0** means it did not
- **4-byte integer counts are replaced by bits, so the bit-vector requires 1/32 of memory**
- Also, decide which items are frequent and list them 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



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 $\leq$ 2 Passes

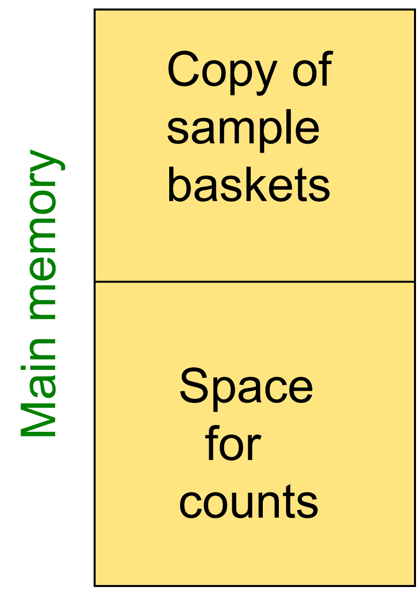
- Simple Algorithm
- Savasere-Omiecinski- Navathe (SON) Algorithm
- Toivonen's Algorithm

Frequent Itemsets in $\leq$ 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
  - 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*.



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 frequent in the whole but not in the sample**
  - Smaller threshold, e.g., $s/125$, helps catch more truly frequent itemsets
    - But requires more space

SON Algorithm – (1)

- **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 ***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

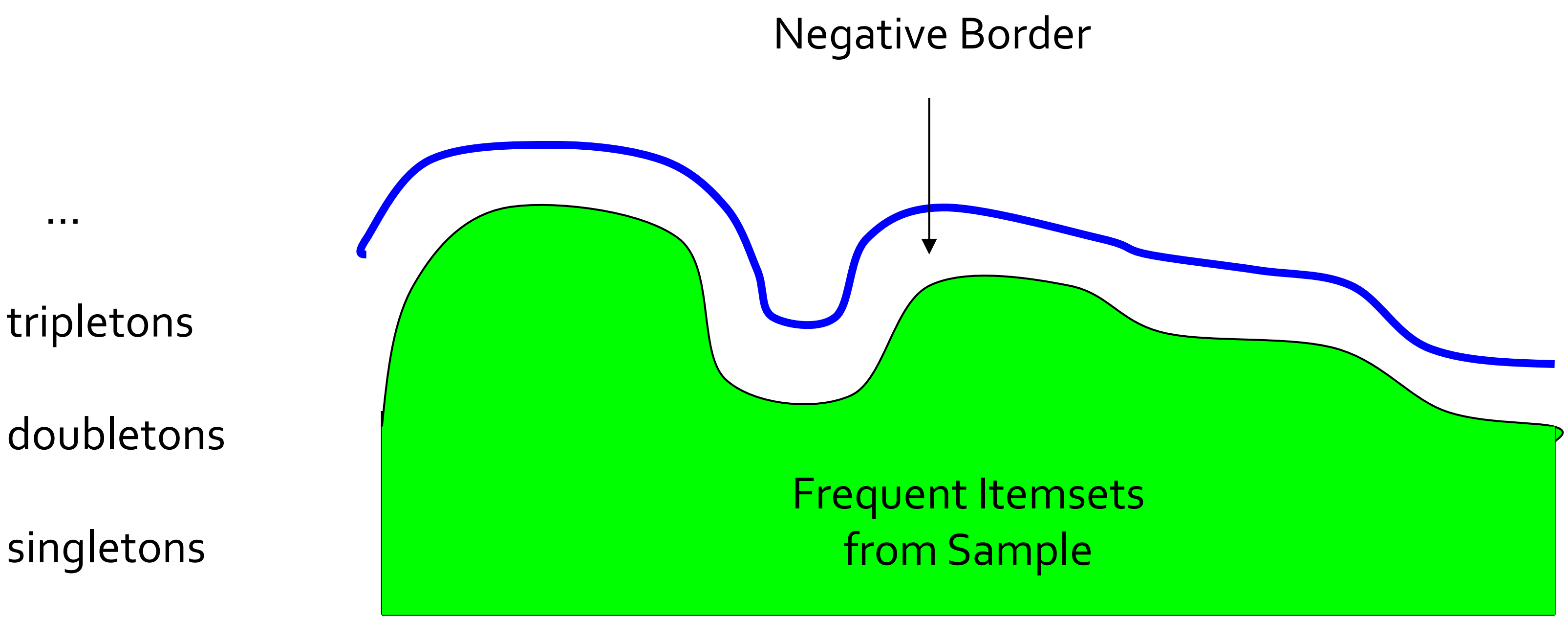
Toivonen's Algorithm: Intro

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

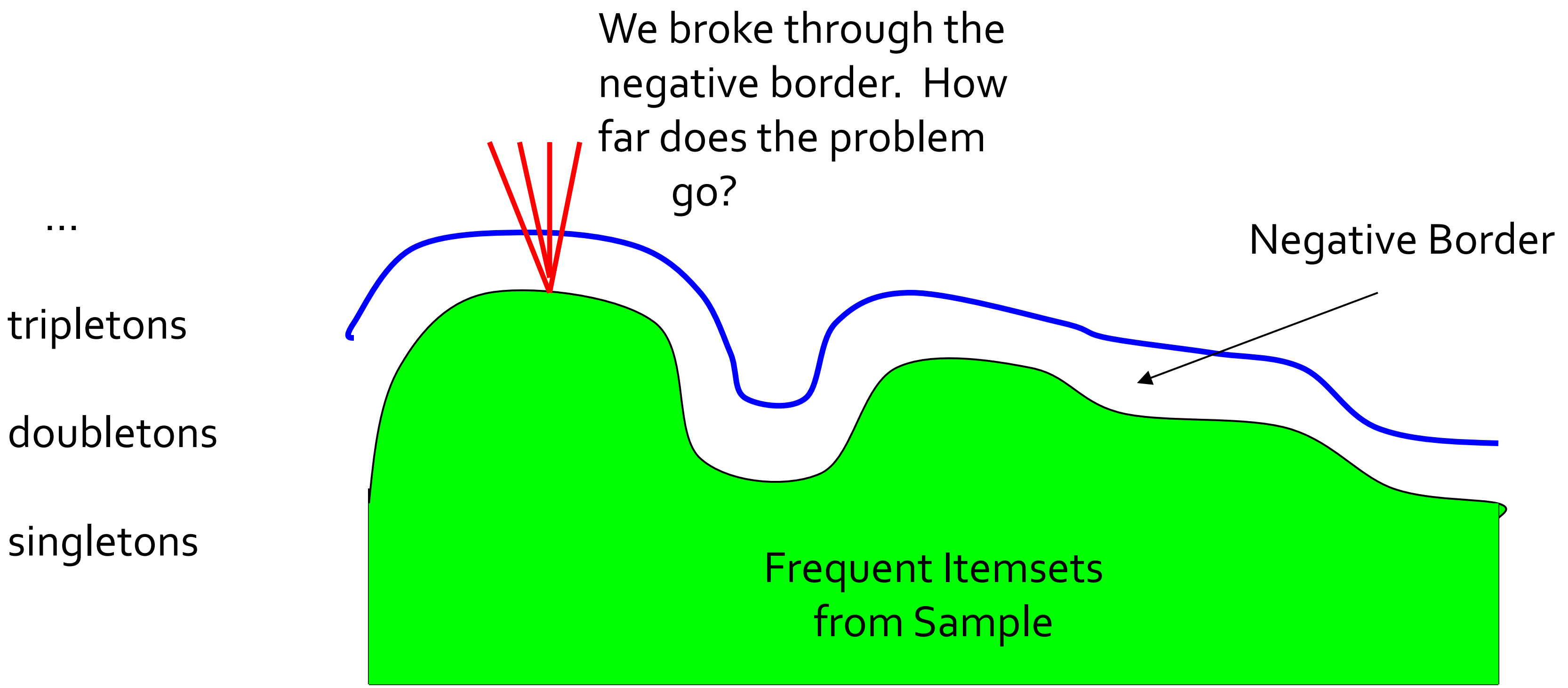


Toivonen's Algorithm

- **Pass 1:**
  - Start as in the SON algorithm, but lower the threshold slightly for the sample
  - Add to the itemsets that are frequent in the sample 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.

If Something in the Negative Border Is Frequent . . .



Theorem:

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

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)