

IERG4300/ IEMS5709
Web-Scale Information Analytics

Frequent Itemsets and
Association Rule Mining

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

- The slides used in this chapter are adapted from:
 - CS246 Mining Massive Data-sets, by Jure Leskovec, Stanford University.

with the author's permission. All copyrights belong to the original author of the material.

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 one buys diaper and milk, then he is likely to buy beer
 - Don't be surprised if you find six-packs next to diapers!

<i>TID</i>	<i>Items</i>
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Rules Discovered:

{Milk} --> {Coke}

{Diaper, Milk} --> {Beer}

The Market-Basket Model

- A large set of *items*

- e.g., things sold in a supermarket

- A large set of *baskets*, each is a small subset of items

- e.g., the things one customer buys on one day

- **A general many-many mapping (association) between two kinds of things**

- But we ask about connections among “items”, not “baskets”

<i>TID</i>	<i>Items</i>
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Association Rules: Approach

- **Given a set of baskets**
- Want to discover **association rules**
 - People who bought $\{x,y,z\}$ tend to buy $\{v,w\}$
 - Amazon!
- **2-step approach:**
 - 1) Find frequent **itemsets**
 - 2) Generate **association rules**

<i>TID</i>	<i>Items</i>
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Rules Discovered:

{Milk} --> {Coke}

{Diaper, Milk} --> {Beer}

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
 - Suggests tie-in “tricks”, e.g., run sale on diapers and raise the price of beer
 - High **support** needed, or no \$\$’s
- **Amazon’s people who bought X also bought Y**

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** = 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

Apriori algorithm

PCY algorithm + refinements

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**

<i>TID</i>	<i>Items</i>
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

Example: Frequent Itemsets

- **Items** = {milk, coke, pepsi, beer, juice}
- **Minimum 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}.

Association Rules

- **Association Rules:**

If-then rules about the contents of baskets

- $\{i_1, i_2, \dots, i_k\} \rightarrow j$ means: “if a basket contains all of i_1, \dots, i_k then it is *likely* to contain j ”

- **In practice there are many rules, want to find significant/interesting ones!**

- **Confidence** of this association rule is the probability of j given $I = \{i_1, \dots, i_k\}$

$$\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)} = \Pr[j | I]$$

*Note: $\text{support}(I \cup j) = \#$ (or %) of baskets contain BOTH I AND j

Interesting Association Rules

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

- The rule $X \rightarrow milk$ 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

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

$$\begin{aligned}\text{Interest}(I \rightarrow j) &= \text{conf}(I \rightarrow j) - \Pr[j] \\ &= \Pr[j | I] - \Pr[j]\end{aligned}$$

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: $\{m, b\} \rightarrow c$

- **Confidence** = $2/4 = 0.5$
- **Interest** = $|0.5 - 5/8| = 1/8$
 - 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 on the left side
- **Hard part:** Finding the frequent itemsets!
 - If $\{i_1, i_2, \dots, i_k\} \rightarrow j$ has high support and confidence, then both $\{i_1, i_2, \dots, i_k\}$ and $\{i_1, i_2, \dots, i_k, j\}$ will be “frequent”

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

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 $A \rightarrow I \setminus A$
 - Since I is frequent, A is also frequent
 - **Variant 1:** Single pass to compute the rule confidence
 - $\text{conf}(A, B \rightarrow C, D) = \text{supp}(A, B, C, D) / \text{supp}(A, B)$
 - **Variant 2:**
 - **Observation**:** If $A, B, C \rightarrow D$ is below confidence, so is $A, B \rightarrow C, D$
 - Can generate “bigger” rules from smaller ones!
 - **Output the rules above the confidence threshold**

Mining Association Rules (cont'd)

- **Claim:** If $A,B,C \rightarrow D$ is below confidence, so is $A,B \rightarrow C,D$

Why ?

Since $\text{Supp}(ABC) \leq \text{Supp}(AB)$

Therefore:

$$\text{Conf.}(ABC \rightarrow D) = \frac{\text{Supp}(ABCD)}{\text{Supp}(ABC)} \geq \frac{\text{Supp}(ABCD)}{\text{Supp}(AB)} = \text{Conf}(AB \rightarrow CD)$$

Thus,

IF $\text{Conf}(AB \rightarrow CD) \geq \text{Threshold}$ THEN $\text{Conf}(ABC \rightarrow D)$ also $\geq \text{threshold}$;

Equivalently,

IF $\text{Conf}(ABC \rightarrow D) < \text{Threshold}$ then $\text{Conf}(AB \rightarrow CD)$ is also below threshold

This means we can first check $\text{Conf}(AB \rightarrow CD)$ if it is above threshold, we can simply generate additional rules, e.g. $ABC \rightarrow D$, $ABD \rightarrow C$.

=> Can generate “bigger” rules from smaller ones!

Example

$$B_1 = \{m, a, b\}$$

$$B_2 = \{m, p, j\}$$

$$B_3 = \{m, a, b, n\}$$

$$B_4 = \{a, j\}$$

$$B_5 = \{m, p, b\}$$

$$B_6 = \{m, a, b, j\}$$

$$B_7 = \{a, b, j\}$$

$$B_8 = \{b, a\}$$

- Min support $s=3$, confidence $c=0.75$

- **1) Frequent itemsets:**

- $\{b, m\}$ $\{a, b\}$ $\{a, m\}$ $\{a, j\}$ $\{m, a, b\}$

- **2) Generate rules:**

- ~~$b \rightarrow m: c=4/6$~~ $b \rightarrow a: c=5/6$ ~~$b, a \rightarrow m: c=3/5$~~
- $m \rightarrow b: c=4/5$... $b, m \rightarrow a: c=3/4$
- ~~$b \rightarrow a, m: c=3/6$~~

A Compact Way to store/track Frequent Itemsets

You only need to store the so-called:

Maximal Frequent itemsets:

Definition: a Frequent set for which **NO** immediate superset is frequent

Nice Properties:

All subsets of a *Maximal Frequent itemset* are frequent

AND

Every Frequent itemset must be a subset of some *Maximal Frequent itemset*

=> By enumerating **ALL subsets** of **all Maximal Frequent Itemsets**, you will **NOT** miss any Frequent Itemset ! Also, every subset you got is a Frequent Itemset !

Example: Maximal Frequent Itemset

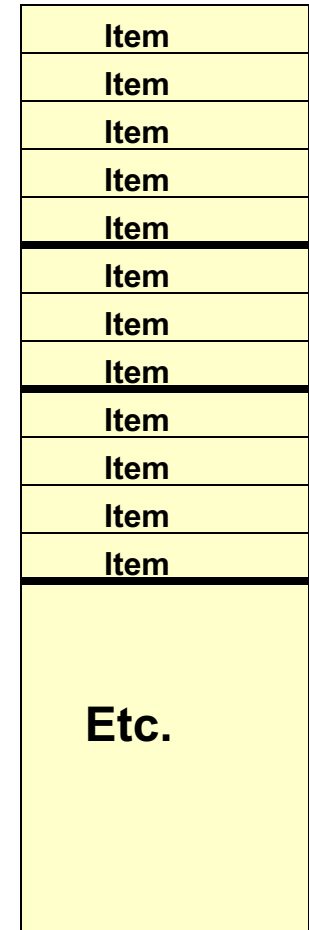
	Count	Maximal (s=3)	
A	4	No	
B	5	No	Frequent, but superset BC also frequent.
C	3	No	
AB	4	Yes	Frequent, and its only superset, ABC, not freq.
AC	2	No	
BC	3	Yes	
ABC	2	No	

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**

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



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

Computation Model

- 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 makes over the data

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 (**why?**)

Finding Frequent Pairs

- **The hardest problem often turns out to be finding the frequent pairs of items $\{i_1, i_2\}$**
 - **Why?** Often frequent pairs are common, frequent 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: $10^5(10^5-1)/2 = 5 \cdot 10^9$
 - Therefore, $2 \cdot 10^{10}$ (20 gigabytes) of memory needed

Counting Pairs in Memory

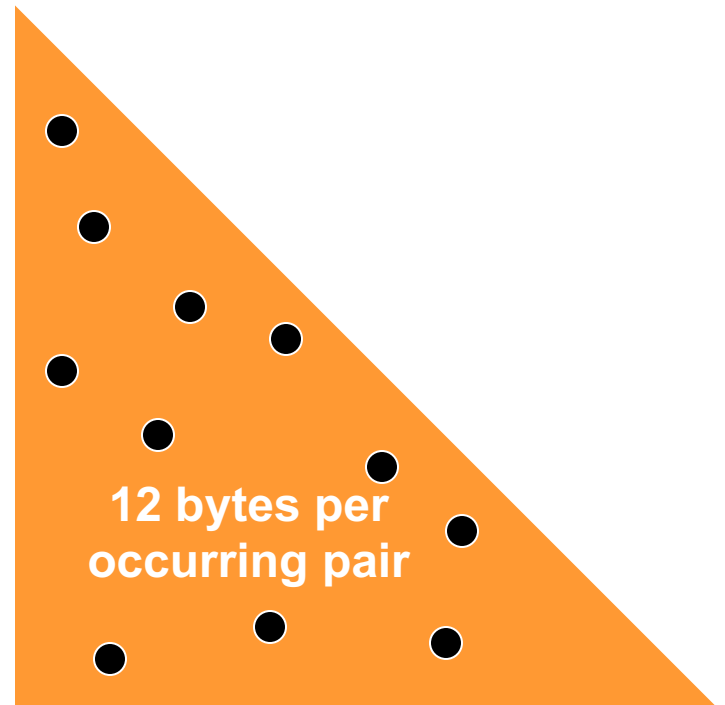
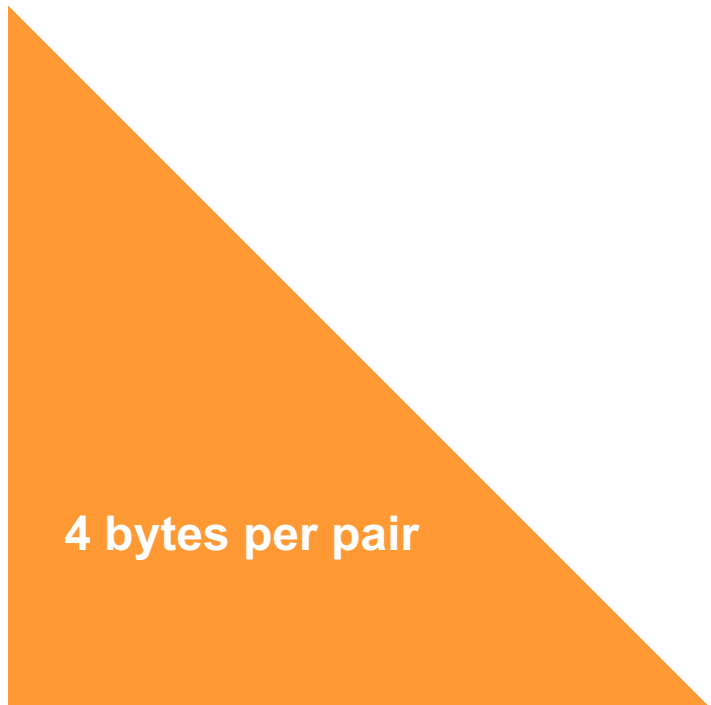
Two Approaches:

- **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

Note:

- **Approach 1** only requires 4 bytes per pair
- **Approach 2** uses 12 bytes per pair (but only for pairs with count > 0)

Comparing the 2 Approaches



Triangular Matrix Approach

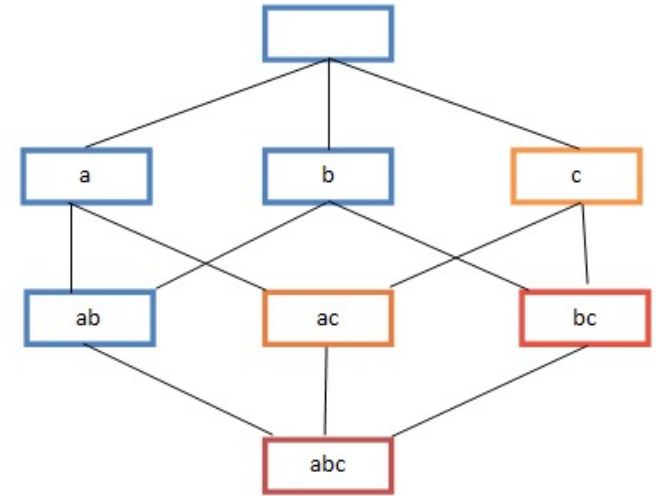
Triangular Matrix Approach

- n = total number items
- Count pair of items $\{i, j\}$ only if $i < j$
- Keep pair counts in lexicographic order:
 - $\{1,2\}, \{1,3\}, \dots, \{1,n\}, \{2,3\}, \{2,4\}, \dots, \{2,n\}, \{3,4\}, \dots$
- Pair $\{i, j\}$ is at position $(i-1)(n-i/2) + j - 1$
- Total number of pairs $n(n-1)/2$; total bytes = $2n^2$
- **Triangular Matrix** requires 4 bytes per pair
- **Approach 2** uses 12 bytes per pair
(*but only for pairs with count > 0*)
 - Beats triangular matrix if less than 1/3 of possible pairs actually occur

The A-Priori Algorithm

A-Priori Algorithm – (1)

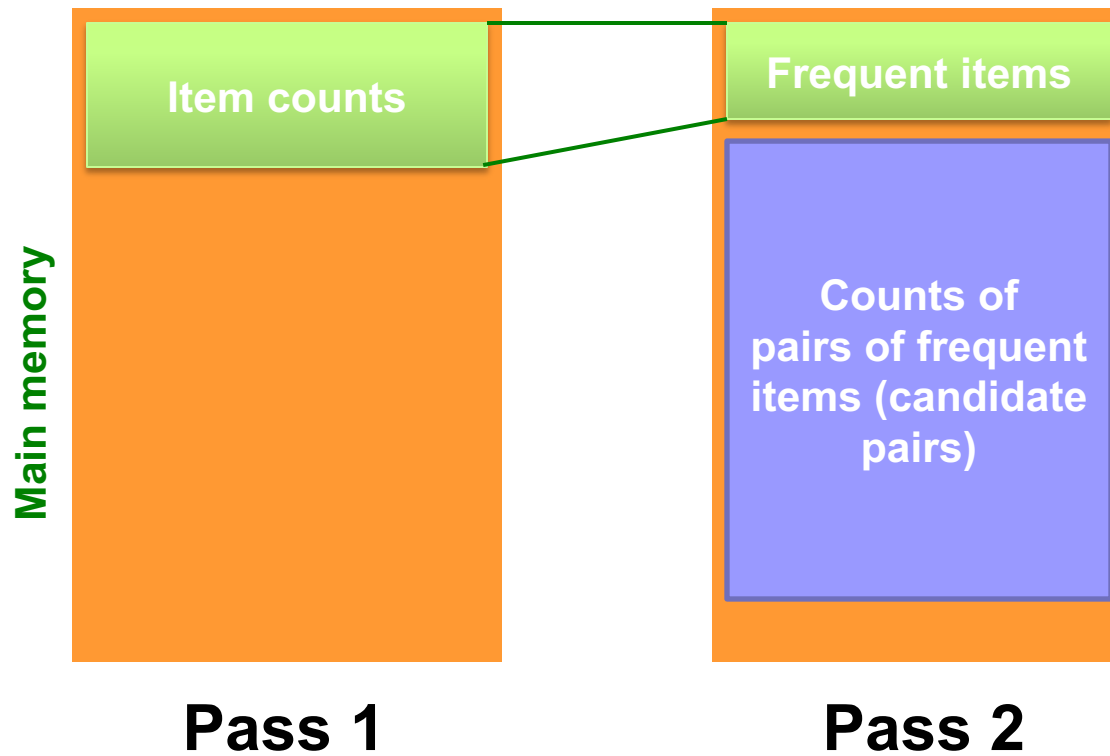
- A **two-pass** approach called *a-priori* limits the need for main memory
- **Key idea: monotonicity**
 - If a set of items I appears at least s times, so does every **subset** J of I .
- **Contrapositive for pairs:**
If item i does not appear in s baskets, then no pair including i can appear in s baskets



A-Priori Algorithm – (2)

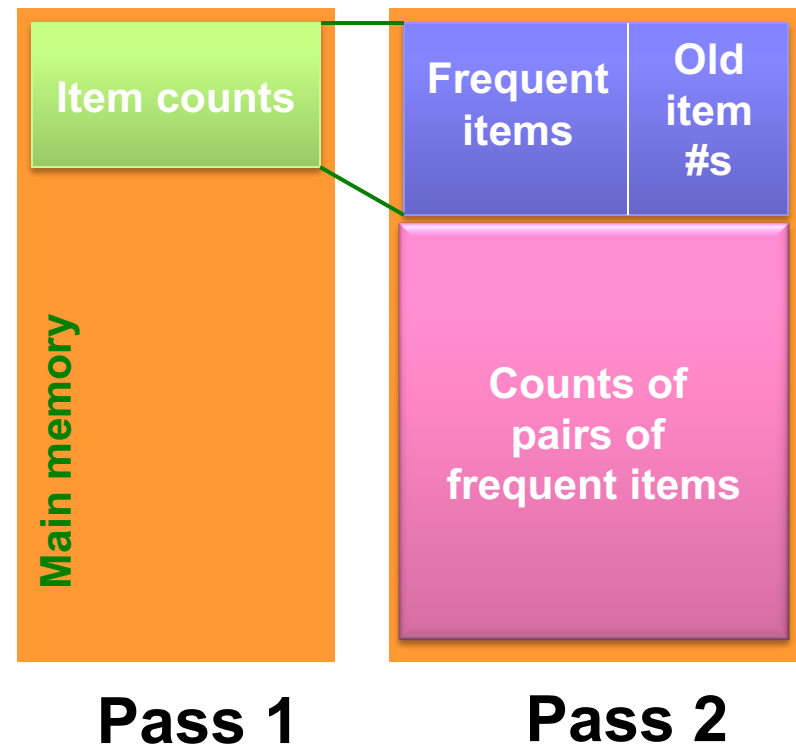
- **Pass 1:** Read baskets and count in main memory the occurrences of each **individual item**
 - Requires only memory proportional to #items, usually enough memory even for 1 billion (= n) different types of items !
- **Items that appear $\geq s$ times are the frequent items**
- **Pass 2:** Read baskets 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



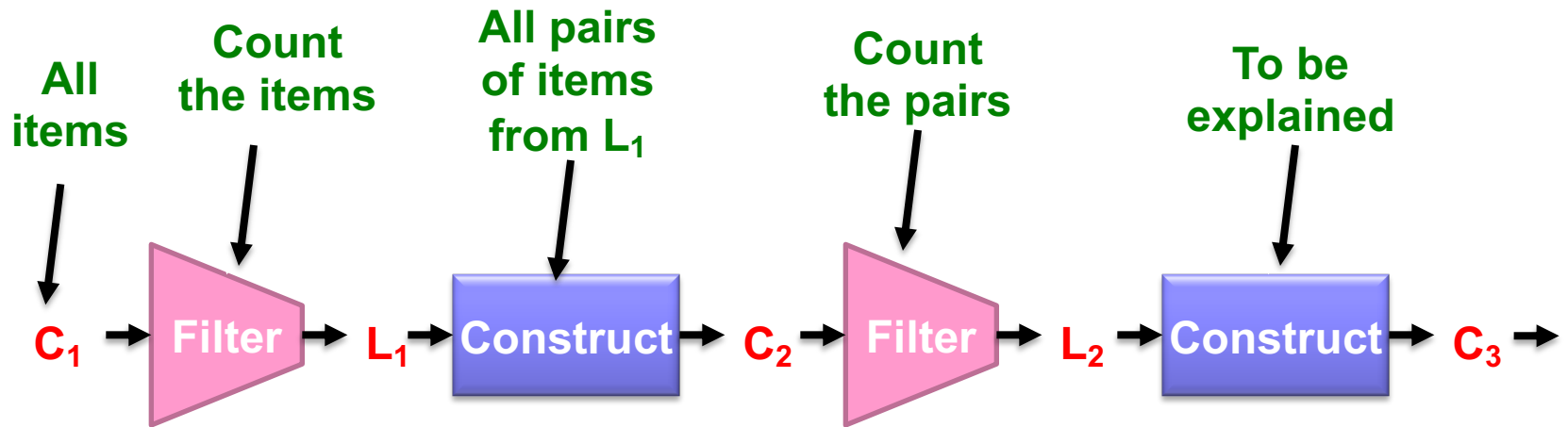
Detail for A-Priori

- 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

○ 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: $L_3 = \{ \{b,c,m\} \}$

**

** 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

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

PCY (Park-Chen-Yu) Algorithm

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
 - **Just the count, not the pairs that hash to the bucket!**

PCY Algorithm – First Pass

```
FOR (each basket) :
```

```
    FOR (each item in the basket) :
```

```
        add 1 to item's count;
```

```
    New { FOR (each pair of items) :  
    in   hash the pair to a bucket;  
    PCY } add 1 to the count for that bucket;
```

- 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

- If a bucket contains a **frequent pair**, then the bucket is surely **frequent**
 - But we cannot based on the hash alone to eliminate any non-frequent member within this bucket
- Even without any frequent pair, a bucket can still be frequent (**why?**)
- **But, for a bucket with total count less than s , none of its pairs can be frequent**
 - Pairs that hash to a **non-frequent bucket** can be eliminated from the candidate list (**even if the pair consists of 2 frequent items**)
- **Pass 2:**
Only count pairs that hash to frequent buckets

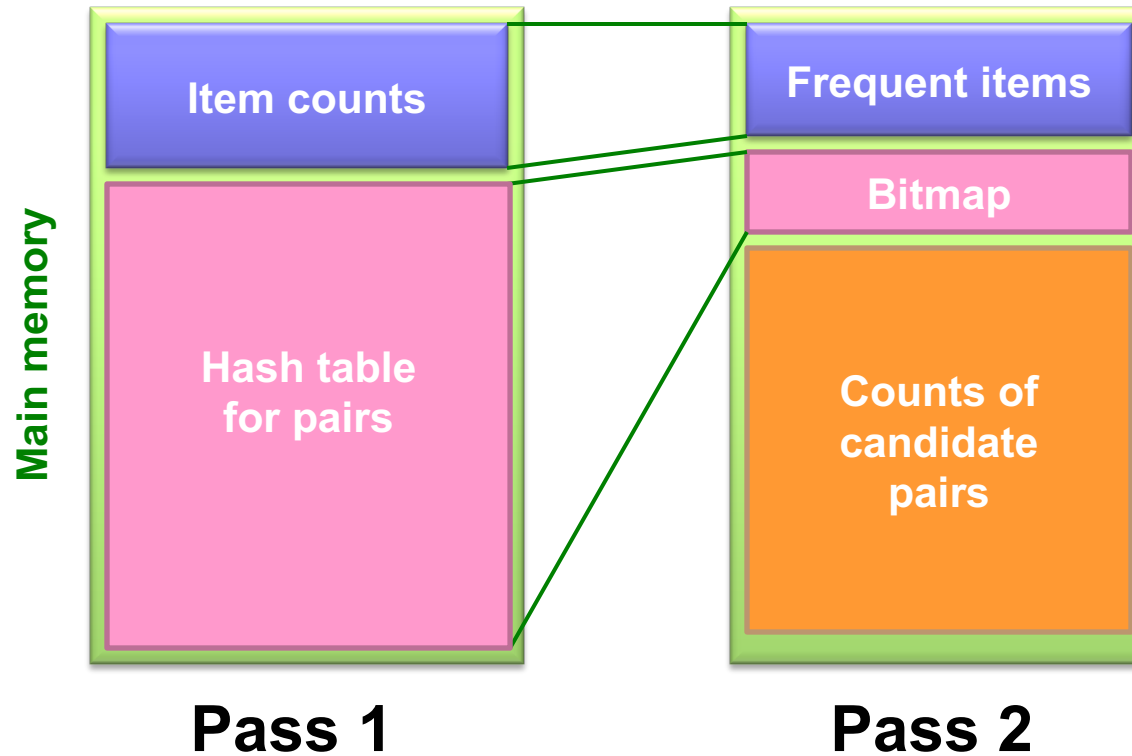
PCY Algorithm – Between Passes

- **Replace the buckets by a bit-vector:**
 - 1 means the bucket count exceeded the support s (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 in Pass 1**
- 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., frequent bucket)
- **Both conditions are necessary for the pair to have a chance of being frequent**

Main-Memory: Picture of PCY



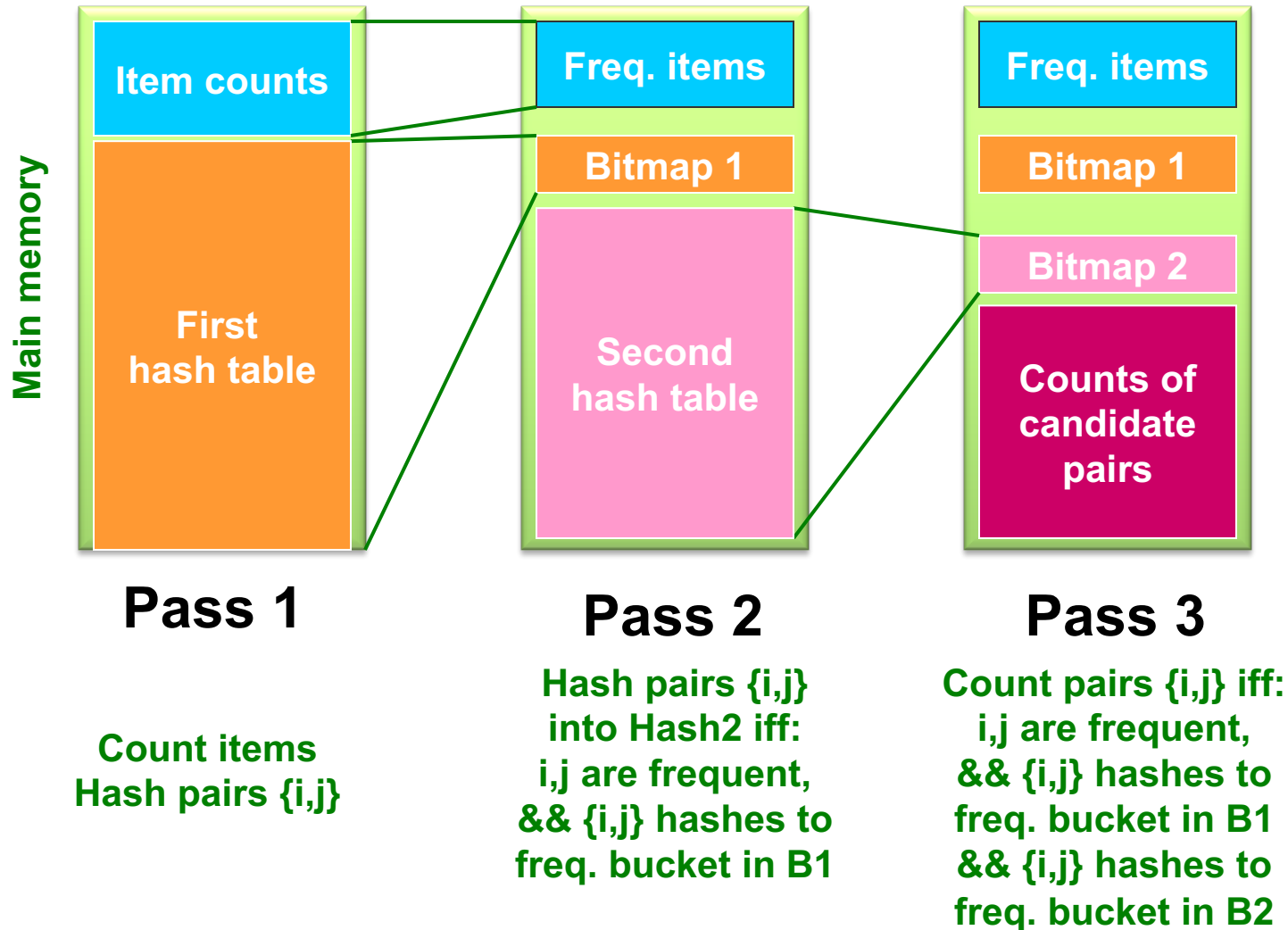
Main-Memory Details

- Buckets require a few bytes each:
 - **Note:** we don't have to count past s
 - #buckets is $O(\text{main-memory size})$
- On second pass, a table of (item, item, count) triples is essential (we cannot use triangular matrix approach, why?)
 - Thus, hash table must eliminate approx. 2/3 of the candidate pairs for PCY to beat A-Priori.

Refinement: Multistage Algorithm

- **Limit the number of candidates to be counted**
 - **Remember:** Memory is the bottleneck
 - Still need to generate all the itemsets but we only want to count/keep track of the ones that are frequent
- **Key idea:** After Pass 1 of PCY, rehash only those pairs that **qualify** for Pass 2 of PCY
 - i and j are frequent, and
 - $\{i, j\}$ hashes to a frequent bucket from Pass 1
- On middle pass (i.e. the new Pass 2), fewer pairs contribute to buckets, so fewer **false positives**
- **Requires 3 passes over the data**

Main-Memory: Multistage



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.

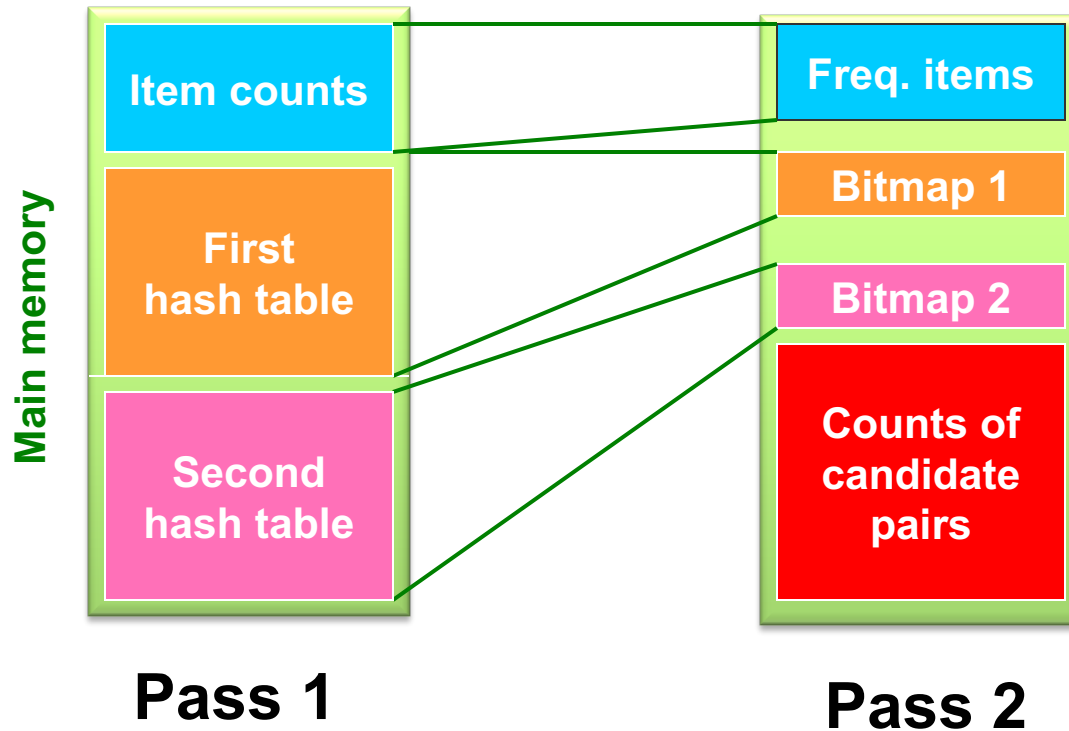
Important Points

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

Refinement: 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

Main-Memory: Multihash



PCY: Extensions

- Either **multistage** or **multihash** can use more than two hash functions
- In **multistage**, there is a point of diminishing returns, since the bit-vectors eventually consume all of main memory
- For **multihash**, the bit-vectors occupy exactly what one PCY bitmap does, but given the constant amount of main memory to hold all hash tables, **too many hash functions makes all counts $\geq s$, and thus, fails to eliminate any non-frequent pairs !**

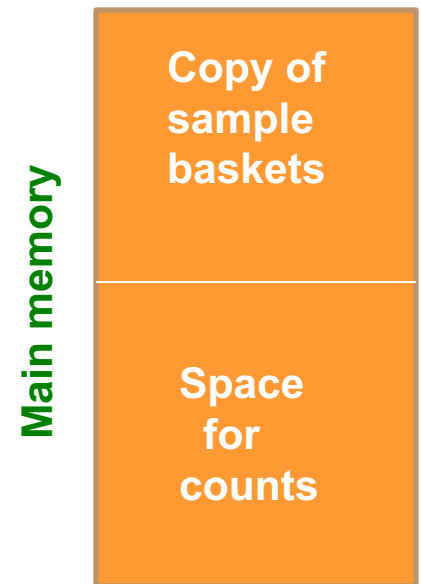
Frequent Itemsets in ≤ 2 Passes

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
 - Random sampling
 - SON (Savasere, Omiecinski, and Navathe)
 - Toivonen (see textbook [MMDS Ch6])

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
 - **MUST reduce support threshold proportionally to match the 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)
- 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)

- 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 idea:** an itemset cannot be frequent in the entire set of baskets **unless it is frequent in at least one subset.**

SON – Distributed Version

- SON lends itself to distributed/ parallel implementation, e.g. using MapReduce
- Baskets distributed among many nodes
 - Compute frequent itemsets at each node
 - Distribute candidate itemsets to all nodes
 - Accumulate the counts of all candidates
- Can be done with two MapReduce jobs:
 - First MapReduce job to produce the candidate itemsets
 - Second MapReduce job to calculate the true frequent itemsets.

SON: Map/Reduce

- **Job 1:** Find candidate itemsets
 - Map?
 - Reduce?

- **Job 2:** Find true frequent itemsets
 - Map?
 - Reduce?

SON: MapReduce Implementation

Mapper for Job 1

- Run A-Priori algorithm on the chunk using support threshold ps
- Output the **frequent itemsets** for that chunk (F, c) , where F is the key (itemset) and c is count (or proportion)

Reducer for Job 1

- Output the candidate itemsets to be verified in the Job 2
- Given (F, c) , discard c and output all candidate itemsets F 's

SON: MapReduce Implementation (cont'd)

Mapper for Job 2

- For all the candidate itemsets produced by Job 1, count the frequency in local chunk

Reducer for Job 2

- Aggregate the o/p of the Mapper of Job 2 and sum the count to get the frequency of each candidate itemsets across the entire input file
- Filter out the itemsets with support smaller than s