**Lecture 1**

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

Baskets = sentences; Items = documents containing those sentences

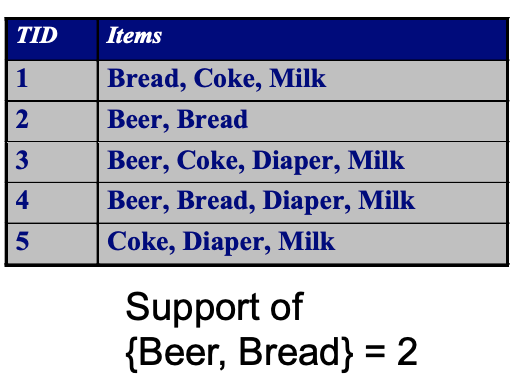
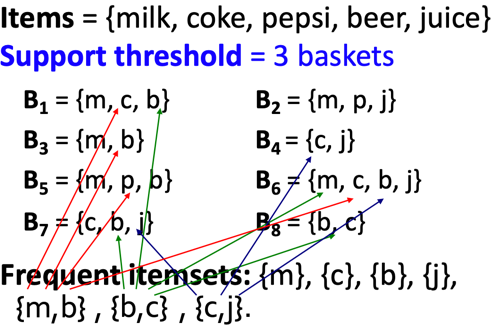
Baskets = patients; Items = drugs & side-effects

There are also many-to-many mapping association between two kinds of things.

Baskets = patients; Items = drugs & side-effects

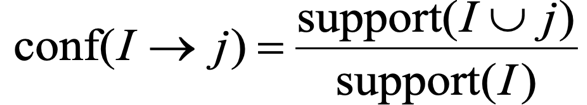
**Support** for itemset **I**: Number of baskets containing all items in **I.**

Given a **support threshold s**, then sets of items that appear in at least **s** baskets are called ***frequent itemsets***

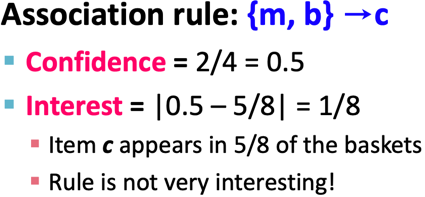
**ASSOCIATION RULES:** If-then rules about the contents of baskets

* **{i1, i2,…,ik}** → j means: “if a basket contains all of i1,…,ik 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 = {i1,…,ik}**



* **Interest** of an association rule **I → j:** difference between its confidence and the fraction of baskets that contain





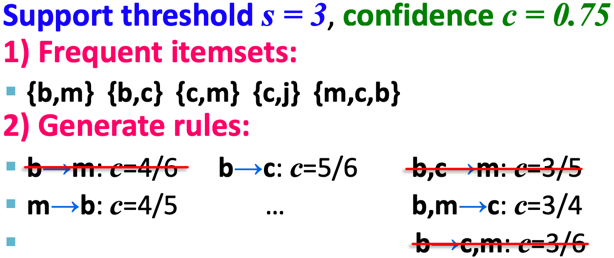
MINING ASSOCIATION RULES:

Step 1: find all frequent itemsets I

Step 2: Rule generation



Example



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.

For many frequent itemst algorithms, main-memory is the critical resource – As we read baskets, we need to count something, e.g., occurrences of pairs of items.

The hardest problem often turns out to be finding the frequent pairs of items {i1, i2,} –

why? Because frequent pairs are common, frequent triples are rare.

why? Because probability of being frequent drops exponentially with size.

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

Counting Pairs in Memory

Two approaches:

Approach 1: count all pairs using a matrix – only requires 4 bytes per pair

* 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},…, {1,n}, {2,3}, {2,4},…,{2,n}, {3,4},…

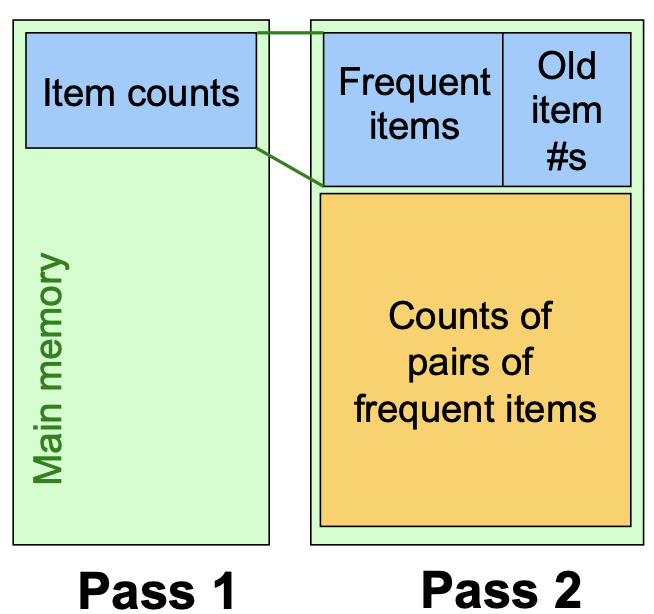
* pair {i, j} is at position (i – 1)(n – i/ 2) + j -1
* Total number of pairs **n(n-1)/2**; total bytes = **2n2**

Approach 2: keep triples [i, j, c] = “the count of the pair of items {i, j} if c – uses 12 bytes per pair (but only for pairs with count > 0)

**A-Priori Algorithm** – A two-pass approach called A-Priori limits the need for main memory

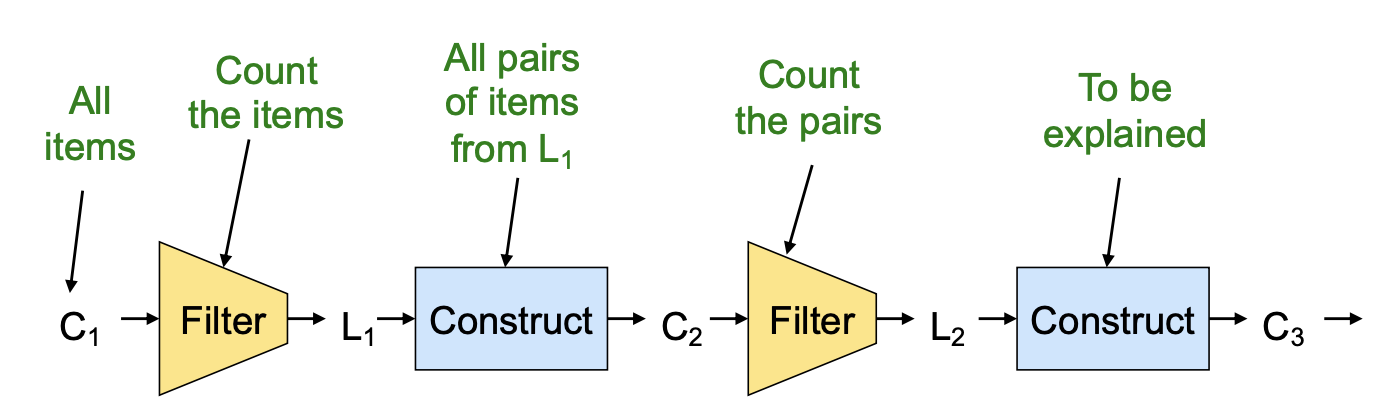
Key idea: monotonicity – if a set of items **I** appears at least **s** times, so does every **subset J** of **I**

Contrapositive for pairs: if item **I** does not appear in **s** baskets, then no pair including **i** can appear in **s** baskets.

* Pass 1: Read baskets and count in main memory the occurrences of each individual item
* Items that appear >= 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)
* **
* For each k, we construct

§ **Ck** = candidate = those that might be frequent sets (support **>= s**) based on information from the pass for **k–1**

§ **Lk** = the set of truly frequent **k**-tuples

* **

Hypothetical steps of the A-Priori algorithm

* C1 = { {b} {c} {j} {m} {n} {p} }
* Count the support of items in C1
* Prune non-frequent: L1 = { b, c, j, m }
* Generate C2 = { {b,c} {b,j} {b,m} {c,j} {c,m} {j,m} }
* Count the support of items in C2
* Prune non-frequent: L2 = { {b,m} {b,c} {c,m} {c,j} }
* Generate C3 = { {b,c,m} {b,c,j} {b,m,j} {c,m,j} }
* Count the support of items in C3
* Count the support of itemsets in C3
* Prune non-frequent: L3 = { {b,c,m} }

**PCY (Park-Chen-Yu) Algorithm** – In A-priori, most memory is idle in pass 1, so PCY questions if that idle memory can be used 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
* That is, 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):

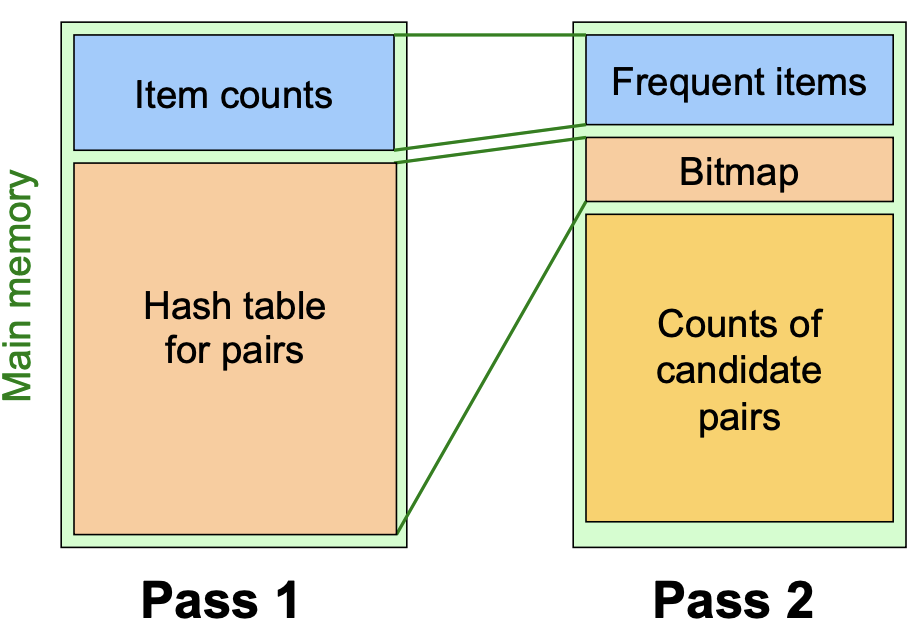
hash the pair to a bucket;

add 1 to the count for that bucket

* NOTE 1
* Pairs of items are not present in file. They need to be generated from the input file.
* We are not interested in the presence of a pair, but we need to see whether it is present at least **s**(support) times.
* NOTE 2
* A bucket with total count less than **s**(support), none of its pairs can be frequent.
* A bucket can be frequent without any frequent pair, but is surely frequent if it contains frequent pairs.
* Pass 2 of PCY: Only count pairs that hash to frequent buckets

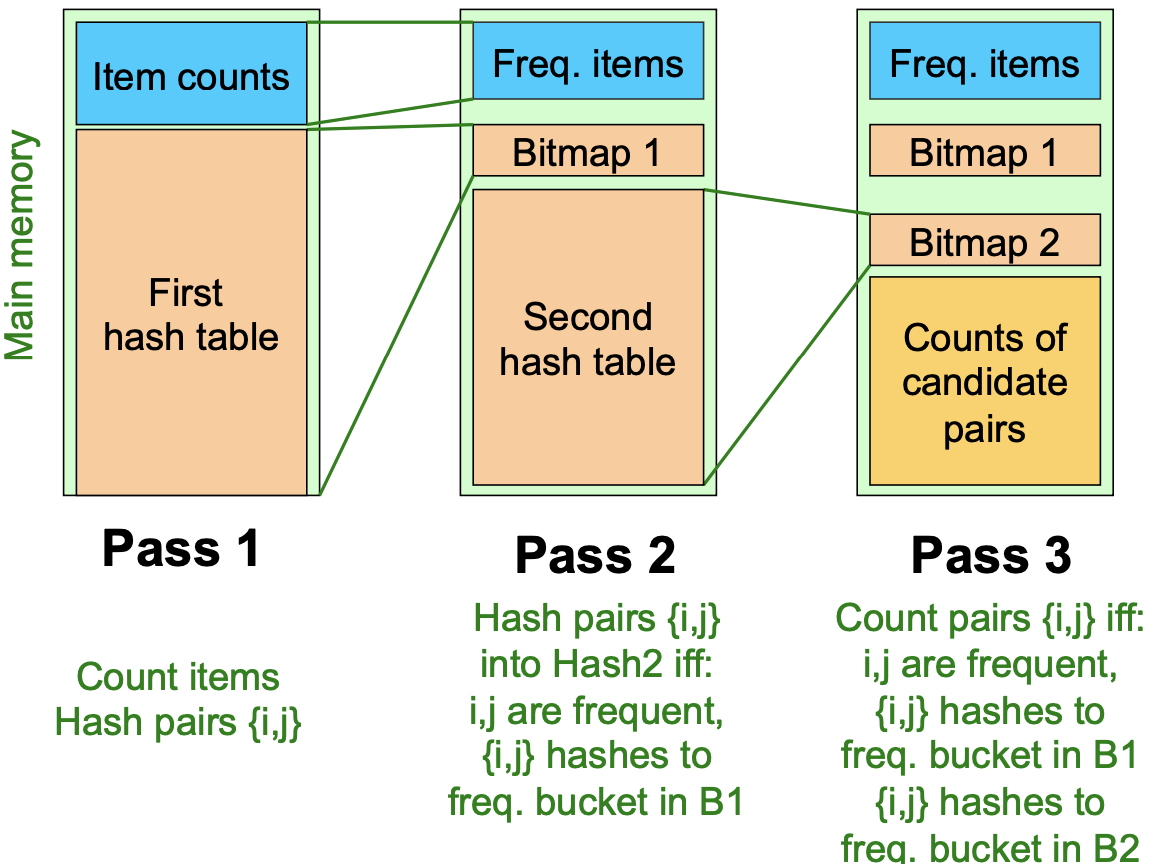
Count all pairs {i, j} that meet the conditions for being a candidate pair:

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)

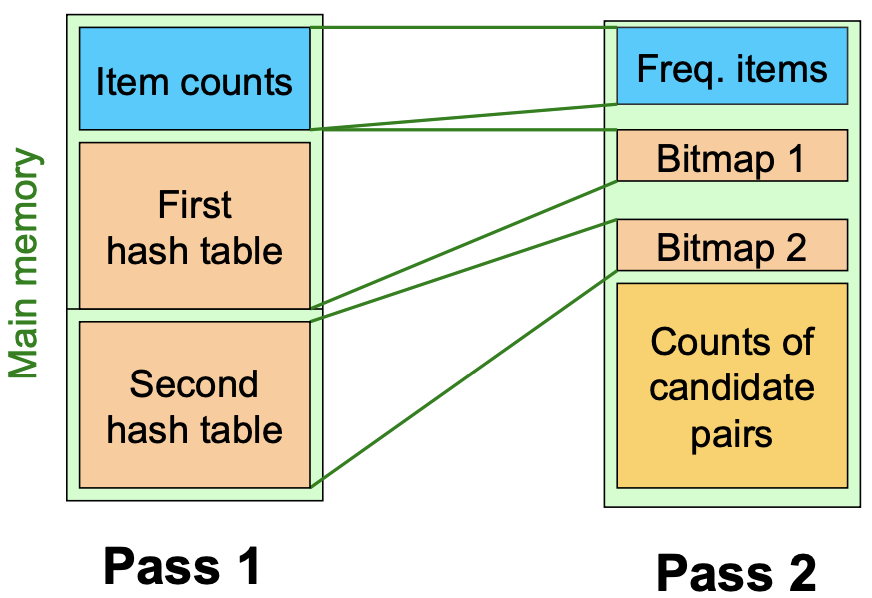


* Key idea: After Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY.
* (Multistage)Pass 3 of PCY: Count only those pairs {i, j} that satisfy these candidate pair conditions:

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



Refinement : (Mutlihash)

****

**PCY: Extensions**

Either multistage or multihash can use more than two hash functions;

1. Multistage - there is a point of diminishing returns, since the bit-vectors eventually consume all of main memory
2. Multihash – the bit-vectors occupy exactly what one PCY bitmap does, but too many hash functions makes all counts **>= s**

**Randomized Algorithms (Sampling) (1)**

* Take a random sample of the market baskets
* Run a-priori or one of its improvements
* Main memory is half (copy of sample baskets) and half (space for counts).

**Randomized Algorithms (Sampling) (2)**

* Avoid false positives
* Smaller threshold helps catch more truly frequent itemsets

**SON Algorithm (Segments) (1)**

* Repeatedly read small subsets of the baskets into main memory and run an in memory algorithm to find all frequent itemsets. (processing entire file in memory-sized chunks not sampling)
* An itemset becomes a candidate if it is found to be frequent in any one or more subsets of the baskets.

**SON Algorithm (Segments) (2)**

* On a second pass
* Count all the candidate items
* 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 on subset.

**SON – Distributed Version**

* SON lends itself to distributed data mining
* Baskets distributed among many nodes
* **Compute** frequent itemsets at each node
* **Distribute** candidates to all nodes
* **Accumulate** the counts of all candidates