Frequent Itemsets

Revise

- Market Basket Model
- Association rule mining
- Apriori Algorithm

Apriori Algorithm

T_ID	Itemsets
T_1000	M,O,N,K,E,Y
T_1001	D,O,N,K,E,Y
T_1002	M,A,K,E
T_1003	M,U,C,K,Y
T_1004	C,O,O,K,E

Itemset	Support
{M}	3
{O}	3
{N}	2
{K}	5
{E}	4
{Y}	3
{D}	1
{A}	1
{U}	1
{C}	2

Pass 1

Itemset	Support
{M}	3
{O}	3
{K}	5
{E}	4
{Y}	3

Pass 2

Itemset	Support
{M,O}	1
{M,K}	3
{M,E}	2
{M,Y}	2
{O,K}	3
{O,E}	3
{O,Y}	2
{K,E}	4
{K,Y}	3
{E,Y}	2

Memory requirement of Apriori Algo

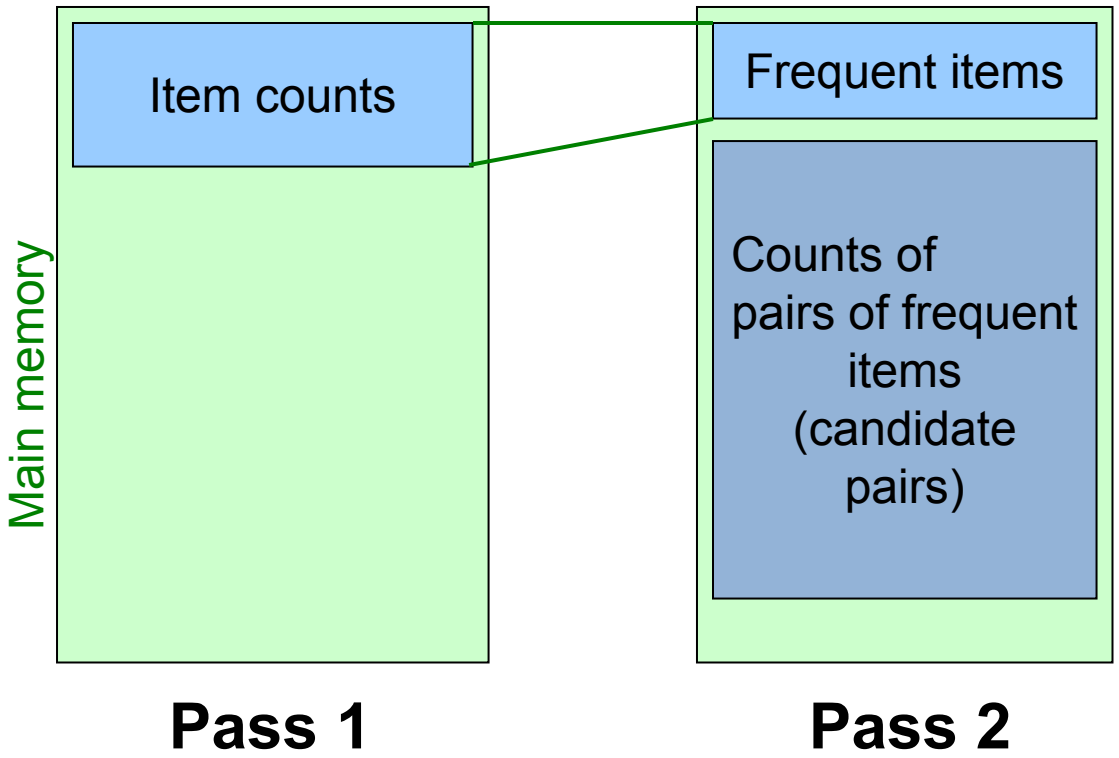
Pass 1: Read baskets and count in main memory the occurrences of each item.

- Requires only memory proportional to #items.

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

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

Main-Memory: Picture of A-Priori



PCY (Park-Chen-Yu) Algorithm

PCY Algorithm – First Pass

```
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;
```

New in PCY

Bitmap

- **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.**
- **It is the second condition that distinguishes PCY from A-Priori.**

MIN Support = 2

Transaction ID	Items
1	1,3,4,5
2	2,3,4
3	1,2,3,5
4	2,5

Itemset	Support
{1}	2
{2}	3
{3}	3
{4}	2
{5}	3

T1: {1,3} {1,4} {1,5} {3,4} {3,5} {4,5}

T1: **2** **1** **2** **2** **2** **1**

T2: {2,3} {2,4} {3,4}

T2: **2** **1** **2**

T3: {1,2} {1,3} {1,5} {2,3} {2,5} {3,5}

T3: **1** **2** **2** **2** **2** **2**

T4: {2,5}

T4: **2**

Bucket	1	2	3	4	5
Count	4	12	0	0	0

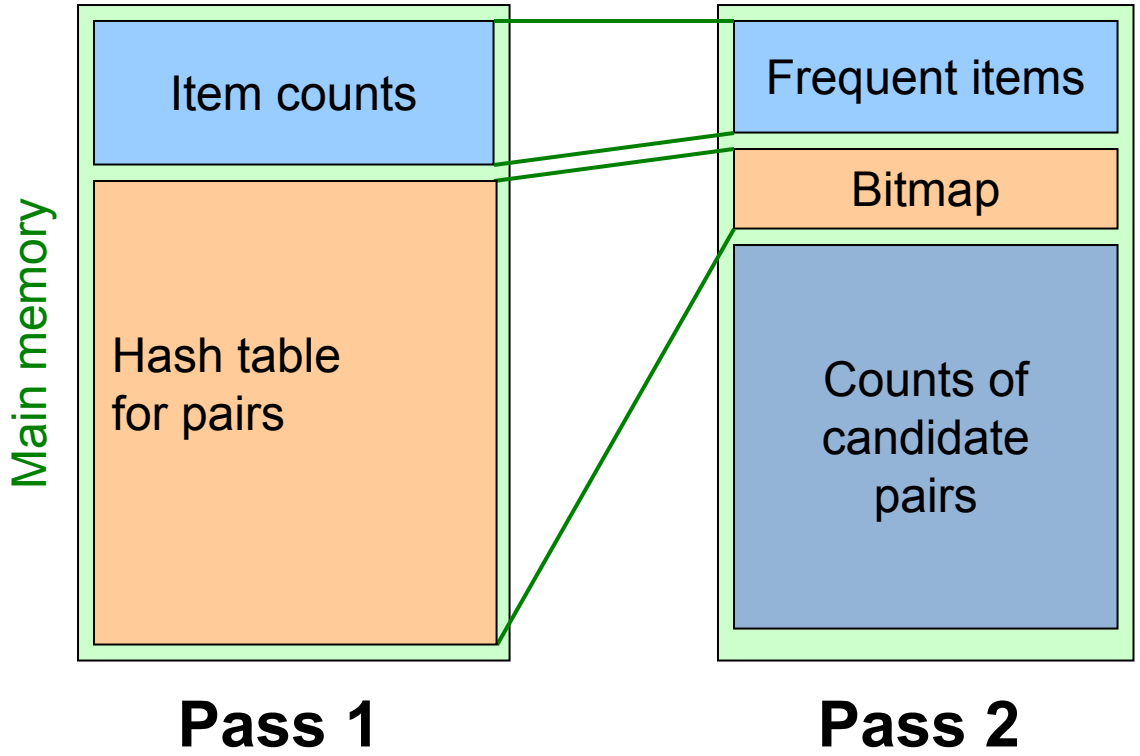
- During Pass 1 of A-priori, most memory is idle.
- Use that memory to keep counts of buckets into which pairs of items are hashed.
  - Just the count, not the pairs themselves.
- For each basket, enumerate all its pairs, hash them, and increment the resulting bucket count by 1.
- A bucket is *frequent* 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 that hash to 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!**

Main-Memory: Picture of PCY



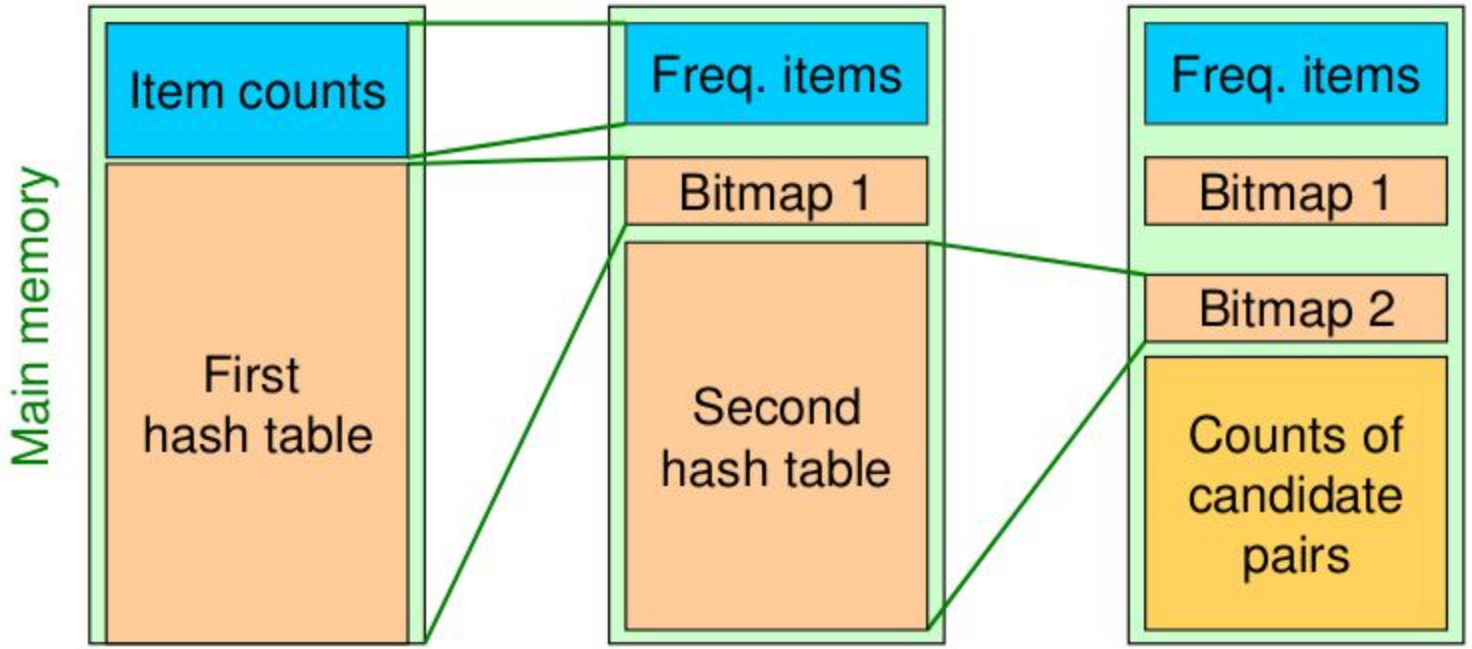
Multistage PCY algorithm

- It is an improvement to PCY.
- It uses more than 2 passes to find frequent pairs.
- Let us discuss 3 pass version of PCY.
- Benefit of extra pass is that on the final pass when we have to count candidate pairs, we have eliminated many of the pairs that PCY would have counted but that turn out not to be frequent.
- Unlike PCY, multistage does not hash all the pairs to bitmap. It hashes only those pairs which are required for next pass.

Multistage PCY algorithm

- Multistage PCY begins with pass1 same as PCY.
- Second pass is however repeat of pass 1 of PCY with different hash function.
- Third pass – Candidate pair must hash not only to a frequent bucket on the firtst pass but also hash to a frequent bucket on the second pass

Main-Memory: Multistage



Pass 1

Count items
Hash pairs {i,j}

Pass 2

Hash pairs {i,j} into Hash2 iff: i,j are frequent, {i,j} hashes to freq. bucket in B1

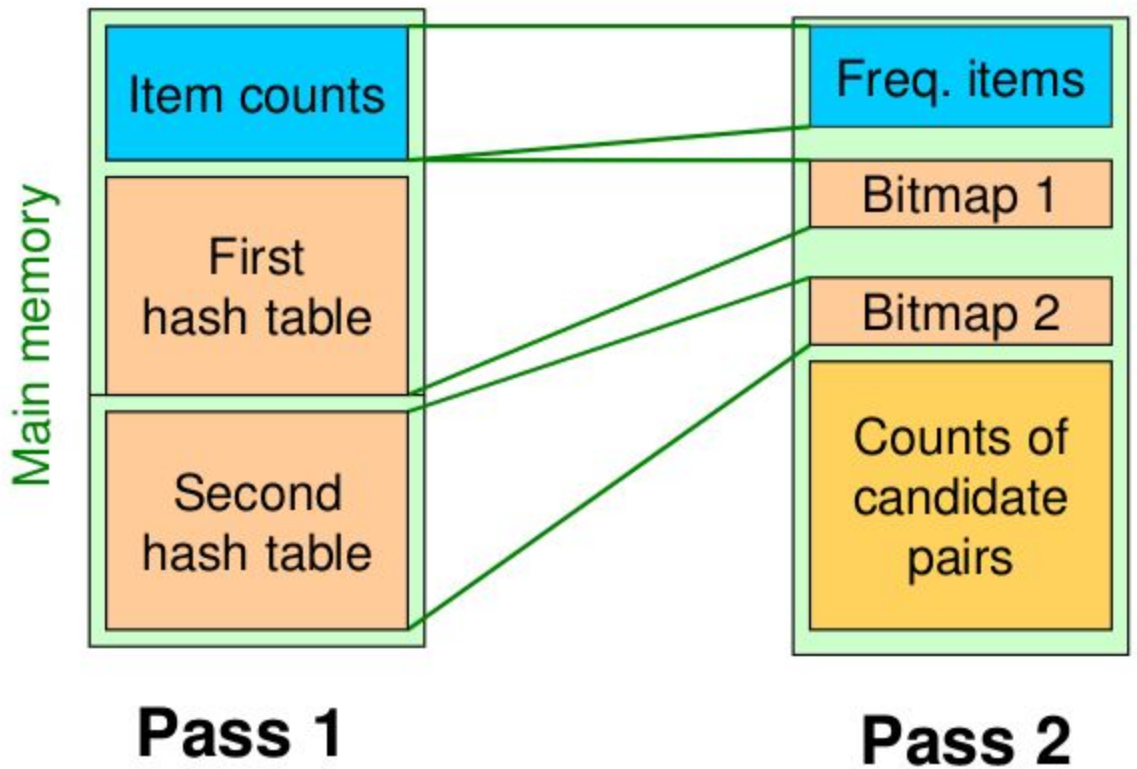
Pass 3

Count pairs {i,j} iff: i,j are frequent, {i,j} hashes to freq. bucket in B1 {i,j} hashes to freq. bucket in B2

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**

Main-Memory: Multihash



In multihash Bitmap size remains same for both bitmaps which is same as PCY.

In multistage however this size is not same. Size depends on the number of pairs eligible for next pass.

Limited Pass Algorithm

Savasere, Omiecinski & Navathe [SON] Algorithm

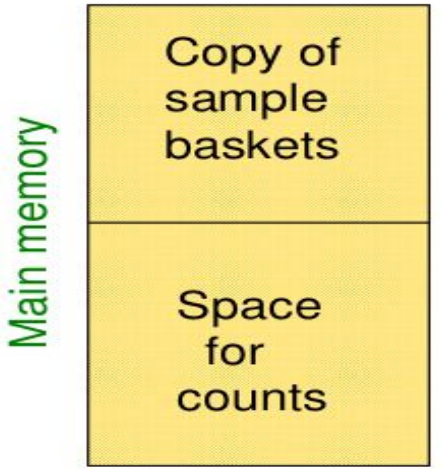
Frequent Itemsets in ≤ 2 Passes

- 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
  - Random sampling
  - SON (Savasere, Omiecinski, and Navathe)

Simple Algorithm

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



Random Sampling (2)

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

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

The SON Algorithm and MapReduce

First Map Function

Take the assigned subset of the baskets and find the itemsets frequent in the subset using the PCY/simple algorithm . Lower the support threshold from s to ps if each Map task gets fraction p of the total input file. The output is a set of key-value pairs (F, 1),where F is a itemset from the sample.

First Reduce Function

Each Reduce task is assigned a set of keys, which are itemsets. The value is ignored, and the Reduce task simply produces those keys (itemsets) that appear one or more times. Thus, the output of the first reduce function is the candidate itemsets.

Second Map Function

The Map tasks for the second Map function take all the output from the first Reduce Function (the candidate itemsets) and a portion of the input data file. Each Map task counts the number of occurrences of each of the candidate itemsets among the baskets in the portion of the dataset that it was assigned. The output is a set of key-value pairs (C, v), where C is one of the candidate sets and v is the support for that itemset among the baskets that were input to this Map task.

Second Reduce Function

The Reduce tasks take the itemsets they are given as keys and sum the associated values for each itemsets. The result is the total support of the itemsets that the Reduce task was assigned to handle. Those itemsets whose sum of values is at least s are frequent in the whole dataset, so the Reduce task outputs these itemsets with their counts. Itemsets that do not have total support at least s are not transmitted to the output of the Reduce task.