Introduction to Data Mining

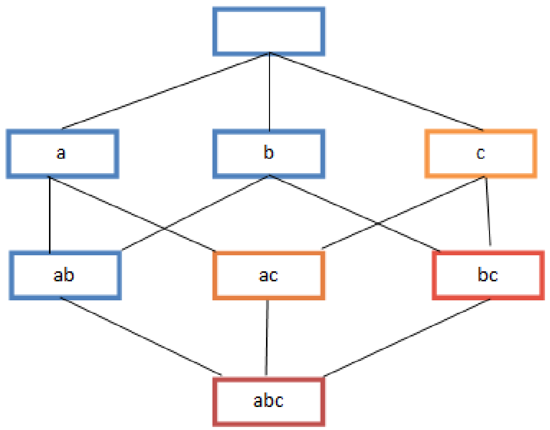
Lecture #13: Frequent Itemsets-2

U Kang
Seoul National University

Outline

- **A-Priori Algorithm**
- PCY Algorithm
- Frequent Itemsets in $\leq$ 2 Passes

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***
  - E.g., if {A,C} is frequent, then {A} is frequent (so does {C})
- **Contrapositive for pairs:**
  If item ***i*** does not appear in ***s*** baskets, then no pair including ***i*** can appear in ***s*** baskets
  - E.g., if {A} is not frequent, then {A,C} is not frequent
- **So, how does A-Priori find freq. pairs?**

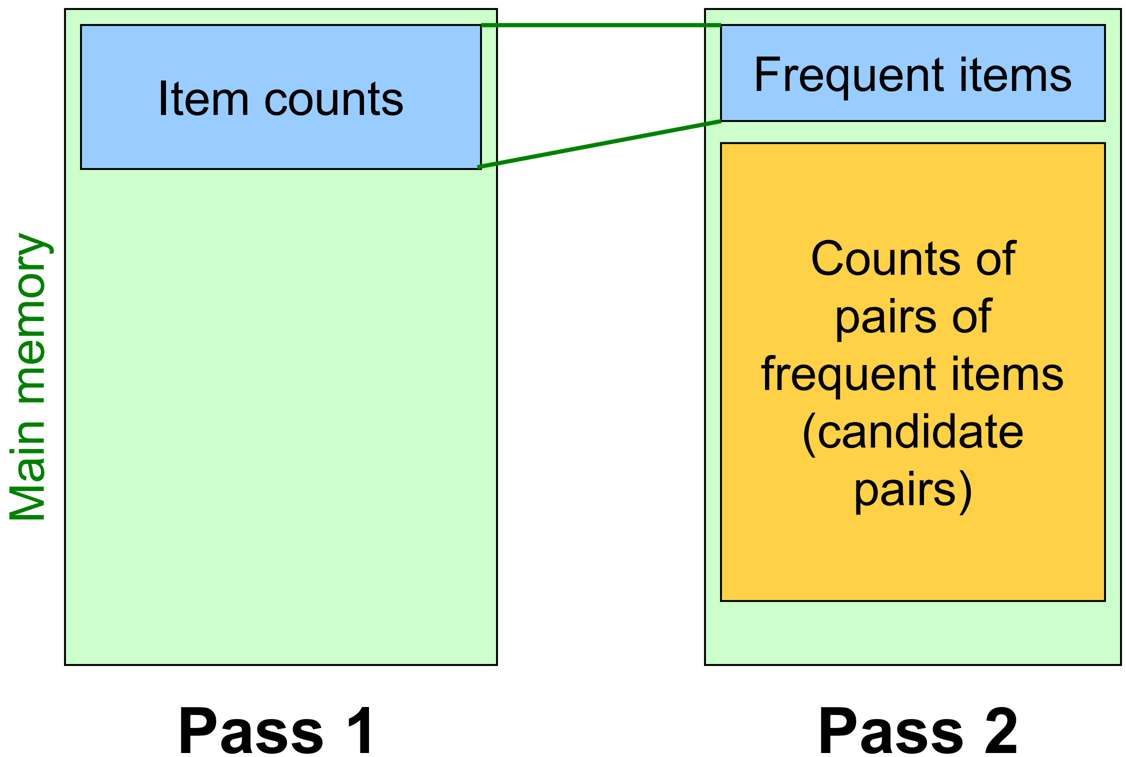
A-Priori Algorithm – (2)

- **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**

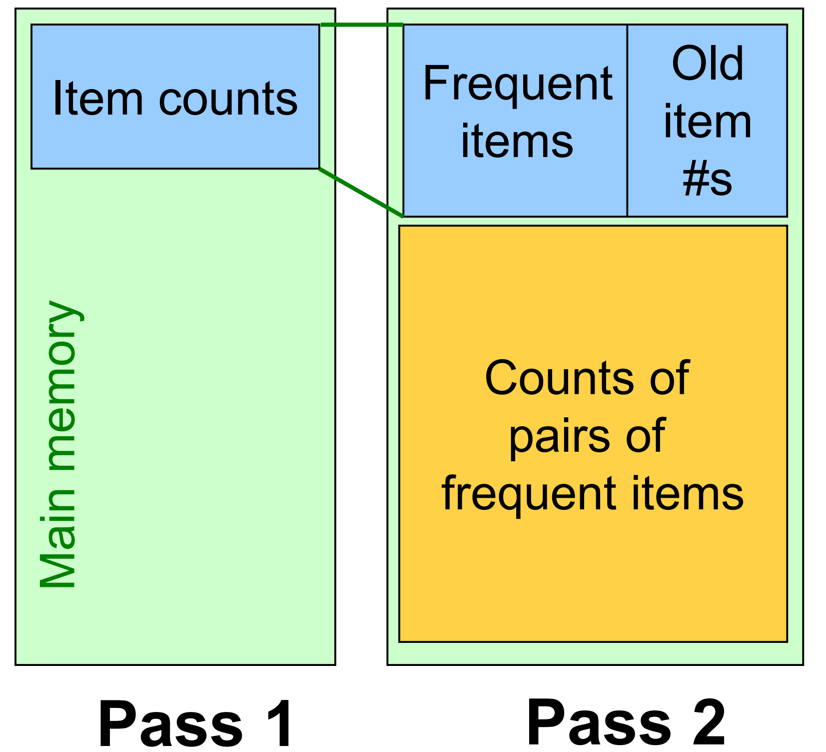
- **Pass 2:** Read baskets again and count in main memory only those pairs where both elements are frequent (from Pass 1)
  - Requires memory proportional to square of **frequent** items only (for counts)
  - Plus a list of the frequent items (so you know what must be counted)

Main-Memory: Picture of A-Priori



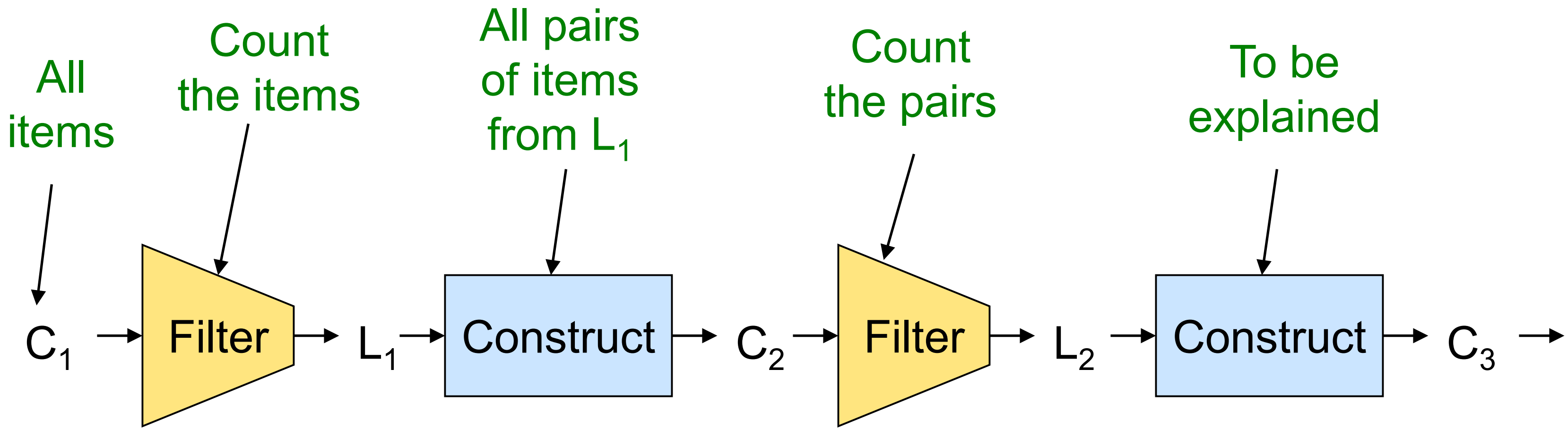
Detail for A-Priori

- You can use the triangular matrix method with ***n*** = number of frequent items
  - Why?
  - => 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,c} {b,m} {c,j} {c,m} }
  - 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}.
But 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

Generating C_3 From L_2

- Assume {x1, x2, x3} is frequent.
- Then, {x1,x2}, {x1, x3}, {x2, x3} are frequent, too.
- => if any of {x1,x2}, {x1, x3}, {x2, x3} is NOT frequent, then {x1, x2, x3} is NOT frequent!

- So, to generate C_3 from L_2,
  - Find two frequent pairs in the form of {a, b}, and {a, c}
    - This can be done efficiently if we sort L_2
  - Check whether {b,c} is also frequent
  - If yes, include {a,b,c} to C_3

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 minimum support (e.g., 1%), ***k* = 2** requires the most memory

- **Many possible extensions:**
  - Association rules with intervals:
    - For example: Men over 60 have 2 cars
  - Association rules when items are in a taxonomy
    - Bread, Butter → FruitJam
    - BakedGoods, MilkProduct → PreservedGoods
  - Lower the min. support ***s*** as itemset gets bigger

Outline

- ☑ A-Priori Algorithm
- ☐ **PCY Algorithm**
- ☐ Frequent Itemsets in $\leq$ 2 Passes

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

```
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

- **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

Example

- **Assume min. support = 10**
  - Sup(1,2) = 10
  - Sup(3,5) = 10
  - Sup(2,3) = 5
  - Sup(1,5) = 4
  - Sup(1,6) = 7
  - Sup(4,5) = 8

{1,2} {3,5}	Total count: 20
{2,3} {1,5}	Total count: 9
{1,6} {4,5}	Total count: 15

Note that {2,3}, and {1,5} cannot be frequent itemsets. (Why?)

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)
    - E.g., even though {A}, {B} are frequent, count of the bucket containing {A,B} might be < s

- **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**

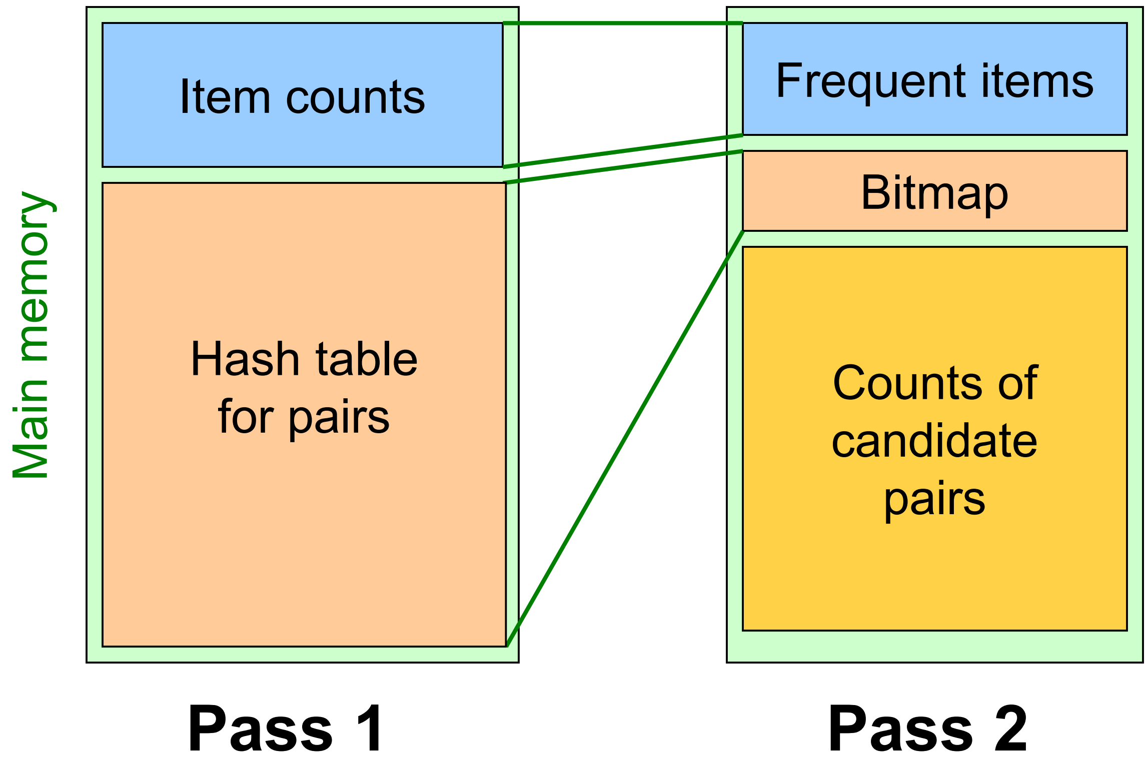
- Also, decide which items are frequent and list them for the second pass

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



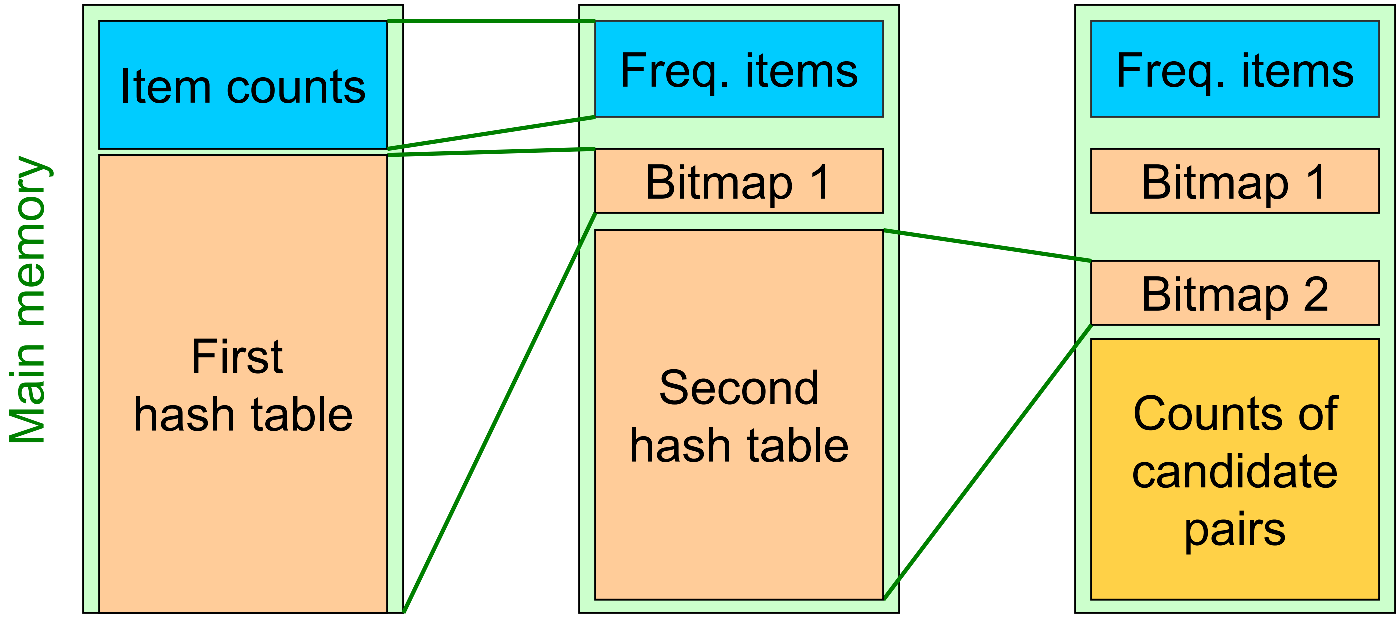
Main-Memory Details

- **Buckets require a few bytes each:**
  - **Note:** we do not have to count past s
    - If s < 256, then we need at most 1 byte for a bucket
  - #buckets is *O(main-memory size)*
    - Large number of buckets helps. (How?)
    - => decreases false positive

Refinement: Multistage Algorithm

- **Limit the number of candidates to be counted**
  - **Remember:** Memory is the bottleneck
  - 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, fewer pairs contribute to buckets, so fewer ***false positives***
- **Requires 3 passes over the data**

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

질문: B1에 있는 것을 뽑아서 다시 Hashing하는 것이고 그 결과가 B2인데 B1을 가지고 있을 필요가 있느냐?
답변: False positive를 막기 위해서? 같은 곳에 들어갔을 때?

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**

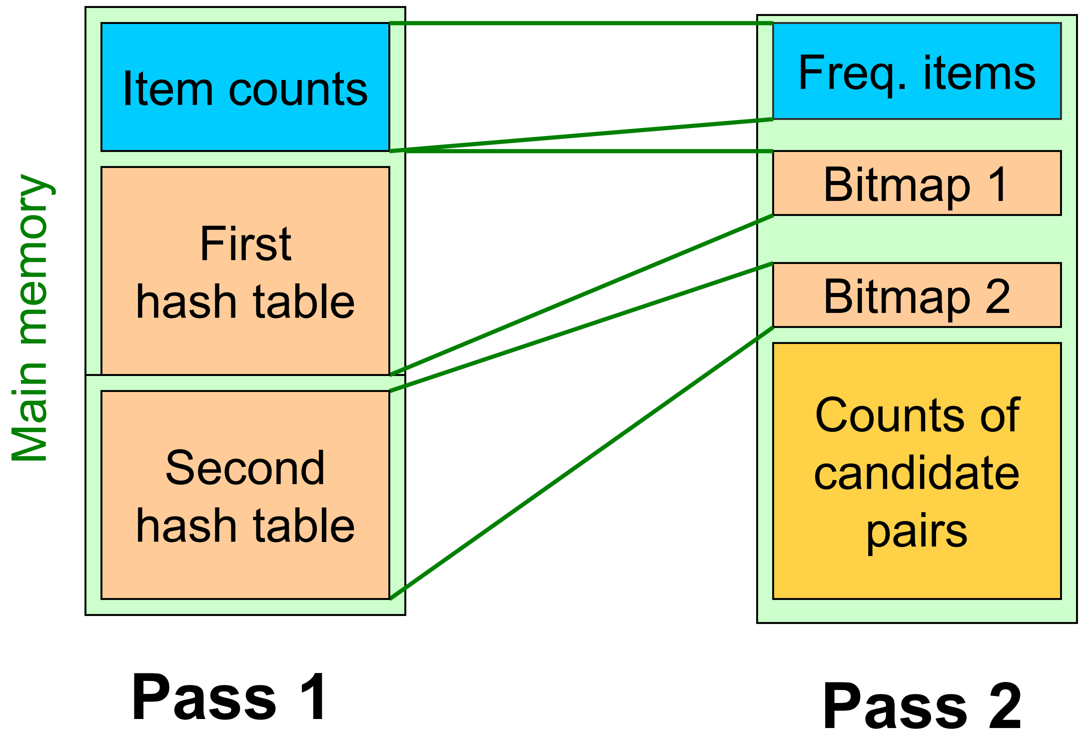
Important Points

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

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
  - If we spend too much space for bit-vectors, then we run out of space for candidate pairs

- For multihash, the bit-vectors occupy exactly what one PCY bitmap does, but too many hash functions make all counts $\geq$ ***s***

Outline

- ☑ A-Priori Algorithm
- ☑ PCY Algorithm
- ☐ **Frequent Itemsets in $\leq$ 2 Passes**

Frequent Itemsets in $\leq$ 2 Passes

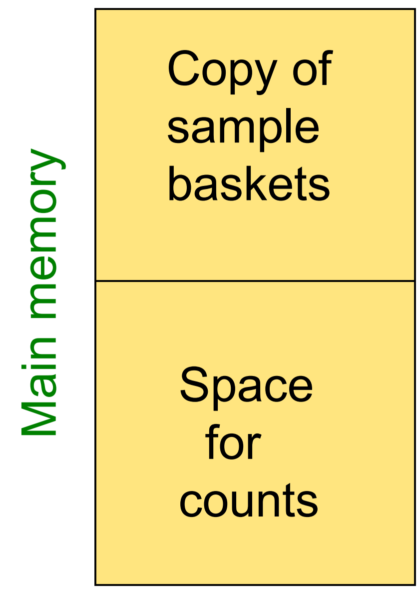
- A-Priori, PCY, etc., take *k* passes to find frequent itemsets of size *k*
- **Can we use fewer passes?**
- Method that uses 2 or fewer passes for all sizes:
  - Random sampling
  - SON (Savasere, Omiecinski, and Navathe)
  - Toivonen (see textbook)

Random Sampling (1)

- Take a random sample of the market baskets

Data양이 너무 많으니까 sampling해서 memory위에 올리고 결과를 찾도록 하자.

- 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 min. support 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 (cannot avoid false negatives)
  - Smaller min. support, e.g., *s*/125, helps catch more truly frequent itemsets
    - But requires more space

      False positive또한 증가할 것이다.

SON Algorithm – (1)

- Repeatedly read small subsets of the baskets into main memory and run an in-memory algorithm to find all frequent itemsets
  - We are not sampling, but processing the entire file in memory-sized chunks
  - Min. support decreases to (s/k) for k 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.
  - Task: find frequent (>= s) itemsets among n baskets
  - n baskets divided into k subsets
  - Load (n/k) baskets in memory, look for frequent (>= s/k) pairs

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

SON: Map/Reduce

- **Phase 1:** Find candidate itemsets
  - Map? each machine finds frequent itemsets for the subset of baskets assigned to it
  - Reduce? Collect and output candidate frequent itemsets (remove duplicates)

- **Phase 2:** Find true frequent itemsets
  - Map? Output (candidate_itemset, count) for the subset of baskets assigned to it
  - Reduce? Sum up the count, and output truly frequent (>= s) itemsets

Summary

- Frequent Itemsets
  - One of the most ‘classical’ and important data mining task
- Association Rules: {A} -> {B}
  - Confidence, Support, Interestingness
- Algorithms for Finding Frequent Itemsets
  - A-Priori
  - PCY
  - <= 2-Pass algorithm: Random Sampling, SON

Questions?