Comparing the two approaches

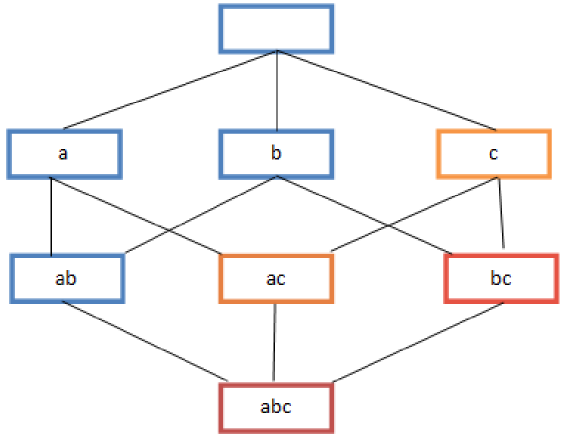
- **Approach 1: Triangular Matrix**
  - **n** = total number items
  - Count pair
  - Keep pair
    - {1,2}, {1,3
  - Pair {*i*, *j*} is
  - Total num
  - **Triangular**
- **Approach** *but only for pairs with count > 0)*
  - Beats Approach 1 if less than **1/3** of possible pairs actually occur

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

Can we do better?

A-Priori Algorithm

A-Priori Algorithm – (1)

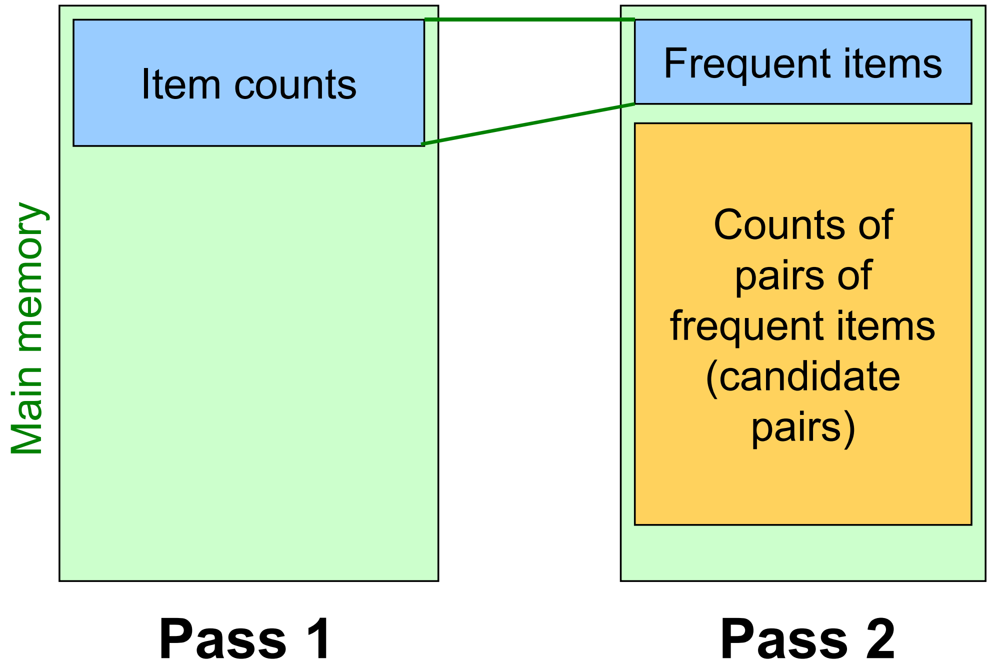


- A **two-pass** approach called ***A-Priori*** limits the need for main memory
- **Key idea: *monotonicity (单调性)***
  - If a set of items I appears at least s times, so does every **subset** J of I
- **Contrapositive for pairs:**
  If item i does not appear in s baskets, then no pair including i can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

A-Priori Algorithm – (2)

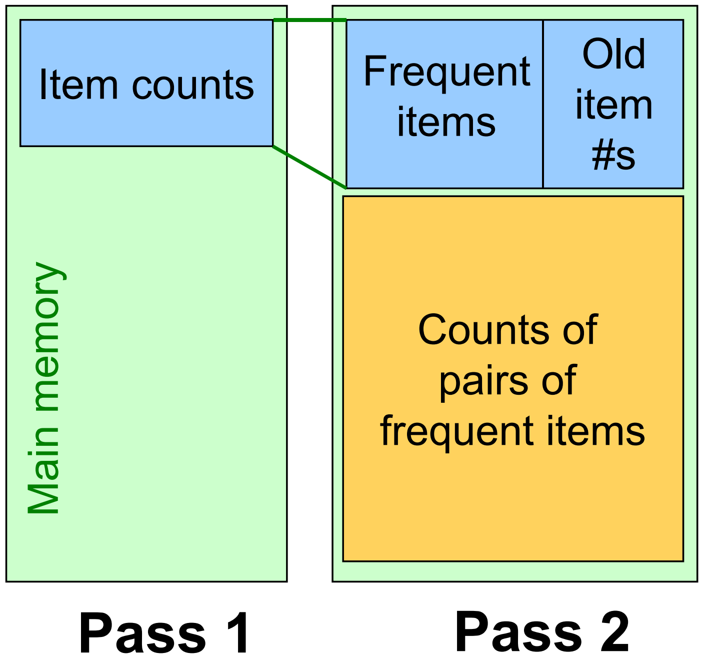
- **Pass 1:** Read baskets and count in main memory the occurrences of each **individual item**
  - Requires only memory proportional to #items
- **Items that appear $\geq s$ times are the frequent items**
- **Pass 2:** Read baskets again and count in main memory only those pairs where both elements 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



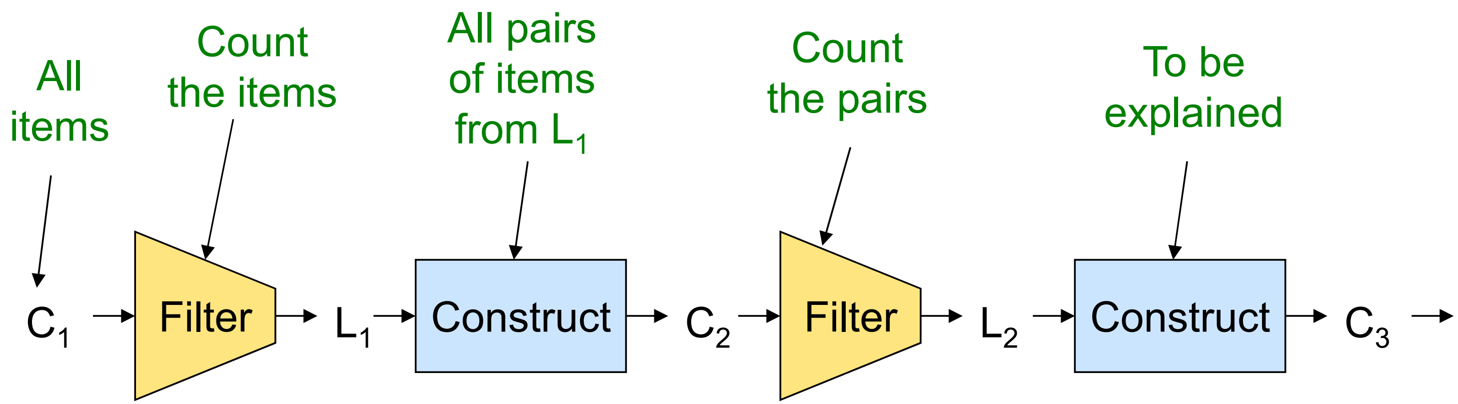
Detail for A-Priori

- 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 ≥ **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,m}, {c,j} }
  - Generate C_3 = {{b,c,j}, {b,c,m}, {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
- What if A-Priori runs out of memory for ***k* = 2**?

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
    - **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

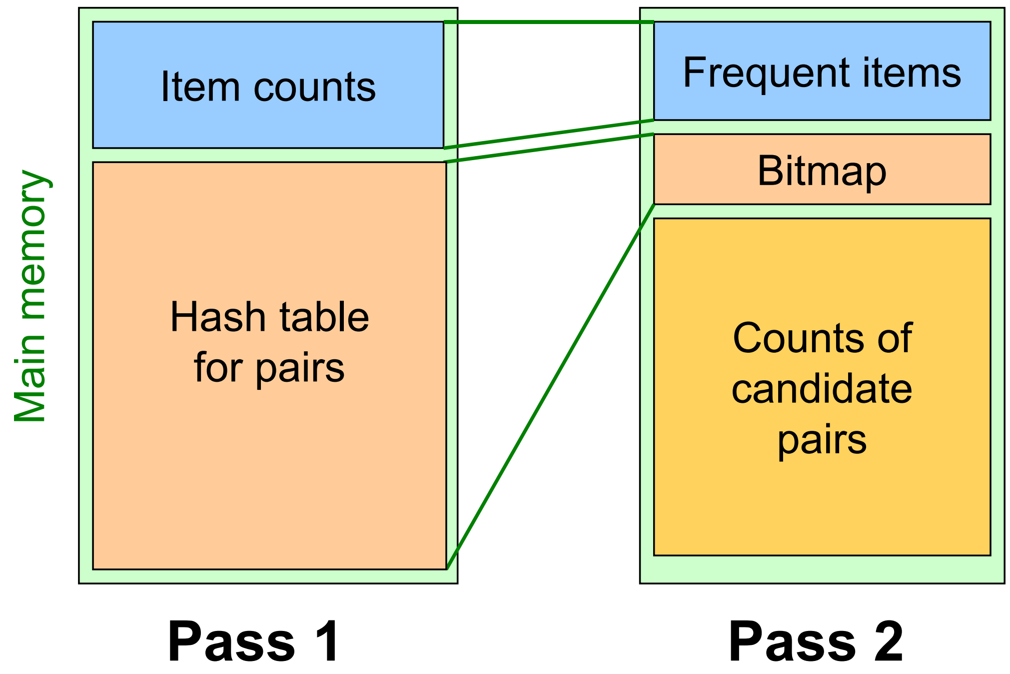
- **Replace the buckets by a bit-vector:**
  - **1** means the bucket count exceeded the support s (call it a **frequent bucket**); **0** means it did not
- **4-byte integer counts are replaced by bits, so the bit-vector requires 1/32 of memory**
- Also, decide which items are frequent and list them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



Main-Memory Details

- **Buckets require a few bytes each:**
  - **Note:** we do not have to count past s
  - #buckets is *O(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 & Multihash

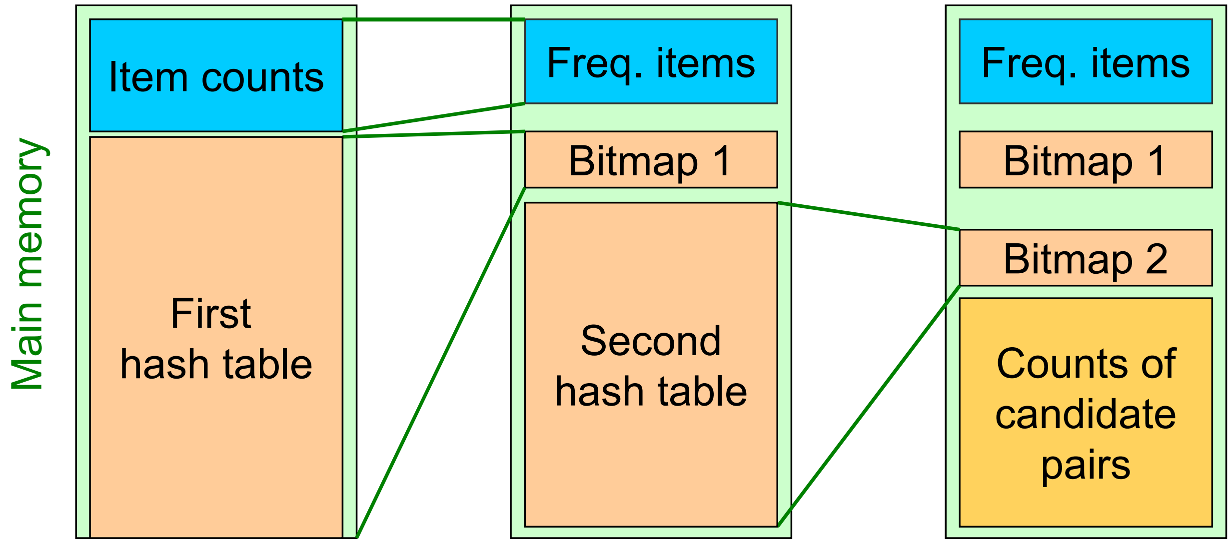
Refinement: Multistage Algorithm

- **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
  - 1) ***i*** and ***j*** are frequent, and
  - 2) ***{i, j}*** hashes to a frequent bucket from **Pass 1**
- On middle pass, fewer pairs contribute to buckets, so fewer ***false positives***
- **Drawback: Requires 3 passes over the data**

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: Multistage



Pass 1

Count items
Hash pairs {i,j}

Pass 2

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

Pass 3

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

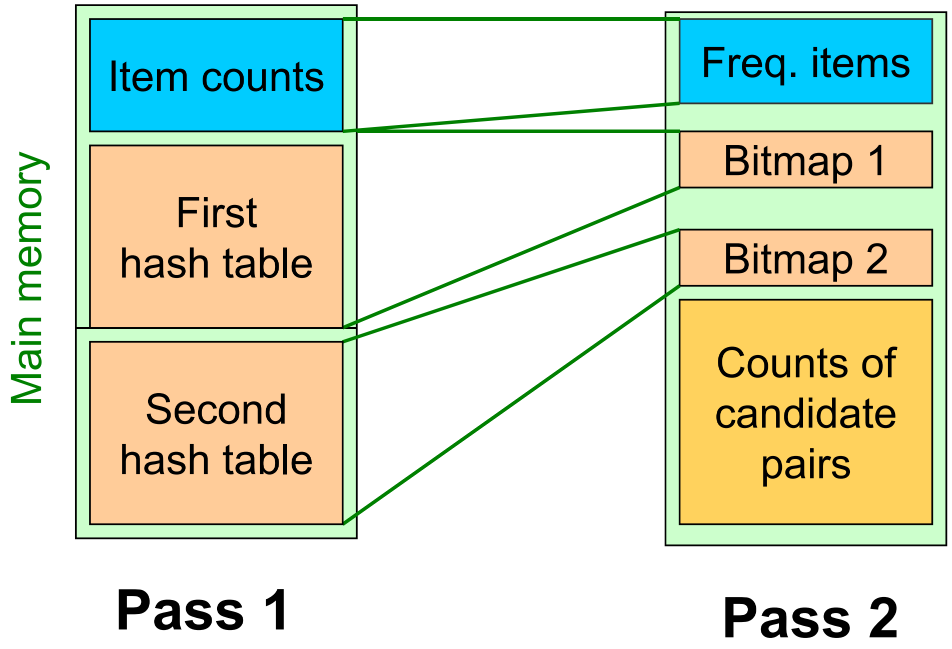
Important Points

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

Frequent Itemsets in $\leq$ 2 Passes:
Random sampling&SON&Toivonen(托伊沃宁算法)

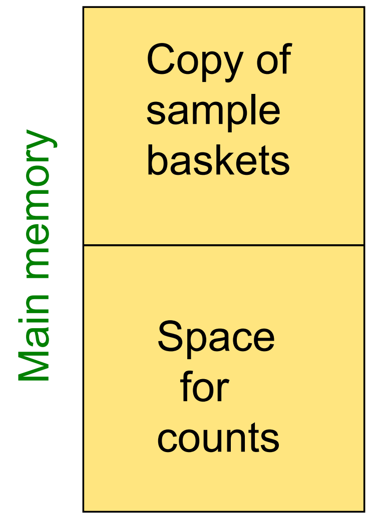
Frequent Itemsets in $\leq$ 2 Passes

- 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
  - SON (Savasere, Omiecinski, and Navathe)
  - Toivonen (托伊沃宁算法, see textbook)

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)

- 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 "monotonicity" 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 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?
  - Reduce?

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

第一次作业

- Here is a collection of twelve baskets. Each contains three of the six items 1 through 6. {1, 2, 3} {2, 3, 4} {3, 4, 5} {4, 5, 6}{1, 3, 5} {2, 4, 6} {1, 3, 4} {2, 4, 5}{3, 5, 6} {1, 2, 4} {2, 3, 5} {3, 4, 6} Suppose the support threshold is 4. On the first pass of the PCY Algorithm we use a hash table with 11 buckets, and the set {i, j} is hashed to bucket i × j mod 11.
  - (a) By any method, compute the support for each item and each pair of items.
  - (b) Which pairs hash to which buckets?
  - (c) Which buckets are frequent?
  - (d) Which pairs are counted on the second pass of the PCY Algorithm?

备注：在学习通《大数据分析》(邀请码84967919) 中作业里提交答案。最晚提交时间: 2022.12.31下午6点