A-Priori for All Frequent Itemsets

- One pass for each k
- Needs room in main memory to count each candidate k–set
- For typical market-basket data and reasonable support (e.g., 1%), $k = 2$ requires the most memory.

PCY 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 will fit in memory

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
    }
}
```

Observations About Buckets

1. For a bucket with total count less than *s*, none of its pairs can be frequent
2. A bucket that a frequent pair hashes to is surely frequent
3. Even without any frequent pair, a bucket can be frequent

We can surely eliminate all pairs that hash into buckets of Type (1)

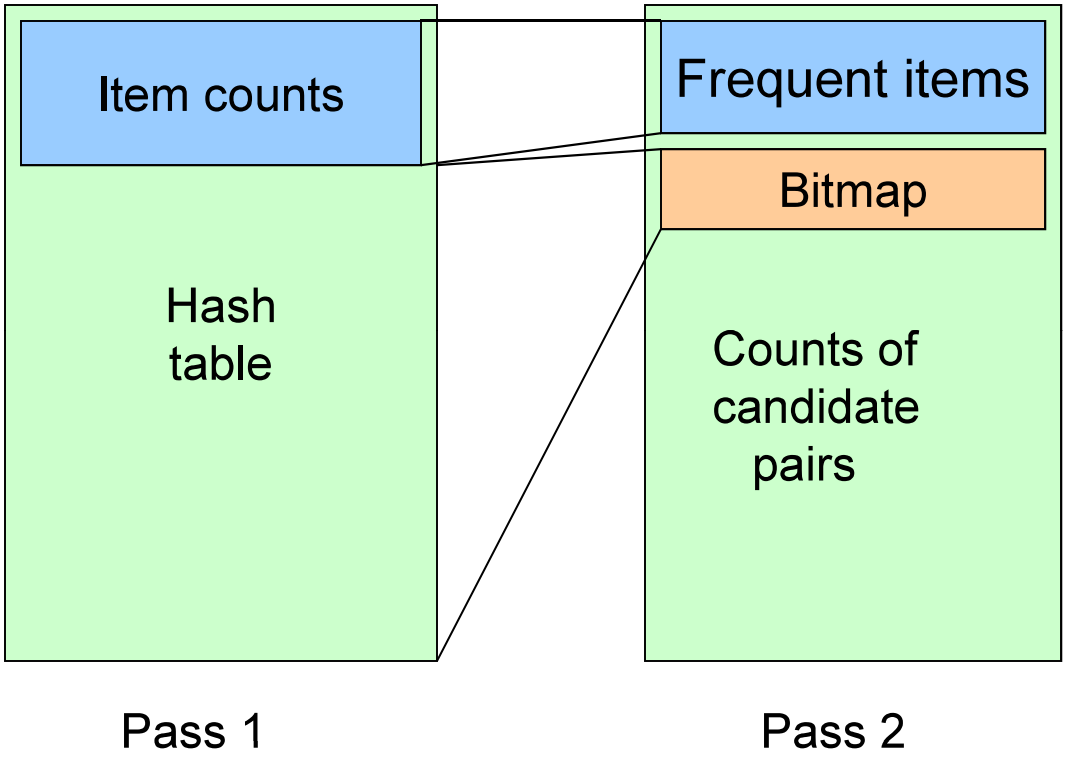
PCY Algorithm – Between Passes

- Replace the buckets by a bit-vector:
  - 1 means the bucket is frequent; 0 means it is not.
- 4-byte integers are replaced by bits, so the bit-vector requires 1/32 of memory.

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 number whose bit in the bit vector is 1.
- Notice all these conditions are necessary for the pair to have a chance of being frequent.

Picture of PCY



Memory Details

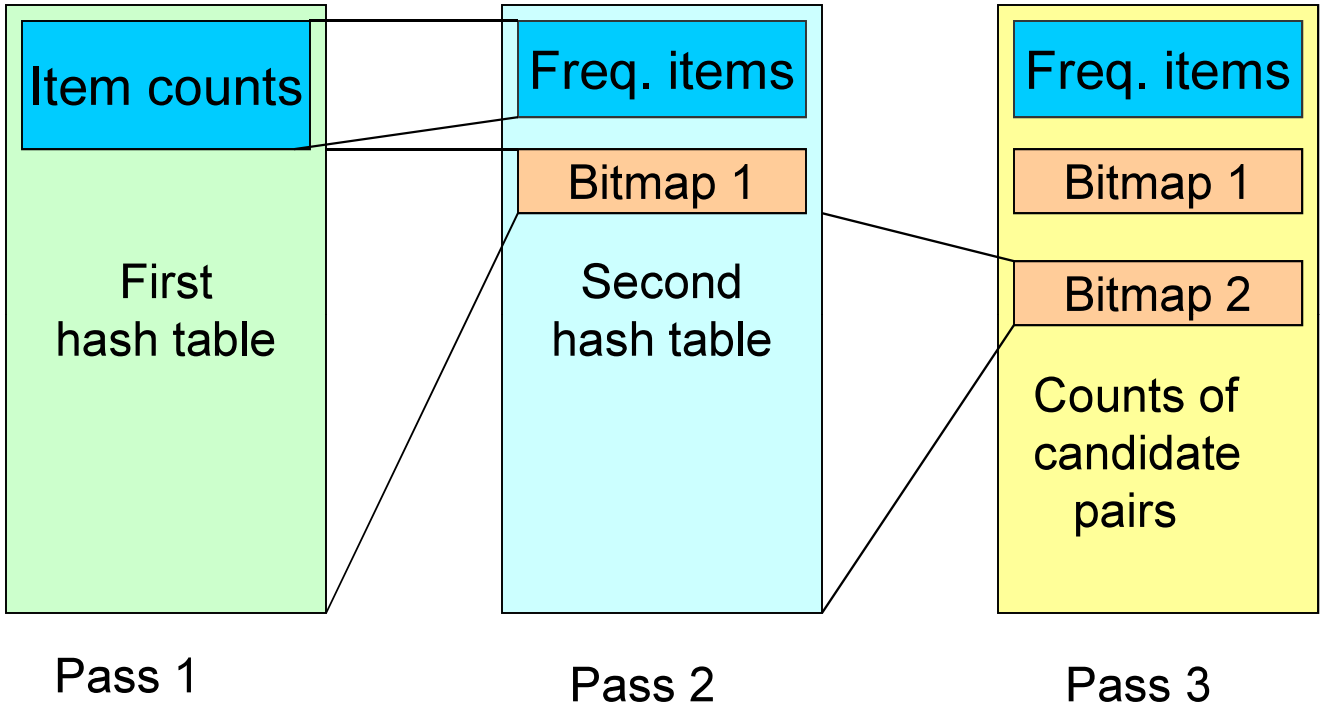
- Buckets require a few bytes each.
  - Note: we don't have to count past *s*.
  - # buckets is O(main-memory size)

- On second pass, a table of (item, item, count) triples is essential (why?)
  - Hash table must eliminate approx. 2/3 of the candidate pairs for PCY to beat a-priori.

Multistage Algorithm

- Key idea: After Pass 1 of PCY, rehash only those pairs that qualify for Pass 2 of PCY
- On middle pass, fewer pairs contribute to buckets, so fewer *false positives*–frequent buckets with no frequent pair.

Multistage Picture



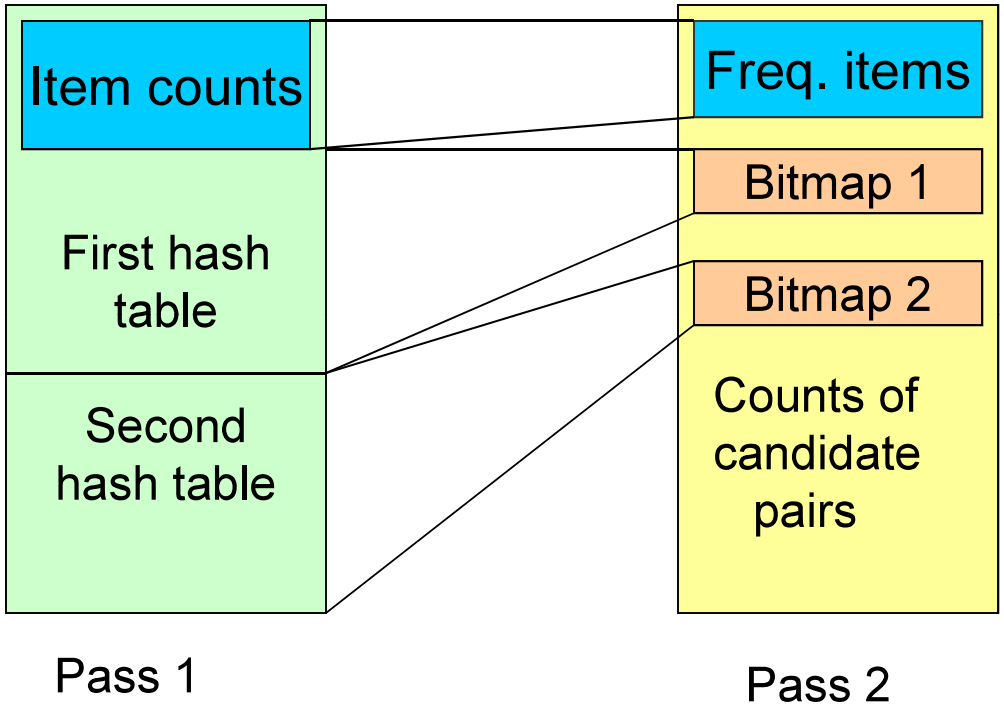
Multistage – Pass 3

- 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.

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.

Multihash Picture



Frequent Itemsets In ≤ 2 Passes

- A-Priori, PCY, etc., take *k* passes to find frequent itemsets of size *k*.
- Other techniques use 2 or fewer passes for all sizes, but may miss some frequent itemsets
  - Random sampling
  - SON (Savasere, Omiecinski, and Navathe)
  - Toivonen (see textbook)

Random Sampling (1)

- Take a random sample of the market baskets
- Run a-priori or one of its improvementsin 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 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)

- May miss some frequent itemsets
  - Smaller threshold helps catch more truly frequent itemsets.

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 Algorithm – 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.