

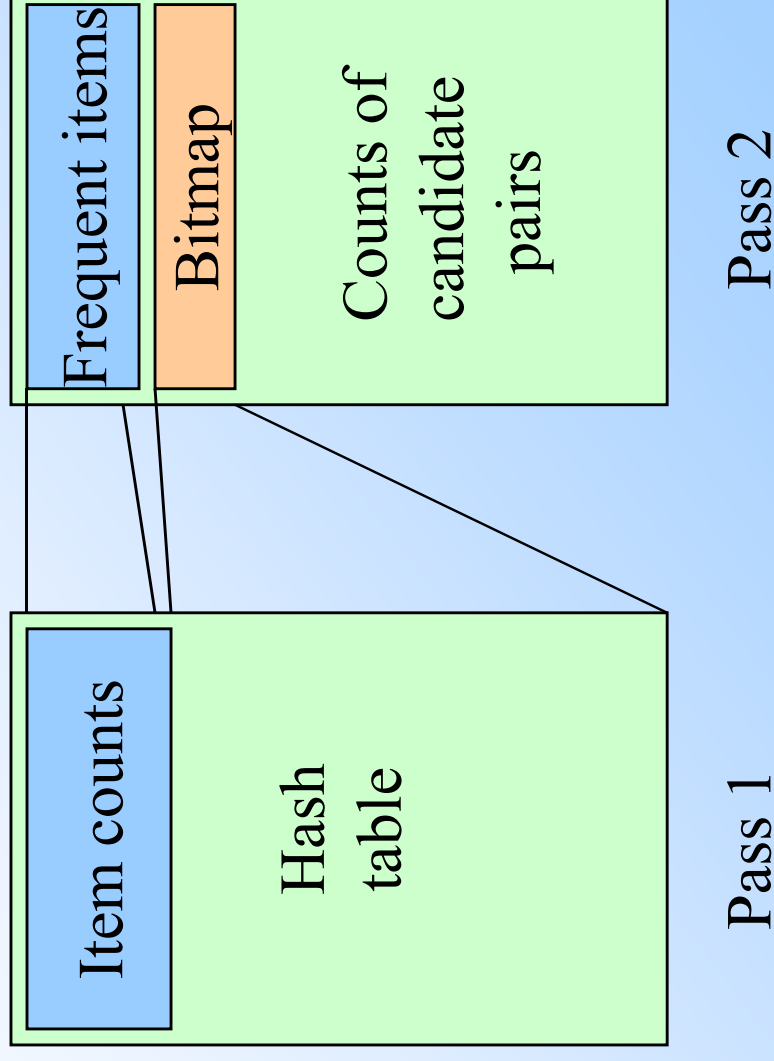
Hash-Based Improvements to A-Priori

Park-Chen-Yu Algorithm
Multistage Algorithm
Approximate Algorithms

PCY Algorithm

- ◆ Hash-based improvement to A-Priori.
- ◆ 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.
- ◆ Gives extra condition that candidate pairs must satisfy on Pass 2.

Picture of PCY



PCY Algorithm --- Before Pass 1

- ◆ Organize main memory:
 - ◆ Space to count each item.
 - One (typically) 4-byte integer per item.
 - ◆ Use the rest of the space for as many integers, representing buckets, as we can.

PCY Algorithm --- Pass 1

```
FOR (each basket) {  
  FOR (each item)  
    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  
  }  
}
```

PCY Algorithm --- Between Passes

- ◆ Replace the buckets by a bit-vector:
 - ◆ 1 means the bucket count \geq the support s (**frequent bucket**); 0 means it did not.
- ◆ Integers are replaced by bits, so the bit vector requires little second-pass space.
- ◆ 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:
 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.

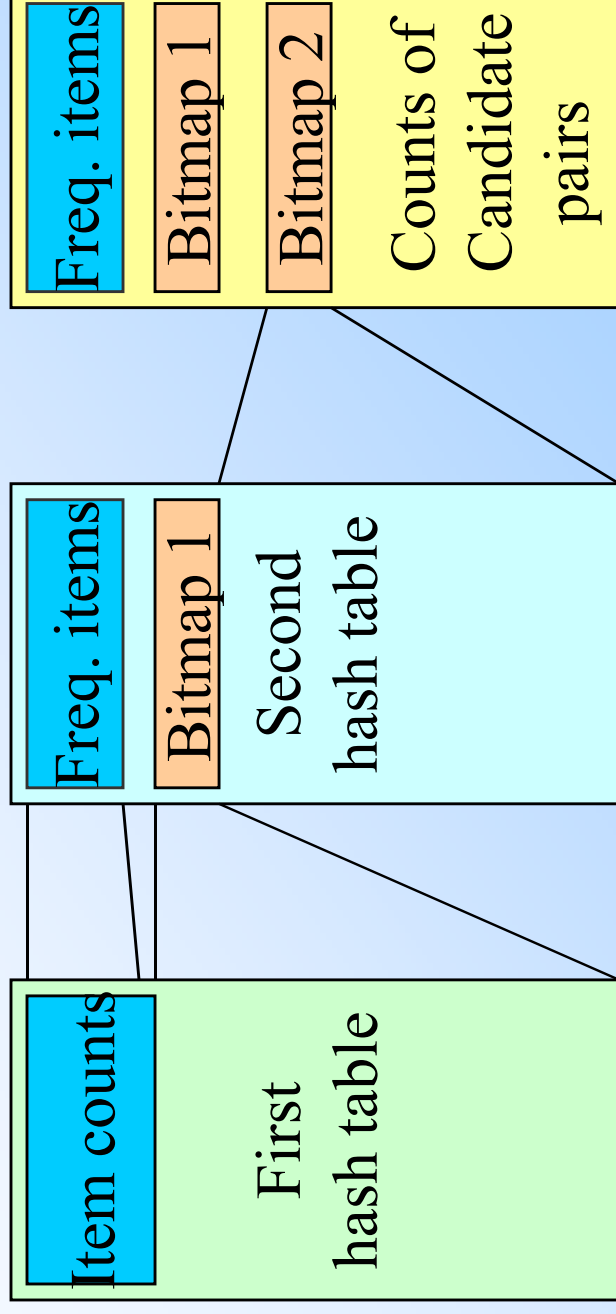
Memory Details

- ◆ Hash table requires buckets of 2-4 bytes.
 - ◆ Number of buckets thus almost $1/4$ - $1/2$ of the number of bytes of main memory.
- ◆ On second pass, a table of (item, item, count) triples is essential.
 - ◆ Thus, we need to eliminate $2/3$ of the candidate pairs 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 drops* --- frequent buckets with no frequent pair.

Multistage Picture



Multistage --- Pass 3

- ◆ Count only those pairs $\{i, j\}$ that satisfy:
 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.

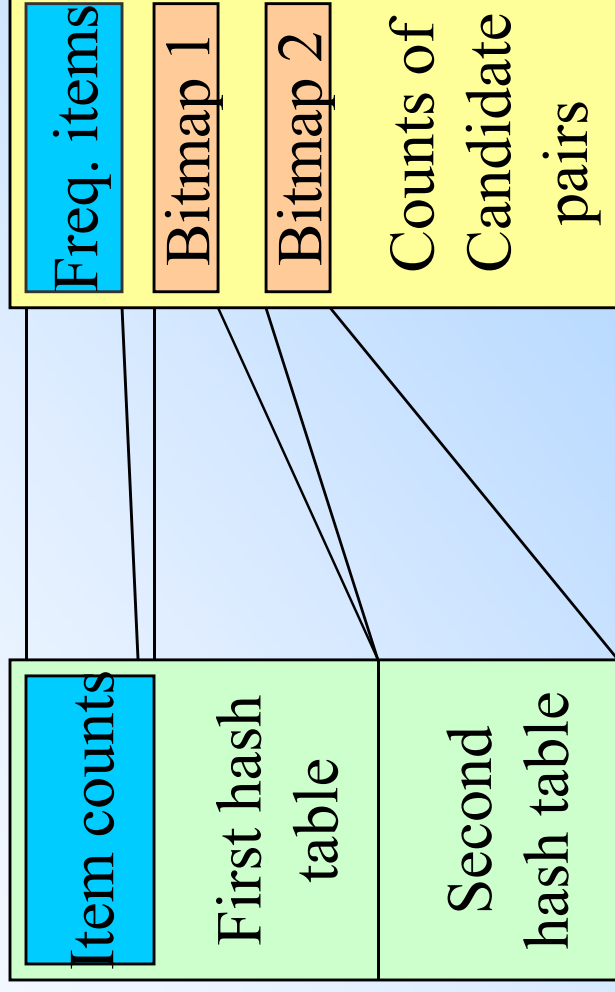
Important Points

1. The two hash functions have to be independent.
2. We need to check both hashes on the third pass.
 - ◆ If not, the pair could pass tests (1) and (3), yet it was never hashed on the second pass because it was in a low-count bucket on the first pass.

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



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 total exactly what one PCY bitmap does, but too many hash functions makes all counts $\geq s$.

All (Or Most) Frequent Itemsets In ≤ 2 Passes

- ◆ Simple algorithm.
- ◆ SON (Savasere, Omiecinski, and Navathe).
- ◆ Toivonen.

Simple Algorithm --- (1)

- ◆ Take a main-memory-sized random sample of the market baskets.
- ◆ Run a-priori or one of its improvements (for sets of all sizes, not just pairs) in main memory, so you don't pay for disk I/O each time you increase the size of itemsets.
 - ◆ Be sure you leave enough space for counts.

The Picture

Copy of sample baskets	Space for counts
------------------------------	------------------------

Simple Algorithm --- (2)

- ◆ Use as your support threshold a suitable, scaled-back number.
 - ◆ E.g., if your sample is $1/100$ of the baskets, use $s/100$ as your support threshold instead of s .
- ◆ Verify that your guesses are truly frequent in the entire data set by a second pass.
- ◆ But you don't catch sets frequent in the whole but not in the sample.
 - ◆ Smaller threshold, e.g., $s/125$, helps.

SON Algorithm --- (1)

- ◆ Repeatedly read small subsets of the baskets into main memory and perform the first pass of the simple algorithm on each subset.
- ◆ 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.

Toivonen's Algorithm --- (1)

- ◆ Start as in the simple algorithm, but lower the threshold slightly for the sample.
 - ◆ **Example:** if the sample is 1% of the baskets, use $s/125$ as the support threshold rather than $s/100$.
 - ◆ Goal is to avoid missing any itemset that is frequent in the full set of baskets.

Toivonen's Algorithm --- (2)

- ◆ Add to the itemsets that are frequent in the sample the *negative border* of these itemsets.
- ◆ An itemset is in the negative border if it is not deemed frequent in the sample, but all its immediate subsets are.

Example

- ◆ $ABCD$ is in the negative border if and only if it is not frequent, but all of ABC , BCD , ACD , and ABD are.

Toivonen's Algorithm --- (3)

- ◆ In a second pass, count all candidate frequent itemsets from the first pass, and also count the negative border.
- ◆ If no itemset from the negative border turns out to be frequent, then the candidates found to be frequent in the whole data are *exactly* the frequent itemsets.

Toivonen's Algorithm --- (4)

- ◆ What if we find something in the negative border is actually frequent?
- ◆ We must start over again!
- ◆ Try to choose the support threshold so the probability of failure is low, while the number of itemsets checked on the second pass fits in main-memory.