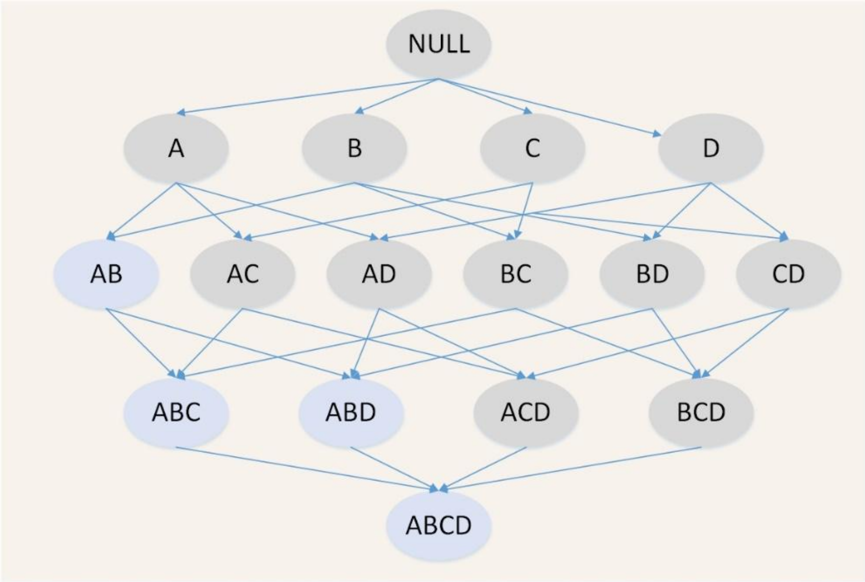
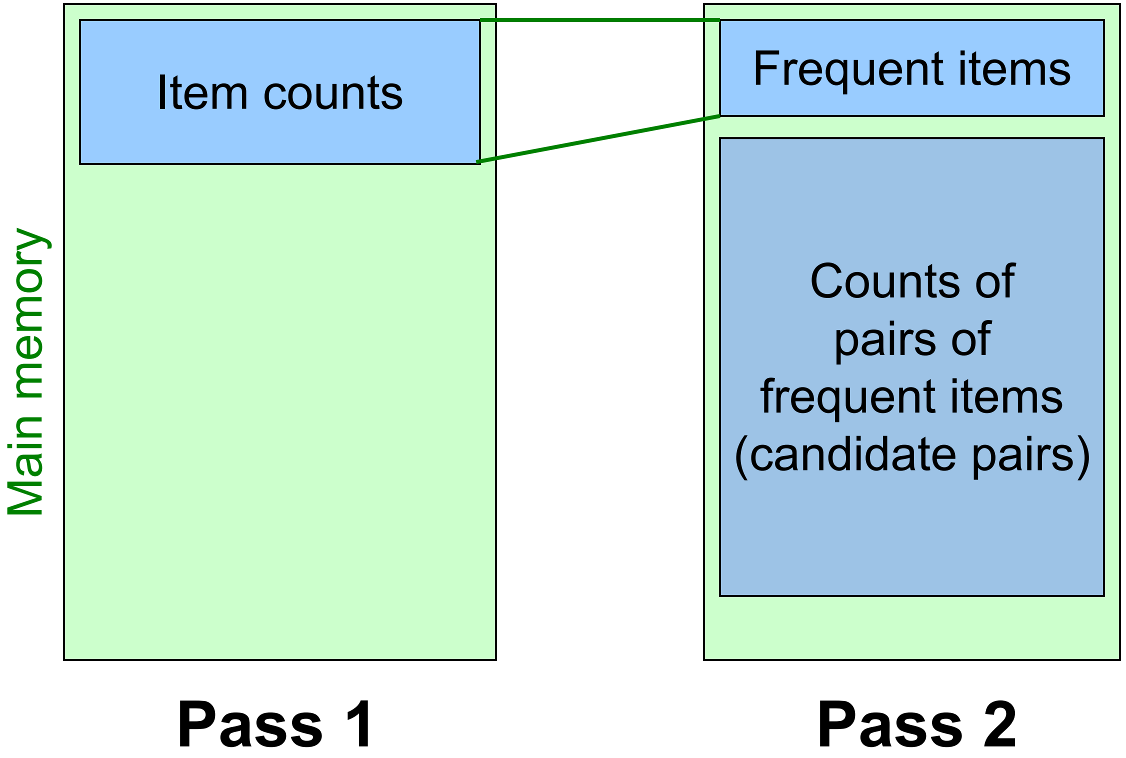


Section 3.5: PCY Algorithm

Content

3.5.1 PCY Algorithm – First Pass

- **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?**



Main-memory picture of A-Priori

3.5.1 PCY Algorithm – First Pass

- **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 (Park-Chen-Yu) Algorithm:** In addition to item counts, maintain **a hash table** with as many buckets as fit in memory (桶计数哈希表)
  - 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!

Note: Bucket≠Basket

3.5.1 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

3.5.2 PCY Algorithm – Between Passes

- **Observation: If a bucket contains a frequent pair, then the bucket is surely frequent (called frequent bucket, 频繁桶)**
- 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 (called infrequent bucket, 非频繁桶), 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

3.5.2 PCY Algorithm – Between Passes

- **Replace the buckets by a bit-vector (位图):**
  - **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

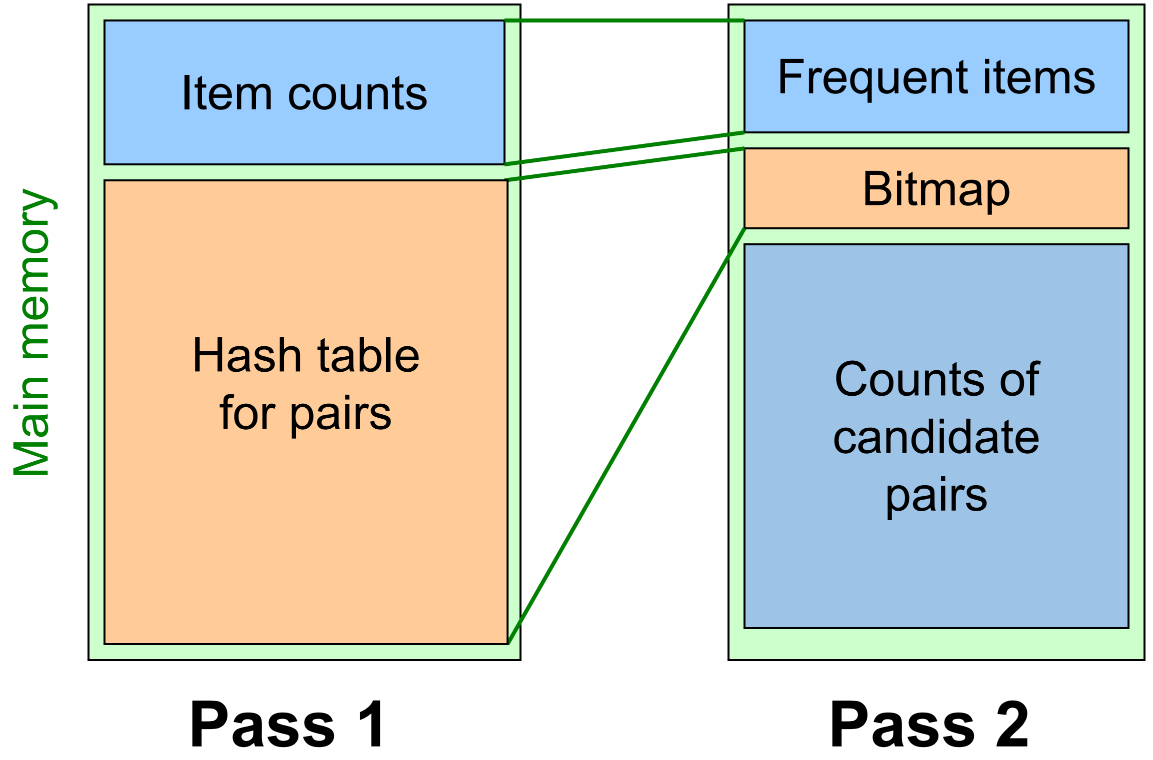
3.5.3 PCY Algorithm – Pass Two

- 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**

3.5.4 Main-Memory in PCY Algorithm

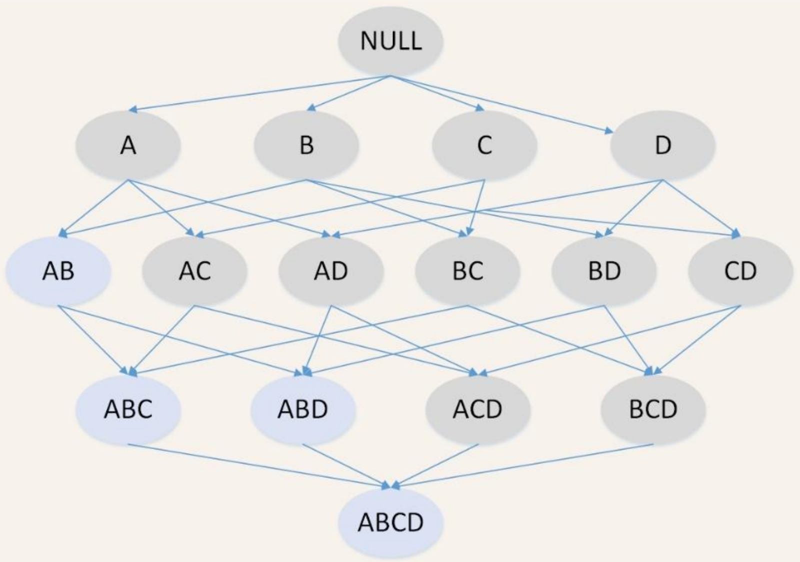
- Main-memory picture of PCY：



3.5.4 Main-Memory in PCY Algorithm

- **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 (三元组方法) 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



Section 3.6: Two Refinement Algorithms

Content

1. Multistage Algorithm
2. Multihash Algorithm
3. Summary for PCY extensions

3.6.1 Multistage Algorithm

华中科技大学
计算机科学与技术学院
School of Computer Science &Technology, HUST

- **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** for **multistage algorithm (多阶段算法):** 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* (伪阳性、伪正性、假阳性)**
- **Drawback: Requires 3 passes over the data**

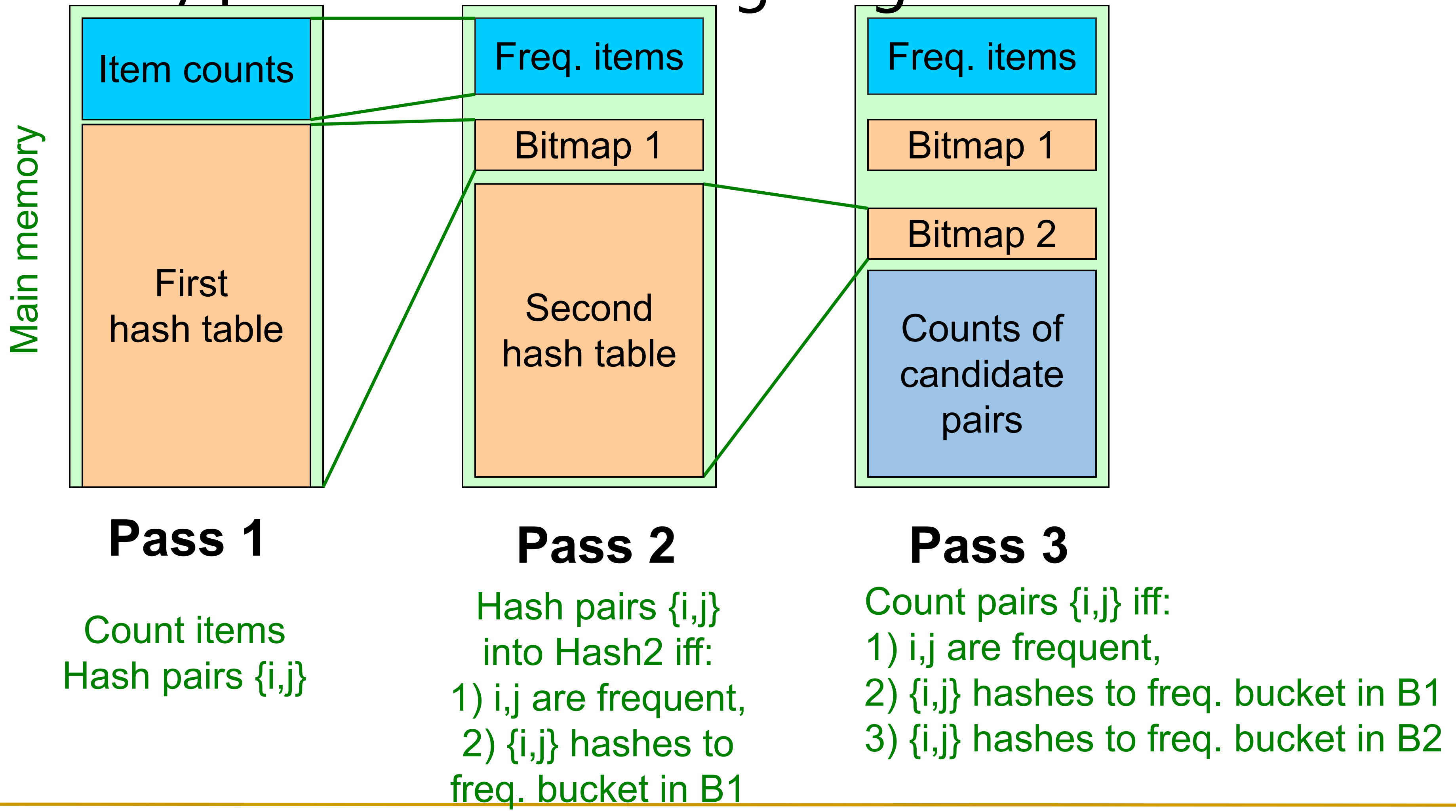
假阳性:测试结果呈阳性, 但事实上却是没有

3.6.1 Multistage Algorithm

- Pass 3 of Multistage Algorithm: 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

3.6.1 Multistage Algorithm

- Main-memory picture of multistage algorithm：



3.6.1 Multistage Algorithm

- **Important points in multistage algorithm:**

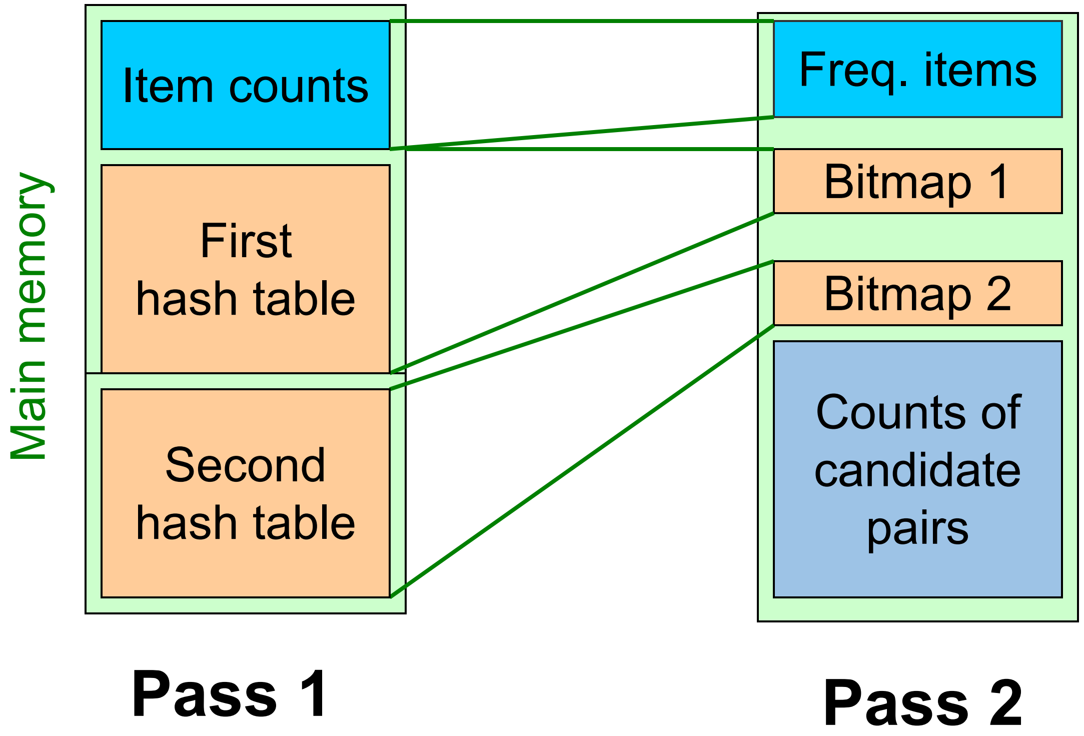
1. **The two hash functions have to be independent**
2. **We need to check both hashes on the third pass**
   - If not, we would end up counting pairs of frequent items that hashed first to an infrequent bucket but happened to hash second to a frequent bucket

3.6.2 Multihash Algorithm

- **Multihash algorithm(多哈希算法) 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

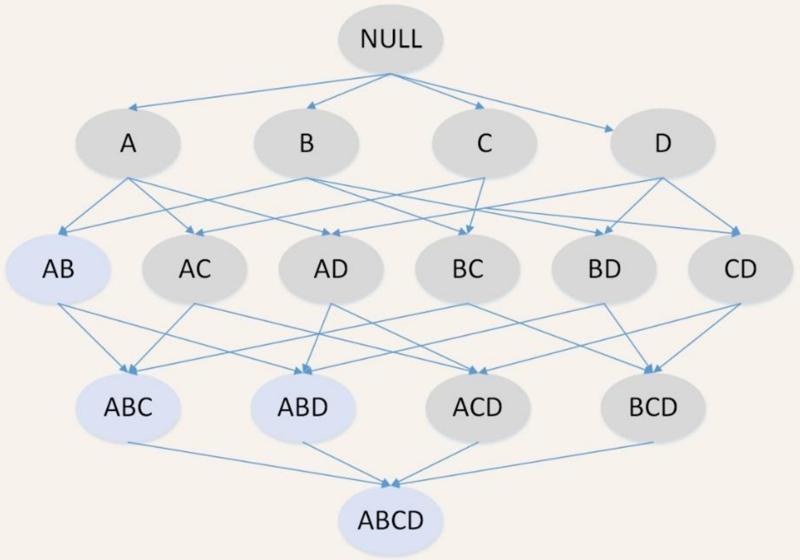
3.6.2 Multihash Algorithm

- Main-memory picture of multihash algorithm:



3.6.3 Summary for 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 > s



Section 3.7: Frequent Itemsets in <= 2 Passes

Random sampling & SON & Toivonen

Content

1. Random sampling
2. SON Algorithm
3. Toivonen Algorithm

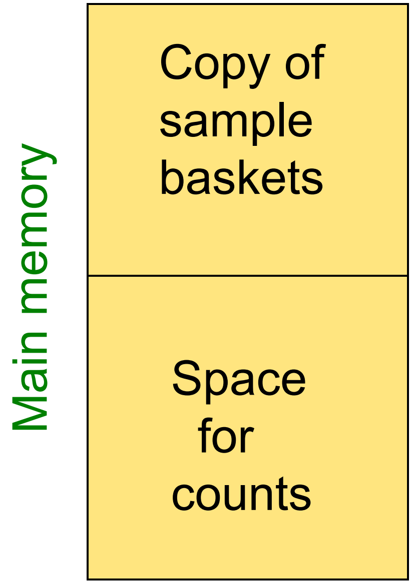
3.7 Frequent Itemsets in $\leq$ 2 Passes

- 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
  - **3.7.1: Random sampling**
  - **3.7.2: SON (Savasere, Omiecinski, and Navathe)**
  - **3.7.3: Toivonen (托伊沃宁算法)**

3.7.1 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



3.7.1 Random Sampling – (2)

- 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
- Problem for random sampling:
  - False positive(伪正例):某个项集在整个数据集上是不频繁的, 但它在抽样样本中频繁
  - False negative(伪反例):某个项集在整个数据集上是频繁的, 但它在抽样样本中不频繁
- Optionally, verify that the candidate pairs are truly frequent in the entire data set by a second pass (avoid false positives)
- But we cannot avoid false negatives

3.7.2 SON Algorithm – (1)

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology，HUST

- To avoid false negative (伪反例) and false positive(伪正例), **SON (Savasere, Omiecinski, and Navathe) algorithm** is designed, using two passes.
- **Key "monotonicity" idea:** an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset.

3.7.2 SON Algorithm – (2)

- On a <u>**first pass**</u>, **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.

- On a <u>**second pass**</u>, **SON algorithm** counts all the candidate itemsets and determines which are frequent in the entire set

3.7.2 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
- **Phase 1:** Find candidate itemsets
  - Map?
  - Reduce?
- **Phase 2:** Find true frequent itemsets
  - Map?
  - Reduce?

3.7.3 Toivonen Algorithm – (1)

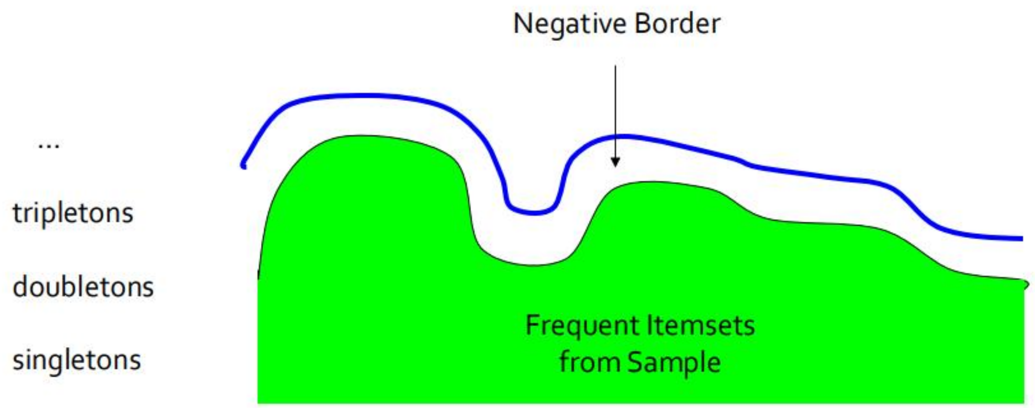
华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

- **Toivonen algorithm (托伊沃宁算法)**, using 2 passes, will give neither false negatives (伪反例) nor false positives (伪正例), but there is a small yet nonzero probability that it will fail to produce any answer at all.

- **Step 1 in Pass 1:** Toivonen algorithm starts as in the simple algorithm, and also lowers the threshold slightly for the sample data to find frequent itemsets
  - Example: if the sample is 1% of the baskets, use 0.008s as the support threshold rather than 0.01s.
  - Goal is to avoid missing any itemset that is frequent in the full set of baskets.

3.7.3 Toivonen Algorithm – (2)

- **Step 2 in Pass 1:** Then, find the **negative border (反例边界)** in the sample.
  - **反例边界:** An itemset is in the **negative border** if it is **not deemed frequent** in the sample, but all its **immediate subsets are frequent**.
- Example: ABCD is in the negative border if and only if it is not frequent, but all of ABC , BCD , ACD , and ABD are.



备注:immediate subsets (直接子集),
删除集合中的一个元素构建的集合

3.7.3 Example

- Example: Let items = {A,B,C,D,E,F} and there are frequent itemsets:{A}, {B}, {C}, {F}, {A,B}, {A,C}, {A,F}, {C,F}, {A,C,F}. Find whole negative border
- Ans:
  - {D}, {E}
  - {B,C}, {B,F}

反例边界: 在数据上满足如下性质的非频繁项集组成，即这些项集的直接子集都是频繁的

3.7.3 Toivonen Algorithm – (3)

- **Step 1 in Pass 2**: Make a pass through the entire dataset, counting all candidate frequent itemsets and the negative border (from the sample data).
  - **Case 1:** If no itemset from the negative border turns out to be frequent, then whichever candidates prove to be frequent in the whole data are exactly the frequent itemsets.
  - **Case 2:** Some itemsets from the negative border are frequent. Then how to deal with it?
  - Ans: We must start over again! We must repeat the algorithm with a new random sample.
  - Note: By choosing the support threshold for the sample wisely, we can make the probability of failure low, while still keeping the number of itemsets checked on step 3 low enough for main-memory.

Chapter 3总结

- Frequent itemsets, Association rules
- 三角矩阵存储方法 Vs. 三元组存储方法
- Algorithms for finding frequent itemsets:
  - A-Priori algorithm
  - PCY algorithm
  - Multistage algorithm (多阶段算法)
  - Multihash algorithm (多哈希算法)
  - Random sampling
  - SON algorithm
  - Toivonen algorithm (托伊沃宁算法)