Improvements to A-Priori

Bloom Filters

Park-Chen-Yu Algorithm

Multistage Algorithm

Approximate Algorithms

Compacting Results

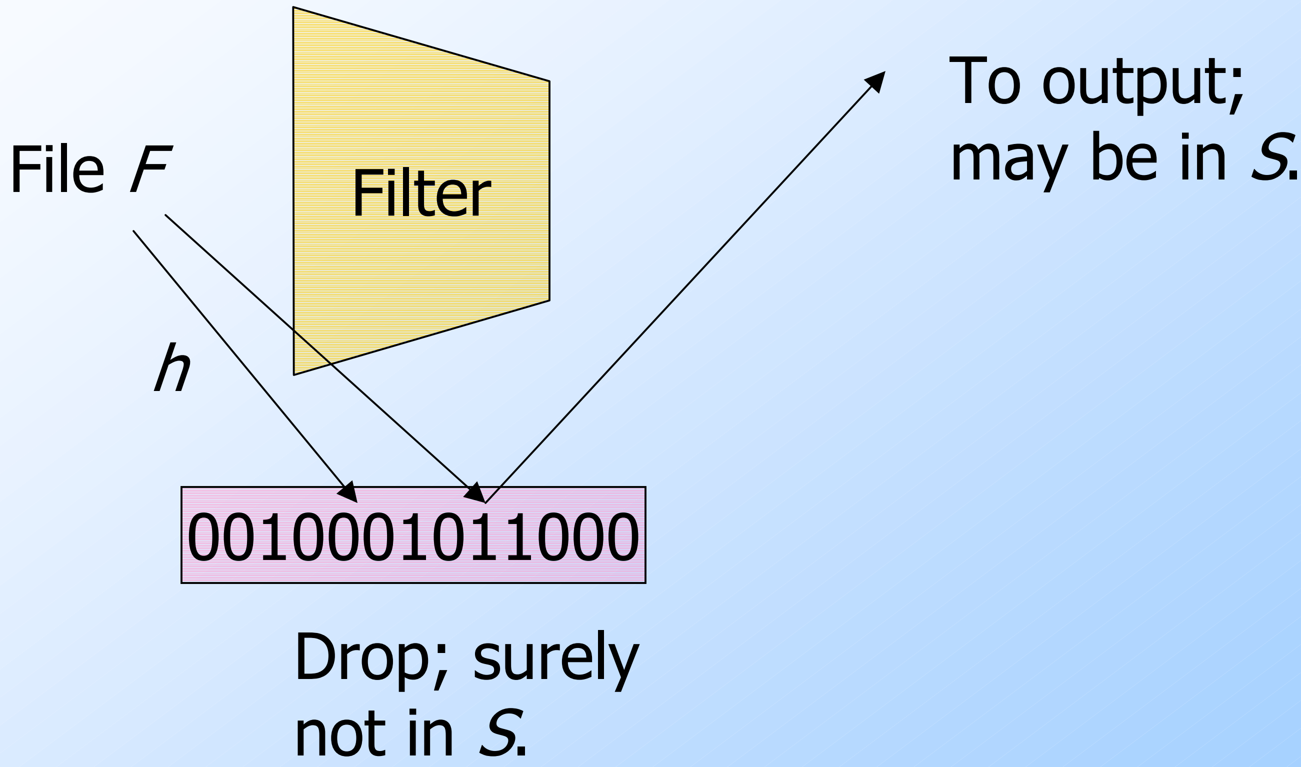
Aside: Hash-Based Filtering

- Simple problem: I have a set *S* of one billion strings of length 10.
- I want to scan a larger file *F* of strings and output those that are in *S*.
- I have 1GB of main memory.
  - So I can't afford to store *S* in memory.

Solution – (1)

- Create a bit array of 8 billion bits, initially all 0's.
- Choose a hash function *h* with range $[0, 8*10^9)$, and hash each member of *S* to one of the bits, which is then set to 1.
- Filter the file *F* by hashing each string and outputting only those that hash to a 1.

Solution – (2)



File *F*
Filter
h
0010001011000
To output;
may be in *S*.
Drop; surely
not in *S*.

Solution – (3)

- As at most 1/8 of the bit array is 1, only 1/8th of the strings not in *S* get through to the output.
- If a string is in *S*, it surely hashes to a 1, so it always gets through.
- Can repeat with another hash function and bit array to reduce the *false positives* by another factor of 8.

Solution – Summary

- Each filter step costs one pass through the remaining file *F* and reduces the fraction of false positives by a factor of 8.
  - Actually $1/(1-e^{-1/8})$.
- Repeat passes until few false positives.
- Either accept some errors, or check the remaining strings.
  - e.g., divide surviving *F* into chunks that fit in memory and make a pass though *S* for each.

Aside: Throwing Darts

- A number of times we are going to need to deal with the problem: If we throw k darts into n equally likely targets, what is the probability that a target gets at least one dart?
- Example: targets = bits, darts = hash values of elements.

Throwing Darts – (2)

Equals 1/e as $n \rightarrow \infty$

Equivalent

$$1 - (1 - 1/n)^{n(k/n)}$$

$$1 - e^{-k/n}$$

Probablity target not hit by one dart

Probability at least one dart hits target

Throwing Darts – (3)

- If $k << n$, then $e^{-k/n}$ can be approximated by the first two terms of its Taylor expansion: $1 - k/n$.
- Example: 10^9 darts, $8*10^9$ targets.
  - True value: $1 - e^{-1/8} = .1175$.
  - Approximation: $1 - (1 - 1/8) = .125$.

Improvement: Superimposed Codes (Bloom Filters)

- We could use two hash functions, and hash each member of S to two bits of the bit array.
- Now, around ¼ of the array is 1's.
- But we transmit a string in F to the output only if both its bits are 1, i.e., only 1/16th are false positives.
  - Actually $(1-e^{-1/4})^2 = 0.0493$.

Superimposed Codes – (2)

- Generalizes to any number of hash functions.
- The more hash functions, the smaller the probability of a false positive.
- Limiting Factor: Eventually, the bit vector becomes almost all 1's.
  - Almost anything hashes to only 1's.

Aside: History

- The idea is attributed to Bloom (1970).
- But I learned the same idea as "superimposed codes," at Bell Labs, which I left in 1969.
  - Technically, the original paper on superimposed codes (Kautz and Singleton, 1964) required *uniqueness* : no two small sets have the same bitmap.

PCY Algorithm – An Application of Hash-Filtering

- 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.

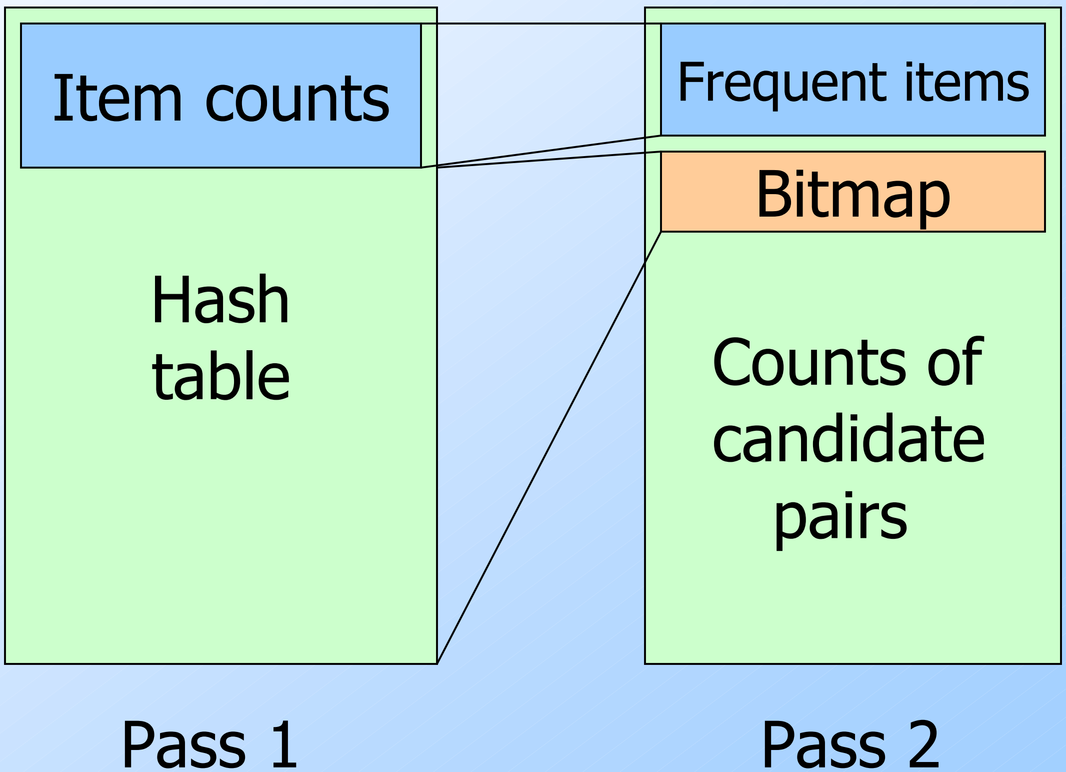
Needed Extensions to Hash-Filtering

1. Pairs of items need to be generated from the input file; they are not present in the file.
2. 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.

PCY Algorithm – (2)

- A bucket is *frequent* if its count is at least the support threshold.
- If a bucket is not frequent, no pair that hashes to that bucket could possibly be a frequent pair.
- On Pass 2, we only count pairs that hash to frequent buckets.

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 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. A bucket that a frequent pair hashes to is surely frequent.
   - We cannot use the hash table to eliminate any member of this bucket.
2. Even without any frequent pair, a bucket can be frequent.
   - Again, nothing in the bucket can be eliminated.

Observations – (2)

3. But in the best case, the count for a bucket is less than the support *s*.
   - Now, all pairs that hash to this bucket can be eliminated as candidates, even if the pair consists of two frequent items.

- Thought question: under what conditions can we be sure most buckets will be in case 3?

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.
- 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 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.

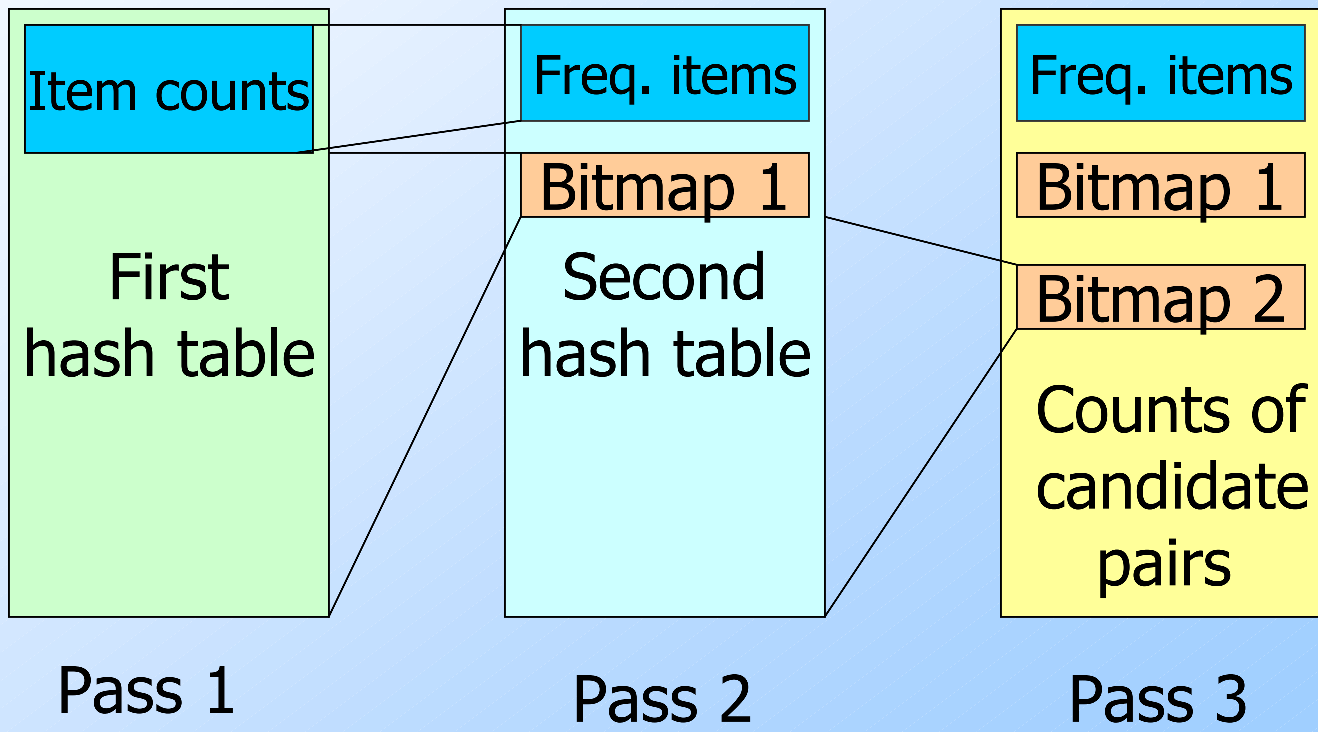
Memory Details

- 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?).
  - Thus, hash table must eliminate 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.

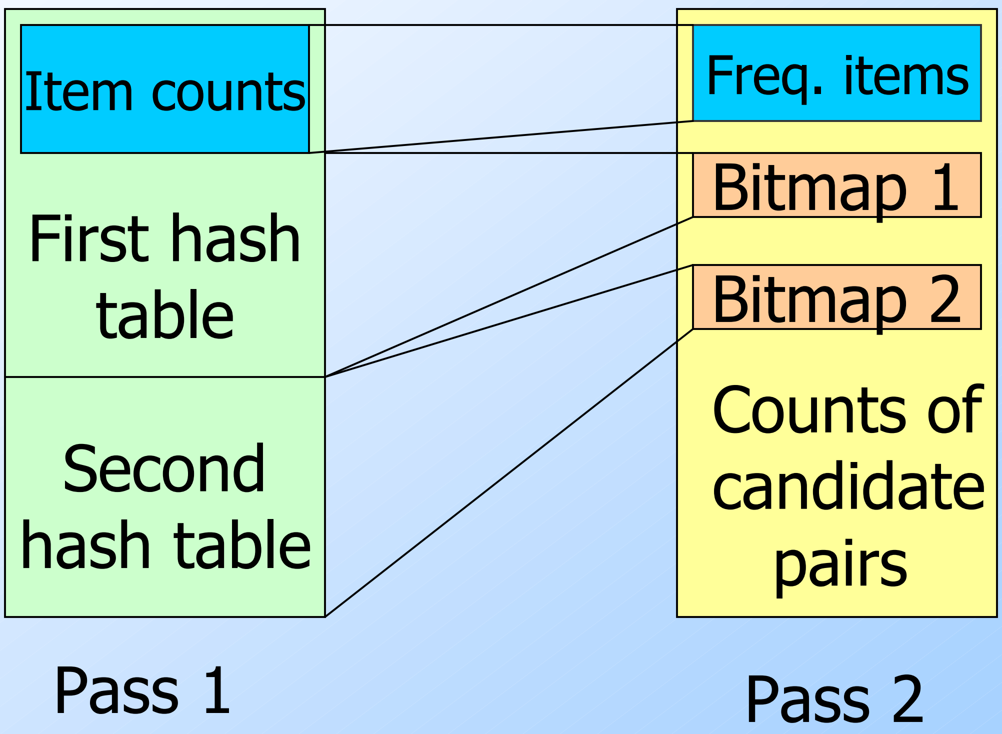
Important Points

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

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

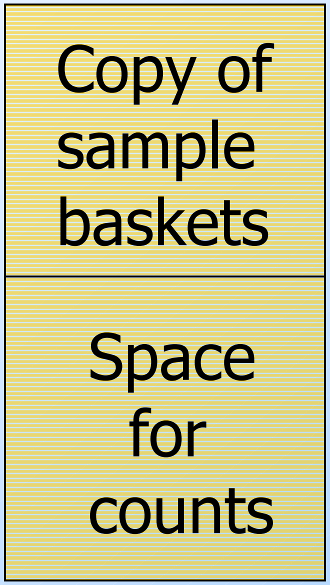
All (Or Most) Frequent Itemsets In $\leq$ 2 Passes

- A-Priori, PCY, etc., take *k* passes to find frequent itemsets of size *k*.
- Other techniques use 2 or fewer passes for all sizes:
  - Simple algorithm.
  - SON (Savasere, Omiecinski, and Navathe).
  - Toivonen.

Simple Algorithm – (1)

- Take a 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.

Main-Memory Picture



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.

Simple Algorithm – Option

- Optionally, 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 catch more truly frequent itemsets.
    - But requires more space.

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.

SON Algorithm – Distributed Version

- This idea lends itself to distributed data mining.
- If baskets are distributed among many nodes, compute frequent itemsets at each node, then distribute the candidates from each node.
- Finally, accumulate the counts of all candidates.

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.

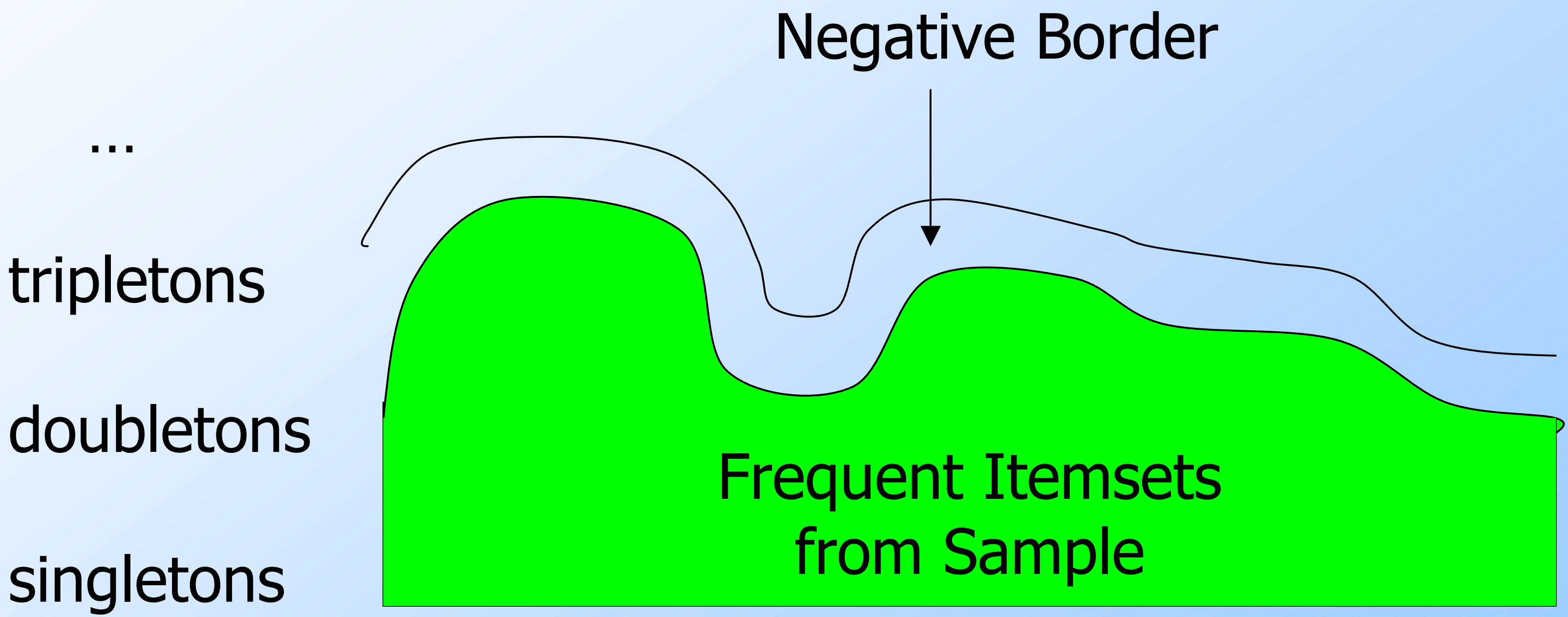
Toivonen's Algorithm – (2)

- 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: Negative Border

- *ABCD* is in the negative border if and only if:
  1. It is not frequent in the sample, but
  2. All of *ABC*, *BCD*, *ACD*, and *ABD* are.
- *A* is in the negative border if and only if it is not frequent in the sample.
  - Because the empty set is always frequent.
    - Unless there are fewer baskets than the support threshold (silly case).

Picture of Negative Border



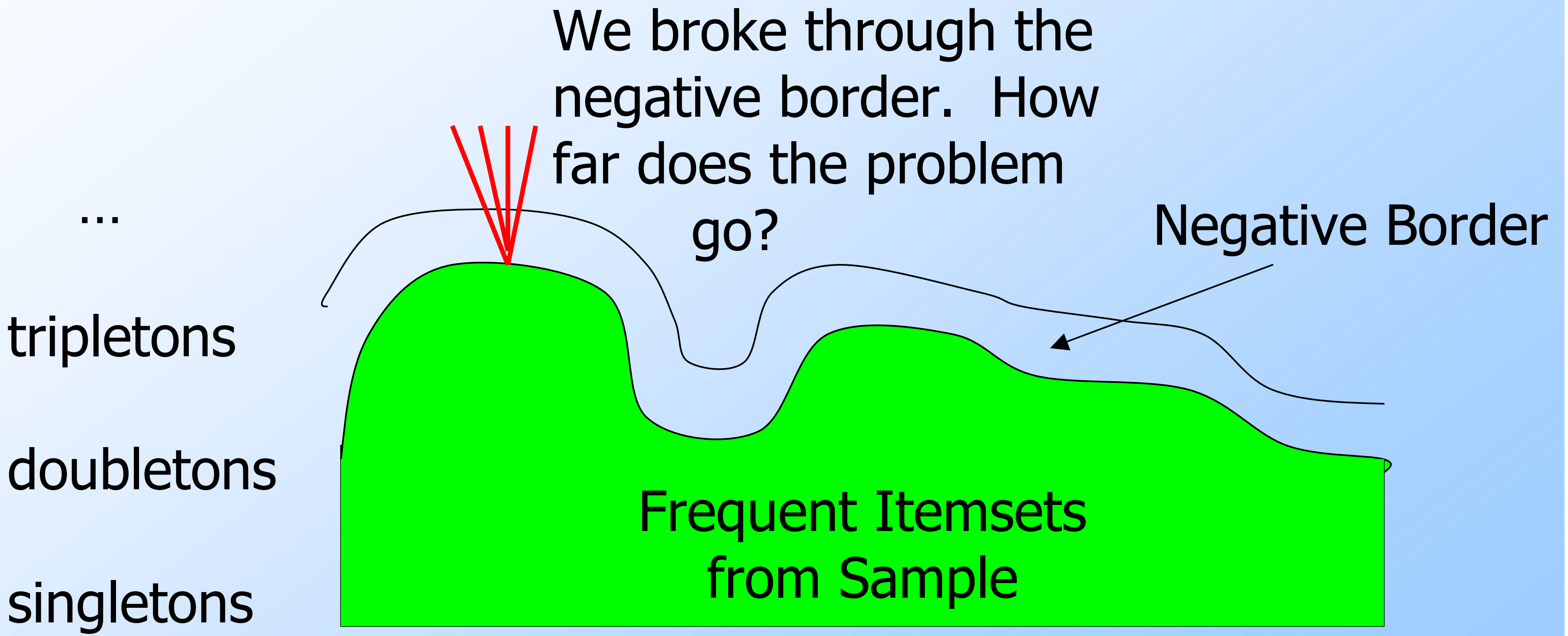
Toivonen's Algorithm – (3)

- In a second pass, count all candidate frequent itemsets from the first pass, and also count their 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 that 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.

If Something in the Negative Border is Frequent . . .



Theorem:

- If there is an itemset that is frequent in the whole, but not frequent in the sample, then there is a member of the negative border for the sample that is frequent in the whole.

- Proof: Suppose not; i.e.;
  1. There is an itemset *S* frequent in the whole but not frequent in the sample, and
  2. Nothing in the negative border is frequent in the whole.
- Let *T* be a smallest subset of *S* that is not frequent in the sample.
- *T* is frequent in the whole (*S* is frequent + monotonicity).
- *T* is in the negative border (else not "smallest").

Compacting the Output

1. *Maximal Frequent itemsets* : no immediate superset is frequent.
2. *Closed itemsets* : no immediate superset has the same count (> 0).
   - Stores not only frequent information, but exact counts.

Example: Maximal/Closed

	Count	Maximal (s=3)	Closed
A	4	No	No
B	5	No	Yes
C	3	No	No
AB	4	Yes	Yes
AC	2	No	No
BC	3	Yes	Yes
ABC	2	No	Yes

Frequent, but superset BC also frequent. (→ B, Maximal: No)

Frequent, and its only superset, ABC, not freq. (→ AB, Maximal: Yes)

Superset BC has same count. (→ C, Closed: No)

Its only super-set, ABC, has smaller count. (→ BC, Closed: Yes)