Frequent Itemsets 2

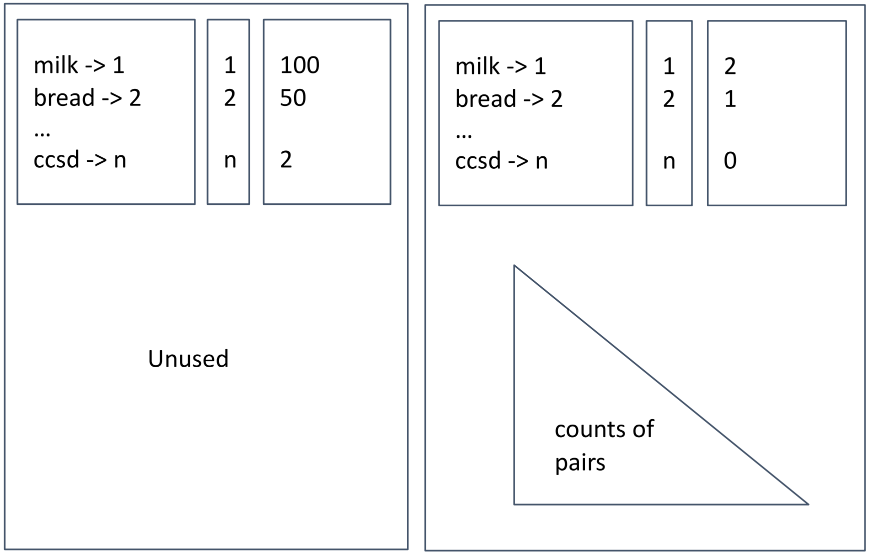
EE412: Foundation of Big Data Analytics

Announcements

- Homeworks
  - HW0 (due: 09/21)
  - HW1 will be posted this Thursday (due: 10/05)
- Classes
  - No classes at 09/19 and 09/21 (videos will be uploaded)

Recap: Frequent Itemsets

- **Market-basket model of data**
  - Support
  - Association rule $I \rightarrow j$
  - Confidence & interest
- **A-Priori algorithm**
  - Counting *pairs* is the hardest
  - Triangular vs triples method
  - Monotonicity
  - Roles of the two passes
  - Generalization to larger itemsets



<Pass 1> <Pass 2>

Outline

1. **PCY Algorithm**
2. Extensions of PCY
3. Limited-pass Algorithms

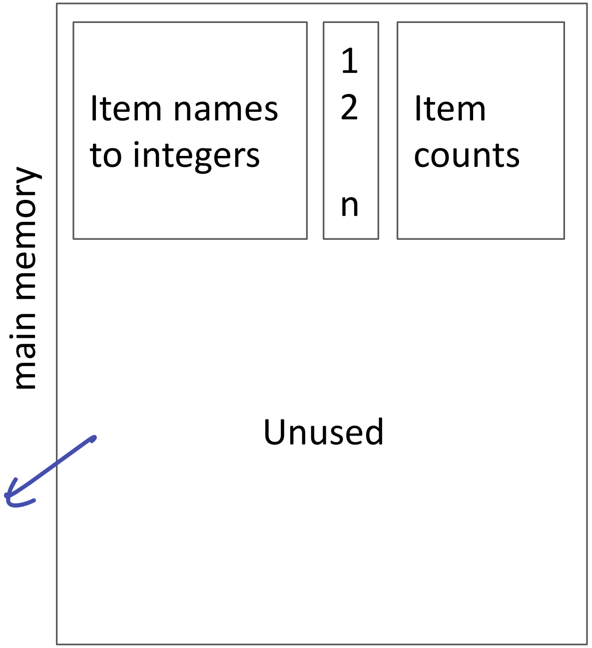
Handling Larger Data in Memory

- **The A-Priori algorithm is fine as long as:**
  - Counting candidate pairs C_2 can be done in main memory
- **Q:** What if the memory is not enough?
- **A:** We need algorithms that cut down the size of C_2:
  - **PCY algorithm**
  - **Multistage algorithm**
  - **Multihash algorithm**

PCY (Park-Chen-Yu) Algorithm

- In Pass 1 of A-Priori, most memory is idle
  - We store only individual item counts
- **Q:** Can we use the idle memory for pass 2?

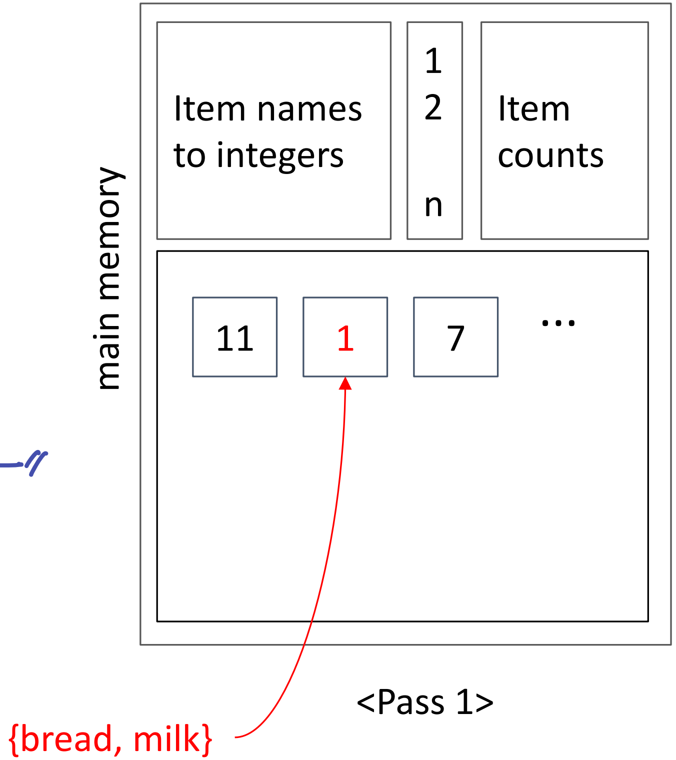
don't waste this space



<Pass 1>

PCY (Park-Chen-Yu) Algorithm

- In Pass 1 of A-Priori, most memory is idle
  - We store only individual item counts
- **Q:** Can we use the idle memory for pass 2?
- **Idea of PCY: Maintain a hash table**
  - Make it have as many buckets as fit in memory
  - Hash each pair and add 1 to the hashed bucket
    - For each bucket just keep the count



Observations about Buckets

- **Q:** Why do we do that?
- **A: To eliminate pairs that has no chance to be frequent!**
  - Each bucket B has count of all pairs that hash to B
    - If count $\geq$ support threshold s, pairs could all be frequent
    - Otherwise (i.e., infrequent bucket), **no pair can be frequent**
  - So in Pass 2, define candidate pairs C_2 as $\{i, j\}$ such that:
    - Both i and j are frequent items
    - $\{i, j\}$ hashes to a frequent bucket → **distinguishes PCY from A-Priori**
  - Later stages are similar to A-Priori

Between Passes in PCY

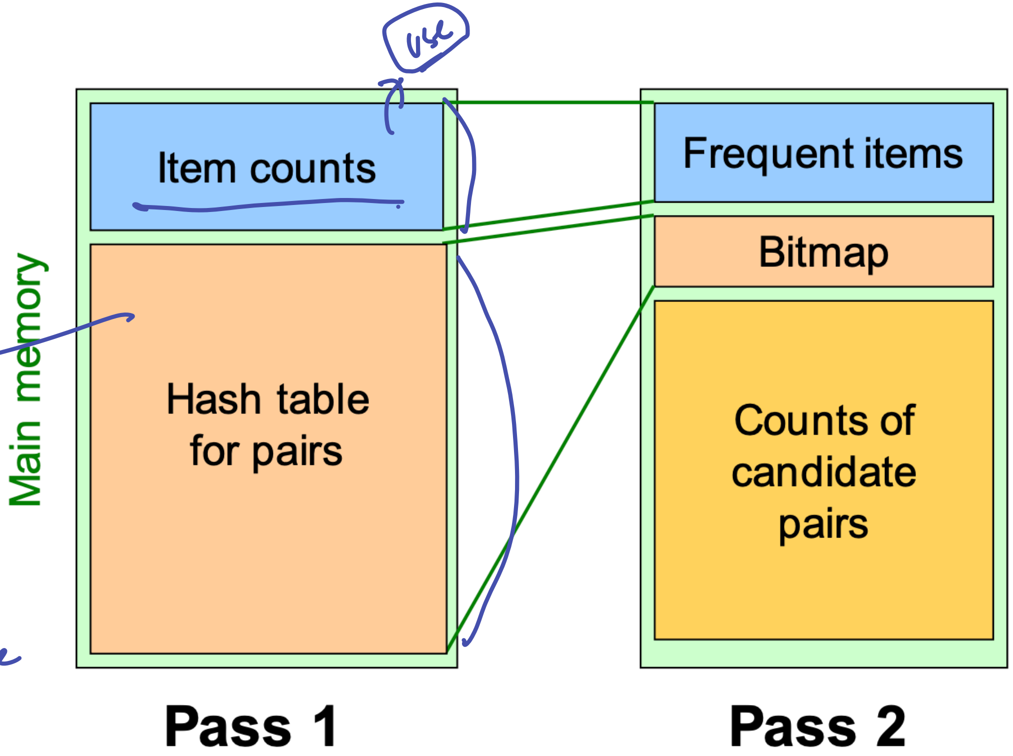
- **Replace the buckets by a bit-vector:**
  - 1 means the bucket count exceeded the support s (i.e., a frequent bucket)
  - 0 means it did not
- **The bitmap reduces the space by 1/32**
  - Since an integer is 32 bits

Main Memory: Picture of PCY

large =>

How many things to eliminate

small => not much of the meaning



Source: Stanford CS246 (2022)

One Subtlety

- PCY cannot use triangular-matrix method for counting pairs
  - Because PCY utilizes the sparsity of candidate pairs
  - No way to “compact” triangular matrix removing infrequent pairs
- **So PCY is always forced to use the triples method**

Pop Quiz

item counts // and hash table for pairs // simultaneously

- Here is a collection of baskets: $\{1,2,3\},\{3,4,5\},\{2,4,5\}$
- Suppose the support threshold is $s = 2$
- On the first pass of PCY, we use a hash table with 3 buckets
  - The set $\{i, j\}$ is hashed to the bucket $[i \times j \bmod 3]$
- **Q:** Which pairs are counted on the second pass of PCY?

(1,2,3) → (1,2) → 2
(1,3) → 0
(2,3) → 0
(3,4,5) → (3,4) → 0
(3,5) → 0
(4,5) → 0

(2,4,5) → (2,4) → 2
(2,5) → 1
(4,5) → 2

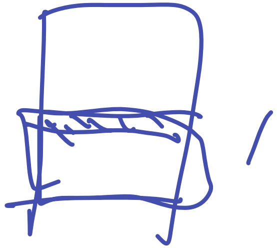
0	5 → freq
1	1 → okay
2	3

Outline

1. PCY Algorithm
2. **<u>Extensions of PCY</u>**
3. Limited-pass Algorithms

Extensions of PCY

- **Two extensions of PCY:** Multistage and Multihash algorithms
  - Both create multiple hash tables to reduce # of candidate pairs
- The way they create multiple tables is different
  - Multistage: Create tables through successive passes
  - Multihash: Create tables at the same time (i.e., in parallel)



The Multistage Algorithm

- **Improves PCY by creating several successive hash tables:**
  - **Pass 1:** Same as PCY
  - **Pass 2:** Use the second hash table to generate another bitmap
    - Hash table has 31/32 of the number of buckets due to first bitmap
    - Here a pair $\{i, j\}$ is hashed only if
      - Both i and j are frequent
      - The pair hashed to a frequent bucket in Pass 1
  - **Idea:** The second table can contain much fewer frequent buckets

Conditions for the Candidates

- A pair $\{i, j\}$ is in C_2 if and only if
    1. Both i and j are frequent
    2. Pair $\{i, j\}$ hashed to a frequent bucket in first hash table
    3. Pair $\{i, j\}$ hashed to a frequent bucket in second hash table
- **Q:** Is Condition 2 necessary?

No

Conditions for the Candidates

- A pair $\{i, j\}$ is in C_2 if and only if
  1. Both i and j are frequent
  2. Pair $\{i, j\}$ hashed to a frequent bucket in first hash table
  3. Pair $\{i, j\}$ hashed to a frequent bucket in second hash table
- **Q:** Is Condition 2 necessary? **Yes!**
  - Let's say $\{i, j\}$ is hashed to a infrequent bucket in the first table
    - Then, it is not counted in Pass 2
  - However, it doesn't mean $\{i, j\}$ cannot hash to frequent bucket
  - An infrequent pair $\{i, j\}$ may satisfy Conditions 1 and 3, but not 2

infrequent 모든 bucket이
second table에 frequent 하게 들어갈 수 있음

first table에서 거부된 걸 포함 안 하려면
2번 조건 필요함

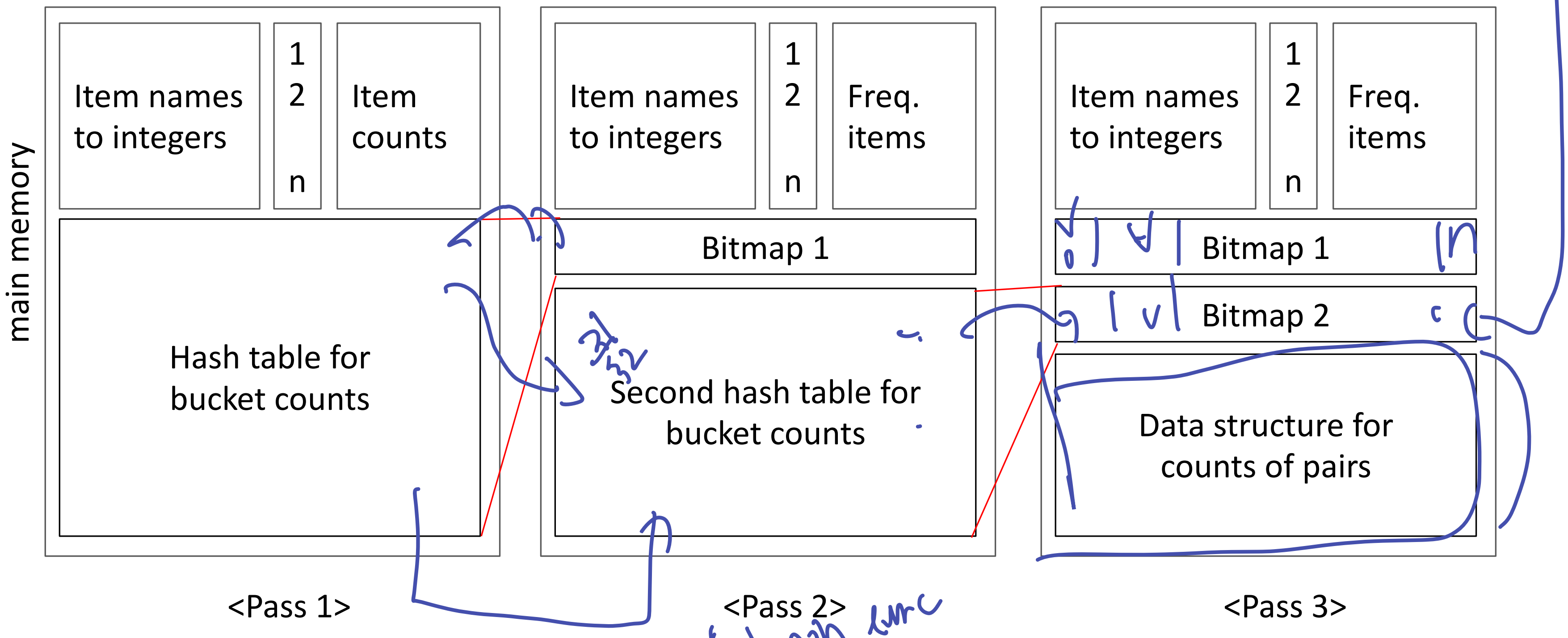
해야 함

hash

The Multistage Algorithm

bitmap (count

(1, 2)



independent hash func

More Passes in Multistage

- We can add more passes until there is not enough space
- Truly frequent pairs will always hash to frequent buckets
    - No matter how many passes we use

The Multihash Algorithm

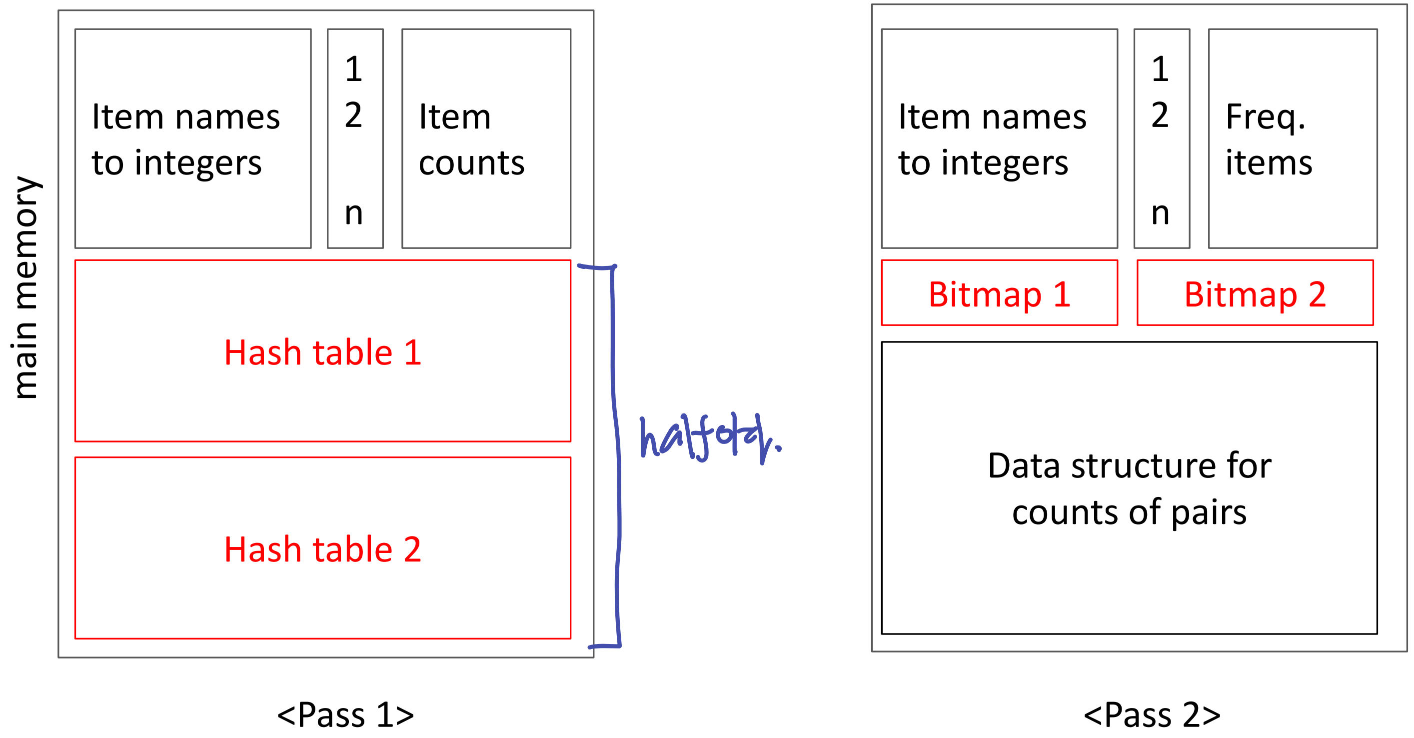
multistage => pass가 하나 더 필요

- **Idea:** Get benefits of extra passes in a single pass
- Create two hash tables with different hash functions on Pass 1
- We can still expect most buckets to be infrequent if
  - The average count of a bucket is lower than the support threshold
- A pair $\{i, j\}$ is in C_2 if
  1. Both i and j are frequent
  2. The pair hashes to a frequent bucket in both hash tables

half size인 hash bucket을 table 둘 위해

The Multihash Algorithm

average count



More Tables in Multihash

늘리게 되면 size↓, avg count↑ 효율적X

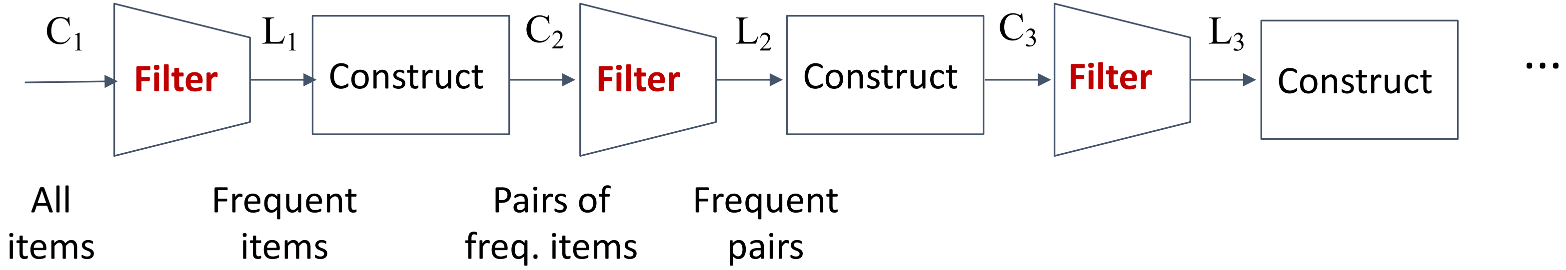
- We can increase the number n of tables as much as we want
- **Q:** How can we choose the optimal value of n?
- ?? Let's say p^n is the probability of an infrequent pair being in C_2
  - p: Probability of a pair to be frequent in each table
- In limited memory, using more tables increases both n and p
  - At some point, increasing # of tables increases the value of p^n

Outline

1. PCY Algorithm
2. Extensions of PCY
3. **<u>Limited-pass Algorithms</u>**

Recap: Generalization of A-Priori

- **So far, we used one pass for each size of itemsets**
  - The creation of L_k requires a new pass
  - C_k: Set of candidate k-tuples that might be frequent sets
  - L_k: Set of truly frequent k-tuples



Limited-Pass Algorithms

↳ approximate accuracy

- **Many applications can sacrifice accuracy for speed**
  - E.g., enough to find most of the frequent itemsets in supermarket
- We can find all or most frequent itemsets using ≤ 2 passes
  - **Random sampling:** Simplest approach
  - **Toivonen:** Two passes on average, but also may not terminate
  - **SON:** Divide the data into multiple chunks

Random Sampling

- **Idea:** Take a random sample of the market baskets
- Then, run a-priori or one of its improvements in main memory:
  - If sample is $p\%$ of baskets, adjust support threshold s to $ps/100$

B1 = {m, c, b}
B2 = {m, p, j}
B3 = {m, c, b, n}
B4 = {c, j}
B5 = {m, p, b}
B6 = {m, c, b, j}

Sample $p\%$ ⇨

B1 = {m, c, b}
B3 = {m, c, b, n}
B4 = {c, j}

⇨ A-Priori (or any other algorithm) using support threshold $ps/100$

sample 50% => half of threshold

Avoiding Errors in Sampling

- Sampling introduces both false positives and false negatives
  - **False positive:** Itemset frequent in sample, but not the whole → multistage에서는 이게 허용되는 경우
  - **False negative:** Itemset frequent in the whole, but not the sample → we miss
- **Eliminate false positives** by making another pass through full dataset and counting all frequent itemsets in sample — not difficult
- **Reduce false negatives** by reducing the sup. threshold (say to $0.9ps$)

more frequent items for safety in sample

pass를 반복해서 비교해보면 된다.

Toivonen’s Algorithm

extension of random sample
↳ avoid false negatives

- **Q:** Can we eliminate false negatives at all?
- **Idea:** Construct negative border for an efficient safety check
- **Toivonen’s Algorithm**
  - Use one pass over samples and one pass over the full data
  - No false negatives or positives
  - However, there is a small chance it will not produce answer

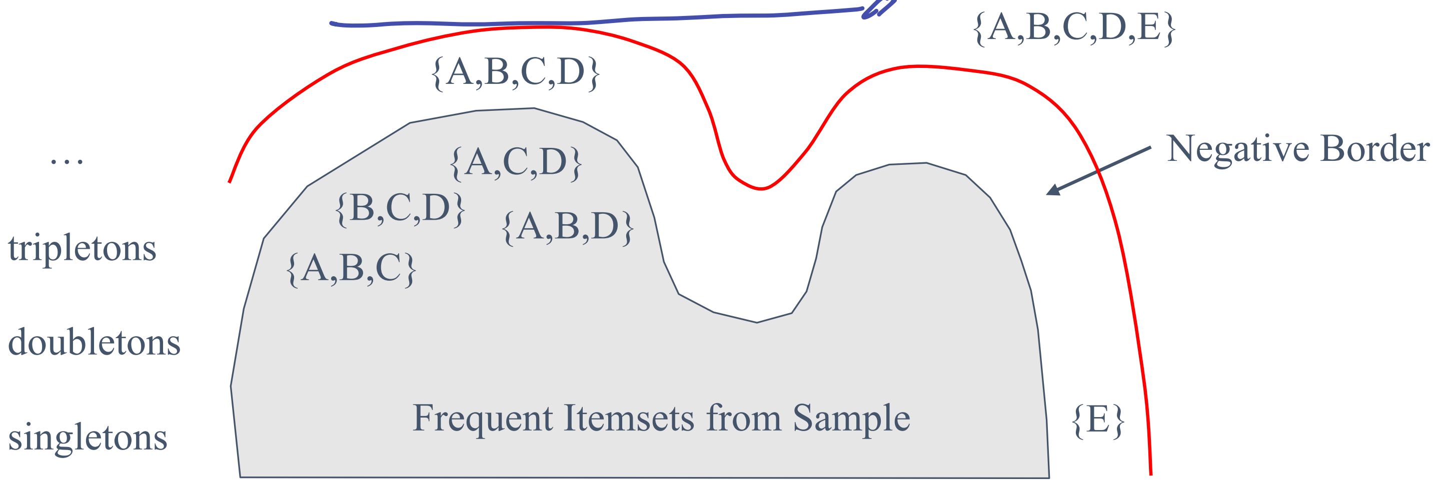
    rerun whole algorithm
    - The average number of passes to produce answer is still a constant

Toivonen's Algorithm (Pass 1)

- **Find candidates from a small sample**
  - Use support threshold less than proportional value ps (say $0.9ps$)
- **Construct** **negative border**
  - Collection of itemsets that are not frequent in sample, but all of their immediate subsets are frequent in sample
  - **Immediate subset:** subsets constructed by deleting exactly one item

Negative Border Example

- $\{A, B, C, D\}$ is in the negative border if and only if:
  - It is not frequent in the sample, but
  - All of $\{A, B, C\}$, $\{B, C, D\}$, $\{A, C, D\}$, and $\{A, B, D\}$ are frequent in sample
- $\{E\}$ is in the negative border if it is not frequent
  - Its immediate subset ∅ is considered frequent



Toivonen's Algorithm (Pass 2)

- **Make a pass on entire dataset**
  - Counting frequent itemsets in sample and itemsets in negative border
- **If** no member of negative border is frequent in whole dataset
  - We have found the correct set of frequent itemsets
- **Otherwise,** there may be a larger set that is frequent
  - Give no answer and repeat the algorithm with new random sample

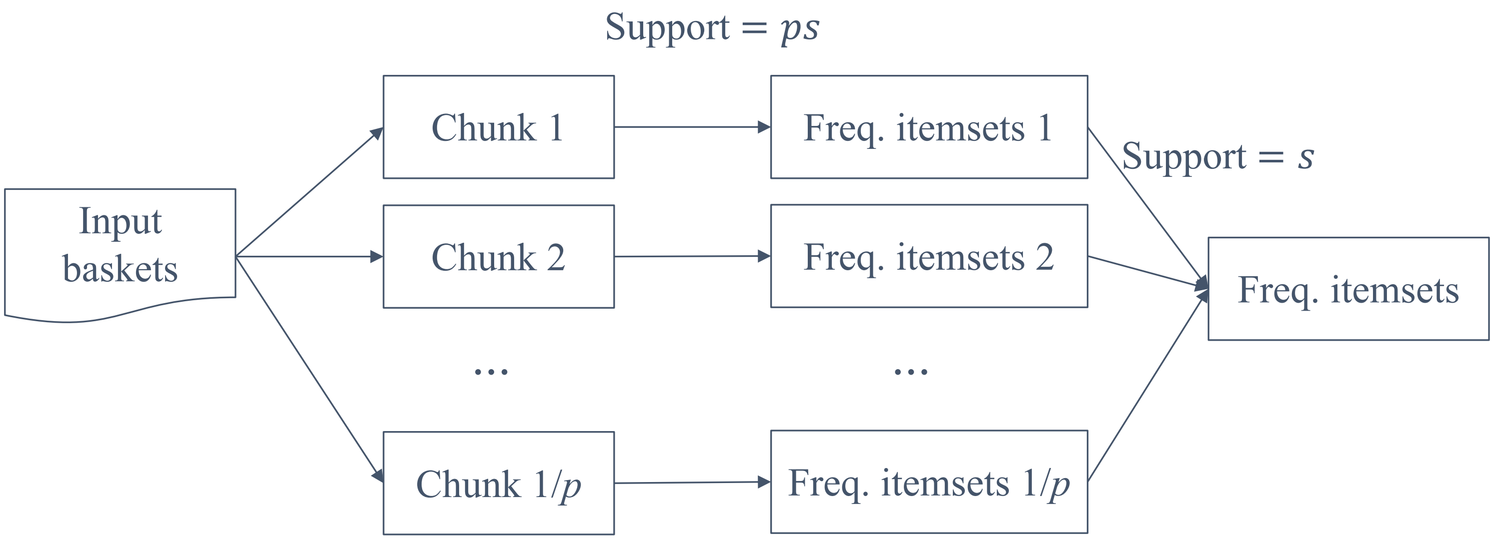
at least 1 member, not sure the result any more → rerun

Why Toivonen's Algorithm Works

- **No false positives:** Clearly because of Pass 2 전부확인
- **No false negatives:**
  - If there is an itemset S frequent in whole data, but not in sample
  - Then the border contains at least one itemset frequent in the whole
  - **Why?** One of the following holds:
    - S is in the negative order
    - S has an immediate subset T frequent in whole data, but not in the sample
      - This goes recursively until there is a subset in the negative order
      - Recall that every singleton is in the negative order

read //.

SON (Savasere, Omiecinski, Navathe)



SON Algorithm

- Avoids both false negatives and positives with two full passes
- **Pass 1:**
  - Divided input file into $1/p$ chunks
  - Run A-Priori with ps as support threshold
- **Pass 2:**
  - Take union of all frequent itemsets and select those with support $\geq s$
- **There are no false negatives**
  - A frequent itemset must be frequent in at least one chunk
  - If an itemset is not frequent in any chunk, its support is $< (1/p)ps \ = \ s$

SON with MapReduce

- MapReduce (or any other parallel computation) can be used
- **Pass 1:** Find candidate itemsets
  - **Map:** Take a chunk and return $(F, 1)$ for each frequent itemset F
    - Support threshold is ps
  - **Reduce:** Ignore value and produce itemsets that appear at least once
- **Pass 2:** Find true frequent itemsets
  - **Map:**
    - Take candidate itemsets from Pass 1 and a portion of the input data
    - Return (C, v) where C is a candidate itemset and v is the support for this portion
  - **Reduce:** For each itemset, sum the values and output if the sum $\geq s$

Summary

1. PCY Algorithm
2. Extensions of PCY
   - Multistage algorithm
   - Multihash algorithm
3. Limited-pass Algorithms
   - Random sampling
   - Toivonen's algorithm
   - SON algorithm