Finding Frequent Itemsets: Limited Pass Algorithms

Thanks for source slides and material to:

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets

http://www.mmds.org

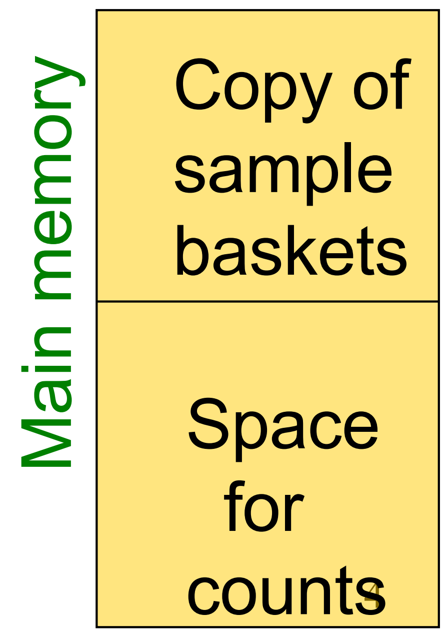
Limited Pass Algorithms

- Algorithms so far: compute **exact** collection of frequent itemsets of size *k* in *k* passes
  - A-Priori, PCY, Multistage, Multihash
- Many applications where it is not essential to discover **every** frequent itemset
  - Sufficient to discover most of them
- Next: algorithms that find all or most frequent itemsets using at most 2 passes over data
  - Sampling
  - SON
  - Toivonen's Algorithm

Random Sampling of Input Data

Random Sampling

- Take a **random sample** of the market baskets **that fits in main memory**
  - Leave enough space in memory for counts
- Run a-priori or one of its improvements in main memory
  - **For sets of all sizes**, not just pairs
  - Don't pay for disk I/O each time we increase the size of itemsets
  - Reduce support threshold proportionally to match the sample size



How to Pick the Sample

- Best way: read entire data set
- For each basket, **select that basket for the sample with probability *p***
  - For input data with m baskets
  - At end, will have a sample with size close to pm baskets

- If file is part of distributed file system, can **pick chunks at random** for the sample

Support Threshold for Random Sampling

- **Adjust support threshold** to a suitable, scaled-back number
  - To reflect the smaller number of baskets

Support Threshold for Random Sampling

- **Adjust support threshold** to a suitable, scaled-back number
  - To reflect the smaller number of baskets
- Example
  - If sample size is 1% or 1/100 of the baskets
  - Use *s* /100 as your support threshold
  - Itemset is **frequent in the sample** if it appears in at least s/100 of the baskets in the sample

Random Sampling: Not an exact algorithm

- With a single pass, **cannot guarantee:**
  - That algorithm will **produce all itemsets** that are frequent in the whole dataset
    - **False negative:** itemset that is frequent in the whole but not in the sample

Random Sampling: Not an exact algorithm

- With a single pass, **cannot guarantee:**
  - That algorithm will **produce all itemsets** that are frequent in the whole dataset
    - **False negative:** itemset that is frequent in the whole but not in the sample
  - That it will **produce only itemsets** that are frequent in the whole dataset
    - **False positive:** frequent in the sample but not in the whole
- If the sample is large enough, there are unlikely to be serious errors

Random Sampling: Avoiding Errors

- **Improvement**
  - Make a **second pass through the full dataset**
  - Count all itemsets that were identified as frequent in the sample
  - Verify that the candidate pairs are truly frequent in entire data set

Random Sampling: Avoiding Errors

- **Eliminate false positives**
  - Make a **second pass through the full dataset**
  - Count all itemsets that were identified as frequent in the sample
  - Verify that the candidate pairs are truly frequent in entire data set
- But this **doesn't eliminate false negatives**
  - Itemsets that are frequent in the whole but not in the sample
  - Remain undiscovered
- **Reduce false negatives**
  - Before, we used threshold *ps* where *p* is the sampling fraction
  - Reduce this threshold: e.g., *0.9ps*
  - More itemsets of each size have to be counted
  - If memory allows: requires more space
  - Smaller threshold helps catch more truly frequent itemsets

Savasere, Omiecinski and Navathe (SON) Algorithm

SON Algorithm

- Avoids false negatives and false positives
- Requires two full passes over data

SON Algorithm – (1)

- **Repeatedly read small subsets of the baskets into main memory**
- Run an in-memory algorithm (e.g., a priori, random sampling) 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

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 – (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
  - Subset or chunk contains fraction p of whole file
  - 1/p chunks in file
  - If itemset is not frequent in any chunk, then **support in each chunk is less than ps**
  - **Support in whole file is less than s: not frequent**

SON – Distributed Version

- **SON lends itself to distributed data mining**
  - **MapReduce**
- Baskets distributed among many nodes
  - Subsets of the data may correspond to one or more chunks in distributed file system
  - Compute frequent itemsets at each node
  - Distribute candidates to all nodes
  - **Accumulate the counts of all candidates**

SON: Map/Reduce
Phase 1: Find candidate itemsets

- **Map**
  - Input is a chunk/subset of all baskets; fraction p of total input file
  - **Find itemsets frequent in that subset** (e.g., using random sampling algorithm)
  - Use support threshold ps
  - **Output is set of key-value pairs (F, 1) where F is a frequent itemset from sample**

- **Reduce**
  - Each reduce task is assigned set of keys, which are itemsets
  - **Produces keys that appear one or more time**
  - **Frequent in some subset**
  - **These are candidate itemsets**

SON: Map/Reduce
Phase 2: Find true frequent itemsets

- **Map**
  - **Each Map task takes output from first Reduce task AND a chunk of the total input data file**
  - **All candidate itemsets go to every Map task**
  - Count occurrences of each candidate itemset among the baskets in the input chunk
  - **Output is set of key-value pairs *(C, v),* where *C* is a candidate frequent itemset and *v* is the support for that itemset** among the baskets in the input chunk
- **Reduce**
  - **Each reduce tasks is assigned a set of keys (itemsets)**
  - Sums associated values for each key: total support for itemset
  - **If support of itemset >= s, emit itemset and its count**

Toivonen's Algorithm

Toivonen's Algorithm

- Given sufficient main memory, uses **one pass over a small sample** and **one full pass over data**
- **Gives no false positives or false negatives**
- BUT, there is a **small but finite probability it will fail to produce an answer**
  - Will not identify frequent itemsets
- Then **must be repeated** with a different sample until it gives an answer
- Need only a small number of iterations

Toivonen's Algorithm (1)

First find candidate frequent itemsets from sample

- **Start as in the random sampling 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 $s/100$
  - For fraction p of baskets in sample, use 0.8ps or 0.9ps as support threshold
- **Goal is to avoid missing any itemset that is frequent in the full set of baskets**
- **The smaller the threshold:**
  - The more memory is needed to count all candidate itemsets
  - The less likely the algorithm will not find an answer

Toivonen's Algorithm – (2)

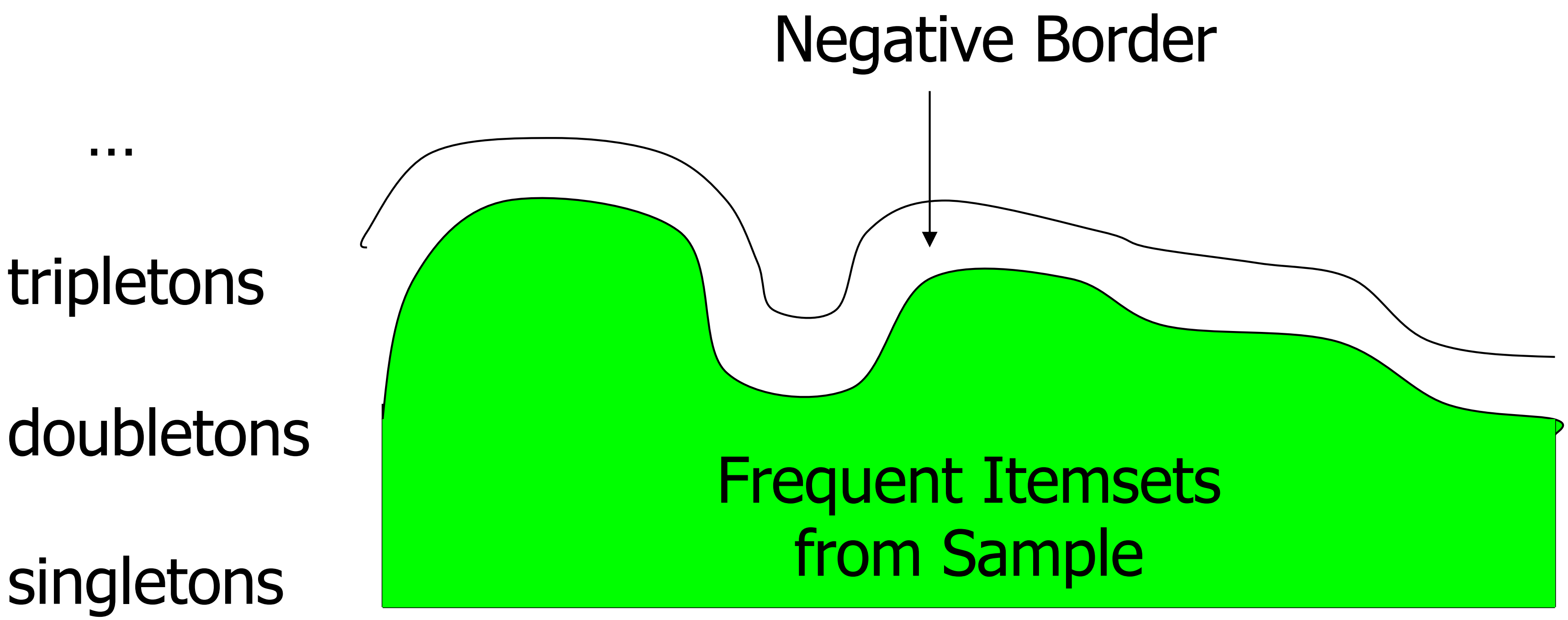
After finding frequent itemsets for the sample, construct the *negative border*

- **Negative border:** Collection of itemsets that are **not frequent** in the sample but **all of their immediate subsets are frequent**
  - Immediate subset is constructed by deleting exactly one item

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 frequent
     - Immediate subsets: formed by deleting an item
- ***A* is in the negative border if and only if it is not frequent in the sample**
  - Note: The empty set is always frequent

Picture of Negative Border



Toivonen's Algorithm (1)

First pass:

(1) First find candidate frequent itemsets from sample

- **Sample on first pass!**
- **Use lower threshold:** For fraction p of baskets in sample, use 0.8ps or 0.9ps as support threshold

- **Identifies itemsets that are frequent for the sample**

(2) Construct the ***negative border***

- **Itemsets that are not frequent** in the sample but **all of their immediate subsets are frequent**

Toivonen's Algorithm – (3)

- **In the second pass, process the whole file (no sampling!!)**
- **Count:**
  - all **candidate frequent itemsets** from first pass
  - all **itemsets on the negative border**
- **Case 1: No itemset from the negative border turns out to be frequent in the whole data set**
  - Correct set of frequent itemsets is ***exactly*** the itemsets from the sample that were found frequent in the whole data
- **Case 2: Some member of negative border is frequent in the whole data set**
  - Can give no answer at this time
  - **Must repeat algorithm with new random sample**

Toivonen's Algorithm – (4)

- **Goal: Save time by looking at a sample on first pass**
  - **But is the set of frequent itemsets for the sample the correct set for the whole input file?**
- **If some member of the negative border is frequent in the whole data set, can't be sure that there are not some even larger itemsets that:**
  - Are **neither in the negative border nor in the collection of frequent itemsets for the sample**
  - **But are frequent in the whole**
- **So start over with a new sample**

- Try to **choose the support threshold** so that **probability of failure is low,** while **number of itemsets checked on the second pass fits in main-memory**

A few slides on Hashing

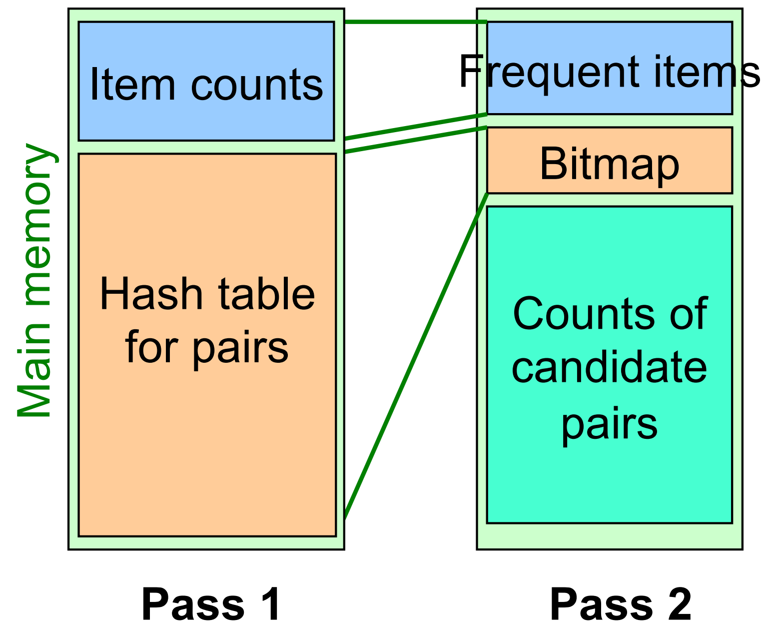
Introduction to Data Mining with Case Studies
Author: G. K. Gupta
Prentice Hall India, 2006.

Hashing

In PCY algorithm, when generating L_1, the set of frequent itemsets of size 1, the algorithm also:

- generates all possible pairs for each basket
- hashes them to buckets
- keeps a count for each hash bucket
- Identifies frequent buckets (count >= s)

Recall:
Main-Memory
Picture of PCY



Example

Consider a basket database in the first table below
All itemsets of size 1 determined to be frequent on previous pass
The second table below shows all possible 2-itemsets for each basket

Basket ID	Items
100	Bread, Cheese, Eggs, Juice
200	Bread, Cheese, Juice
300	Bread, Milk, Yogurt
400	Bread, Juice, Milk
500	Cheese, Juice, Milk

100	(B, C) (B, E) (B, J) (C, E) (C, J) (E, J)
200	(B, C) (B, J) (C, J)
300	(B, M) (B, Y) (M, Y)
400	(B, J) (B, M) (J, M)
500	(C, J) (C, M) (J, M)

Example Hash Function

- For each pair, a numeric value is obtained by first representing B by 1, C by 2, E 3, J 4, M 5 and Y 6.
- Now each pair can be represented by a two digit number
  - (B, E) by 13 (C, M) by 26
- **Use hash function** on these numbers: e.g., **number modulo 8**
  - **Hashed value is the bucket number**
- Keep count of the number of pairs hashed to each bucket
- Buckets that have **a count above the support value are frequent buckets**
  - Set corresponding bit in bit map to 1; otherwise, bit is 0
- All pairs in rows that have zero bit are removed as candidates

Hashing Example

Support Threshold = 3

The possible pairs:

100	(B, C) (B, E) (B, J) (C, E) (C, J) (E, J)
200	(B, C) (B, J) (C, J)
300	(B, M) (B, Y) (M, Y)
400	(B, J) (B, M) (J, M)
500	(C, J) (C, M) (J, M)

(B,C) -> 12, 12%8 = 4; (B,E) -> 13, 13%8 = 5; (C, J) -> 24, 24%8 = 0

Mapping table

B	1
C	2
E	3
J	4
M	5
Y	6

Bit map for frequent buckets	Bucket number	Count	Pairs that hash to bucket
1	0		
0	1		
0	2		
0	3		
0	4		
1	5		
1	6		
1	7		

Hashing Example

Support Threshold = 3

The possible pairs:

100	(B, C) (B, E) (B, J) (C, E) (C, J) (E, J)
200	(B, C) (B, J) (C, J)
300	(B, M) (B, Y) (M, Y)
400	(B, J) (B, M) (J, M)
500	(C, J) (C, M) (J, M)

(B,C) -> 12, 12%8 = 4; (B,E) -> 13, 13%8 = 5; (C, J) -> 24, 24%8 = 0

Mapping table

B	1
C	2
E	3
J	4
M	5
Y	6

Bit map for frequent buckets	Bucket number	Count	Pairs that hash to bucket
1	0		
0	1		
0	2		
0	3		
0	4	2	(B, C)
1	5	3	(B, E) (J, M)
1	6		
1	7		

Bucket 5 is frequent. Are any of the pairs that hash to the bucket frequent?

Does Pass 1 of PCY know which pairs contributed to the bucket?

Hashing Example

Support Threshold = 3

The possible pairs:

100	(B, C) (B, E) (B, J) (C, E) (C, J) (E, J)
200	(B, C) (B, J) (C, J)
300	(B, M) (B, Y) (M, Y)
400	(B, J) (B, M) (J, M)
500	(C, J) (C, M) (J, M)

(B,C) -> 12, 12%8 = 4; (B,E) -> 13, 13%8 = 5; (C, J) -> 24, 24%8 = 0

Mapping table

B	1
C	2
E	3
J	4
M	5
Y	6

Bit map for frequent buckets	Bucket number	Count	Pairs that hash to bucket
1	0	5	(C, J) (B, Y) (M, Y)
0	1	1	(C, M)
0	2	1	(E, J)
0	3	0	
0	4	2	(B, C)
1	5	3	(B, E) (J, M)
1	6	3	(B, J)
1	7	3	(C, E) (B, M)

At end of Pass 1, know only which buckets are frequent

All pairs that hash to those buckets are candidates and will be counted

Reducing number of candidate pairs

- Goal: reduce the size of candidate set C_2
  - Only have to count candidate pairs
  - Pairs that hash to a frequent bucket
- Essential that the hash table is large enough so that collisions are few
- Collisions result in loss of effectiveness of the hash table
- In our example, three frequent buckets had collisions
- Must count all those pairs to determine which are truly frequent