Finding Frequent Itemsets: Improvements to A-Priori

Park-Chen-Yu Algorithm

Multistage Algorithm

Multi-Hash Algorithm

Approximate Algorithms

Professor Wei-Min Shen

University of Southern California

Thanks for source slides and material to:

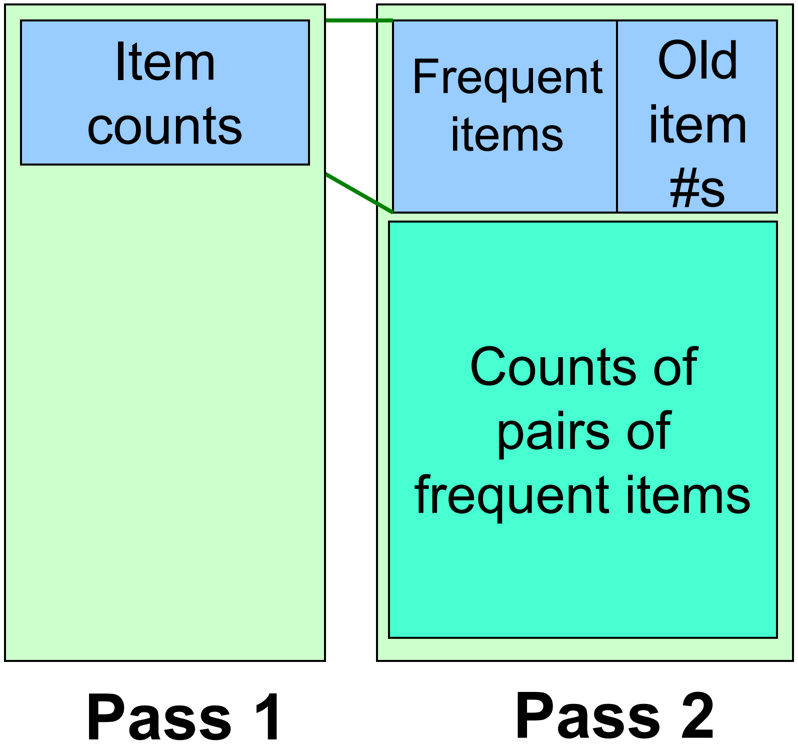
J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets

http://www.mmds.org

Park-Chen-Yu (PCY) Algorithm

Can we do better than A-Priori?

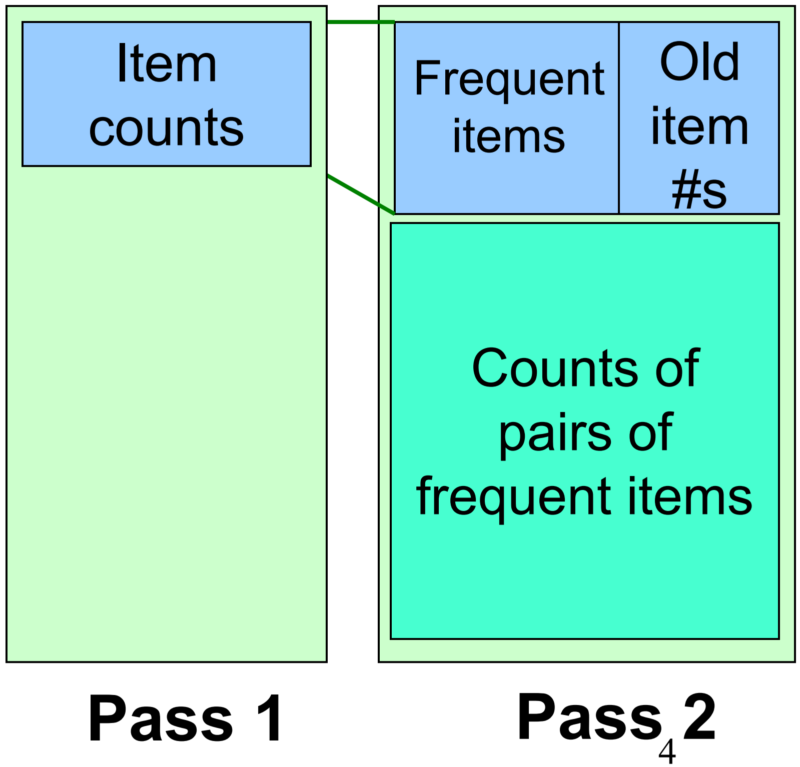
A Priori Memory Usage:



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

A Priori Memory Usage:



PCY (Park-Chen-Yu) Algorithm

- **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:** In addition to item counts, maintain a hash table with as many <u>buckets</u> as fit in memory

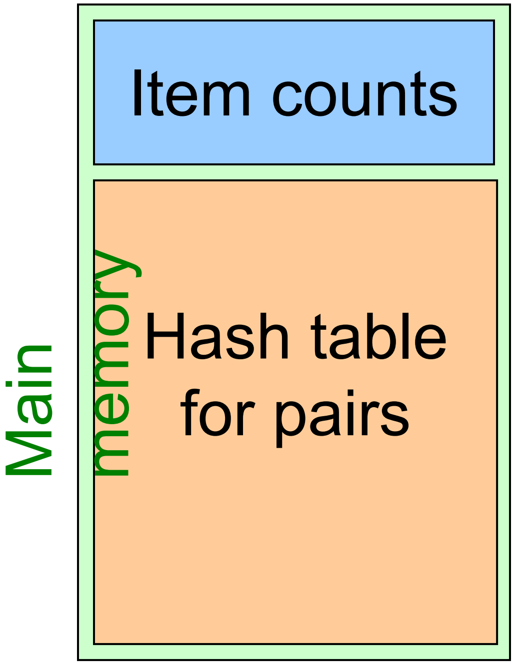
  not "baskets"
  - Keep a **count** for each bucket into which **pairs** of items are hashed
    - **For each bucket just keep the count, not the actual pairs that hash to the bucket!**

PCY Algorithm – First Pass

```
FOR (each basket) :
  FOR (each item in the basket) :
     add 1 to item's count;
New FOR (each pair of items) :
 in  hash the pair to a bucket;
PCY  add 1 to the count for that bucket;
```

- **A 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

Main-Memory: Picture of PCY Pass 1



Pass 1

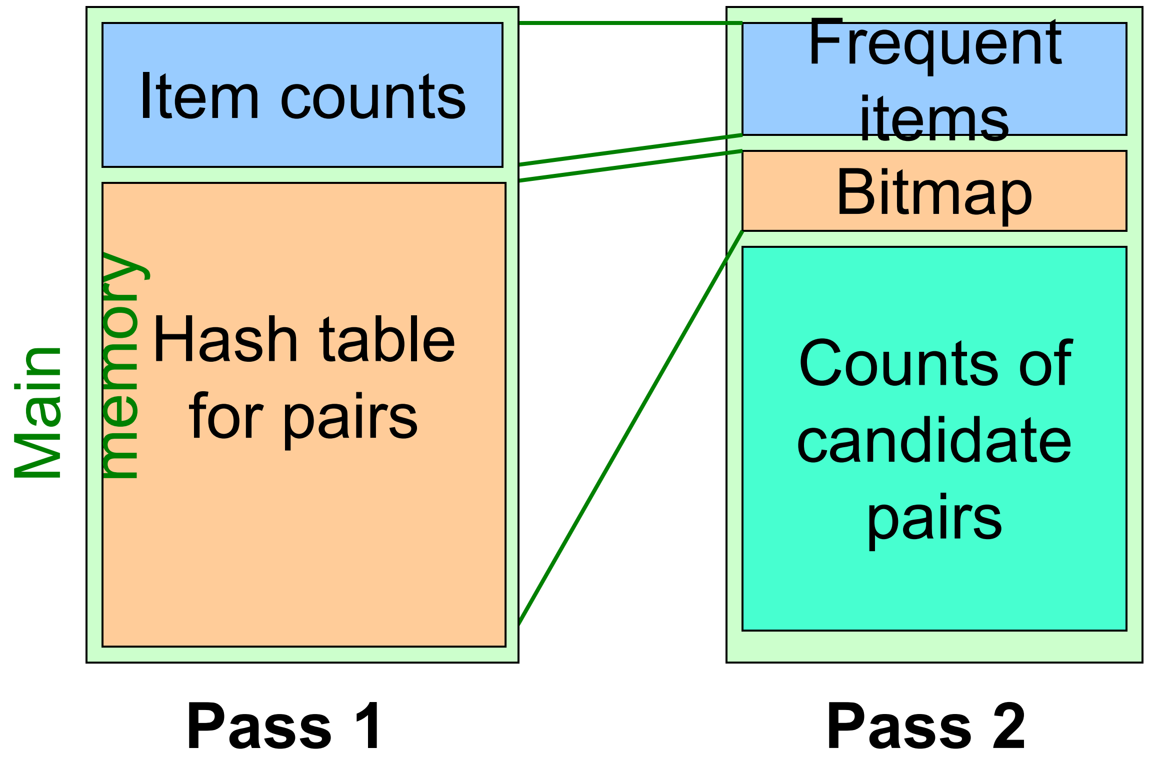
Observations about Buckets

- A bucket is *frequent* if its count is at least the support threshold s
- Observation: If a bucket contains a frequent pair, then the bucket is surely frequent (why?)
- A bucket may contain more than one pairs (same hash), thus, a bucket may still be frequent, but the pairs in that bucket may not be "truly 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*, then none of its pairs can be frequent (why?)**
  - 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

PCY Algorithm – Between Passes

- **Replace the buckets by a bit-vector:**
  - **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**
- As for A Priori, decide which items are frequent in first pass and list them for the second pass

Main-Memory: Picture of PCY



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}* has been hashed into 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**

Hashing 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 25
- **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

Bucket 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

Bucket 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

Bucket 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 (not pairs) are frequent

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

Another Hashing Example
(Try yourself)

Exercise 6.3.1: Here is a collection of twelve baskets. Each contains three of the six items 1 through 6.

$$\begin{array}{cccc} \{1,2,3\} & \{2,3,4\} & \{3,4,5\} & \{4,5,6\} \\ \{1,3,5\} & \{2,4,6\} & \{1,3,4\} & \{2,4,5\} \\ \{3,5,6\} & \{1,2,4\} & \{2,3,5\} & \{3,4,6\} \end{array}$$

Hash functions: i+j mod 9

Main-Memory Details

- **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 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

Why Can't We use a Triangular Matrix on Phase 2 of PCY?

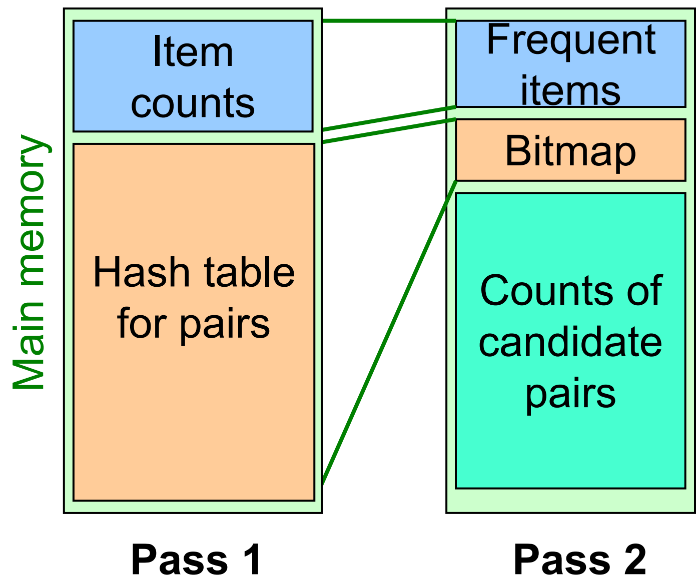
- Recall: in A-Priori, the frequent items could be renumbered in Pass 2 from *1* to *m*
- Can't do that for PCY
- Pairs of frequent items that PCY lets us avoid counting are placed randomly within the triangular matrix
  - Pairs that happen to hash to an infrequent bucket on first pass
  - No known way of compacting matrix to avoid leaving space for uncounted pairs
- Must use the triples method

Hashing (summary)

In PCY algorithm, when generating L_1, the set of frequent single items, 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



The Goal of Hashing: Reduce the number of candidate pairs

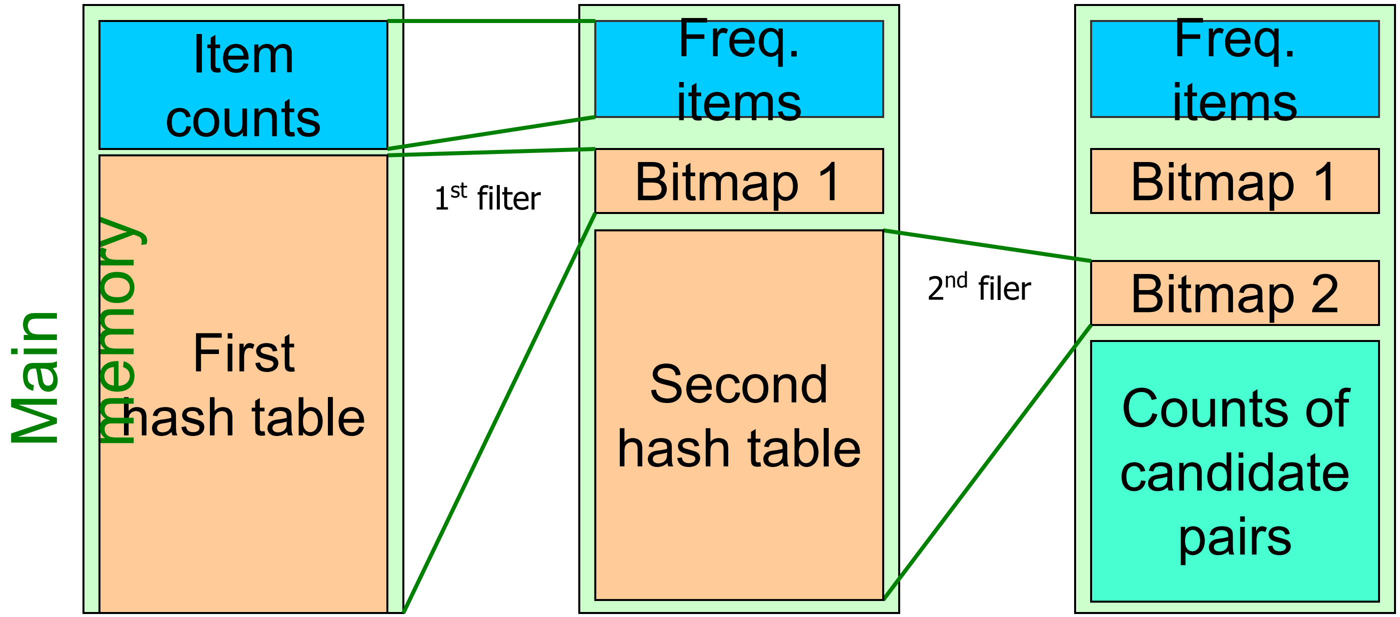
- Goal: reduce the size of candidate set C_2
  - Only have to count candidate pairs (not all pairs)
  - Candidate pairs are that hash to a frequent bucket
- Essential that the hash table is large enough so that collisions are few
- Collisions result in loss of effectiveness of the hash table
- In our example, three frequent buckets had collisions
- Must count all those pairs to determine which are truly frequent

Multi-Stage Algorithm

Refinement: Multistage Algorithm

- **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:** After Pass 1 of PCY, **rehash only those pairs** that qualify for Pass 2 of PCY
  - ***i*** and ***j*** are frequent, and
  - ***{i, j}*** hashes to a frequent bucket from **Pass 1**
- On middle pass, fewer pairs contribute to buckets, so fewer ***false positives***
- **Requires 3 passes over the data**

Main-Memory: Multistage



Pass 1

Count items
Hash pairs {i,j}

Pass 2

Hash pairs {i,j} into Hash2 iff:
i,j are frequent,
{i,j} hashes to freq. bucket in B1

Pass 3

Count pairs {i,j} iff:
i,j are frequent,
{i,j} hashes to freq. bucket in B1
{i,j} hashes to freq. bucket in B2

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 end up counting pairs of frequent items that hashed first to an infrequent bucket but happened to hash second to a frequent bucket
   - Would be a false positive

Multi-stage Example

bucket support threshold = 3

Exercise 6.3.1: Here is a collection of twelve baskets. Each contains three of the six items 1 through 6.

$\{1,2,3\}$	$\{2,3,4\}$	$\{3,4,5\}$	$\{4,5,6\}$
$\{1,3,5\}$	$\{2,4,6\}$	$\{1,3,4\}$	$\{2,4,5\}$
$\{3,5,6\}$	$\{1,2,4\}$	$\{2,3,5\}$	$\{3,4,6\}$

Try the multistage algorithm:

- 1st Hash function: i+j mod 8
- 2nd Hash function: i+j mod 5

1st hash

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)

2nd hash

??? (you fill in)

Key Observation

- Can insert any number of hash passes between first and last stage
  - Each one uses <u>an independent hash function</u>
  - Eventually, all memory would be consumed by bitmaps, no memory left for counts
  - Cost is another pass of reading the input data
- **The truly frequent pairs will always hash to a frequent bucket**
- So we will count the frequent pairs no matter how many hash functions we use

Multi-Hash Algorithm

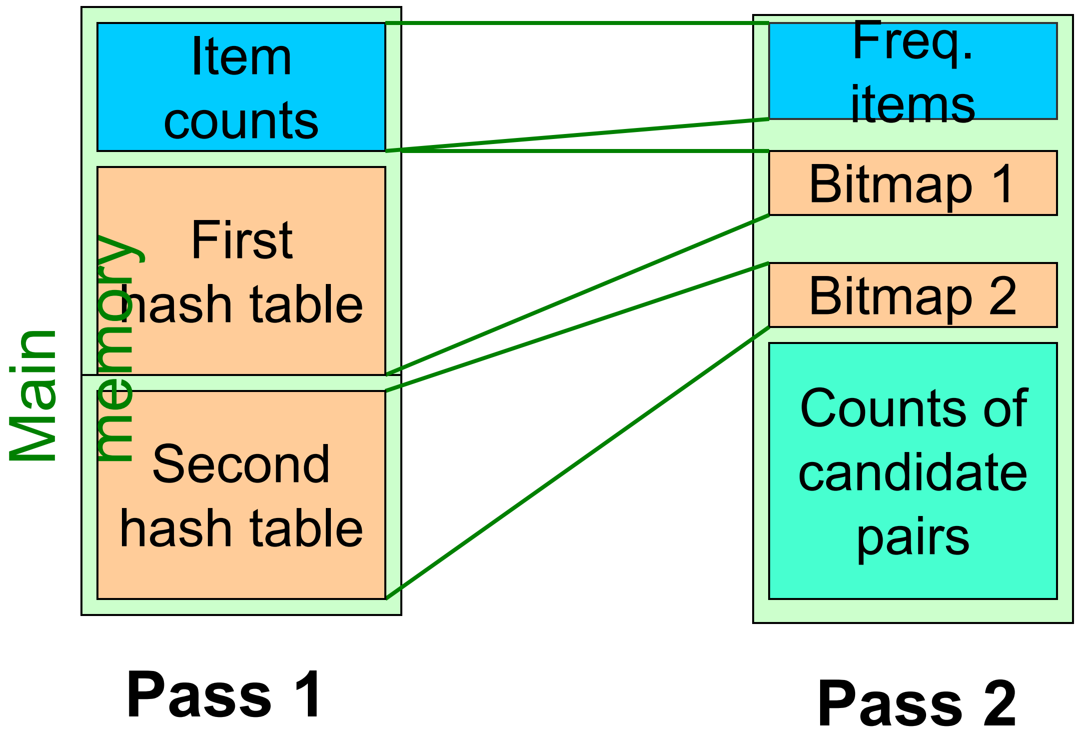
Thanks for source slides and material to:
J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets
http://www.mmds.org

Refinement: 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

Main-Memory: Multihash



Multihash – Pass 2

- Same condition as Multistage but checked in second pass
- **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**

Example

Exercise 6.3.1 : Here is a collection of twelve baskets. Each contains three of the six items 1 through 6.

$$\begin{array}{cccc}
\{1,2,3\} & \{2,3,4\} & \{3,4,5\} & \{4,5,6\} \\
\{1,3,5\} & \{2,4,6\} & \{1,3,4\} & \{2,4,5\} \\
\{3,5,6\} & \{1,2,4\} & \{2,3,5\} & \{3,4,6\}
\end{array}$$

Try multi-hash algorithm:

1^{st} Hash function: i+j mod 5

2^{nd} Hash function: i+j mod 4

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 $\geq$ ***s***

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 (Approximate) Algorithms

- There are many applications where it is sufficient to find most but not all frequent itemsets
- Algorithms to find these in at most 2 passes

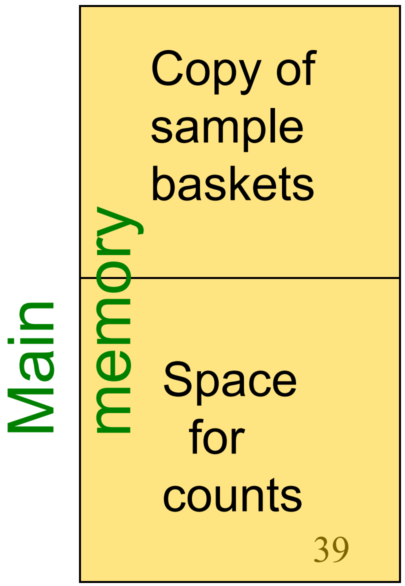
Limited Pass Algorithms

- 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

- The 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?
  - Reduce?

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

SON: Map/Reduce
Phase 1: Find local 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 a scaled-down support threshold p*s
  - **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 0.8*ps* or 0.9*ps* 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*

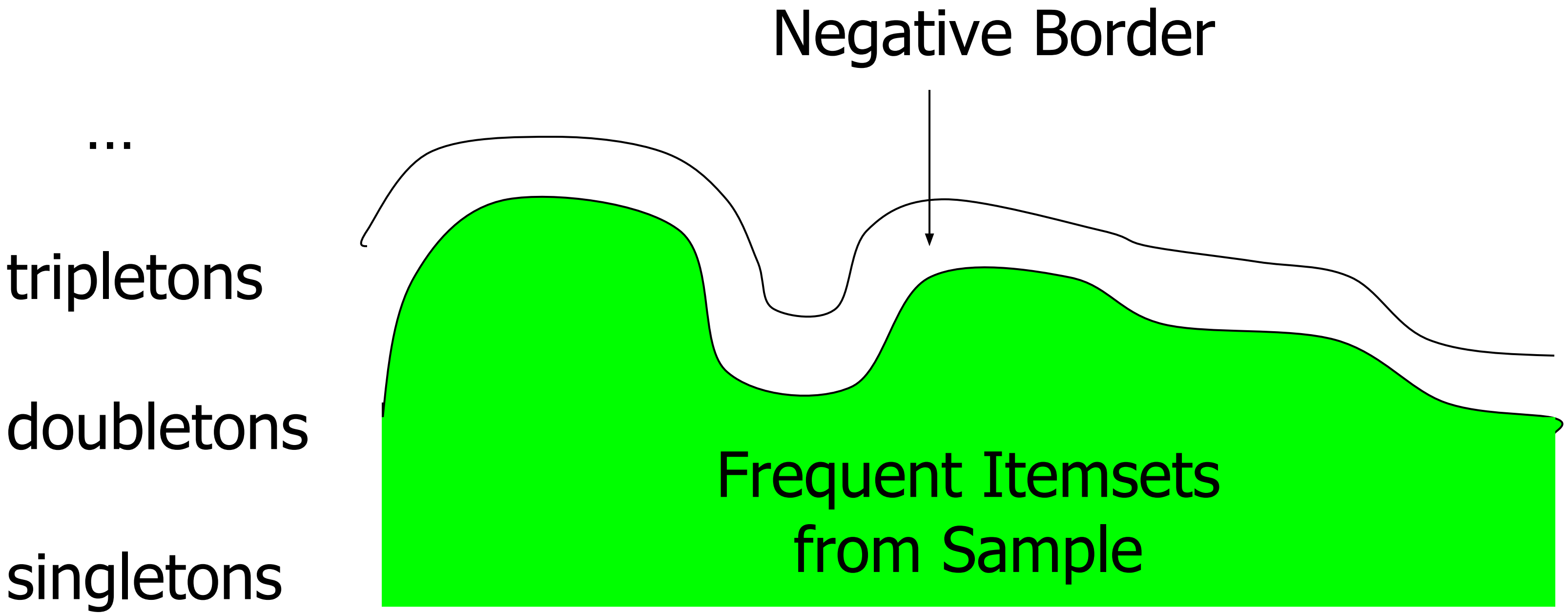
- **Negative border:** Collection of itemsets that are **not frequent** in the sample but **all of their immediate subsets are frequent**
  - Immediate subset is constructed by deleting exactly one item

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

- ***X* 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 (3)

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 – (4)

- **In the second pass, process the whole file (not sample)**
- **Count:**
  - all **candidate frequent itemsets** from first pass
  - all **itemsets on the negative border**
- **Case 1 (the border is correct): No itemset from the negative border is 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 (the border is wrong): 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 – (5)

- **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**