Frequent Item Sets

Chau Tran & Chun-Che Wang

Outline

1. Definitions
    - Frequent Itemsets
    - Association rules
2. Apriori Algorithm

Frequent Itemsets

What? Why? How?

Motivation 1: Amazon suggestions

Frequently Bought Together

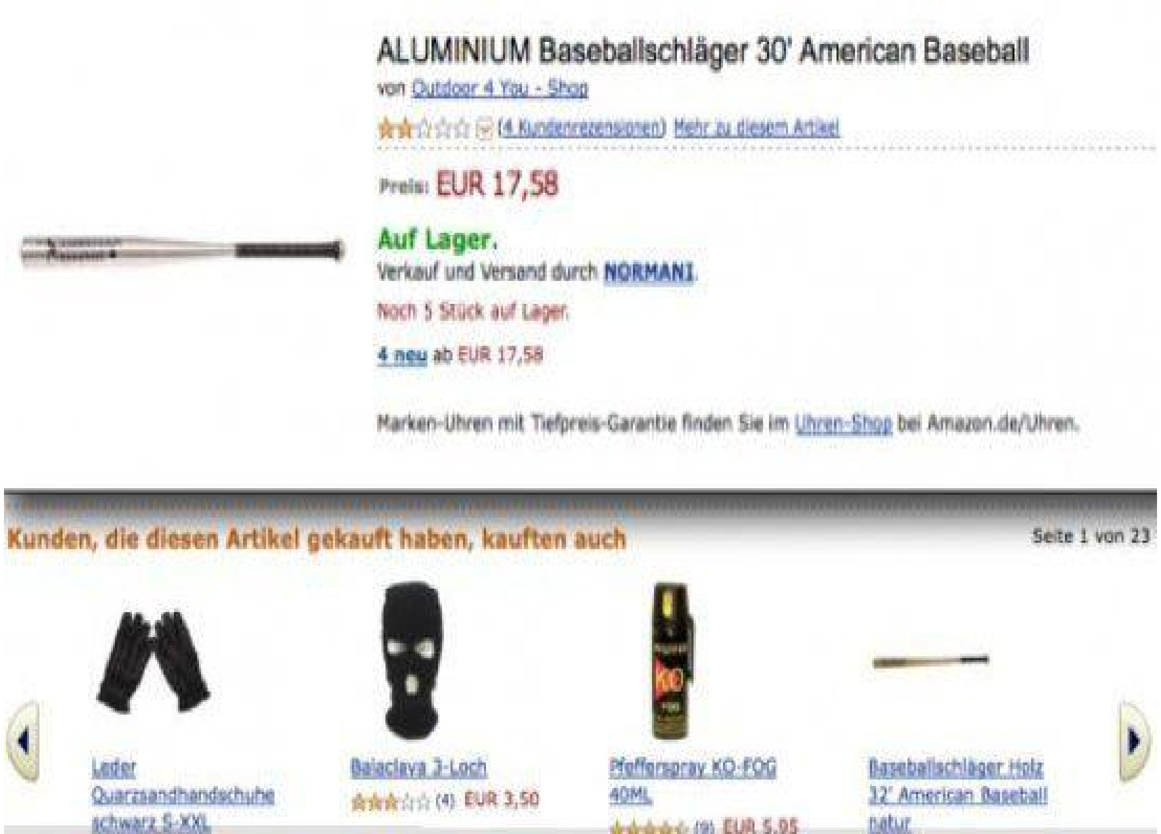
Price for all three: $85.90

Add all three to Cart | Add all three to Wish List

Show availability and shipping details

- ☑ **This item:** Kit Kat Candy Bar, Crisp Wafers in Milk Chocolate, 1.5-Ounce Bars (Pack of 36) $28.63 ($0.53 / oz)
- ☑ Reese's Peanut Butter Cups, 1.5-Ounce Packages (Pack of 36) $24.30 ($0.45 / oz)
- ☑ Twix-chocolate Caramel Cookie Bars, 36ct $32.97 ($9.16 / 10 Items)

Amazon suggestions (German version)



Motivation 2: Plagiarism detector

- Given a set of documents (eg. homework handin)
  - Find the documents that are similar

Motivation 3: Biomarker

- Given the set of medical data
  - For each patient, we have his/her genes, blood proteins, diseases
  - Find patterns
    - which genes/proteins cause which diseases

What do they have in common?

- A large set of items
  - things sold on Amazon
  - set of documents
  - genes or blood proteins or diseases
- A large set of baskets
  - shopping carts/orders on Amazon
  - set of sentences
  - medical data for multiple of patients

Goal

- Find a general many-many mapping between two set of items
  - {Kitkat} ⇒ {Reese, Twix}
  - {Document 1} ⇒ {Document 2, Document 3}
  - {Gene A, Protein B} ⇒ {Disease C}

Approach

- A = {A1, A2,..., Am}
- B = {B1, B2,..., Bn}

A, B are subset of
I = set of items

$$P(B|A) = \frac{P(A,B)}{P(A)}$$

$$= \frac{Count(A,B)}{Count(A)}$$

Definitions

- Support for itemset A: Number of baskets containing all items in A
  - Same as Count(A)
- Given a support threshold s, the set of items that appear in at least s baskets are called frequent itemsets

Example: Frequent Itemsets

- Items = {milk, coke, pepsi, beer, juice}

B1 = {m,c,b}	B2 = {m,p,j}
B3 = {m,b}	B4 = {c,j}
B5 = {m, p, b}	B6 = {m,c,b,j}
B7 = {c,b,j}	B8 = {b,c}

- Frequent itemsets for support threshold = 3:
  - {m}, {c}, {b}, {j}, {m,b}, {b,c}, {c,j}

Association Rules

- $A \Rightarrow B$ means: "if a basket contains items in A, it is likely to contain items in B"
- There are exponentially many rules, we want to find significant/interesting ones
- Confidence of an association rule:
  - $\mathrm{Conf}(A \Rightarrow B) = P(B \mid A)$

Interesting association rules

- Not all high-confidence rules are interesting
  - The rule $X \Rightarrow$ milk may have high confidence for many itemsets X, because milk is just purchased very often (independent of X), and the confidence will be high
- Interest of an association rule:
  - $\text{Interest}(A \Rightarrow B) = \text{Conf}(A \Rightarrow B) - P(B)$

$$= P(B \mid A) - P(B)$$

- Interest(A $\Rightarrow$ B) = P(B | A) - P(B)
  - > 0 if P(B | A) > P(B)
  - = 0 if P(B | A) = P(B)
  - < 0 if P(B | A) < P(B)

Example: Confidence and Interest

B1 = {m,c,b}	B2 = {m,p,j}
B3 = {m,b}	B4 = {c,j}
B5 = {m, p, b}	B6 = {m,c,b,j}
B7 = {c,b,j}	B8 = {b,c}

- Association rule: $\{m,b\} \Rightarrow c$
  - Confidence = 2/4 = 0.5
  - Interest = $0.5 - \frac{5}{8} = -\frac{1}{8}$
    - High confidence but not very interesting

Overview of Algorithm

- Step 1: Find all frequent itemsets I
- Step 2: Rule generation
  - For every subset A of I, generate a rule $A \Rightarrow I \setminus A$
    - Since I is frequent, A is also frequent
  - Output the rules above the confidence threshold

Example: Finding association rules

B1 = {m,c,b}	B2 = {m,p,j}
B3 = {m,b}	B4 = {c,j}
B5 = {m, p, b}	B6 = {m,c,b,j}
B7 = {c,b,j}	B8 = {b,c}

- Min support s=3, confidence c=0.75
- 1) Frequent itemsets:
  - {b,m} {b,c} {c,n} {c,j} {m,c,b}
- 2) Generate rules:
  - ~~b ⇒ m = 4/6~~ b ⇒ c = 5/6 b,m ⇒ c = 3/4
  - m ⇒ b = 4/5 … ~~b,c ⇒ m = 3/5~~

…

How to find frequent itemsets?

- Have to find subsets A such that Support(A) > s
  - There are 2^n subsets
  - Can’t be stored in memory

How to find frequent itemsets?

- Solution: only find subsets of size 2

Really?

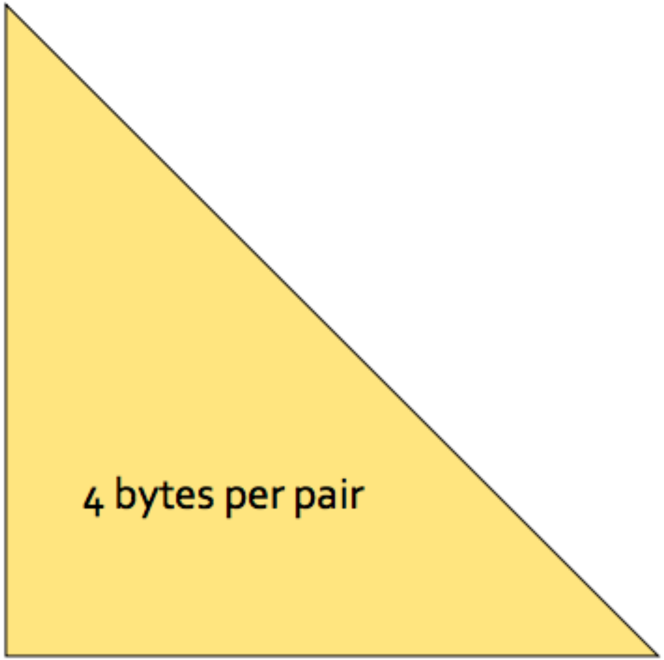
- Frequent pairs are common, frequent triples are rare, don't even talk about n=4
- Let's first concentrate on pairs, then extend to larger sets (wink at Chun)
- The approach
  - Find Support(A) for all A such that |A| = 2

Naive Algorithm

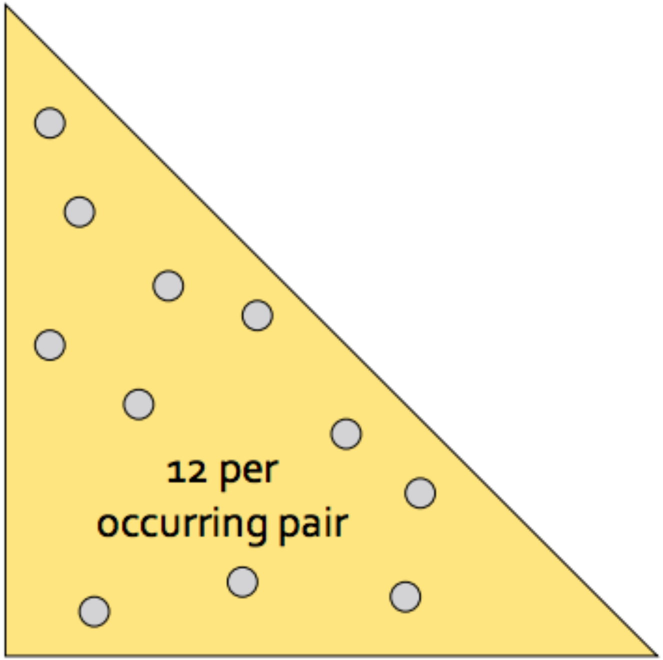
- For each basket b:
  - for each pair (i1,i2) in b:
    - increment count of (b1,b2)
- Still fail if (#items)^2 exceeds main memory
  - Walmart has 10^5 items
  - Counts are 4-byte integers
  - Number of pairs = 10^5*(10^5-1) /2 = 5 * 10^9
  - 2 * 10^10 bytes (20 GB) of memory needed

Not all pairs are equal

- Store a hash table
  - (i1, i2) => index
- Store triples [i1, i2, c(i1,i2)]
  - uses 12 bytes per pair
  - but only for pairs with count > 0
- Better if less than ⅓ of possible pairs actually occur



Triangular Matrix



Triples

Summary

- What?
  - Given a large set of baskets of items, find items that are correlated
- Why?
- How?
  - Find frequent itemsets
    - subsets that occur more than s times
  - Find association rules
    - $\text{Conf}(A \Rightarrow B) = \text{Support}(A,B) / \text{Support}(A)$

A-Priori Algorithm

Naive Algorithm Revisited

- Pros:
  - Read the entire file (transaction DB) once
- Cons
  - Fail if (#items)^2 exceeds main memory

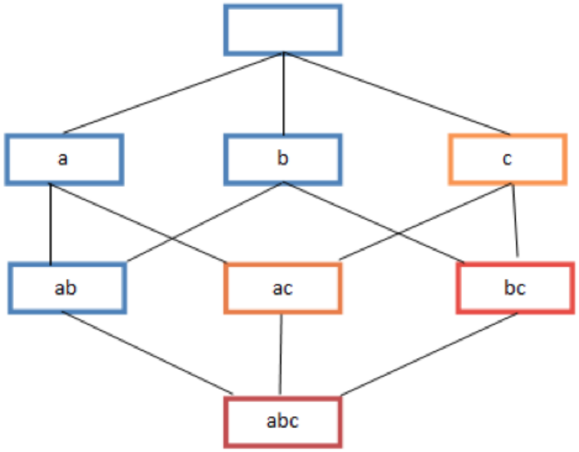
A-Priori Algorithm

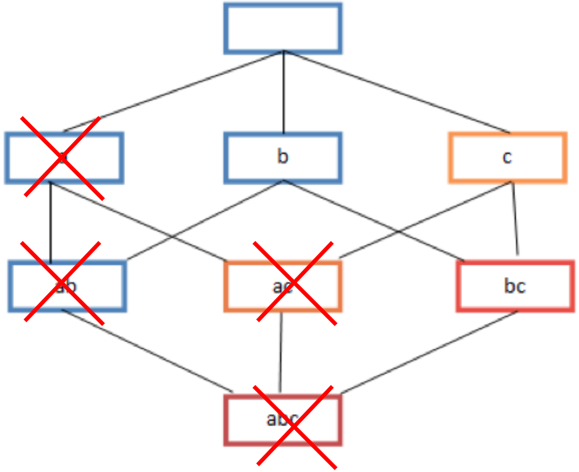
- Designed to reduce the number of pairs that need to be counted
- How?

  hint: There is no such thing as a free lunch
- Perform 2 passes over data

A-Priori Algorithm

- **Key idea : monotonicity**
  - If a set of items appears at least *s* times, so does every subset
- **Contrapositive for pairs**
  - If item *i* does not appear in *s* baskets, then no pair including *i* can appear in *s* baskets

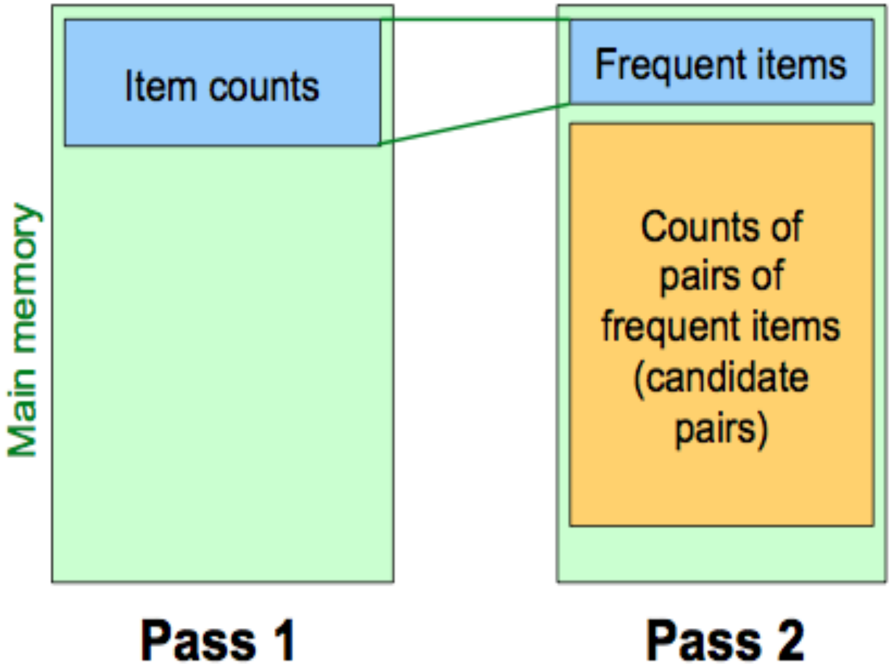




A-Priori Algorithm

- Pass 1:
  - Count the occurrences of each **individual item**
  - items that appear at least s time are the frequent items
- Pass 2:
  - Read baskets again and count in only those pairs where both elements are frequent (from pass 1)

A-Priori Algorithm

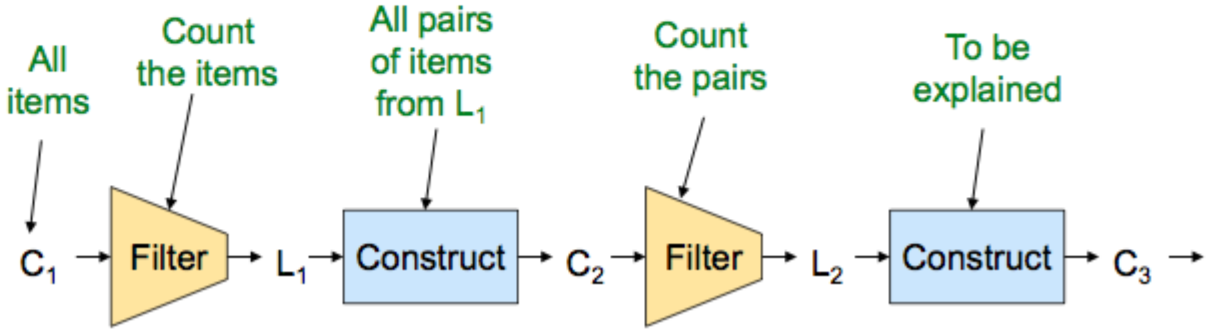


Frequent Tripes, Etc.

For each k, we construct two sets of *k-tuples*

C_k Candidate k-tuples = those might be frequent sets (support > s)

L_k The set of truly frequent k-tuples



Example

- **Hypothetical steps of the A-Priori algorithm**
  - C_1 = { {b} {c} {j} {m} {n} {p} }
  - Count the support of itemsets in C_1
  - Prune non-frequent: L_1 = { b, c, j, m }
  - Generate C_2 = { {b,c} {b,j} {b,m} {c,j} {c,m} {j,m} }
  - Count the support of itemsets in C_2
  - Prune non-frequent: L_2 = { {b,m} {b,c} {c,m} {c,j} }
  - Generate C_3 = { {b,c,m} {b,c,j} {b,m,j} {c,m,j} }
  - Count the support of itemsets in C_3
  - Prune non-frequent: L_3 = { {b,c,m} }

A-priori for All Frequent Itemsets

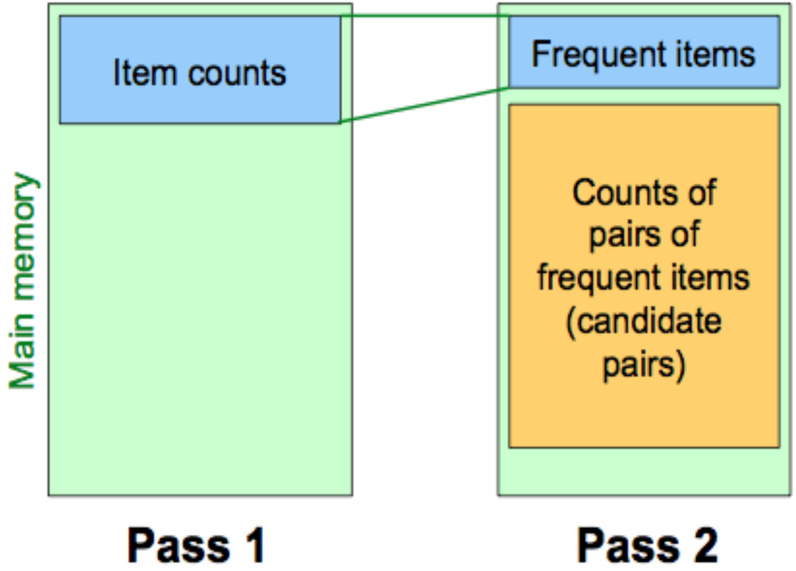
- For finding frequent k-tuple: Scan entire data k times
- Needs room in main memory to count each candidate k-tuple
- Typical, k = 2 requires the most memory

What else can we improve?

- Observation

In pass 1 of a-priori, most memory is idle !

Can we use the idle memory to reduce memory required in pass 2?



PCY Algorithm

- PCY (Park-Chen-Yu) Algorithm
- Take advantage of the idle memory in pass1
    - During pass 1, maintain a hash table
    - Keep a count for each bucket into which pairs of items are hashed

```
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

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;
```

New in PCY

Items A = {milk, coke, cookies, pepsi, juice},. Baskets are

B_1 = {m, c, b} B_2 = {m, p, j}
B_3 = {m, b} B_4 = {c, j}
B_5 = {m, p, b} B_6 = {m, c, b, j}
B_7 = {c, b, j} B_8 = {b, c}

Support threshold s=3,

If we assign milk=1, coke=2, cookies= 3, pepsi=4, and juice=5
Then,

B_1 = {1, 2, 3} B_2 = {1, 4, 5}
B_3 = {1, 3} B_4 = {2, 5}
B_5 = {1, 3, 4} B_6 = {1, 2, 3, 5}
B_7 = {2, 3, 5} B_8 = {2, 3}

Define the hash function: h(i, j) = (i + j) % 5 = K
(Hashing pair (i, j) to bucket K)

Pass 1

Item #	Count
1	5
2	5
3	5
4	2
5	4

The hash table is

Bucket #	Count
0	6
1	2
2	4
3	4
4	5

For s=3, L_1 = {1, 2, 3, 5}, and Bitmap {1, 0, 1, 1, 1}

Observations about Buckets

The hash table is

Bucket #	Count
0	6
1	2
2	4
3	4
4	5

For s=3, L_1 = {1, 2, 3, 5}

- If the count of a bucket is >= support s, it is called a **frequent bucket**
- For a bucket with total count less than s, none of its pairs can be frequent. Can be eliminated as candidates!
- For Pass 2, only count pairs that hash to frequent buckets

PCY Algorithm - Pass 2

- Count all pairs {i, j} that meet the conditions
  1. Both *i* and *j* are frequent items
  2. The pair {i, j} hashed to a frequent bucket (count >= s)
- All these conditions are necessary for the pair to have a chance of being frequent

PCY Algorithm - Pass 2

Hash table after pass 1:

The hash table is

Bucket #	Count
0	6
1	2
2	4
3	4
4	5

For s=3, L_1 = {1, 2, 3, 5}, and Bitmap {1, 0, 1, 1, 1}

Pass 2

Frequent items are {1, 2, 3, 5}

Candidate pairs and their counts

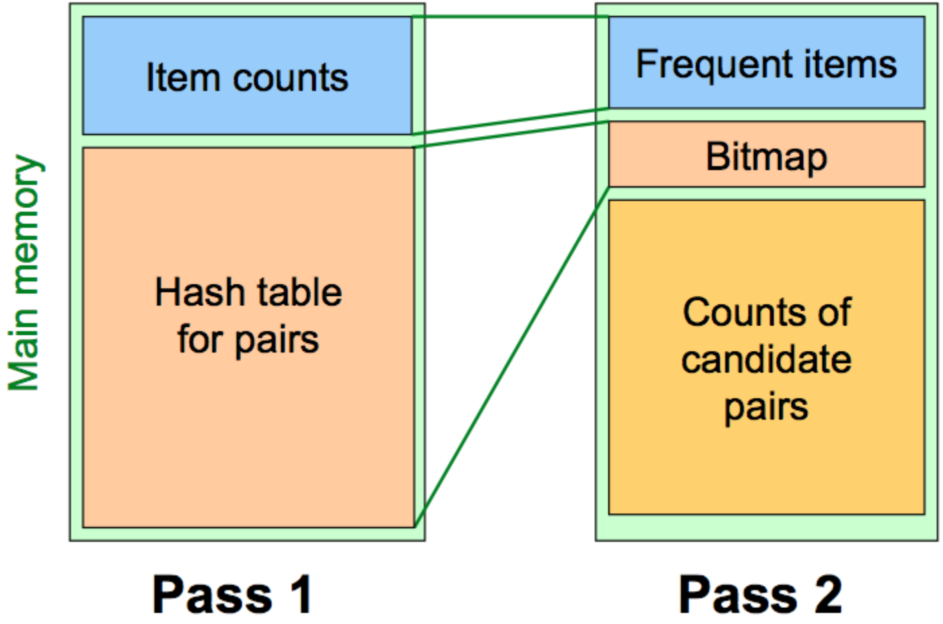
(1, 4), (2, 3)➔h(i, j)= 0
(1, 5), (2, 4)➔h(i, j)=1
(2, 5), (3, 4)➔h(i, j)= 2
(1, 2), (3, 5)➔h(i, j)= 3
(1, 3), (4, 5)➔h(i, j)= 4

Pair	Count
(2,3)	4
(2,5)	3
(1,2)	2
(3,5)	2
(1,3)	4

frequent itemsets are

{1}, {2}, {3}, {5}, {1, 3}, {2, 3}, {2, 5}

Main-Memory: Picture of PCY

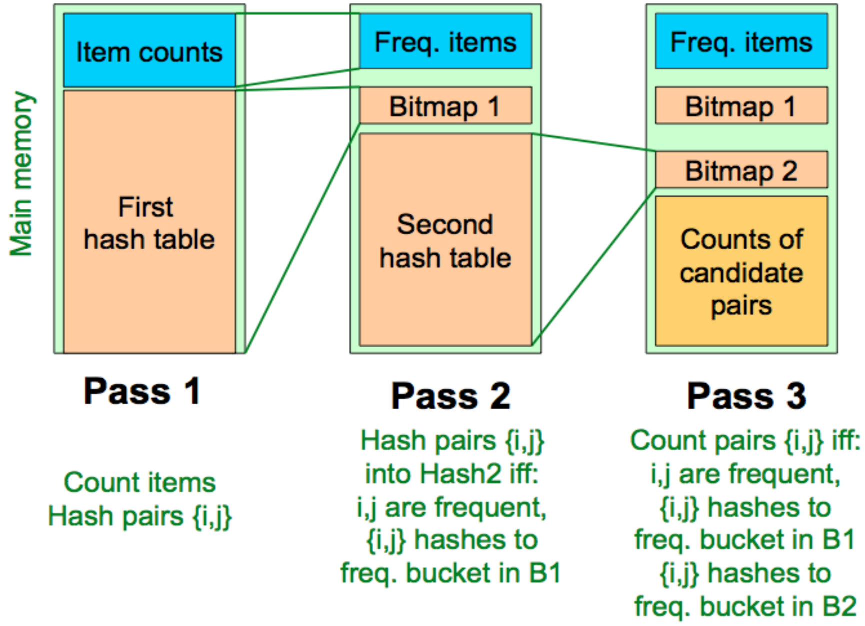


Refinement

- Remember: Memory is the **bottleneck**!
- Can we further limit the number of candidates to be counted?
- Refinement for PCY Algorithm
  - Multistage
  - Multihash

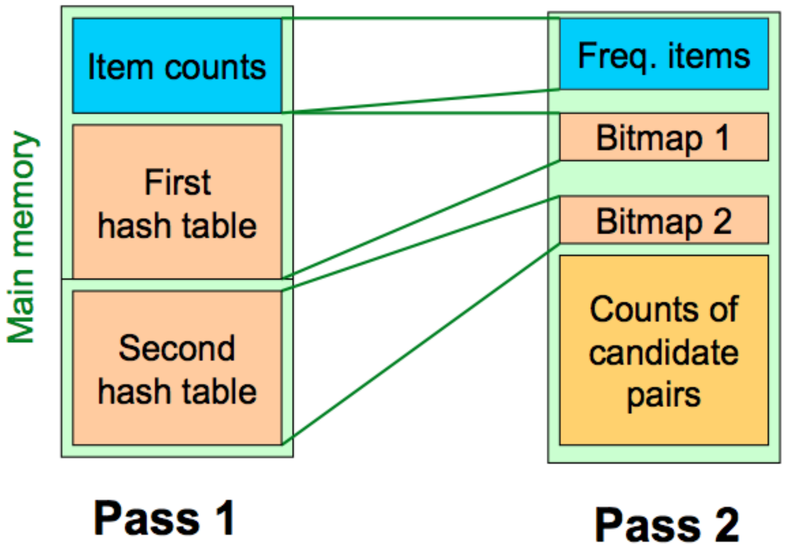
Multistage Algorithm

- Key Idea: After Pass 1 of PCY, rehash only those pairs that qualify for pass 2 of PCY
- Require additional pass over the data
- Important points
  - Two hash functions have to be independent
  - Check both hashes on the third pass



Multihash Algorithm

- Key Idea: Use several independent hash functions on the first pass
- Risk: Halving the number of buckets **doubles** the average count
- If most buckest still not reach count s, then we can get a benefit like multistage, but in only 2 passes!
- Possible candidate pairs {i, j}:
  - i, j are frequent items
  - {i, j} are hashed into both frequent buckets

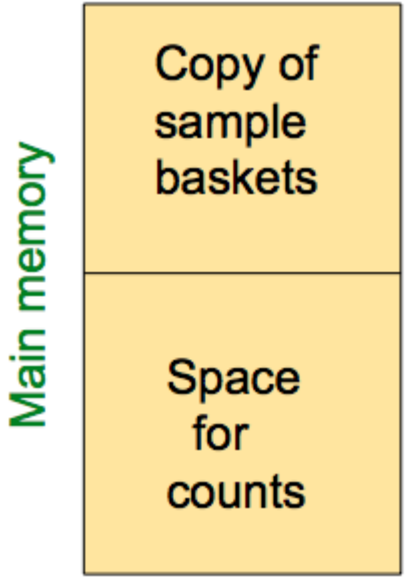


Frequent Itemsets in <= 2 Passes

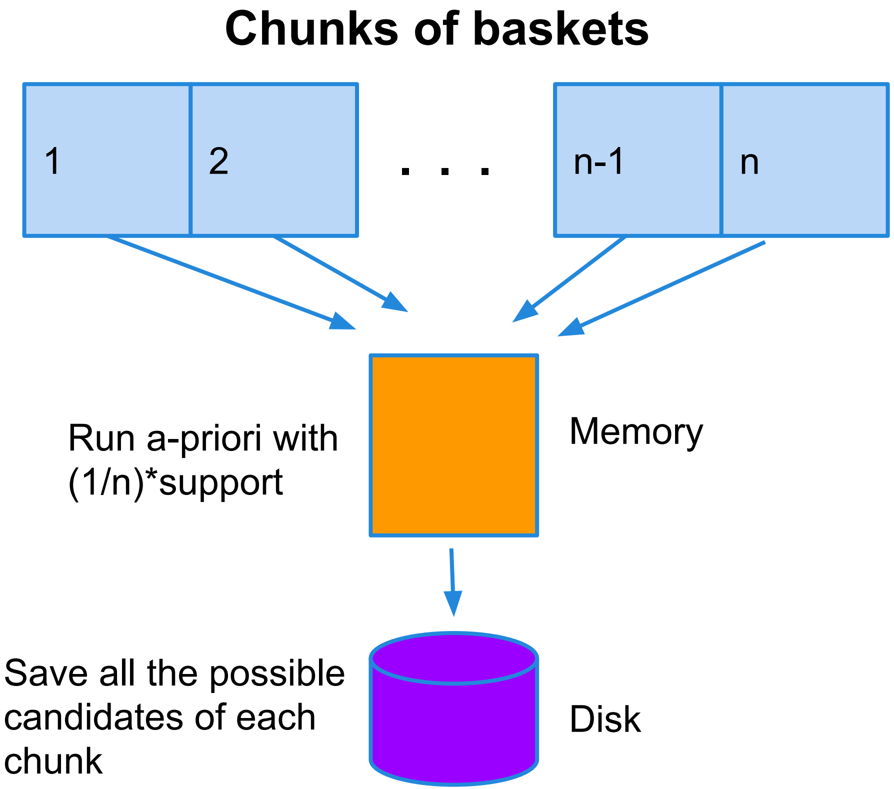
- A-Priori, PCY, etc., take k passes to find frequent itemsets of size k
- Can we use fewer passes?
- Use 2 or fewer passes for all sizes
  - Random sampling
    - may miss some frequent itemsets
  - SON (Savasere, Omiecinski, and Navathe)
  - Toivonen (not going to conver)

Random Sampling

- Take a random sample of the market baskets
- Run A-priori in **main memory**
  - Don’t have to pay for disk I/O each time we read over the data
  - Reduce the support threshold proportionally to match the sample size (e.g. 1% of Data, support => 1/100*s)
- Verify the candidate pairs by a second pass



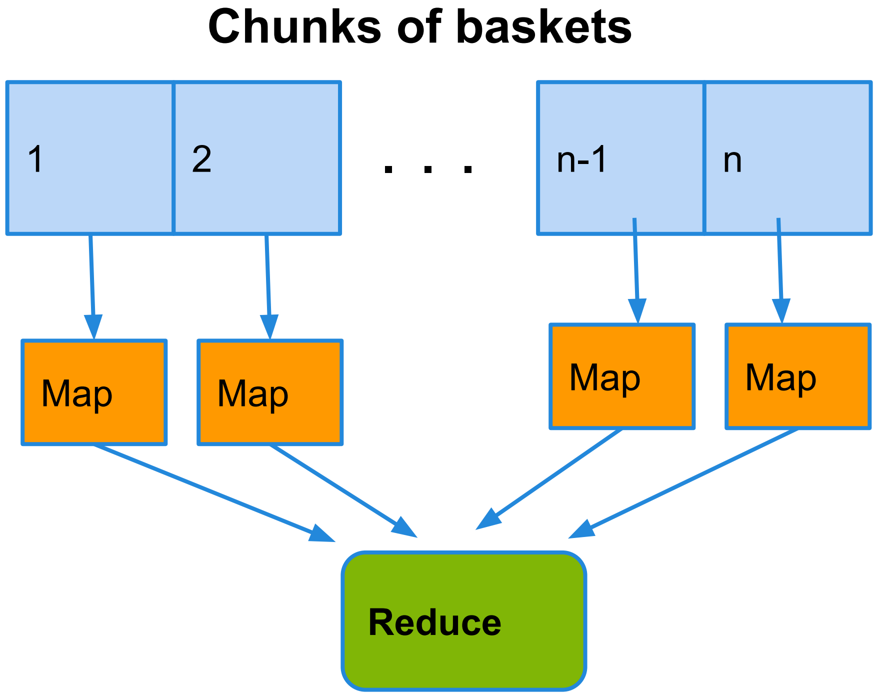
SON Algorithm



- Repeatedly read small subsets of the baskets into main memory and run an in-memory algorithm to find all frequent itemsets
- Possible candidates:
  - Union all the frequent itemsets found in each chunk
  - why? “monotonicity” idea: an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset
- On a **second pass**, count all the candidate

SON Algorithm- Distributed Version

- **MapReduce** for Pass 1



- Distributed data mining
- Pass 1: Find candidate itemsets
  - Map: (F,1)
    - F : frequent itemset
  - Reduce: Union all the (F,1)
- Pass 2: Find true frequent itemsets
  - Map: (C,v)
    - C : possible candidate
  - Reduce: Add all the (C, v)

FP-Growth Approach

Introduction

- A-priori
  - Generation of candidate itemset (Expensive in both space and time)
  - Support counting is expensive
    - Subset checking
    - Multiple Database scans (I/O)

FP-Growth approach

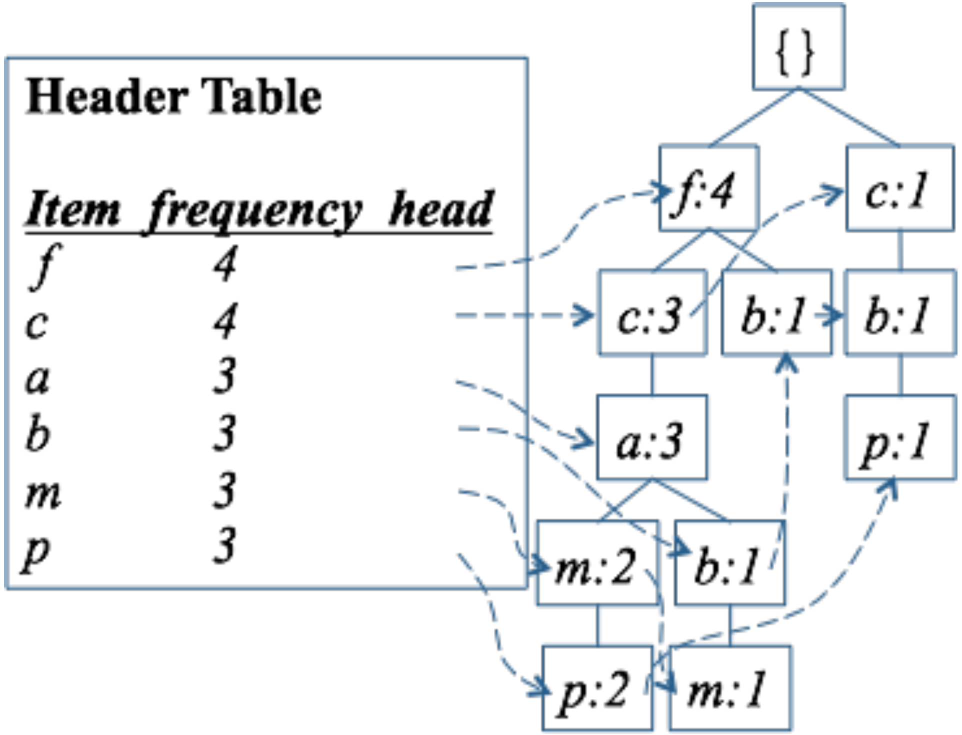
- FP-Growth (Frequent Pattern-Growth)
  - Mining in main memory to reduce (#DBscans)
  - Without candidate itemsets generation
- Two step approach
  - Step 1: Build a compact data structure called the FP-tree
  - Step 2: Extracts frequent itemsets directly from the FP-tree (Traversal through FP-tree)

FP-Tree construction

- FP-Tree construction
  - Pass 1:
    - Find the frequent items
  - Pass 2:
    - Construct FP-Tree

f	4
c	4
a	3
b	3
m	3
p	3

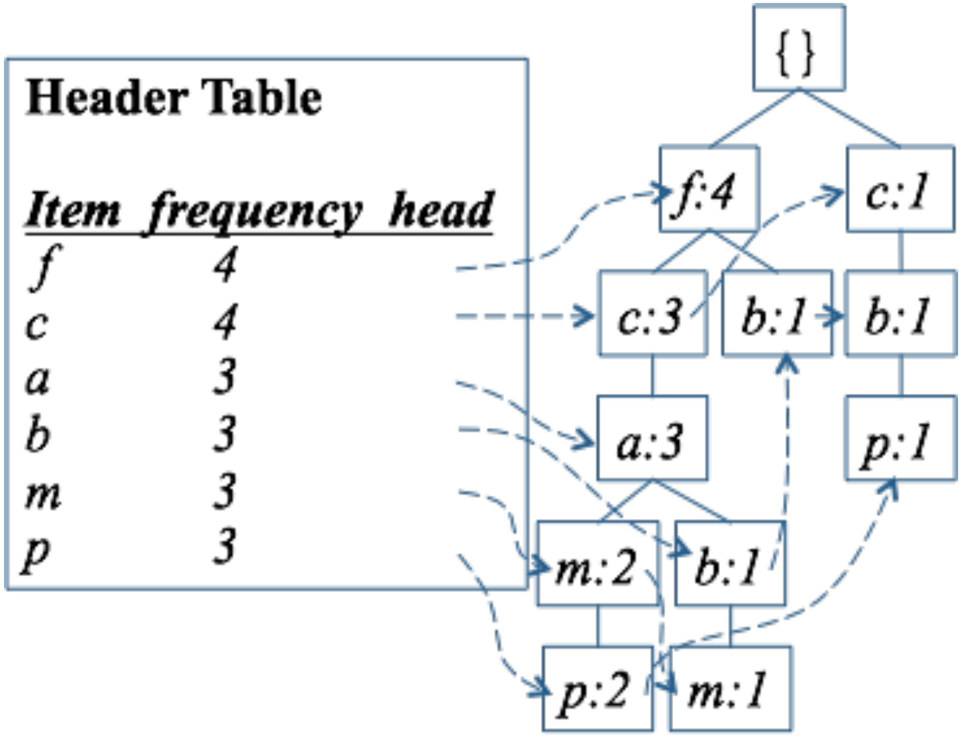
TID	Items bought	Ordered
100	{a, c, d, f, g, i, m, p}	{f, c, a, m, p}
200	{a, b, c, f, i, m, o}	{f, c, a, b, m}
300	{b, f, h, j, o}	{f, b}
400	{b, c, k, s, p}	{c, b, p}
500	{a, c, e, f, l, m, n, p}	{f, c, a, m, p}



FP-Tree

- FP-Tree
  - Prefix Tree
  - Has a much smaller size than the uncompressed data
  - Mining in main memory
- How to find the Frequent itemset?
  - Tree traversal
  - Bottom-up algorithm
    - Divide and conquer
  - More detail:

    http://csc.lsu.edu/~jianhua/FPGrowth.pdf



FP-Growth V.S A-priori

	Apriori	FP-Growth
# Passes over data	depends	2
Candidate Generation	Yes	No

- FP-Growth Pros:
  - “Compresses” data-set, mining in memory
  - much faster than Apriori
- FP-Growth Cons:
  - FP-Tree may not fit in memory
  - FP-Tree is expensive to build
    - Trade-off: takes time to build, but once it is build, frequent itemsets are read off easily

Acknowledgements

- Stanford CS246: Mining Masive Datasets (Jure Leskovec)
- Mining of Massive Datasets (Anand Rajaraman, Jeffrey Ullman)
- Introduction to Frequent Pattern Growth (FP-Growth) Algorithm (Florian Verhein)
- NCCU: Data-mining (Man-Kwan Shan)
- Mining frequent patterns without candidate generation. A frequent-tree approach, SIGMOD '00 Proceedings of the 2000