Hadoop implementation for algorithms (A-Priori, PCY & SON) working on Frequent Itemsets

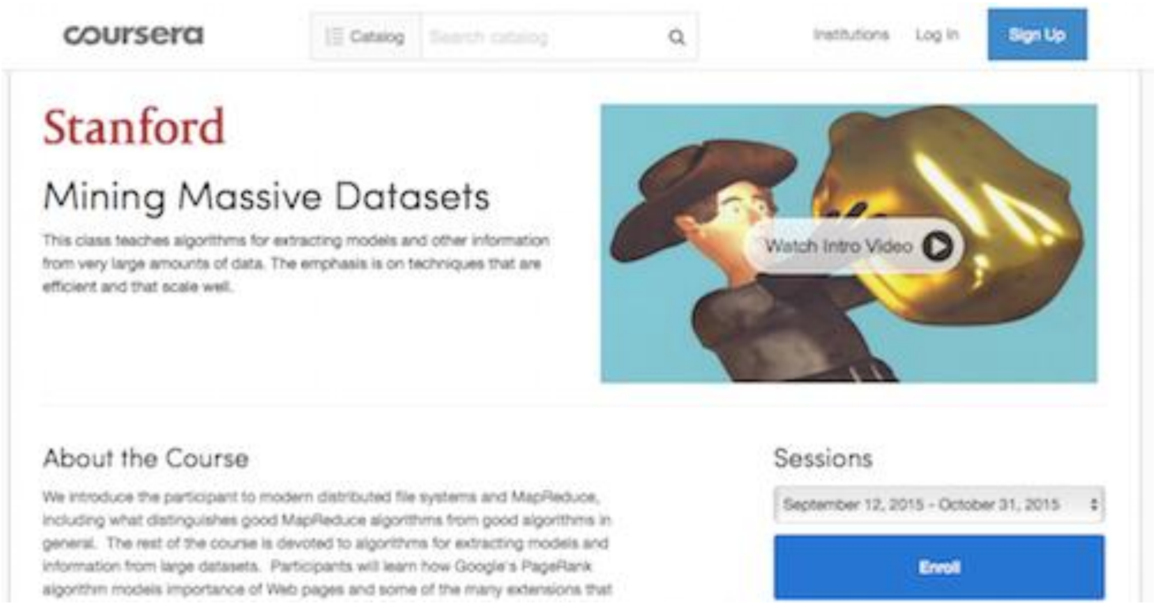
Chengeng Ma

Stony Brook University

2016/03/26

"Diapers and Beer"

- If you are familiar with Frequent Itemsets, please directly go to page 13, because the following pages are fundamental knowledge and methods about this topic, which you can find more details in the book, *Mining of Massive Dataset*, written by Jure Leskovec, Anand Rajaramanand Jeffrey D. Ullman.



Stanford

Mining Massive Datasets

This class teaches algorithms for extracting models and other information from very large amounts of data. The emphasis is on techniques that are efficient and that scale well.

About the Course

We introduce the participant to modern distributed file systems and MapReduce, including what distinguishes good MapReduce algorithms from good algorithms in general. The rest of the course is devoted to algorithms for extracting models and information from large datasets. Participants will learn how Google's PageRank algorithm models importance of Web pages and some of the many extensions that

Sessions

September 12, 2015 - October 31, 2015

1. Fundamentals about frequent itemsets

Frequent itemsets is about things frequently bought together

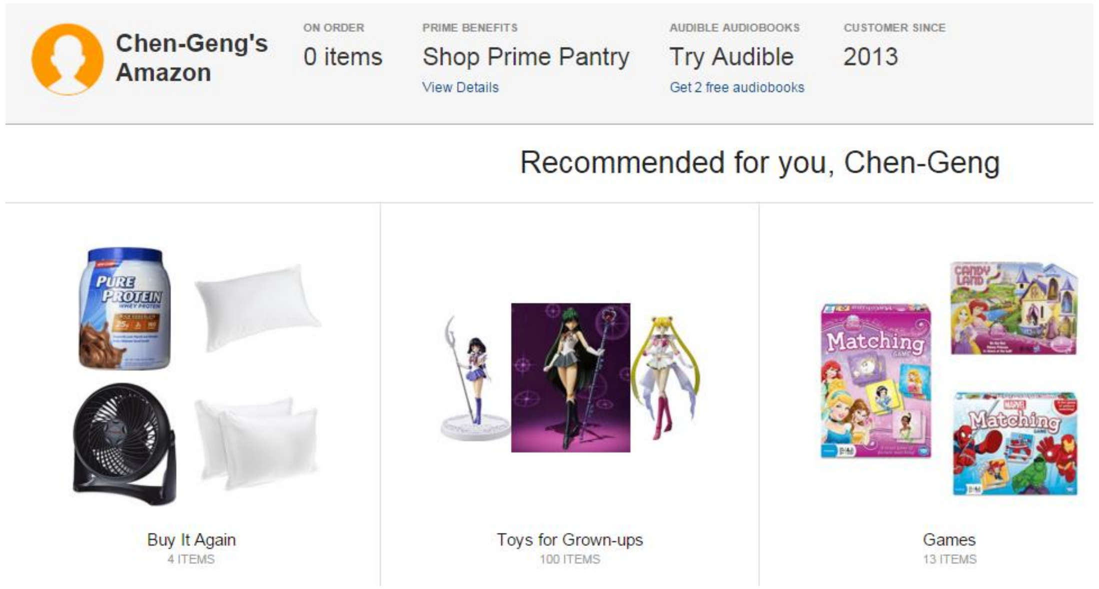
- For large supermarkets, e.g., *Walmart, Stop & Shop, Carrefour …,* finding out what stuffs are usually bought together not only can better serve their customers, but also is very important for their selling strategies.
- For example, people buy hot dogs & mustard together. *Then a supermarket can offer a sale on hot dog but raise the price of mustard.*

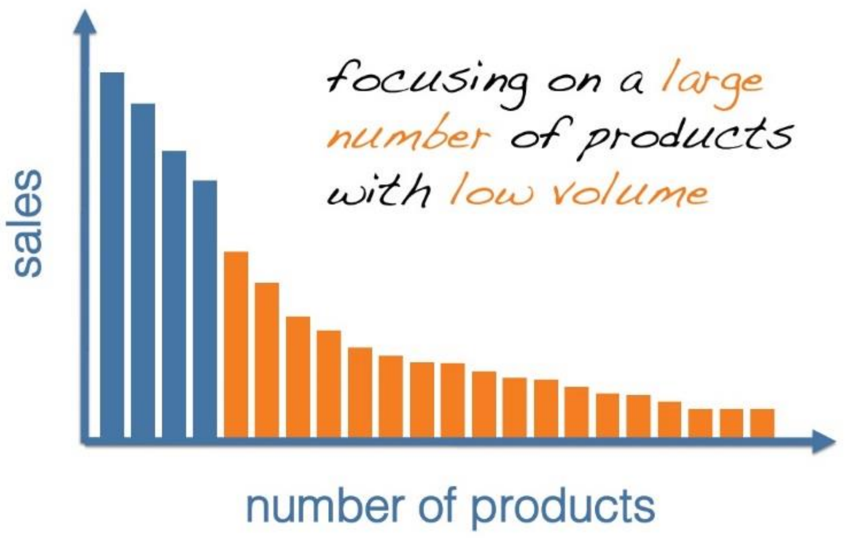
Except for marketing strategies, it is also widely used for other domains, e.g.,

- *Related concepts*, finding a set of words frequently appears in a lot of documents, blogs, Tweets;
- *Plagiarism checking*, finding a set of papers that shares a lot of sentences;
- *Biomarkers*, finding a set of biomarkers that frequently appear together for a specific disease.

Why and Who studies frequent itemsets?

- Online shops like Amazon, Ebay, …, can recommend stuffs to all their customers, as long as the computation resources allow.
- They not only can learn what stuffs are popular;
- but also study customers' personal interests and recommend individually.
- So they want to find similar customers and similar products.
- This opportunity of online shops is called *long tail effect*.





Why and Who studies frequent itemsets?

- The brick-and-mortar retailers, however, have limited resources.
- When they advertise a sale, they spend money and space on the advertised stuff.
- They cannot recommend to their customers individually, so they have to focus on the most popular goods.
- So they want to focus on frequently bought stuffs and frequent itemsets.

What is the difficulty if the work is just counting?

- Suppose *N* different products, if you want to find frequent single items, you just need a hash table of *N* key value pairs.
- If you want to find frequent pairs then the size of the hash table becomes *N(N-1)/2*.
- If you want to find frequent length-k-itemsets, you need C_N^k key value pairs to store your counts.

$$\boxed{C(n,k) = \frac{n!}{(n-k)!k!}}$$

- *When k gets large, it grows as N^k/k, it's crazy!*

Who is the angel to save us?

- Monotonicity of items: *if a set I is frequent, then all the subsets of I is frequent.*
- *If a set J is not frequent, then all the sets that contains J cannot be frequent.*

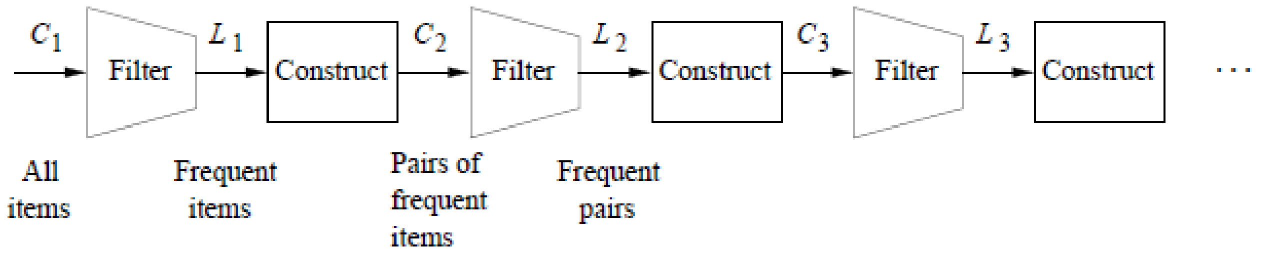


Figure 6.4: The A-Priori Algorithm alternates between constructing candidate sets and filtering to find those that are truly frequent

- *Ck: set of candidate itemsets of length k*
- *Lk: set of frequent itemsets of length k*

- In the real world, reading all the single items usually does not stress main memory too much. Even a giant supermarket Corp. cannot sell more than one million stuffs.
- Since *the candidates of frequent triples (C3) are based on frequent pairs (L2) instead of all possible pairs (C2)*, the memory stress for finding frequent triples and higher length itemsets is not that serious.
- *Finally, the largest memory stress happens on finding frequent pairs.*

Classic A-Priori: takes 2 passes of whole data

- 1st pass holds a hash table for the counts of all the single items.
- After the 1st pass, the frequent items are found out,
- now you can use a BitSet to replace the hash table by setting the frequent items to 1 and leaving others to 0;
- or you can only store the indexes of frequent items and ignore others.

- 2nd pass will take the 1st pass' results as an input besides of reading through the whole data second time.

- During 2nd pass, you start from an empty hash table, whose key is pair of items and value is its count.
- For each transaction basket, exclude infrequent items,
- within the frequent items, form all possible pairs, and add *1* to the count of that pair.

- After this pass, filter out the infrequent pairs.

A-Priori's shortages

- 1. During the 1st pass of data, there are a lot of memory unused.
- 2. The candidates of frequent pairs (C2) can be still a lot if the dataset is too large.
- 3. *Counting is a process that can be map-reduced. You do not need a hash table to really store everything during the 2nd pass.*

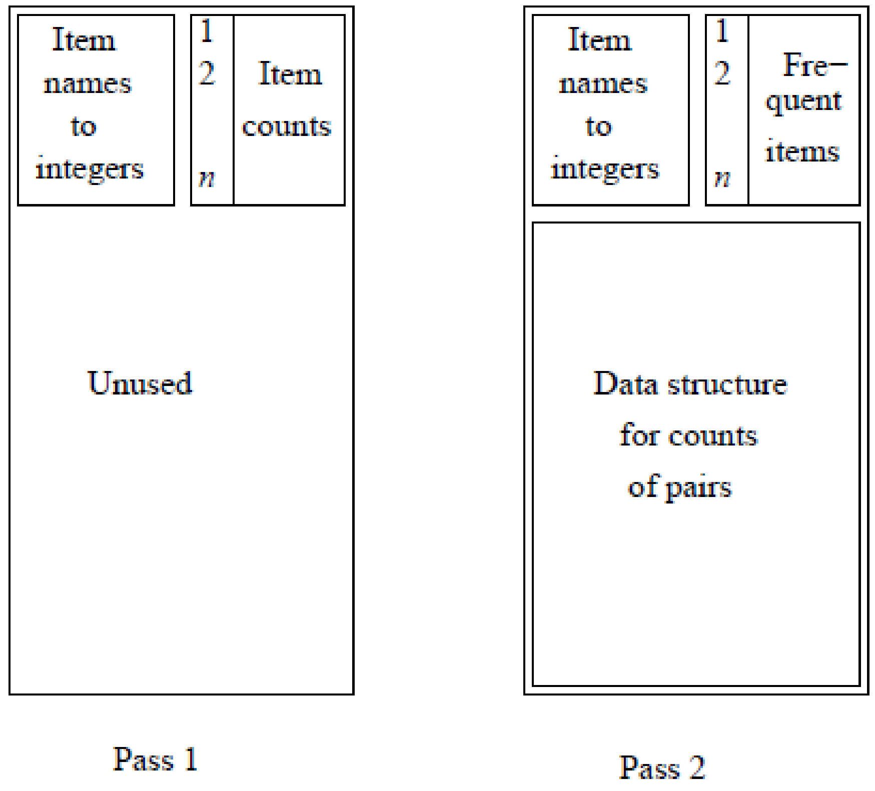


Figure 6.3: Schematic of main-memory use during the two passes of the A-Priori Algorithm

Some improvements exists, *PCY, multi-hash, multi-stage*, …

PCY (The algorithm of Park, Chen and Yu) makes use of the unused memory during the 1st pass

- During the 1st pass, we creates 2 empty hash tables, the 1st is for counting single items, the 2nd is for hashing pairs.
- When processing each transaction basket, you not only count for the singletons,
- But also generate all the pairs within this basket, and hash each pair to a bucket of the 2nd hash table by adding 1 to that bucket.

- So the 1st hash table is simple, it must has *N* keys and values, like: *hashTable1st(item ID)=count*
- For the 2nd hash table, its size (how many buckets) is depended on the specific problem.
- Its key is just a bucket position and has no other meanings, its value represents how many pairs are hashed onto that bucket, like: *hashTable2nd(bucket pos)=count*

The 2nd pass of PCY algorithm

- After the 1st pass, in the 1st hash table, you can find frequent singletons,
- In the 2nd hash table, you can find the frequent buckets,
- Replace the 2 hash tables to 2 BitSet, or just discard the not-frequent items and bucket positions and save the remaining into 2 files separately.

- The 2nd pass will read in the 2 files generated from the 1st pass, besides of reading through the whole data.

- During the 2nd pass, for each transaction basket, exclude items that are infrequent,
- within the frequent items, generate all possible pairs,
- for each pair, hash it to a bucket (same hash function as 1st pass),
- if that bucket is frequent according to 1st pass, then add 1 to the count of that pair.

- After this pass, filter out the infrequent pairs.

Why hashing works?

- If a bucket is infrequent, then all the pairs hashed into that bucket cannot be frequent, because their sum of counts is less than *s (threshold)*.
- If a bucket is frequent, then the sum of counts of pairs hashed into that bucket is larger than *s*. Then some pairs inside that bucket can be frequent, some can be not, or all of them are not frequent individually.
- Note we use deterministic hash function not randomly hashing.

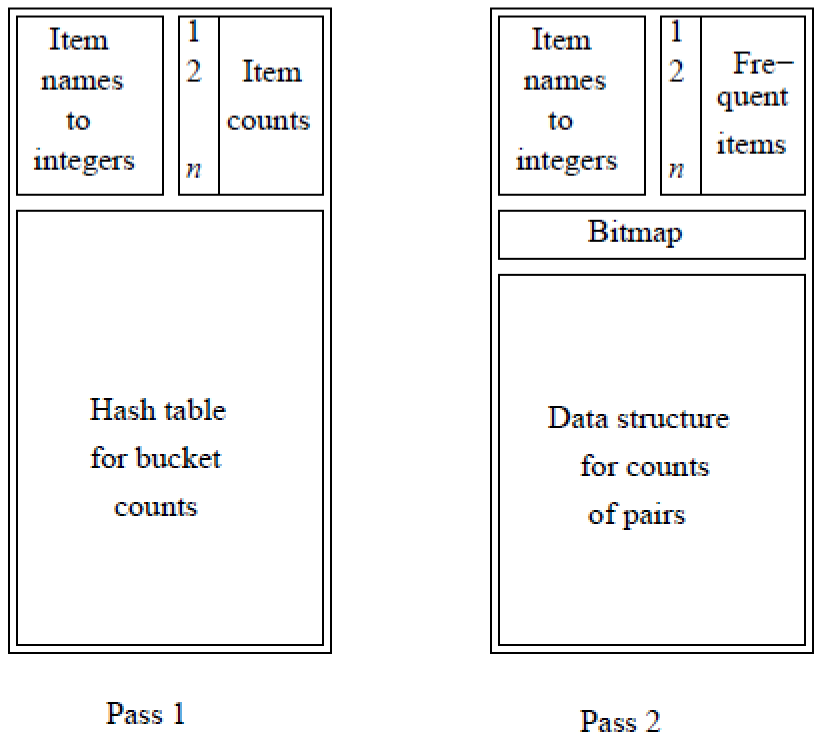


Figure 6.5: Organization of main memory for the first two passes of the PCY Algorithm

- If you use multiple hash functions in 1st pass, it's called multi-hash algorithm.
- If you add one more pass and use another hash function in 2nd pass, it's called multi-stage algorithm.

Multistage takes more than two passes to find the frequent pairs. An outline of the Multistage Algorithm is shown in Fig. 6.6.

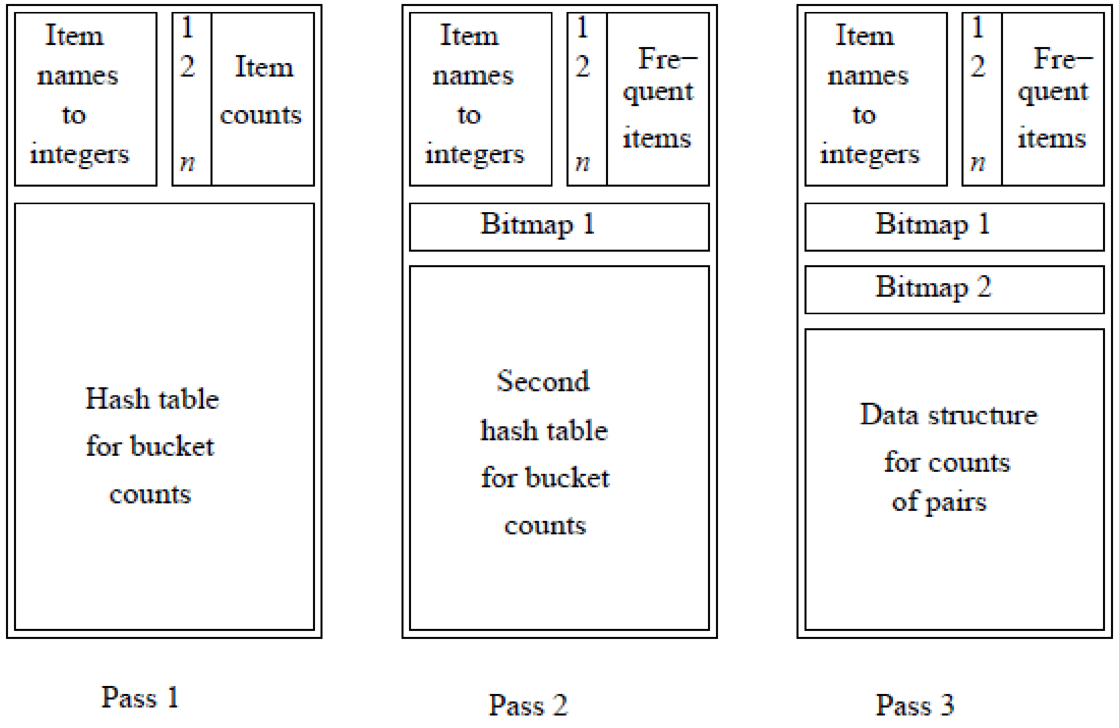


Figure 6.6: The Multistage Algorithm uses additional hash tables to reduce the number of candidate pairs

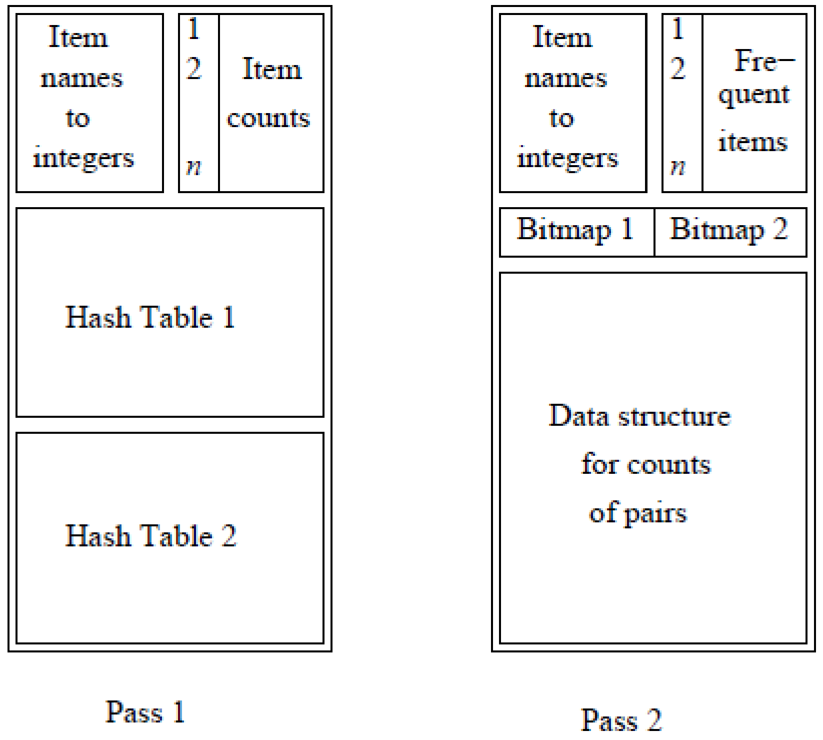


Figure 6.7: The Multihash Algorithm uses several hash tables in one pass

- If you add one more pass and use another hash function in 2nd pass, it’s called multi-stage algorithm.

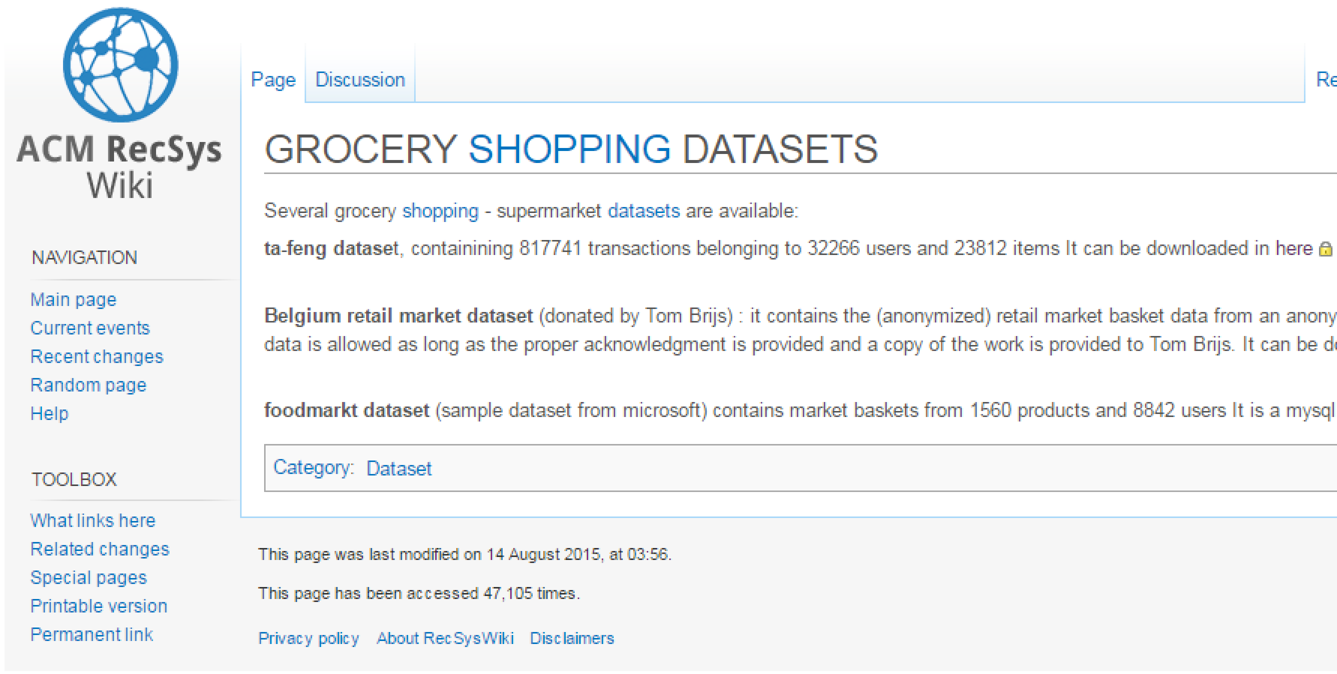
- If you use multiple hash functions in 1st pass, it’s called multi-hash algorithm.

How many buckets do you need for hashing?

- Suppose totally there are *T* transactions, each of which contains averagely *A* items, the threshold for becoming frequent is *s* and the number of buckets you create for PCY hashing is *B*.
- Then approximately there are $T\frac{A(A-1)}{2}$ possible pairs.
- Each bucket will get $\frac{TA^2}{2B}$ counts in average.
- If $(\frac{TA^2}{2B})$ is around *s*, then only few buckets are infrequent, then we do not gain much performance by hashing.
- But if $(\frac{TA^2}{2B})$ is much smaller than *s (<0.1)*, then only few buckets can get frequent. So we can filter out a lot of infrequent candidates before we really count on them.

2. Details of my class project: dataset

- *Ta-Feng Grocery Dataset* for all transactions during *Nov 2000* to *Feb 2001.*
- *Ta-Feng* is a membership retailer warehouse in *Taiwan* which sells mostly food-based products but also office supplies and furniture (*Hsu et. al, 2004*).
- The dataset contains *119,578* transactions, which sum up to *817,741* single item selling records, with *32,266* different customers and *23,812* unique products.
- Its size is about 52.7 MB. But as a training, I will use *Hadoop* to find the frequent pairs.



GROCERY SHOPPING DATASETS

Several grocery shopping - supermarket datasets are available:

ta-feng dataset, containining 817741 transactions belonging to 32266 users and 23812 items It can be downloaded in here

Belgium retail market dataset (donated by Tom Brijs) : it contains the (anonymized) retail market basket data from an anony
data is allowed as long as the proper acknowledgment is provided and a copy of the work is provided to Tom Brijs. It can be d

foodmarkt dataset (sample dataset from microsoft) contains market baskets from 1560 products and 8842 users It is a mysql

Category: Dataset

This page was last modified on 14 August 2015, at 03:56.

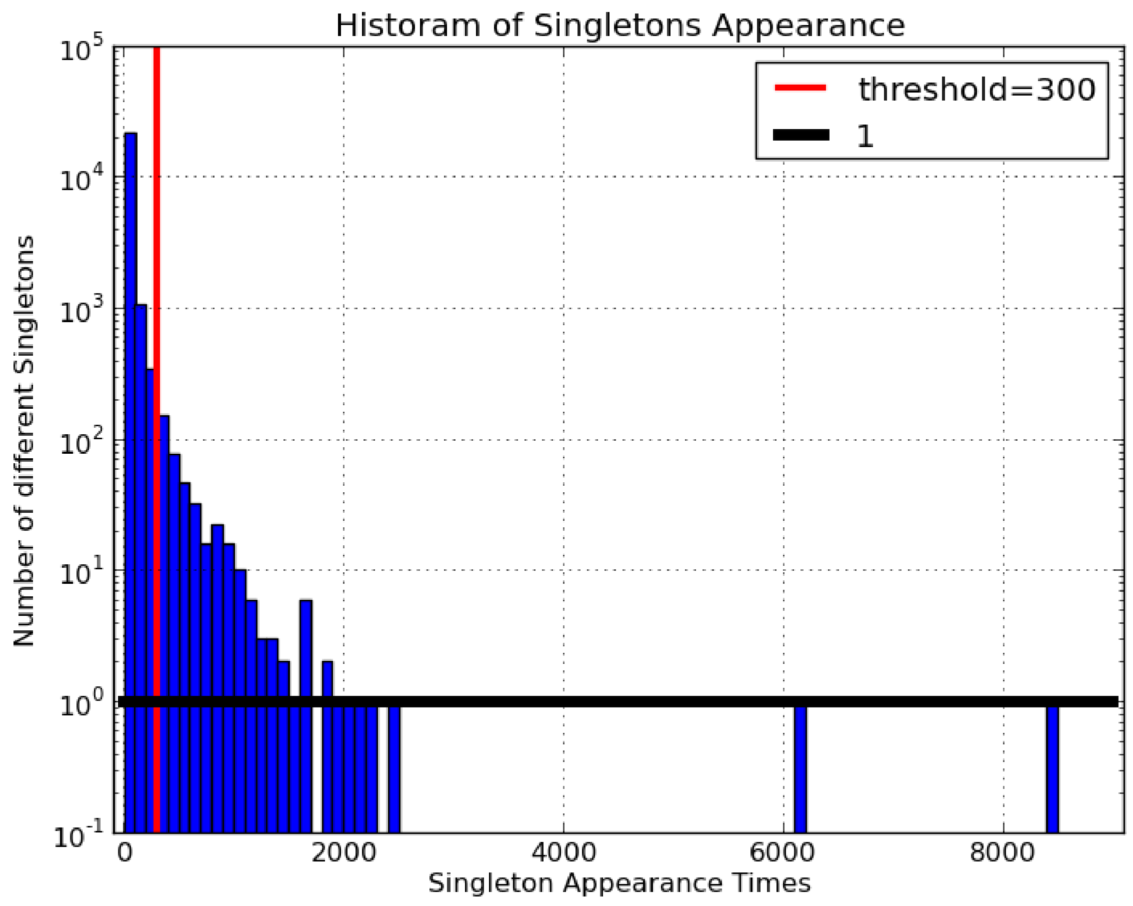
This page has been accessed 47,105 times.

Privacy policy About RecSysWiki Disclaimers

- It can be download in *http://recsyswiki.com/wiki/Grocery_shopping_datasets*

Prepare work

- The product ID originally is in 13 digits, we re-indexing it within [0, 23811].
- Usually the threshold *s* is set as *1%* of transaction numbers.
- However, this dataset is not large, we cannot set too high threshold. Finally *0.25%* of transaction numbers is used as threshold, which is *300*.
- Only *403 frequent singletons* (from *23,812* unique products)
- For PCY, since we derive $\frac{TA^2}{2B} \ll s$, so $B \gg \frac{TA^2}{2s} = \frac{119578*7^2}{2*300} = 9765$. We set *B* as a prime number *999,983 (=10^6-17).*



- For the hash function, I use the following:
- *H(x, y)={ (x + y)*(x + y + 1) / 2 + max(x, y) } mod B*

Mapreduced A-Priori: Hadoop is good at counting

- Suppose you already have *frequent singletons*, read from Distributed Cache and stored in a BitSet as 1/0.

- Mapper input: $\{i, [\ M_{i1}, M_{i2}, \ldots, M_{iS_i}\ ]\}$, as one transaction, where i is the customer ID, S_i items are bought.
- Within $[M_{i1}, M_{i2} \ldots, M_{iS_i}]$, exclude infrequent items, only frequent items remains as $[F_{i1}, F_{i2} \ldots, F_{if_i}\ ]$, where $f_i \leq S_i$.
-
```
$for\ x \in [F_{i1}, F_{i2} \ldots, F_{if_i}\ ]$:
    $for\ y \in [F_{i1}, F_{i2} \ldots, F_{if_i}\ ]$:
        if x < y: output {(x, y), 1}
```

- Reducer input: {(x, y), [a, b, c, …]};
- Take the sum of the list as t;
- *if t>=s: Output {(x, y), t}.*

- Combiner is the same as the reducer, except only taking the sum but not filtering infrequent pairs.

- Original A-Priori needs a data structure to store all candidate pairs in memory to do the counting.
- Here we deal with each transaction individually. Hadoop will help us gather each pair's count without taking large memory.

Mapreduced PCY 1st pass

- Mapper input: $\{i, [M_{i1}, M_{i2}, \ldots, M_{iS_i}]\}$, as one transaction, where i is the customer ID, S_i items are bought.

```
• for item ∈ [$M_{i1}, M_{i2} \ldots, M_{iS_i}$]:
        Output {(item, "single"), 1}
• for x ∈ [$M_{i1}, M_{i2} \ldots, M_{iS_i}$]:
      for y ∈ [$M_{i1}, M_{i2} \ldots, M_{iS_i}$]:
          If x < y: Output {(hash(x, y), "hash"), 1}
```

- Reducer input : {(item, "single"), [a, b, c, …]} or {(hashValue, "hash"), [d, e, f, …]}, but cannot contain both;
- Take the sum of the list as t;
- *If t>=s: Output {key, t}.*

(where the key can be (item, "single") or (hashValue, "hash")).

Combiner is the same as the reducer, except only taking the sum but not filtering infrequent pairs.

Mapreduced PCY 2nd Pass: counting

- Suppose you already have *frequent singletons and frequent buckets*, read from Distributed Cache and stored in 2 BitSets as 1/0.
- Mapper input: $\{i, [\, M_{i1}, M_{i2}, \ldots, M_{iS_i} \,]\}$, as one transaction, where i is the customer ID, S_i items are bought.
- Within $[M_{i1}, M_{i2} \ldots, M_{iS_i}]$, exclude infrequent items, only frequent items remains as $[F_{i1}, F_{i2} \ldots, F_{if_i}]$, where $f_i \leq S_i$.
-
```
for $x \in [F_{i1}, F_{i2} \ldots, F_{if_i}]$:
    for $y \in [F_{i1}, F_{i2} \ldots, F_{if_i}]$:
        if (x < y) & hash(x, y)∈ frequent buckets:  output {(x, y), 1}
```

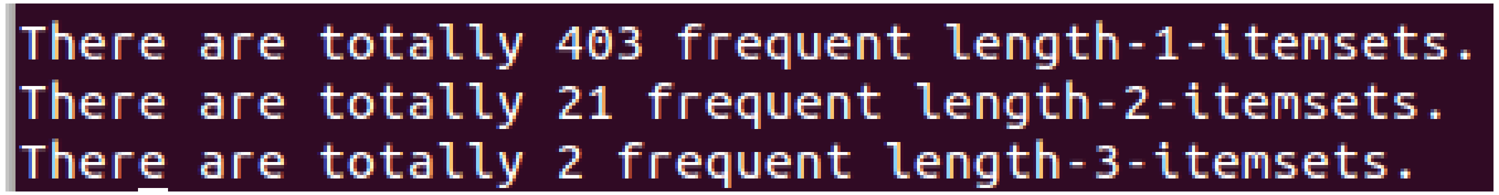
- Reducer input: {(x, y), [a, b, c, …]};
- Take the sum of the list as t;
- *If t>=s: Output {(x, y), t}.*

- Combiner is the same as the reducer, except only taking the sum but not filtering infrequent pairs.

- As we said, Hadoop will help us gather each pair's count without stressing memory. You don't need to store everything in memory.

Higher order A-Priori

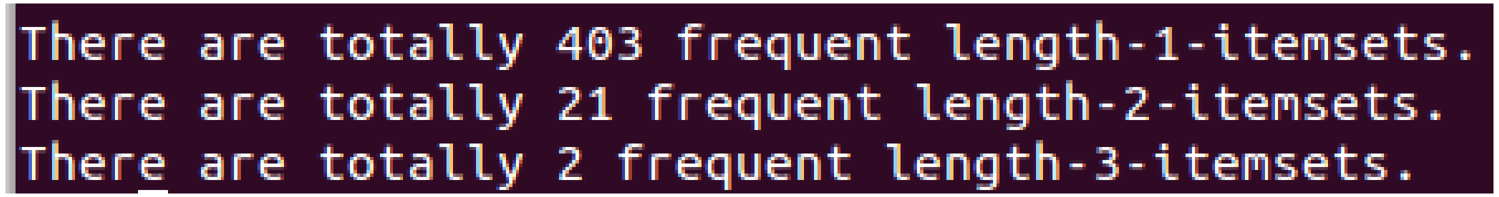
- Now suppose you have a set of frequent length-*k*-itemsets *L(k)*, how to find *L(k+1)*?
- Create an empty hash table *C*, whose key is length-*k*-itemset, and value is count.
- Within *L(k)* using double loops to form all possible itemset pairs,
- for each pair *(x, y),* take the union $z = x \cup y$,
- If the length of z is k+1, then add 1 to the count of z, i.e., C[z]+=1.
- Finally, if the count of z is less than k+1, then z does not have k+1 frequent length-k-subsets, delete z from C.
- The remaining itemsets in C are candidate itemsets.
- Now going through the whole dataset, count for each candidate itemsets and filter out the infrequent candidates.



```
There are totally 403 frequent length-1-itemsets.
There are totally 21 frequent length-2-itemsets.
There are totally 2 frequent length-3-itemsets.
```

Frequent pairs and triples and their count

(11315,16786)	600	(5116,5117)	359
(19405,19406)	351	(6041,6049)	473
(4831,4832)	700	(6041,6068)	447
(4831,4839)	620	(6049,6068)	520
(4831,4840)	580	(6050,6051)	414
(4831,4844)	531	(6052,6053)	351
(4832,4839)	439	(8767,11315)	413
(4832,4840)	365	(9989,11315)	306
(4832,4844)	405		
(4834,4835)	338		
(4839,4840)	379	(4831, 4832, 4839)	339
(4839,4844)	357	(4831, 4832, 4844)	325
(4840,4844)	314		



```
There are totally 403 frequent length-1-itemsets.
There are totally 21 frequent length-2-itemsets.
There are totally 2 frequent length-3-itemsets.
```

- The number of frequent itemsets drops exponentially as its length *(k)* grows.
- Within 18 out of 21 frequent pairs, the two product IDs are close to each other, like (4831, 4839), (6041, 6068), ….

Association rules: $Confidence(I \rightarrow J) = \frac{support(I \cup J)}{support(I)} = P(J|I)$

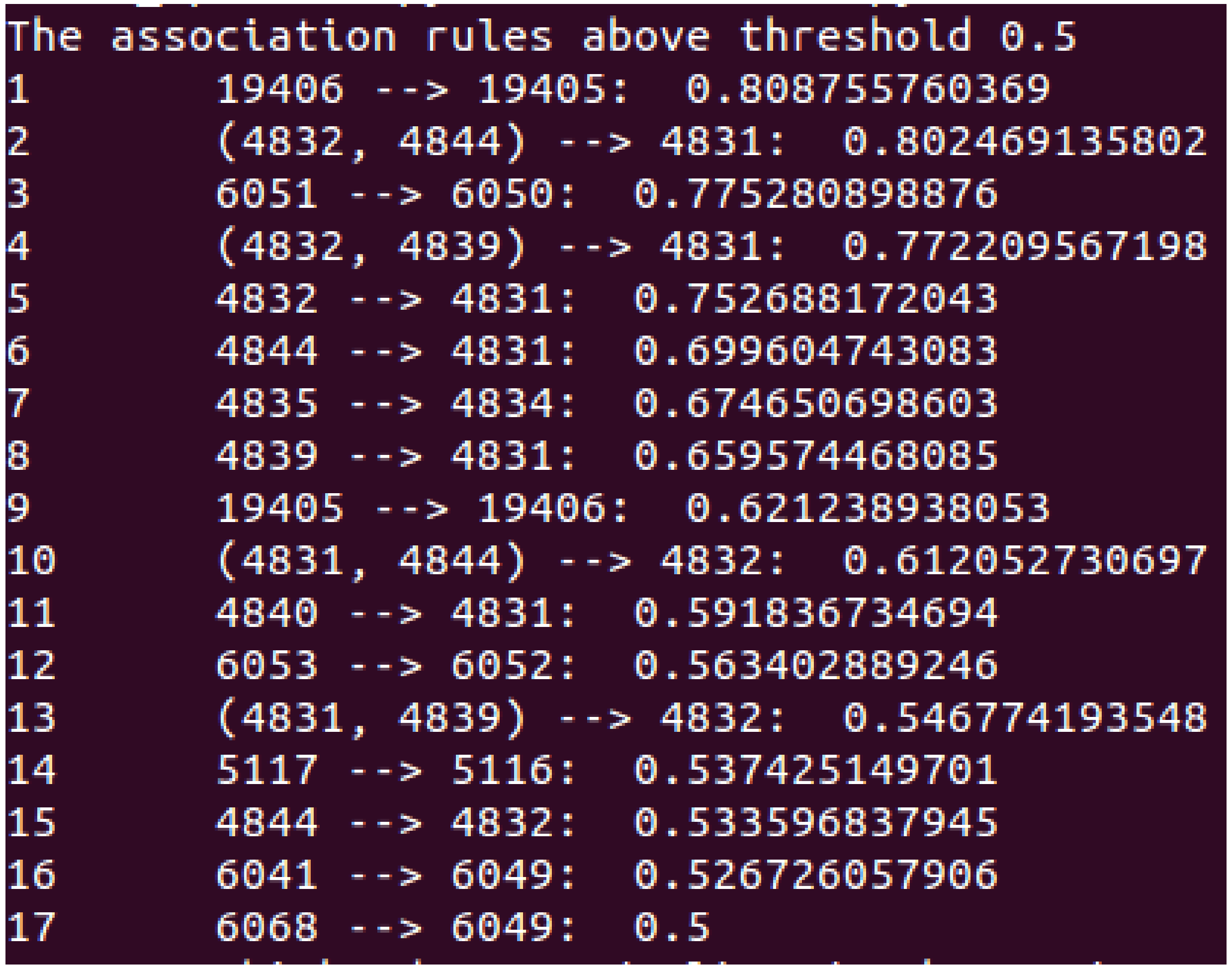
- If you know an itemset *J* is frequent, how do you find all related association rules' confidence?
- There're better ways than this, however, since our results stops only after *k=3*, we will enumerate all of them (*2^k* possibilities).
- Suppose $I \cup J = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, we use binary number to simulate the whether an item appears in *J (1) or not (0).*
- Then $\mathrm{I} = (I \cup J) - \mathrm{J}$.

• J={},	000,	0
• J={a},	001,	1
• J={b},	010,	2
• J={a, b},	011,	3
• J={c},	100,	4
• J={a, c},	101,	5
• J={b, c},	110,	6
• J={a, b, c},	111,	7

Binary digits, each digit represents a specific item appears in J or not

Decimal numbers for you to easily going through

The associations found (confidence threshold=0.5)



```
The association rules above threshold 0.5
1       19406 --> 19405:  0.808755760369
2       (4832, 4844) --> 4831:  0.802469135802
3       6051 --> 6050:  0.775280898876
4       (4832, 4839) --> 4831:  0.772209567198
5       4832 --> 4831:  0.752688172043
6       4844 --> 4831:  0.699604743083
7       4835 --> 4834:  0.674650698603
8       4839 --> 4831:  0.659574468085
9       19405 --> 19406:  0.621238938053
10      (4831, 4844) --> 4832:  0.612052730697
11      4840 --> 4831:  0.591836734694
12      6053 --> 6052:  0.563402889246
13      (4831, 4839) --> 4832:  0.546774193548
14      5117 --> 5116:  0.537425149701
15      4844 --> 4832:  0.533596837945
16      6041 --> 6049:  0.526726057906
17      6068 --> 6049:  0.5
```

- The index within each association rule are quietly close to each other, like 19405 → 19405, (4832, 4839)→ 4831, ….
- We guess the supermarket, when they design the positions or creating IDs for their products, they try to put stuffs that are frequently bought together close on purpose.

SON: (the Algorithm of Savasere, Omiecinski, and Navathe)

This is a simple way of paralleling:

- Each computing unit gets a portion of data that can be fit in its memory (need a data structure to store the local data),
- then use A-Priori or PCY or other methods to find the locally frequent itemsets, which forms the candidate set.
- During this process, the local threshold is adjusted to p*s, (p is the portion of data you get)

- During the 2nd pass, you only count for the candidate itemsets.
- Filter out infrequent candidates.

- If an itemset is not locally frequent from all computing units, then it also cannot be frequent globally ($\sum_i p_i = 1$).
- This algorithm can deal with whatever length itemsets in its paralleling process.
- But it needs to store raw data to local memory (more expensive than storing count, practical only when you have a lot of clusters).

References

- *1. Mining of Massive Dataset, Jure Leskovec, Anand Rajaramanand Jeffrey D. Ullman.*
- *2. Chun-Nan Hsu, Hao-Hsiang Chung and Han-Shen Huang, Mining Skewed and Sparse Tranaction Data for Personalized Shopping recommendation, Machine Learning, 2004, Volume 57, Number 1-2, Page 35*