Improvements to A-Priori

Park-Chen-Yu Algorithm

Sampling

SON Algorithm

PCY (Pass 1)

- Use a hash function which ``bucketizes'' item pairs, that is, maps them to integers in [1,k].
- Each ``bucket'' *i* in [1,k] is associated with a counter c_i.
- During pass 1, as we examine a basket (e.g. {m,b,d,o}):
  - update counters of single items;
  - Generate all item pairs for that basket, hash each of them and add 1 to the corr. counter.

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
    }
}
```

PCY Algorithm

- Main observation: during pass 1 of A-priori, most memory is idle.
- Use that memory to keep additional info to improve storage during pass 2 of A-priori.
- Passes > 2 are the same as in A-Priori.

PCY: Observations

- A bucket is *frequent* if its counter is at least the support threshold *s*.
- If a bucket is not frequent, no pair that hashes to that bucket could possibly be a frequent pair.
- Therefore, on pass 2 we only count pairs that hash to frequent buckets.

PCY: Observations (2)

1. A bucket that a frequent pair hashes to is surely frequent.
2. Even without any frequent pair, a bucket can be frequent.

Observations – (2)

3. But in the best case, the count for a bucket is less than the support *s*.
   - Now, all pairs that hash to this bucket can be eliminated as candidates, even if the pair consists of two frequent items.

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* }, hashes to a frequent bucket.

Ignore all pairs belonging to non-frequent buckets (do not use a counter for them).

All (Or Most) Frequent Itemsets In $\leq$ 2 Passes

- A-Priori, PCY, etc., take *k* passes to find frequent itemsets of size *k*.
- Other techniques use 2 or fewer passes for all sizes:
  - Simple algorithm.
  - SON (Savasere, Omiecinski, and Navathe).

Simple Algorithm – (1)

- Take a random sample of the market baskets.
- Run A-priori or one of its improvements in main memory, so you don't pay for disk I/O each time you give a pass on the data.
  - Be sure you leave enough space for counts.

Sampling

- To sample: give a full pass on the data and keep a basket in main memory with probability p (depending on main memory and input size)
- Why do we need to give a full pass just to retain a fraction p of the data?

Sampling

- To sample: give a full pass on the data and keep a basket in main memory with probability p (depending on main memory and input size)
- Why do we need to give a full pass just to retain a fraction p of the data?
  - A random sample is the best representative of a dataset. Keeping only the first baskets might not contain iPhones for example.

Simple Algorithm – (2)

- Adjust the support threshold s accordingly:

  E.g., if p=1/100 of the baskets, use *s* /100 as your support threshold instead of *s* .

Simple Algorithm: errors

- We might have:
  - *False positives*: items frequent in the sample but not in the dataset.
  - *False negatives*: items not frequent in the sample but frequent in the dataset.
- If the sample is large enough it is unlikely that we get either of them.

Simple Algorithm: Improvement

- If we cannot have a sample large enough then
  - Remove false positives with one more pass (count only frequent itemsets in the sample).
  - False negatives: decrease the support threshold (e.g. 0.9*ps*). This might increase false positives. We might not remove all false negatives.

SON Algorithm

- Two passes,
- No false positives or false negatives.
- Divide the dataset into chunks, where each chunk contains a subset of baskets.

SON Algorithm – Pass 1

- Let *p* such that each each chunk is a fraction *p* of the dataset.
- Divide the dataset into chunks, where each chunk contains a subset of baskets.
- For each chunk compute all frequent itemsets with support *ps* and store them on disk. This is the set of candidates for next pass.

SON Algorithm – Pass 2

- Read all frequent itemsets found in the previous pass (candidates).
- For each of them count the number of occurrences and output only those with support at least *s*.

SON Algorithm – Errors?

- False positives?

SON Algorithm – Errors?

- False positives? No, because we compute the correct support in the second pass.
- False negatives?

SON Algorithm – Errors?

- False positives? No, because we compute the correct support in the second pass.
- False negatives? No. If an itemset is not frequent in any chunk then its support is < $ps(1/p)=s$ (there are $1/p$ chunks). So it is not frequent in the dataset.

SON Algorithm – Errors?

- False positives? No, because we compute the correct support in the second pass.
- False negatives? No. If an itemset is not frequent in any chunk then its support is < ps(1/p)=s (there are 1/p chunks). So it is not frequent in the dataset.
- The SON algorithm lends itself to a MapReduce implementation...

SON Algorithm – MapReduce 1st job

- First Map. Each mapper receives a fraction p of the dataset. It computes frequent itemsets with support at least ps. It outputs for each frequent itemset F, the pair (F,1) (where 1 is irrelevant).
- First Reduce. Output (only once) those itemsets F that appear one or more times.

SON Algorithm – MapReduce 2nd job

- Second Map. Each mapper takes all the frequent itemsets (candidates) from the previous job and a subset of baskets. The output is a set of key-value pairs *(C,v)* where C is a candidate and *v* is the support of C in that set of baskets.
- Second Reduce. It computes the total support of C for each candidate and outputs only those above the support.

References

Frequent Itemsets via sampling, how large the sample?:

Matteo Riondato, Eli Upfal: Efficient Discovery of Association Rules and Frequent Itemsets through Sampling with Tight Performance Guarantees. TKDD 8(4):20 (2014)