Finding Frequent Item Pairs

Lu Lu, Runmin Xu

MIT 6.338/18.337 project report

December 16th, 2013

I. Background

The problem we addressed is to find frequent item pairs over massive transaction data. Many practical applications, such as search log mining and network analysis, are related to this finding frequent itemsets. The motivation of our project is to explore the customer behaviors behind the transaction, for example, what kind of items people are likely to buy together. Results of this analysis can be applied to the development of "recommendation system" and increase the market revenue.

Rather than behavior analysis, we are focusing on the implementation of efficient algorithms in terms of time and space optimization. Different methods will be described and compared to show the steps how we reached the final solution.

II. Methodology

Before exploring the algorithms, let's make some definitions first.

Basket: A set of items some one bought in one time.
Example: {Apple, Milk, Orange}

Frequent item: Given a threshold, an itemset whose frequency in all the baskets is greater than the threshold is called frequent itemset.
Example: In a set of baskets: {a,b,c}, {a,b,d}, {a,b,e}, {a,b,f}; item pair {a,b} are frequent since it appears in every basket.

Naive method

The most straightforward 1-pass algorithm to solve this problem is to count all possible item pairs while traversing the dataset and save the results in a dictionary (item ids, frequency).

All item pairs with counts that are larger than a threshold are frequent item pairs. This method, however, requires huge memory space to save all candidate item pairs and their counts. If there are n distinct items in the dataset, the total number of possible item pairs can be as large as n*(n-1)/2, which indicates an $O(n^2)$ space complexity. Suppose there are 10^5 distinct items and the counts are 4-byte integers, 20 GB memory is needed to apply this brute-force method.

Improving memory usage: 2-pass method and hashing

In our project, memory is the most critical issue since an algorithm with $O(n^2)$-space-complexity is definitely unable to handle a large dataset. The following features, however, are helpful to improve the algorithm:

1. If an item pair is frequent pair, each item in this pair must be frequent

This is not surprising. In one dataset, count(a,b) is obviously not greater than either count(a) or count(b). Benefit from this feature, a frequency list of single items can be created first. Based on this list, the non-frequent items can be removed and only the remaining frequent items are being considered to generate the frequent item pairs. By doing this, a linear space complexity, *O(n)*, is reached in the first pass to maintain the counts of each item. In this second pass, only the frequency of the item pairs generated from frequent items are counted. By doing this a large amount of non-frequent items can be removed. Two method are developed as follows:

Method 1

Generate a list of possible item pairs based on the of single items, which takes $O(m^2)$ space. Then for each basket, iterate through this list and check if each pair exist. This method takes $O(m^2*L*N)$ time, where m is number of frequent single items, L is the average length of baskets and N is the number of baskets.

Method 2

For each basket, generate a list of frequent single items and all possible item pair based on this list. Then iterate through all the basket. This method takes $O(L^2*N)$, where L is the average length of baskets and N is the number of baskets.

In practical datasets with reasonable thresholds, L is typically much smaller than m^2. Therefore, method 2 is a more efficient way to count item pairs in the second pass.

2. Typically only a small fraction of the item pairs are frequent

There is usually only a very small fraction of all the possible item pairs are frequent. So why do we waste a large proportion of memory to store the count of non-frequent item pairs? Here a hash table is implemented which creates a set of bucket and hashes each item to a certain bucket, In general, the number of bucket is much smaller than the number of possible item pairs, which saves a large amount of memory. With this hash mechanism, only the frequency of each bucket are counted and stored, not the item pairs.

What can we benefit from the hash table? Intuitively, an item pair is frequent only if the bucket it was hashed to is frequent, since the count of a bucket is the sum of the counts of each item pair in it. By doing this a large amount of non-frequent item pairs can be removed.

Improving computational time: parallelization

A MapReduce framework is applied to parallelize the 2-pass method as shown in Figure 1. It is supposed to improve the running time performance of the process when running with multi-processors. This framework is applied when counting the frequencies of single items and when counting the frequencies of item pairs after filtering out non-frequent single items.

The procedure is described in below:

Step 1*.* If on one computer, load the dataset into main memory, divide the data into chunks with similar size. The number of chunks is decided based on the number of processors to use. If on a cluster with multiple computers, divide the data into chunks beforehand and load each chunk into the main memory of each computer. Then, on each computer, divide the data into chunks based on the number of processors to use.

Step 2. For each data chunk, use a mapper function to count items (item pairs) while traversing the chunk and save the results in dictionaries.

Step 3. Use a reducer function to combine different dictionaries from the mapper functions and get a dictionary to save the total counts of each item (item pair) over the entire dataset.

Step 4*.* Apply a filter to get frequent items (item pairs) based on a threshold.

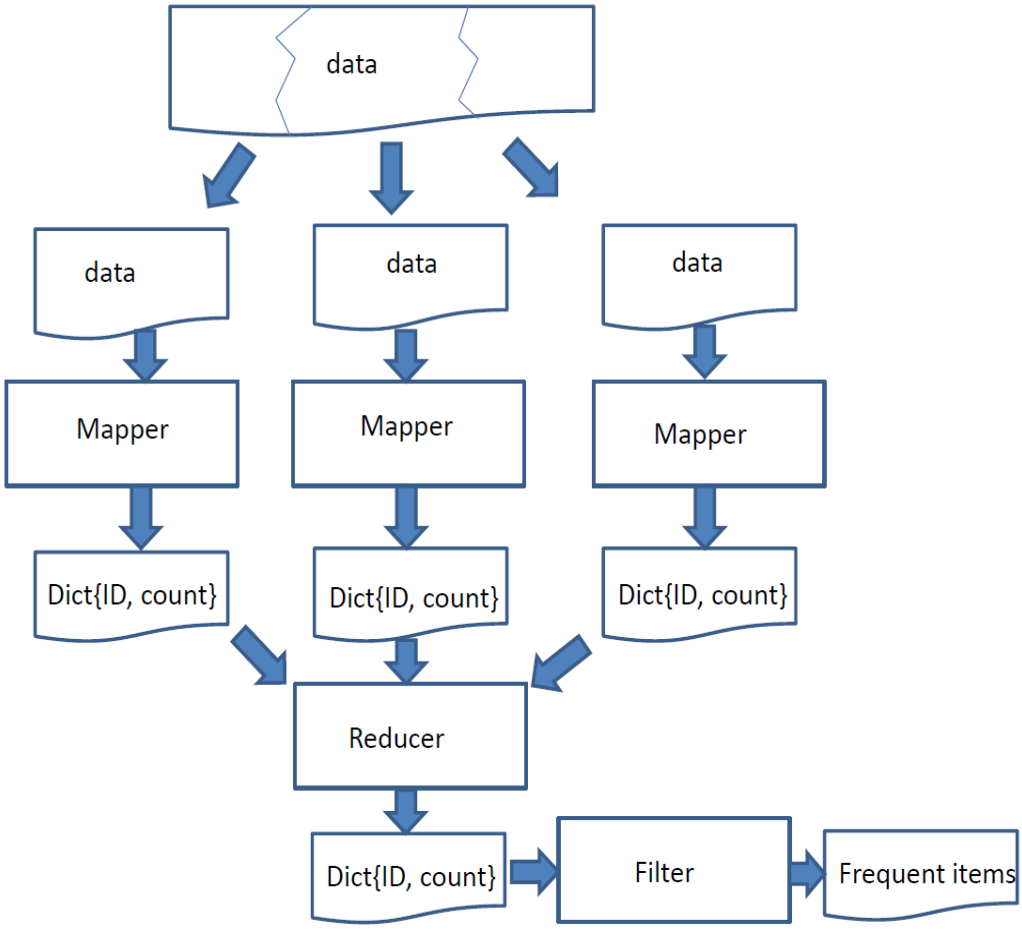

Figure 1: The MapReduce framework to parallelize the 2-pass method

III. Implementation

The 2-pass method is implemented using Julia. Julia is a high-level technical computing language with high performance. It is very convenient to implement parallel computing framework in Julia.

To illustrate the implementation of the framework using Julia, the core part of the Julia code for parallel counting of single items is shown below. A similar method is applied to counting item pairs with more complicated details in the map and reduce functions.

```
# Splits input string into nprocs() equal-sized chunks
# (last one rounds up), and @spawns wordcount() for each chunk to
```

```
# run in parallel. Then fetch()s results and performs wcreduce().

function parallel_item_count(text)
    lines=split(text,'\n',false)
    np=nprocs()
    unitsize=ceil(length(lines)/np)
    item_counts={}
    rrefs={}
    # spawn procs
    for i in 1:np
        first=unitsize*(i-1)+1
        last=unitsize*i
        if last>length(lines)
            last=length(lines)
        end
        subtext=join(lines[int(first):int(last)],"\n")
        push!(rrefs, @spawn map_count( subtext ) )
    end
    # fetch results
    while length(rrefs)>0
        push!(item_counts,fetch(pop!(rrefs)))
    end
    # reduce
    count=reduce_count(item_counts)
    return count
end

# "Map" function.
# Takes a string. Returns a Map with the number of each item.
function map_count(text)
    items=split(text,[' ','\n','\t','-','.',',',':','_','"',';','!'])
    counts=Dict{ASCIIString, Int64}()
    for item in items
        counts[item]=get(counts,item,0)+1
    end
    return counts
end

# "Reduce" function.
# Takes a collection of Map in the format returned by map_count()
# Returns a Map in which words that appear in multiple inputs have
their totals added together.
function reduce_count(item_counts)
```

```
    reduced_counts = Dict{ASCIIString, Int64}()
    for item_count in item_counts
        for (item,count) in item_count
            reduced_counts[item] = get(reduced_counts,item,0)+count
        end
    end
    return reduced_counts
end
```

IV. Performance

The method was tested using a sampled dataset with 999,002 transaction records and 41,270 distinct items. The test machine has a 2-core 1.7GHz Intel Core i5 processor and 4GB memory. The naive method requires more than 6 GB memory. Therefore, it is not possible to apply it with the test machine. The memory usage with and without hashing and the running time with different number of processes in Julia were tested.

Memory usage saving

Different hash functions could result in different memory usage levels. By implementing the 2-pass method and hashing, roughly 76% memory has been saved on average.

Computational time saving

The running times of single items count, item pairs count, and the total running time are with respect to different numbers of processes are shown in Figure 2. Compared to item pairs counting, single items counting required much less time, and therefore the trend for total time is mainly decided by the running time of item pairs counting. The running time decreases when more processes are used. However, when the number of processes exceeds 5, the overhead from the multi-processing framework exceeds the time saving and the total running time increases when more processes are used. This type of test on a relatively small dataset is important to make decisions about how many processes to use in real large-scale cases.

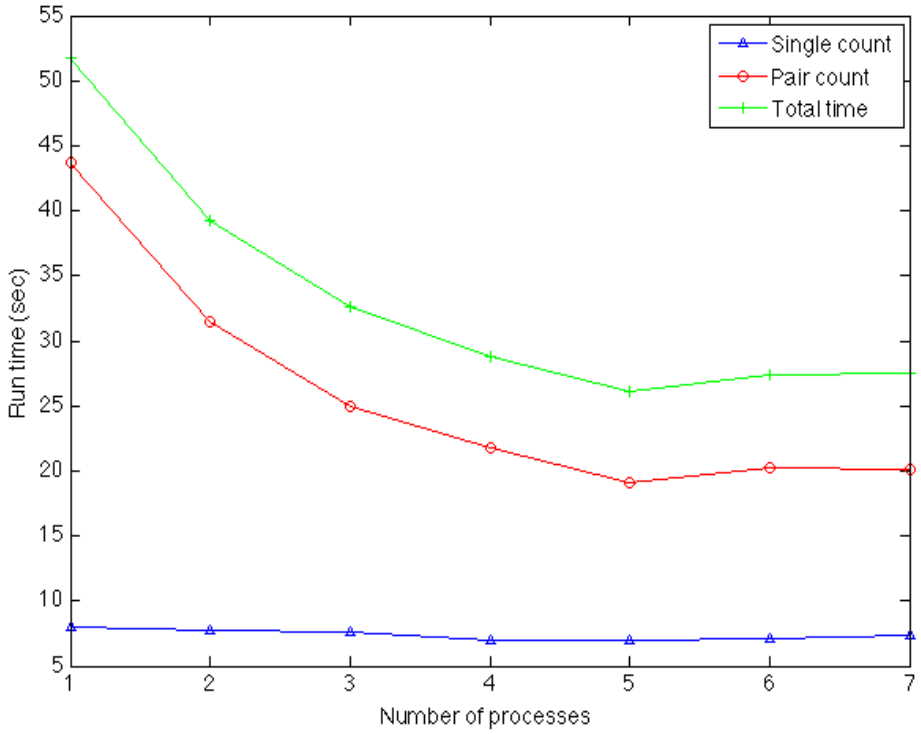

Figure 2: Running time with different number of processes

V. Conclusions

In this project, we attempted to improve the naive solution algorithm of a realistic problem: finding frequent item pairs in massive transaction datasets. Our methods, though relatively intuitive, are capable of reducing both the required memory space and the running time significantly. This process helped us to better understand the MapReduce process and to appreciate Julia's extremely easy-to-use parallel computing features. In future researches, Julia will become a strong candidate when we try to decide a scientific computing language to use.

There are research papers focusing on more sophisticated methods with rigorous mathematical derivations and proofs. For example:

Pasquier, Nicolas, et al. "Discovering frequent closed itemsets for association rules." *Database Theory—ICDT'99*. Springer Berlin Heidelberg, 1999. 398-416.

Zaki, Mohammed J. "Efficiently mining frequent trees in a forest." *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002.

Bodon, Ferenc. "A trie-based APRIORI implementation for mining frequent item sequences." *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*. ACM, 2005.