SON implementation on Spark in python

Carli Alberto VR473845 Fraccaroli Leonardo VR??????

Contents

1 Introduction

Finding frequent itemsets is frequently done in data mining domains to understand which items frequently appear together. The definition of "items" is beyond the scope of this work. Suffice to say that we treated them as binary variables, listed in a basket only when their value is True. In out project a basket is a list of items that appear together.

This is an analysis that can be performed on almost any dataset, but is especially suited for domains like e-commerce. It is an interesting challenge to try and take the system to the extreme where the dataset is so large that it does not fit on any single machine's memory and it is more convenient to distribute analysis to multiple compute units.

We tried to imagine ourselves into a big e-commerce company that stores transaction data on a distributed database and wants to periodically analyse data to understand which items are usually bought together.
The data is simply a list of items bought at any one time; as the distributed database we used MongoDB and thus we needed a version of parallel apriori.

We tried to parallelize apriori by using Spark on python and using the SON algorithm as proposed in chapter 6.4.4 of the Mining Massive Datasets book.

2 SON

In this section we will dissect our implementation of SON through Spark on python, describing all the steps of computation and their result.

2.1 SON Algorithm

The algorithm as described in the book involves two phases of mapreduce as follows:

1. The map basically executes apriori on a batch of baskets, extracting all frequent itemsets. The required frequency (support) is reduced by a factor equal to the number of number of batches the data is partitioned in.
   The reducer is an identity reducer and simply combine all the frequent itemsets from every basket in a single list. These are candidate frequent itemsets.
2. The map function takes a batch of the data and counts every occurrence of every itemset in the candidates list. The reducer adds up the counts from every batch and finally filters out the infrequent ones using the original support.

The result of this is the list of frequent itemsets without false positives.

2.2 Our Implementation

Our implementation expects a *pyspark* RDD

2.2.1 First Map

The input data is in the form of a list of lists, where every internal list holds a set of strings, which are the items (fig: 1a).

Apriori algorithm is applied to every batch, which returns a list of frequent itemsets (fig: 1b). This is done with the *mapPartitions* function of *pyspark*:

```
.mapPartitions(lambda x: apriori(list(x), chunk_support))
```

We will discuss our implementation of apriori, in details, in section 3. We chose not to keep only the largest set because we saw that there may be cases where the larger set is not globally supported, but the smaller ones are.
Then every itemset is "emitted" with a 1, to facilitate the reduce phase:

```
.map(lambda x: (x, 1))
```

2.2.2 First Reduce

Now the result of the previous step is grouped by key (fig: 1c) and the list of counts discarded entirely:

```
.groupByKey()
.map(lambda x: x[0])
```

The result is the list of all itemsets that appear at least once (fig: 1d), which means that they are frequent in at least one batch.

(a) Input data (b) Apriori on batch (c) First reduce (d) Candidate frequent itemsets

2.2.3 Second Map

At this point our candidate frequent itemsets are still partitioned. We need to distribute the whole list to every batch of data. In order to do so we coalesced the candidates RDD into one single partition, group it together and multiply if with every partition of the original data:

```
.repartition(1)
.glom()
.cartesian(baskets.glom())
```

The result is, for every partition, a list containing two lists; the first one is a list of itemsets and the second one is the batch of data (fig: 2a).
Then, for every itemset in the first list, we count in how many baskets is present[1]:

```
.mapPartitions(lambda x : count_frequencies(list(x)[0]))
```

The result is a list of tuples where the first element is the itemset and the second is the count for every batch (fig: 2b).

2.2.4 Second Reduce

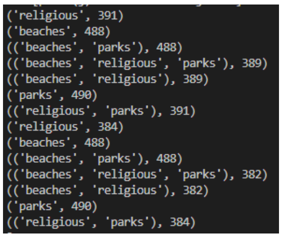
Now the itemsets are reduced across partitions by summing counts (fig: 2c) and then filtered:

```
.reduceByKey(lambda x, y: x + y)
.filter(lambda x: x[1] / data_size >= support)
```

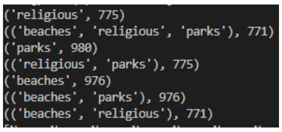
The result is the list of every itemset which is supported enough to be considered frequent. Our next step is to filter out all the smaller sets which are included in larger sets.

[1] count_frequencies is a function we developed that does exactly that: count the occurrences of the elements in the first list in the second list

(a) Cartesian product result



(b) Counts of itemsets for every batch



(c) Sum of counts across partitions

3 apriori

Apriori is the most used algorithm to find frequent itemsets in a dataset. The algorithm relies on the monotonicity of support by building candidates from smaller sets which are knowingly frequent. This is because it is impossible for an itemset to be frequent if every item it is composed of is frequent by itself.

To benchmark SON execution we had to compare it to an implementation of apriori, which we made ourselves:

```
def apriori(data, support):
    basket_size = len(data)
    frequent_itemsets = []
    items = list(set([k for j in data for k in j]))
    temp = count_frequencies((items, data))
    frequent_items = [i[0] for i in temp if i[1]/basket_size >= support]
    new_frequent_itemsets = frequent_items
    while new_frequent_itemsets != []:
        frequent_itemsets += new_frequent_itemsets
        new_candidate_itemsets = [(j, ) + (k, ) if isinstance(j, str) \
                else j + (k,) for j in new_frequent_itemsets \for
                k in frequent_items if k not in j]
        new_candidate_itemsets = [tuple(j) for j in {frozenset(i) \
        for i in new_candidate_itemsets}]
        temp = count_frequencies((new_candidate_itemsets, data))
        new_frequent_itemsets = [i[0] for i in temp \
        if i[1]/basket_size >= support]
    return frequent_itemsets
```

This code first extracts the items from the data, keeps the frequent ones and then uses them to build all the candidate couples, counts them, filters out infrequent ones and then expands those couples to form triples and so on.

4 benchmark

We have ran some benchmarks to better understand how SON compares to apriori.
In order to do so we made a script for the specific purpose of being able to programatically change parameters and subsample datasets. Also, we imple-

mented a gridsearch to test many parameters. We also added a logging feature to our project to better follow executions and ease debug. We compared our implementation, both based on DB and in memory, to a method $FreqItems$ provided by $pyspark.sql$. This function uses a modified version of FP-Tree to be more parallelizeable. This function proved itself to be absurdly fast, at the expense of false positives. Both a 'simple' dataset and a 'harder' one were used to compare performances.

4.1 Datasets

4.2 Results

These tests were performed on an 8 core $i5 - 8250U@1.6GHz$, with 16 GB DDR4 memory @2400MHz

5 Conclusions

After the benchmarks it is clear that apriori is blazingly fast and the only reasons one should instead use SON is when data doesen't fit in memory or many compute units are available.