

# SON implementation on Spark in python

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

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>SON</b>	<b>3</b>
2.1	SON Algorithm . . . . .	3
2.2	Our Implementation . . . . .	3
2.2.1	First Map . . . . .	3
2.2.2	First Reduce . . . . .	3
2.2.3	Second Map . . . . .	4
2.2.4	Second Reduce . . . . .	4
<b>3</b>	<b>apriori</b>	<b>5</b>
<b>4</b>	<b>benchmark</b>	<b>5</b>
4.1	Datasets . . . . .	6
4.2	Results . . . . .	6
<b>5</b>	<b>Conclusions</b>	<b>6</b>

## 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 our 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])
```

[illegible]

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 it with every partition of the original data:

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<sup>1</sup>:

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).

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

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.

4

(a) Cartesian product result

(b) Counts of itemsets for every batch

(c) Sum of counts across partitions

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

The first “easier” dataset is from UCI (Travel Reviews) and is composed of averages of scores that every user gave to some categories of places on TripAdvisor. We rounded the scores to integer and kept only the ones above 3.

This way our dataset is in the form:

user.id: category1, category2 ...

It is considered “easy” because it is very sparse and the number of items is very low.

The second “harder” dataset is a collection of transaction where some user bought some item in a transaction, again from UCI (Online Retail). We decided to form transactions on invoices, so we extracted the list of items bought together. This way our dataset is in the form:

invoice.no: item.code, item.code ...

It is considered “harder” because of the higher number of items.

## 4.2 Results

These tests were performed on an 8 core *i5 – 8250U@1.6GHz*, with 16 GB DDR4 memory @*2400MHz*, but both the driver and the executor were capped at 4 GB. figures 3a, 3b

## 5 Conclusions

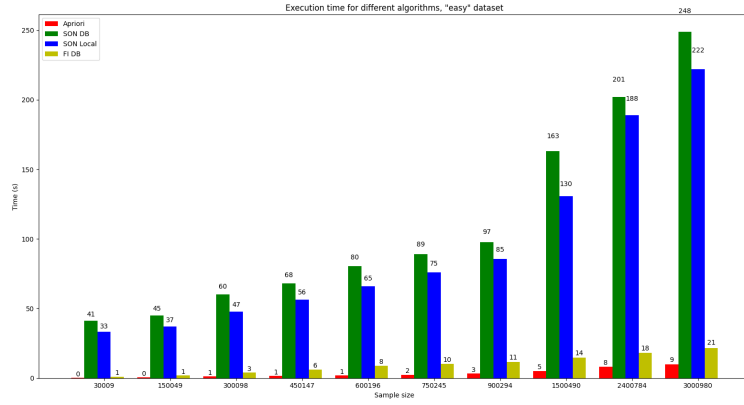
The main advantage of SON, that we could appreciate in our single machine setup, was the ability to use all available core, distributing the load.

But aside from that, we appreciated how the algorithm is slow and only to be used when needed. It is not to be preferred to apriori or FP-Trees.

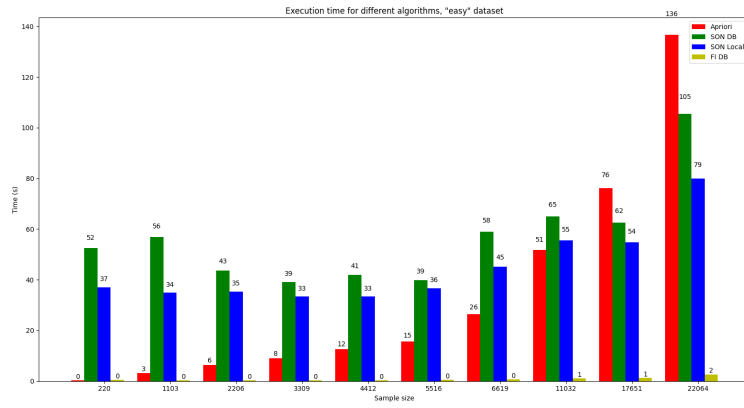
We have seen how easy it is connecting pyspark to a distributed database, eventually sharded.

With the testing we have appreciated how SON execution time remains somewhat constant, while apriori execution time grows exponentially with the number of items.

Also, because of the way SON works, we expect to work better on denser datasets and with a higher number of items, because the splitting of the datasets produces smaller exponentiation in the number of candidates.



(a) Easy dataset benchmark results



(b) Hard dataset benchmark results

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