SON implementation with Spark in Python

Carli Alberto VR473845 Fraccaroli Leonardo VR474539

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 our project a **basket** is a list of items that appear together (for example products bought by a customer).

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 time. To store the data in a distributed database we used MongoDB and thus we needed a version of parallel **Apriori** (an algorithm to find frequent itemsets on a single machine).
We parallelized Apriori by using Spark in 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, 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 mapper basically executes Apriori (or whatever algorithm to find frequent itemsets) on every batch of baskets, extracting all frequent itemsets for each batch. The required frequency (support) is reduced by a factor equal to the number of batches the data is partitioned in, using the following formula:

   $$batch\_support = basket\_size/total\_data\_size * total\_support$$

   The reducer combines all the frequent itemsets from every batch in a single list. These are *candidate frequent itemsets*.
   This step removes false negatives by keeping all the itemsets that are frequent in at least one batch.

2. The map function takes a batch of the data and counts the occurrence of every candidate itemset.
   The reducer adds up the counts from every batch and finally filters out the infrequent ones using the original support.
   This step removes false positives by keeping only the itemsets that are globally supported.

The result of these phases is the list of *frequent itemsets*.

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), support, data_size))
```

where `x` is a batch, `support` is the support for the entire dataset and `data_size` is the total number of baskets in the entire dataset (they are needed in `apriori()` to calculate the batch support). We will discuss our implementation of Apriori, in details, in section 3. We chose not to keep only the largest sets 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:

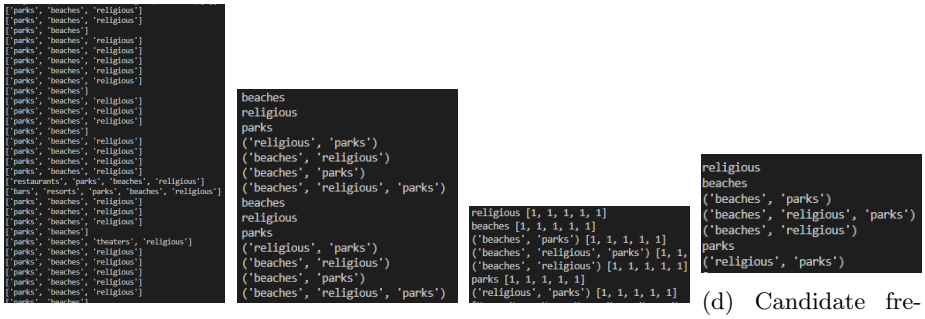
```
.map(lambda x: (x, 1))
```

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.
Then the list is collected and broadcasted to all the nodes.



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

Figure 1: Results of first MapReduce phase

2.2.3 Other considerations and final implementation of first pass of MapReduce

Initially we have implemented the code following the instructions on the book using Spark operations, but then we found another method that has slightly better performances. So the final implementation of the first pass of MapReduce goes from

```
candidate_frequent_itemsets = (baskets
    .mapPartitions(lambda x: apriori(list(x), support, data_size))
    .map(lambda x: (x, 1))
    .groupByKey()
    .map(lambda x: x[0])
    ).collect()
```

to

```
candidate_frequent_itemsets = (baskets
    .mapPartitions(lambda x: apriori(list(x), support, data_size))
    ).collect()
candidate_frequent_itemsets = list(set(candidate_frequent_itemsets)
    )
```

2.2.4 Second Map

Now we need to count the occurrences of every itemset in the whole dataset. We do it on every partition:

```
.mapPartitions(lambda x: count_frequencies(
    candidate_frequent_itemsets.value, list(x)))
```

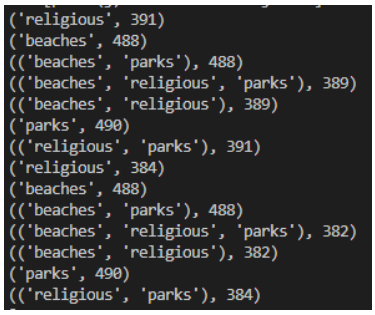
The result is a list of tuples where the first element is the itemset and the second is the count for every batch (fig: 2a).

2.2.5 Second Reduce

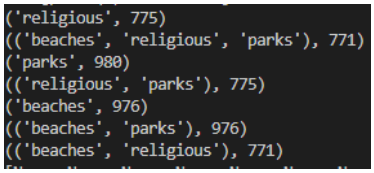
Now the itemsets are reduced across partitions by summing counts (fig: 2b) and then filtered:

```
.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.



(a) Counts of itemsets for every batch



(b) 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. The algorithm 1 shows the Apriori pseudocode.

To benchmark SON execution we had to compare it to an implementation of Apriori, which we made ourselves (*Scripts/apriori.py*).

Algorithm 1 Apriori pseudocode

```
1:  function APRIORI(data, total_support, total_db_size)
2:      L_1 ← {frequent 1 − itemsets}          ▷ L_k is the set of truly frequent itemsets of size k
3:      k ← 2
4:      while L_{k−1} is not empty do
5:          create C_k using L_{k−1} (by adding singlets)   ▷ C_k is the set of candidate itemsets of size k
6:          count occurrences of each itemset of C_k in data
7:          filter out non-frequent itemsets in C_k
8:          L_k ← C_k
9:          k ← k + 1
10:     end while
11: end function
```

In our implementation of the Apriori pseudocode we first extract the items from the data, keep the frequent ones and then use them to build all the candidate couples, count them, filter out infrequent ones and then expand those couples to form triples and so on.
We also added a fail-safe feature to our implementation: if the support in a partition is so low that all possible item sets are frequent, it doesn't even begin to create them. It instead prints some error message and returns None.
The other threads won't stop for this, but detecting at least one null result will void the other results.

Moreover, we found a state-of-the-art implementation of Apriori: it is called Efficient Apriori, and it is the fastest implementation of Apriori. We tried to use it in our SON code and we can appreciate the improvements in execution time.

4 Benchmark

We have run 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 programmatically change parameters and subsample datasets. Also, we implemented a grid search 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** (using the MongoDB Connector for Spark, which is kind of a "real world" example) and **in memory**

(having all the data locally), to a method $FreqItems$ provided by $pyspark.sql$. This function implements the algorithm proposed by Karp, Schenker, and Papadimitriou, which 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 arbitrarily kept only the ones with an average mark above 2.5. This way our dataset is in the form:

$$user\_id : category1, category2, \ldots$$

It is considered "easy" because it is very sparse and the number of items is very low (10 items).

The second "harder" dataset is a collection of transactions where some user bought some items 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:

$$invoice\_no : item\_code1, item\_code2, \ldots$$

It is considered "harder" because of the higher number of items (4059).

4.2 Database

Part of our project was to explore how spark interacts with a database like MongoDB.

We found that there is a MongoDB Connector for Spark to connect to a MongoDB database and read the data from it as well as write the results back to it.

The main difference between a local spark instance (e.g. reads from disk and not a database) is that the environment is a $SparkSession$ (instead of a $SparkContext$), which allows to use connectors to read and write data from databases as well as repartition it.
A *partitioner* can be specified to regulate how the data is handled and to change how it is distributed.
The read data is in the form of a $DataFrame$ which wraps an RDD and allows to use SQL queries on it.

We tried using spark connected to a sharded database, with a sharded partitioner and while it was easy to use (no particular setup on the spark side) it was really difficult to setup the database with no noticeable execution difference.
So we settled on a single node database with a default partitioner, which in the case of our datasets, always produced a single partition.

4.3 Results

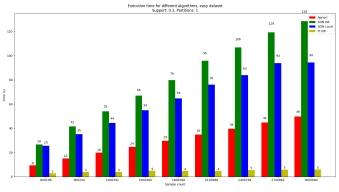
After some testing we figured that both the support and the partitions played a major role, on top of the dataset, in determining an algorithm's performance.

We tested a mix of different supports on each dataset with the growth of both dataset size and partition number.

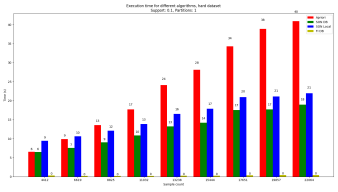
These tests were performed on an 8 core $i5 - 8250U@1.6GHz$, with 16 GB DDR4 memory $@2400MHz$, but both the driver and the executor were capped at 4 GB.

1. **Minimum support** required to produce frequent itemsets. In the case of our datasets, after some testing we figured that this is 0.3 for the "easy" dataset and 0.1 for the "hard" one.
   We tested with 1 (easy 3a, hard 3b) and 2 (easy 4a, hard 4b) partitions.

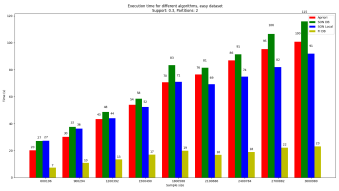
2. **Reasonable support** that is more likely used in a real scenario. Here we tried to see ho performance varied with partitions, using all dataset.
   We tested with support 0.75 (easy 5a, hard 5b) and 0.9 (easy 6a, hard 6b).



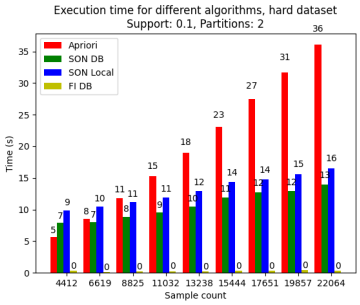
(a) Easy dataset benchmark results



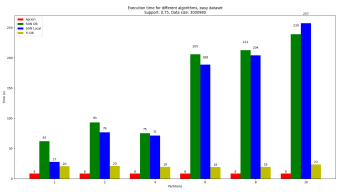
(b) Hard dataset benchmark results



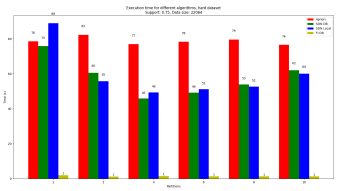
(a) Easy dataset benchmark results



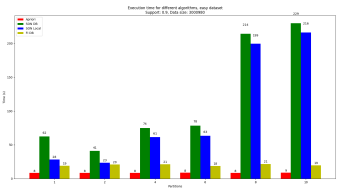
(b) Hard dataset benchmark results



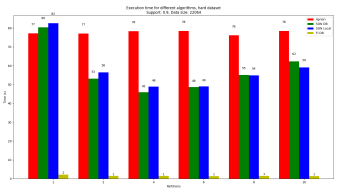
(a) Easy dataset benchmark results



(b) Hard dataset benchmark results



(a) Easy dataset benchmark results



(b) Hard dataset benchmark results

5 Conclusions

The main advantage of SON, that we could appreciate in our single machine setup, was the ability to use all the available cores, distributing the load, which, for some datasets, could prove beneficial to execution time.

We have seen how easy it is connecting pyspark to a distributed database, eventually sharded. We noticed that the database partitioner optimizes for the database connection (based on the size of the database itself), while the local partitioner (*parallelize* function) optimizes for exploiting all the resources of the machine.

We have encountered an issue where, in some cases, the support in a partition was so low that a frequency of one (or less) was considered frequent, which caused an explosion in the number of candidates generated by Apriori, not because they were actually frequent in the partition, but because every item (and all the combinations in each basket) always appear at least once.

In general, the huge reduction in support is an issue that happens every time the global support is too low or when the support is divided over many partitions ($\leq 5\%$ in a single partition).
This is an issue because the core is still Apriori and it's very important that the minimum support is high enough to prune the data as the algorithm works.

In fact, this may be the major factor to consider when using SON, because this can happen whenever a lot of candidates are produced in each partition. Dataset conformation must be taken into account when deciding how to partition the data.