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:

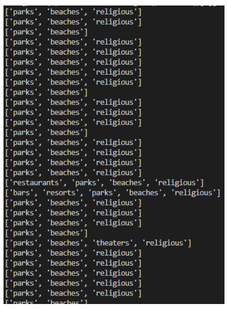
```
.map(lambda x: (x, 1))
```

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:

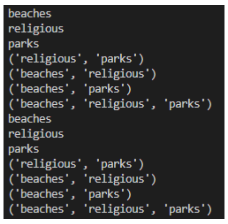
```python
.groupByKey()
.map(lambda x: x[0])
```

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.



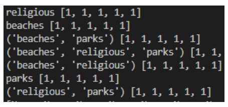
```
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['restaurants', 'parks', 'beaches', 'religious']
['bars', 'resorts', 'parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches']
['parks', 'beaches', 'theaters', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
['parks', 'beaches', 'religious']
```

(a) Input data



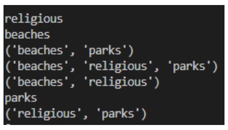
```
beaches
religious
parks
('religious', 'parks')
('beaches', 'religious')
('beaches', 'parks')
('beaches', 'religious', 'parks')
beaches
religious
parks
('religious', 'parks')
('beaches', 'religious')
('beaches', 'parks')
('beaches', 'religious', 'parks')
```

(b) Apriori on batch



```
religious [1, 1, 1, 1, 1]
beaches [1, 1, 1, 1, 1]
('beaches', 'parks') [1, 1, 1, 1, 1]
('beaches', 'religious', 'parks') [1, 1,
('beaches', 'religious') [1, 1, 1, 1, 1]
parks [1, 1, 1, 1, 1]
('religious', 'parks') [1, 1, 1, 1, 1]
```

(c) First reduce



```
religious
beaches
('beaches', 'parks')
('beaches', 'religious', 'parks')
('beaches', 'religious')
parks
('religious', 'parks')
```

(d) Candidate frequent itemsets

Figure 1: Results of first MapReduce phase

2.2.3 Second Map

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

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

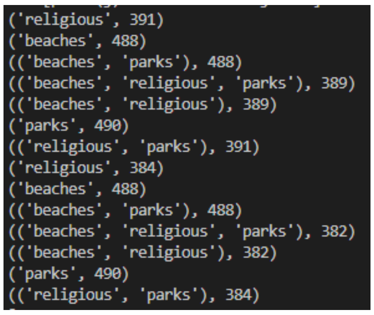
The result is a list of tuples where the first element is the itemset and the second is the count for every batch (fig: 2a).

2.2.4 Second Reduce

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

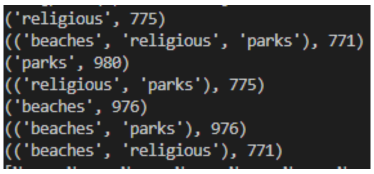
```python
.reduceByKey(lambda x, y: x + y)
.filter(lambda x: x[1] / data_size >= support)
```

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.



```
('religious', 391)
('beaches', 488)
(('beaches', 'parks'), 488)
(('beaches', 'religious', 'parks'), 389)
(('beaches', 'religious'), 389)
('parks', 490)
(('religious', 'parks'), 391)
('religious', 384)
('beaches', 488)
(('beaches', 'parks'), 488)
(('beaches', 'religious', 'parks'), 382)
(('beaches', 'religious'), 382)
('parks', 490)
(('religious', 'parks'), 384)
```

(a) Counts of itemsets for every batch



```
('religious', 775)
(('beaches', 'religious', 'parks'), 771)
('parks', 980)
(('religious', 'parks'), 775)
('beaches', 976)
(('beaches', 'parks'), 976)
(('beaches', 'religious'), 771)
```

(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:      $L_1 \leftarrow \{frequent\ 1 - itemsets\}$    ▷ $L_k$ is the set of truly frequent itemsets of size $k$
3:      $k \leftarrow 2$
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:          $L_k \leftarrow C_k$
9:          $k \leftarrow k + 1$
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.

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 implemented 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** (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 exists 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. We had some difficulties because the connector version 10 is not compatible with any spark version and had to check, as well as finding the right place to download the connector from.
Also it is very difficult to work in the environment spark+connector, because the connector is written in scala and spark uses function written in java.

The main difference between a local spark instance (e.g. reads from disk and not a database) is that the environment is a *SparkSession*, 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 used this connector to test the difference when the data is stored in a database versus when it is read from disk and found out that there is a appreciable difference in performance, but not as much as we expected.

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

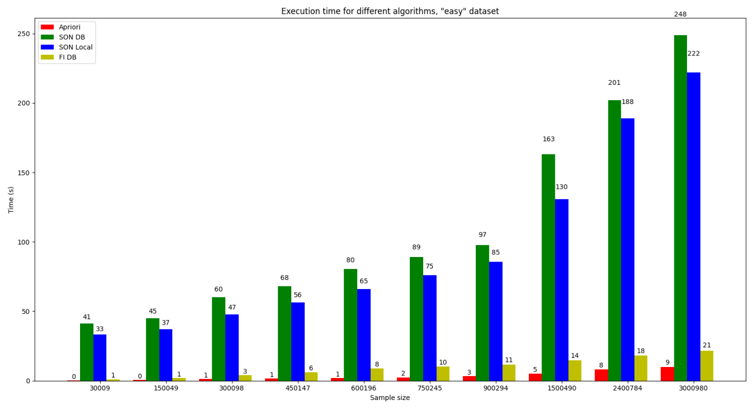
These tests were performed on an 8 core $i5 - 8250U@1.6GHz$, with 16 GB DDR4 memory $@2400MHz$, but both the driver and the executor were capped at 4 GB. figures 3a, 3b

5 Conclusions

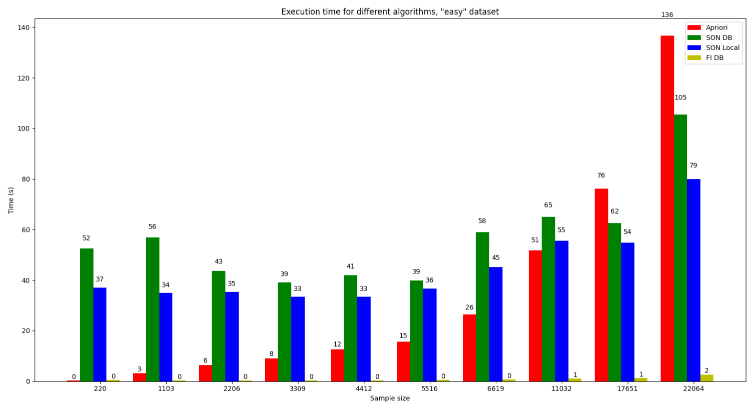
The main advantage of SON, that we could appreciate in our single machine setup, was the ability to use all the available cores, distributing the load. We also have seen how easy it is connecting pyspark to a distributed database, eventually sharded.

Partitioning in multiple threads seems to not be rewarded both in the local and the database case. We came to the conclusion that the database partitioner optimizes for the database connection, while the local partitioner (*parallelize* function) optimizes for exploiting all the resources of the machine.
We also found out that a single partition per CPU and not CORE is very efficient.

With the testing we have appreciated how SON execution time remains somewhat constant across the dataset sizes, while Apriori execution time grows exponentially with the number of items. We could also notice the overhead of the database connection as well as the overhead of spark framework management.



(a) Easy dataset benchmark results



(b) Hard dataset benchmark results