SON implementation with Spark in Python

Carli Alberto VR473845 Fraccaroli Leonardo VR474539

Contents

1 Introduction **2**

2 SON **3**
2.1 SON Algorithm 3
2.2 Our Implementation 3
2.2.1 First Map 3
2.2.2 First Reduce 4
2.2.3 Second Map 4
2.2.4 Second Reduce 4
2.2.5 Other considerations and final implementation of SON . . 5

3 Apriori **6**

4 Benchmark **6**
4.1 Datasets 7
4.2 Database 7
4.3 Results 8

5 Conclusions **9**

A Apriori benchmarks **10**

B Benchmark Images **10**

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. We will discuss our forst implementation, following the steps as described in the book, and then we will show our second implementation, which is a bit different and cuts around some corners that we found out to be unnecessary in spark.

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. We performed a map operation to transform the data into a list of sets, where every set is a basket (fig.1a). This is done with the *map* function of *pyspark*:

```
.map(set)
```

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(i for i in 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). x is an itertools.chain object, so we needed to convert it to a list. After some testing we found out that the fastest way is to use a generator object instead of the *list*() constructor.
We will discuss our implementation of Apriori, in details, in section 3. We chose not to keep only the frequent supersets 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 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)))
```

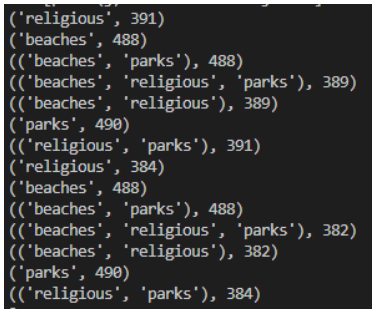
The result is a list of tuples where the first element is the itemset and the second is the count for every batch (fig.2a).

2.2.4 Second Reduce

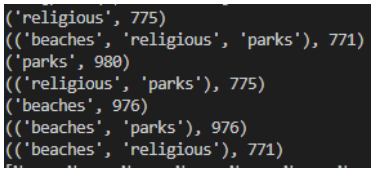
Now the itemsets are reduced across partitions by summing counts (fig.2b) and then filtered based on the global support:

```
.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. The last step would be to keep only the supersets, removing smaller sets contained in larger ones.



(a) Counts of itemsets for every batch



(b) Sum of counts across partitions

2.2.5 Other considerations and final implementation of SON

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

followed by the broadcast of the variable to all nodes

```
  candidate_frequent_itemsets = baskets.context.broadcast(
    candidate_frequent_itemsets)

```

The second MapReduce pass is the same as before, because we didn't manage to identify unnecessary steps.

3 Apriori

Apriori is the most known 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 not frequent by itself. The algorithm 1 shows the Apriori pseudocode.

To benchmark SON execution we compared 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. Then we 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 itemsets 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 all 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 in fig.3 and fig.4, which will be further discussed in section 4.3.

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:

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

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:

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

It is considered "harder" because of the higher number of items (4059) and the large variability in basket size.

Since the most part of the algorithm consists in counting frequencies of itemsets in the dataset, we tried to optimize time a little bit, by converting the baskets of items from a list of lists to a list of sets. This way every "in" operation in python takes constant time instead of linear.

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 by providing the abstraction layer to work on a database like an RDD. The partitioner tries to distribute access load to the database and can be configured to accustom the specifics of every database. For instance the default partitioner is a *SamplePartitioner*, which distributes

the data based on pages count up to a maximum partition size, then there is the $PaginateBySizePartitioner$ which creates partitions of specified weight and $PaginateIntoPartitionsPartitioner$ which creates the reuired number of partitions, regardless of their weights.

The read data is in the form of a $DataFrame$. The main difficulty when using a $DataFrame$ is that the underlaying data is represented as rows, with specific access methods. Despite that, it appears to be a wrapper of an RDD which allows to even use SQL queries on the data. It is still possible to extract the RDD and use it directly.
That's what we did: we used the $DataFrame$ to read data from a database, but then extracted the RDD and used the commonly known methods to perform the actual computation to avoid struggling around the $DataFrame$.

We tried executing SON on a sharded database. Spark makes it transparent to the user, because the connector simply connects to the database master node and it is the database that handles the connection to the shards. On the other side, the setup of the sharded database is not trivial.
After the connection, spark provides a specific partitioner, the $ShardedPartitioner$ which theoretically creates a partition per shard and assigns it to the same machine, to maximize data locality.

We tried this configuration and although it is theoretically better for load distribution and allows exploiting data locality, it was difficult to setup the database sharding, it added complexity in handling and distributing data while not providing any performance improvement. In fact, working on a single machine it was impossible to properly test performances and although we didn't notice difference in execution time, we couldn't determine if that was because of our setup or configuration mistakes.
So, for semplicity, we settled on a single node database with a default partitioner.

4.3 Results

After some testing we figured that both the support and the partitions count played a major role in execution time, on top of the dataset.
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 a 6 core $i5-8600@3.10GHz$, with 16 GB DDR4 memory $@2666Mhz$, but the driver was capped at 4GB and executors were capped at 2GB RAM.

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 execution time as the dataset grows with 1 (fig.5) and 2 (fig.6) partitions.

2. **Reasonable support** that is more likely used in a real scenario. Here we tried to see ho performance varied with partitions, using the datasets

whole.
We tested with support 0.75 (fig.7) and 0.9 (fig.8).

We performed all these tests with our implementation of apriori, to show the components of execution time.

We noticed the overhead added by the second MapReduce phase of SON, which is not present in Apriori, using a single partition when the number of baskets is high (e.g. easy dataset) (fig.7 and fig.8).
This is especially visible when using efficient-apriori because the time spent in the first MapReduce phase is negligible compared to the second one (fig.4).

Not all the difference in time is due to the time taken by the second MapReduce phase. In fact, we found that the collect() operation takes a time proportional to the dataset size and it is not negligible. In our testing we considered it 4s for the 3 million rows of the easy dataset.
This *collect*() time does not depend on pending operations on the collected RDD and it slightly increases with the number of partitions. And this is just on one machine. We expect it to be much more significant when using a cluster with network times.

On the other side we could appreciate the speedup of SON with an higher number of items (e.g. hard dataset) in figs.**??** and fig.**??** as well as the loss in performance linked to lower support in each partition and the consequent higher number of candidates generated by Apriori (e.g. hard dataset, fig.**??** and fig.**??**).

Tests show a clear difference in execution time between the local and DataBase based algorithms (fig.7 and fig.8, easy dataset), despite SON being exactly the same.
This shows that handling the database brings in additional overhead. This is partly because of the additional operations we have to do to strip the data from the *row* structure and to convert it to a list of lists, but mostly due to the database access.
We also tested that the second phase of MapReduce takes a lot longer, although we weren't able to determine if it was due to the database access or to something else.

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 could proves highly beneficial to execution time on larger datasets.

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.

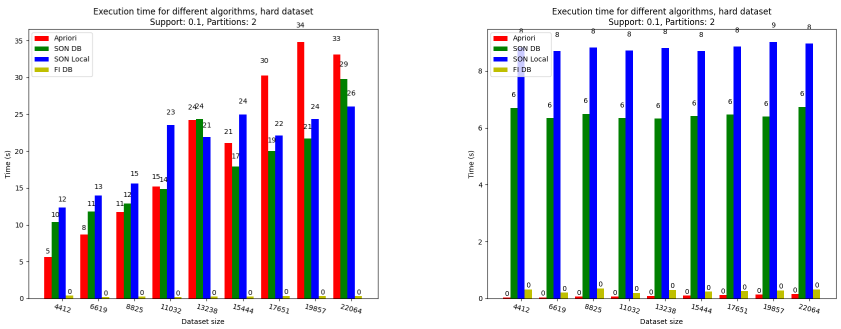
We have encountered an issue where, in some cases, the scaled support in a partition was so low that a frequency of one (or less) basket was considered frequent, which caused an explosion in the number of candidates generated by Apriori, not because they were actually frequent in the partition, but because 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.

We also noticed that the execution time of the second pass of MapReduce scales a lot with the number of baskets, which has to be considered when deciding how to partition the data.

Throughout the project, we have come to the understanding that using a distributed algorithm on spark is a balancing act and that even the simplest parameter, like the number of partitions and, consequently how distributable the data is, has to be perfectly balanced and fine tuned while keeping in mind hardware specifics, like the core count, the required support and the characteristics of the dataset to achieve maximum performance.

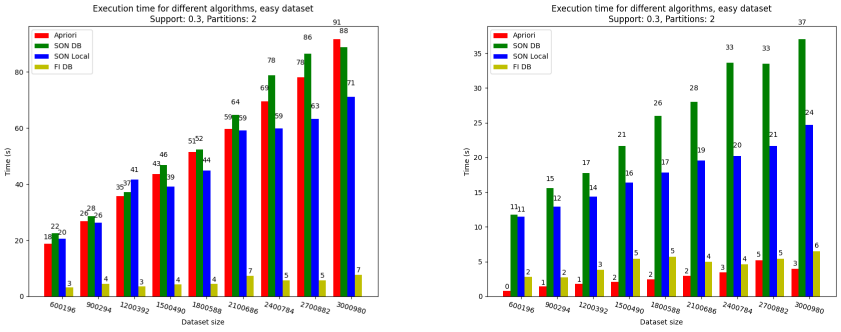
A Apriori benchmarks



(a) Apriori (b) Efficient-apriori

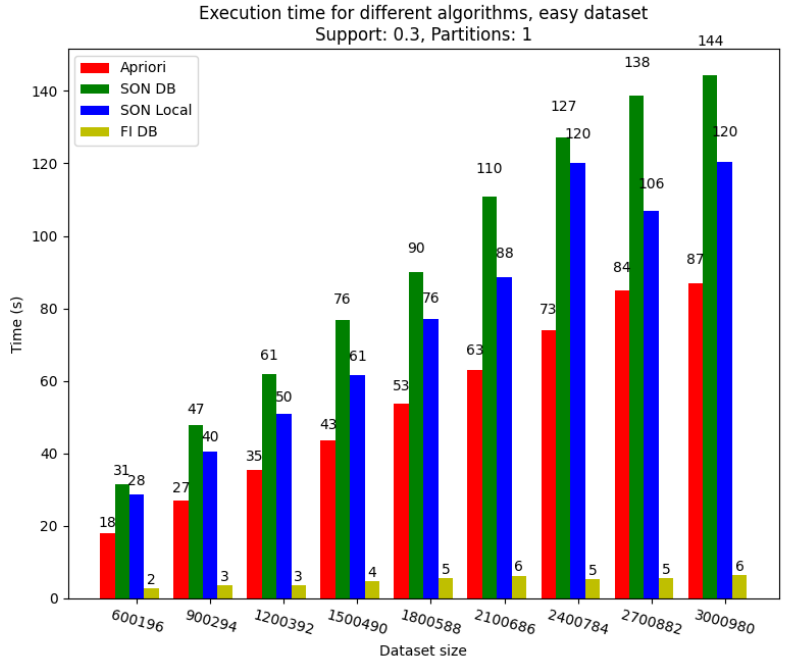
Figure 3: Apriori and Apriori2 benchmark results on the hard dataset

B Benchmark Images

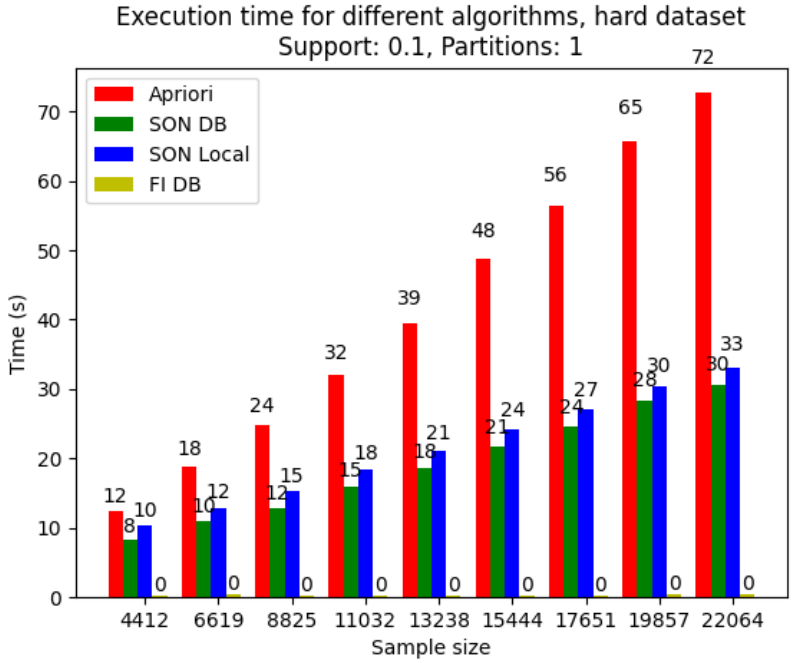


(a) Apriori (b) Efficient-apriori

Figure 4: Apriori and Apriori2 benchmark results on the easy dataset

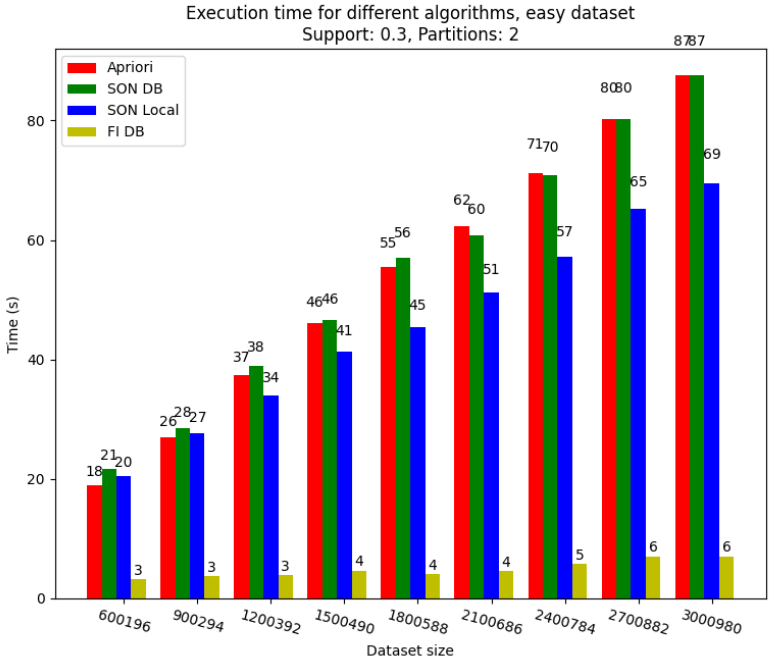


(a) Easy dataset

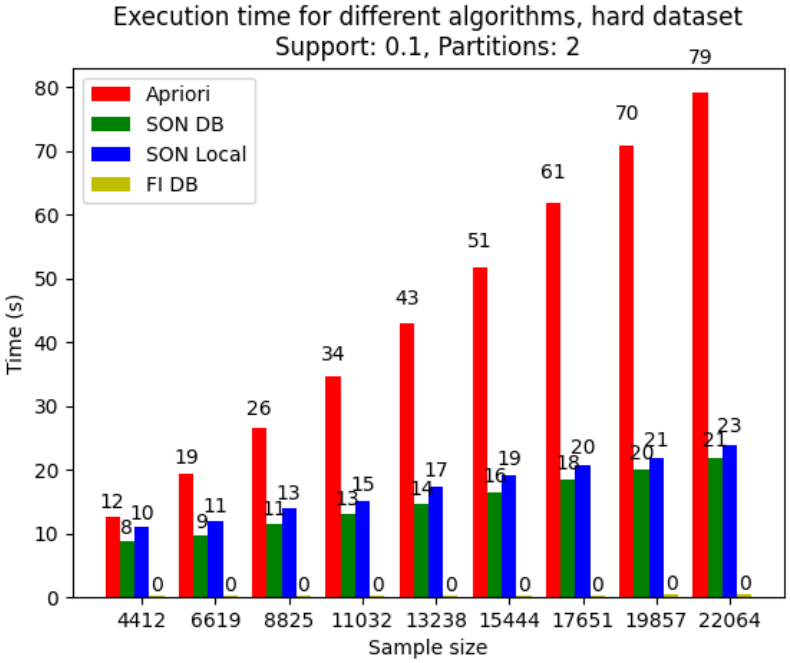


(b) Hard dataset

Figure 5: One partition, minimum support benchmarks

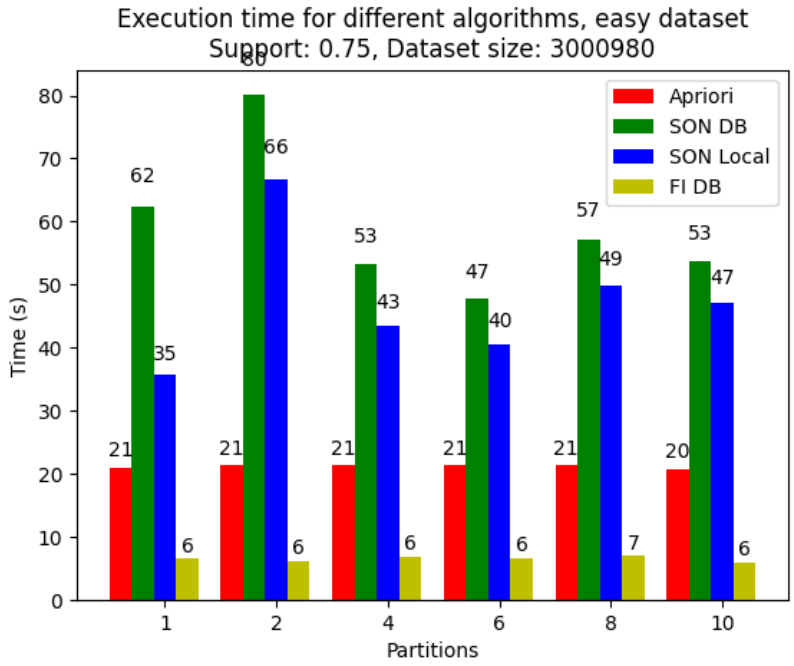


(a) Easy dataset

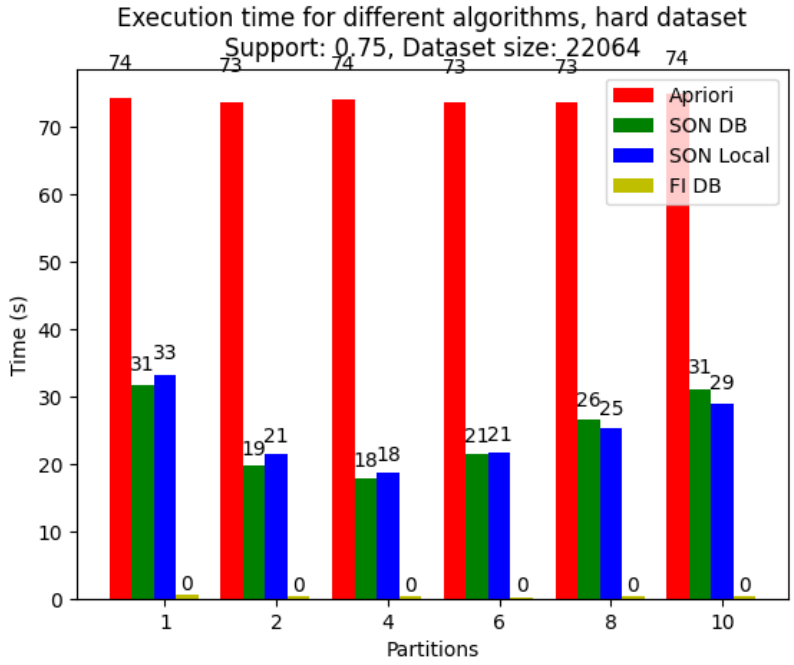


(b) Hard dataset

Figure 6: Two partition, minimum support benchmarks

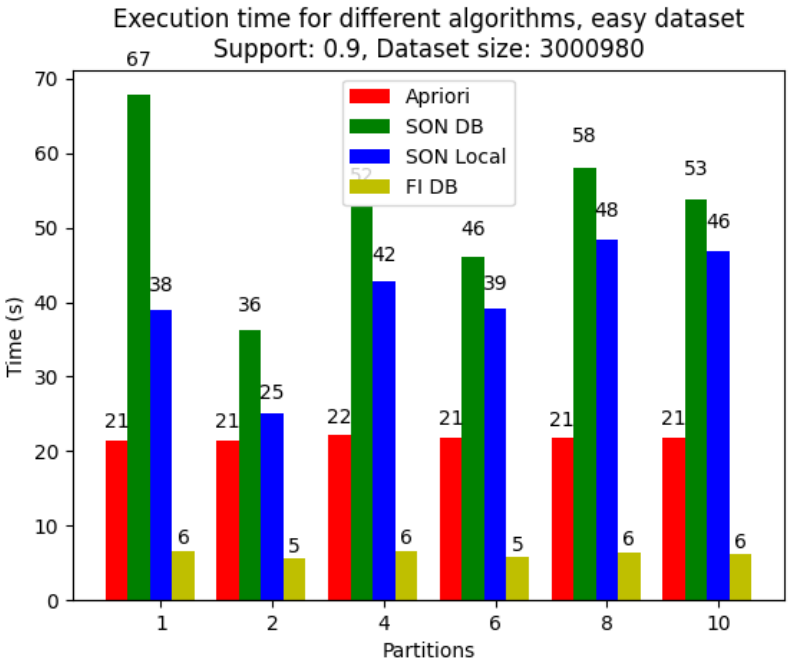


(a) Easy dataset

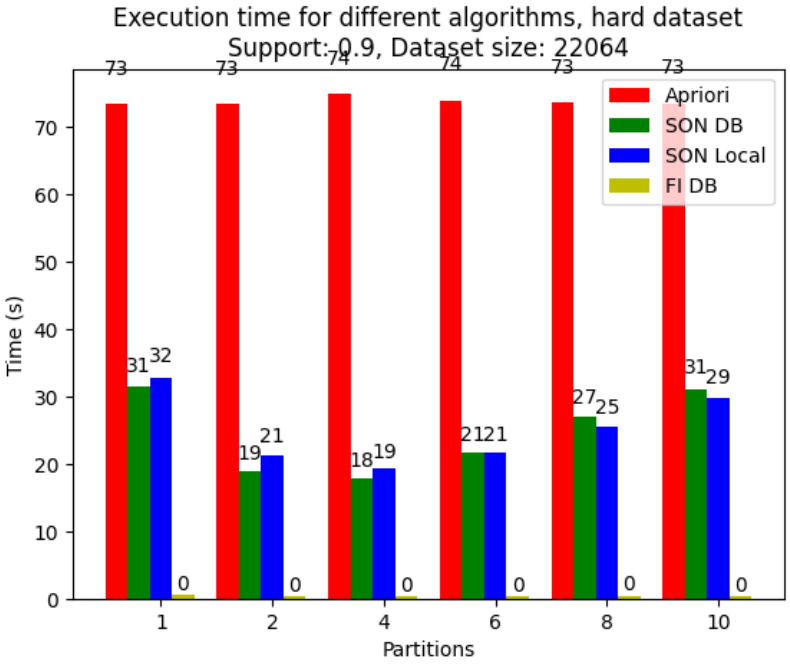


(b) Hard dataset

Figure 7: Partitions with 75% support, full dataset



(a) Easy dataset



(b) Hard dataset

Figure 8: Partitions with 90% support, full dataset