Mining Frequent Itemsets in Apache Spark

Andrea Galloni
University of Trento, Italy
andrea.galloni@studenti.unitn.it

Daniel Bruzual
University of Trento, Italy
daniel.bruzualbalzan@studenti.unitn.it

ABSTRACT

This document presents the work done for the *Big Data and Social Networks* course taken at UNITN during the first semester of 2015-2016. The project consisted of several steps.

First a dataset from traffic accidents in the USA[1] was selected and integrated to ensure consistency across different years. This involved studying the data, reading the description manuals, building a common schema and then converting the data with the help of special data integration software[2].

Afterwards, several known algorithms to calculate frequent itemsets were implemented. *Apache Spark* was used for implementing them in a parallel manner. A new parallel algorithm was also developed, which uses repeated random sampling and concepts from other algorithms to calculate frequent itemsets on big datasets while reducing execution time.

Finally, the algorithms were tested and compared using the integrated datasets, along with other classic datasets in the field of itemset mining.

Keywords

Frequent itemset mining; FP-Growth algorithm; SON algorithm; Apriori algorithm; Apache Spark; NASS General Estimates System (GES)

1. INTRODUCTION

The Frequent Itemset Mining problem consists of finding sets of frequently co-occurring elements in a given corpus of collections of elements. Typically, this problem has been studied under the "market-based" approach [10], in which the elements correspond to products or *items* and the collections of elements to transactions or shopping *baskets*. The problem has its origin in early 90s, when the progress in bar-code technology made it possible for companies to collect and store big amounts of sales data. Typically a date, and the items bought together in a transaction.

Using this approach, store or supermarket managers could identify groups of items which are frequently bought together and make decisions about product placement, promotions, marketing or product pricing. For example, a store might decide to make an offer on diapers, while slightly increasing the price of beer, since it has been determined that these products are typically bought together. In this way, the discount on the first item would attract the customer, whereas the increase on the second one would allow to make up for the lost profit.

Although nowadays, with the surge of online shopping and email marketing, which allow for person-tailored recommendations, other techniques are being used, frequent itemset mining is still valid for traditional retail stores and supermarkets. Moreover, there are also other applications outside the classical item-basket model. For example, one could apply this technique to documents and sentences, to find cases of plagiarism, to analyse click stream data and even to suggest keywords in search engines [6].

One common use of frequent itemsets is to determine association rules. These give a measure of how likely an item X is to belong to a basket, given that an item Y belongs as well. In the case of before, we could have the association rule {Diapers}→{Beer}, which represents that people who buy the first are likely to buy the second. However, the opposite doesn't necessarily hold.

Even though association rules give more interesting information to a store manager than the frequent itemsets themselves. Finding the frequent itemsets is a preliminary step in determining these associations, and is the step which involves most computational complexity, which is why we have decided to focus on this part of the problem.

As will be presented in the following sections, the frequent itemset mining problem is combinatorial by nature. It involves finding all the possible combinations up to size n of items and counting how many times they occur over the whole data. If we assume an inventory with a moderate size of 1000 products, we are already talking about half a million possible pairs of products and more than 150 million combinations of size 3 that need to be considered and counted over the whole collection of baskets.

Several optimizations exist that allow one to reduce the number of combinations generated, and checked. There is particularly an algorithm, FP-Growth, which allows the frequent itemsets to be calculated without explicitly generating sections.

The scope of this project was two-fold. On the one hand we chose a dataset of car crashes which occurred in the USA during the years 2010-2014, studying the data to see if it could be an applications of the frequent itemset problem. On the other hand we implemented and compared several algorithms, in attempts to find an efficient way to calculate frequent itemsets in a distributed manner.

[1]NASS General Estimates System
[2]Talend Studio

2. CONTENTS OF THIS PAPER

This paper is structured as follows. In section 3, the frequent itemset mining problem will be formally introduced, along with some interesting properties which algorithms use to trim the search space.

In section 4, previous works in this field are referenced.

In section 5, the characteristics of the NASS GES dataset are introduced, along with a description of the process done to integrate the data from different years.

In section 6, we present our implementations of different existing algorithms, as well as one of our own creation. This new algorithm is a hybrid which takes interesting concepts from already existing algorithms.

In section 7, the different algorithms are compared, and the results analyzed.

Finally, in section 8 we present our conclusions based on the theory and the results obtained in practice.

3. THEORETICAL PRELIMINARIES

To formally introduce the problem of frequent itemset mining, some preliminary definitions are needed. As stated before, the problem will be studied using the "market-basket" model.

DEFINITION 3.1. *Let the set:*

$$I = \{i_1, i_2, ..., i_n\}$$

be a set of n ***attributes*** *called items.*

DEFINITION 3.2. *The set:*

$$T = \{t_1, t_2, ..., t_j\}$$

be a set of j ***transactions*** *called baskets where:*

$$\{\forall e \in t_i \mid e \in I\}$$

DEFINITION 3.3. *Let $X \subseteq I$ we can define the cover of I as:*

$$K_T(X) = \{k \in \{1, 2, ..., j\} \mid X \subseteq t_k\}$$

DEFINITION 3.4. *The* ***support*** *of X:*

$$supp_T(X) = |K_T(X)|$$

DEFINITION 3.5. *The* ***minimum support****:*

$$s_{min} \in \mathbb{N} : 0 < s_{min} \leq j$$

Given those definitions the solution of Frequent Itemsets Mining problem can be expressed as the set:

$$F_T(s_{min}) = \{X \subseteq I \mid s_T(X) \geq s_{min}\}$$

Before introducing algorithms there is the need to highlight some properties who these exploit:

PROPERTY 1.

$$\forall t : \forall X : \forall Y \supseteq X : Y \subseteq t \Rightarrow X \subseteq t$$

If a set X is subset of Y and if Y is a subset of t then X is a subset of t, follows:

$$\forall X : \forall Y \supseteq X : K_T(Y) \subseteq K_T(X)$$

thus:

$$\forall X : \forall Y \supseteq X : supp_T(Y) \leq supp_T(X)$$

This means that if an set is extended its support can not increase. This is also known as ***anti-monotone property*** *of support.*

PROPERTY 2. *From the previous property directly descend that:*

$$\forall s_{min} : \forall X \supseteq Y : s_T(Y) < s_{min} \Rightarrow s_T(X) < s_{min}$$

That is: a super set of a non frequent item set is not frequent. This is also known as ***apriori property*** *of support.*

PROPERTY 3. *In contrapposition of the previous property one can say:*

$$\forall s_{min} : \forall X \subseteq Y : s_T(Y) \geq s_{min} \Rightarrow s_T(X) \geq s_{min}$$

That is: all subsets of a frequent item set are frequent.

4. RELATED WORK

As stated, the research in this field began in the early 90's. The first algorithm for finding all association rules, referred to as the AIS algorithm, was presented in 1993[2], during the same period another algorithm for this task, called the SETM algorithm, was proposed in [5]. Then a completely new approach was developed, namely *Apriori* and its variant *AprioriTid*, moreover an hybrid solution have been demonstrated to be the most performing in terms of time execution at that time[3]. The Apriori algorithms, thanks to some set theory properties, are based on the generation of a *reduced set of frequent itemsets candidates*, discarding *a priori* some combinations of items; the only drawback is that they need many database scans to achieve the solution.

Moving forward, a sampling method was proposed by Toivonen[9], which allowed to determine the frequent itemsets, without false positives or false negatives, by using a sample and a concept called *negative border*. However, there is the possibility that the algorithm may not always end.

During the following years, the research in frequent itemset mining moved to the use of tree based data structures that reduce the memory space needed to represent the transactions, also making use of parallelization[8][12] and reducing computational costs as well[1].

Nowadays the best performance is achieved by the *FP-Growth* algorithm and its variants. These solutions do not generate candidates itemsets explicitly, they can be run on distributed systems and require only two full database scans to compute the solution [4] [7]. Empirical tests have shown that these solutions are in magnitude one order less than the Apriori-based ones.

5. DATASET

5.1 NASS GES Dataset

The National Highway Traffic Safety Administration is an agency of the U.S. Government, whose one of its primary objectives is to reduce the human toll and property damage that motor vehicle traffic crashes impose on society. For this it collects and studies data of traffic accidents that occur throughout the entire country, and also releases it publicly. One of this data collections is the National Automotive Sampling System (NASS) General Estimates System (GES), which contains a sample of around 50.000 accidents of the more than 5 million which occur in the U.S. per year.

Data is available from 1988 to 2014, at the time of this research.

The data is presented in tabular format, which encodes into columns/attributes a detailed description of the circumstances surrounding each crash, like probable causes, weather, light conditions, alcohol involvement, vehicles involved, the roads where they occurred, etc. In total there are more than 50 variables that describe an accident. Each variable can take on a series of values (numeric) that represent a certain characteristic, as describe in the official encoding manual [11].

Our original idea consisted of using frequent itemset mining, and more specifically, association rules to find strong connections or implications between the crash variables. For example, one might expect that alcohol involvement might lead to a higher fatality rate, but we also expected to find other, let's say *hidden* implications. In order to do this, we first needed to integrate and clean the data, and then find a mapping of this problem into frequent itemset problem (i.e. baskets of items).

5.2 Data Integration

In order to have sufficient data to experiment with, we decided to integrate the last five years of records (2010-2014), corresponding to around 3 million records among accidents, vehicles, persons, and some others (120mb of data in total). Although the data integration was not technically hard, it was one of most time-consuming phases of this project. It involved going over the NASS GES manual and for every year seeing which columns had changed and how values had changed; deciding a new global schema and finally translating from the original files into this new schema.

In particular, some of the inconsistencies between years that were addressed were:

1. Columns changed names or were moved from one file to another.
2. Columns were split into several, or in some cases, some columns were merged into one.
3. Value encoding for a specific column changed in number, but the represented concepts remained the same.
4. Value encoding became more specific. (i.e. categories became more granular).
5. Rows with all many or most values mapped to "unknown".
6. Repeated information between the files that compose one year of data.

For this phase we used a software for data integration, Talend Open Studio [3], which offers a visual interface with which data flow and modifications can be easily visualized. This program allowed us to import the data, take the columns of interest, remap the values into our desired schema and export into a common format. Points 1 and 2 were easily dealt with by renaming or merging/splitting columns. Point 3 required mapping the old numbers into the new values. And in point 4, we opted for mapping the old value to one of the new, more specific values; or in some cases, we merged the specific values into a more generalized value. We removed the rows mentioned in point 6.

[3]https://www.talend.com/products/talend-open-studio

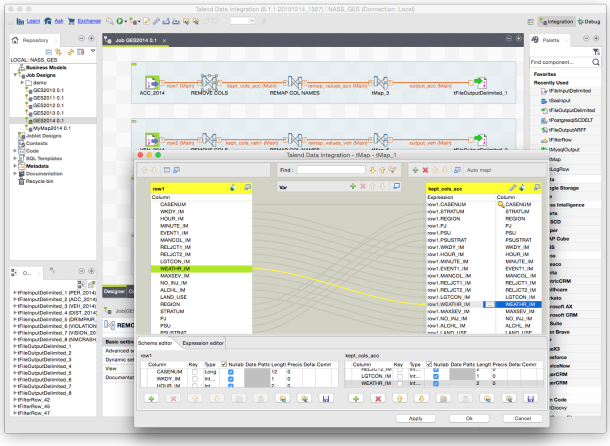


Figure 1: Talend Open Studio. Visual interface. Renaming/remapping columns.

In some cases, although the encoding was consistent, we decided to merge some very specific values into a more general category. Just as a manager in a supermarket might be interested in finding frequent itemsets without looking at specific brands (i.e. Milk and Beer), sometimes having too specific categories might prevent us from finding frequent itemsets. For example the values "crashed into a sign post" and "crashed into a utility post" could be merged into an encompassing one, namely "crashed into a post".

5.3 Data Transformation

To treat this data as a frequent itemset mining problem, we established a mapping from its domain into the frequent itemset domain, that is, baskets and products:

- Every traffic crash is considered to be a basket.
- The characteristics of the crash are considered to be the products in this basket.

Using this logic, we encoded every possible value of every column (i.e. the attribute domain) into a unique integer $a_{ij} \in 1, 2, ...n$, where i is a column and $j \in domain(j)$. In this way, every basket b_k would contain the integers that represent the values crash k contains in its columns. That is,

$$a_{ij} \in b_k \iff \text{Crash } k \text{ has value } j \text{ in column } i$$

There are different files for information related to the crash, the vehicles involved, the drivers, driver impairment, etc; and they are all joined by a *case number id*. By applying this transformation of attributes we were able to merge all files into the baskets pertaining to each crash.

Each basket was treated like a set, without repetitions. If two vehicles of brand `Fold` participated in a crash k, then `Fold` would be in b_k only once. However, if the crash was between a `Chevry` and a `Fold`, they would both appear in b_k. This is akin to the typical market-based approach to itemset mining, where we are interested in seeing *how many* people bought milk and cookies *together*, but not how many

milks or how many cookies. The same holds for every file where there can be many-to-one crash, like vehicles, drivers, or persons involved.

6. PROPOSED SOLUTION

This research project started by first choosing the dataset to work with, and then choosing the technique we would apply to it. However, after initial tests with serial algorithms and small test datasets we quickly realized that the frequent itemset mining problem is one that doesn't easily scale. Even with a collection of 10^3 items and 10^5 baskets, when we calculate all the possible itemsets of size 3, there are already more than 100 million candidates, which need to be counted over the whole baskets. If we take another approach, and generate the candidates from each basket, assuming a basket length of 10 items, there is only around 12 million of them. Although no algorithm is so naive as to generate all possible candidates, even taking advantage of properties that allow to significantly prune the search space, the complexity of the problem remains high.

For these reasons, we decided to spend most of our effort in testing different algorithms to see which one would scale better, and to design one of our own, taking the best parts of each, in hopes of being able to mine the frequent itemsets for our whole integrated dataset in reasonable time. If we could achieve this, then it could be easily be applied in other fields.

In particular, we implemented the following algorithms:

- A serial version of the Apriori algorithm, for comparison purposes.
- The SON algorithm[8], which breaks up the basket document into several chunks that are processed separately, after which the results are merged. We used Apriori on each node.
- FP-Growth, an algorithm which determines frequent itemsets without generating candidates explicitly.
- A new algorithm designed by us, which takes repeated random samples, in order to calculate the frequent itemsets without analysing the whole dataset.

We chose to work with Apache Spark[4], a cluster computing framework based on the map-reduce programming model. Using this framework, one can write programs in a functional paradigm, that get run on any configured cluster. Spark handles the distribution and handling of data between nodes, and is made to be easily scalable.

Spark currently supports Scala, Java and Python programs. For the first two, it compiles into java bytecode which gets run on a Java Virtual Machine. In the case of Python, the program is run with a wrapper, which translates the calls to the JVM. Although there is reason to belief that due to its dynamic nature Python can be slower, we decided to work with it over Scala or Java for a couple of reasons.

First of all, due to familiarity with Python, we could make better use of the data structures that would allow us to implement a better solution. Secondly, because of the availability of an interactive environment (PySpark) that allowed us to easily test and interact with our program without rerunning it every time, as opposed to Java. And finally, because we believe that run-time will be largely determined by how we use Spark's primitives to reduce the amount of data shuffled between the nodes and not the programming language itself.

[4]http://spark.apache.org/

6.1 Apriori Algorithm

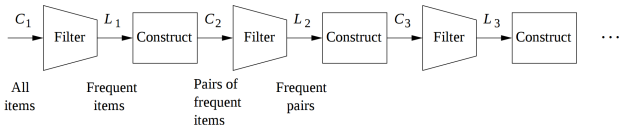
The Apriori algorithm[3] is based on property 2, *a superset of non frequent itemsets is not frequent*, from here comes the apriori name, in fact, this algorithm at every step generates all possible combinations using only frequent itemsets of the previous step. Then checks if, for every new combination, that all its immediate subsets are contained in the frequent itemsets from the previous step. Given this, we will show that the algorithm discards (prunes) a priori some itemsets skipping the count of certainly infrequent combinations.

At the first step the algorithm counts the number of occurrences of each item throughout all baskets, prunes those with support lower than the given minimum support and stores the rest in a list L_1. L_1 is in fact the list of frequent 1-itemsets (itemsets with cardinality 1). Only then, it generates a list C_1 of possible candidates based on L_1 (i.e. pairs of elements composed only by frequent 1-itemsets). This procedure implicitly discards, a priori, couples composed by non frequent 1-itemsets. In following steps this loop is generalized, L_{k-1} is used to generate the set of candidates C_k having cardinality k. The algorithm terminates when the set of C_k is empty.

Note that this algorithm, to accomplish the task, needs to read all over the database at every step.

The k-th step can be summarized as follows:

1. Generate C_k as the set of candidate itemsets of cardinality k from L_{k-1}.
2. For every element c in C_k count its occurrences scanning the whole database.
3. Generate L_k as the list containing only frequent itemsets of C_k. That is, those with count higher than the minimum support.



where the corresponding algorithm of the k-th pass is:

```
1  L_1 = { frequent 1-itemsets }
2  for (k = 2; L_{k-1} ≠ ∅; k++) do
3  |  C_k = apriori-gen(L_{k-1}); //New candidates
4  |  foreach transactions t ∈ D do
5  |  |  C_t = subset(C_k, t); //Candidates contained in t
   |  |    foreach candidates c ∈ C_t do
6  |  |  |  c.count++;
7  |  |  end
8  |  end
9  |  L_k = {c ∈ C_k | c.count ≥ minsup}
10 end
11 Result = ⋃_k L_k;
```

And the **apriori-gen** call corresponds to:

```
1 foreach itemsets c ∈ C_k do
2 |  foreach (k − 1)-subset s of c do
3 |  |  if s ∉ L_{k−1} then
4 |  |  |  delete c from C_k
5 |  |  end
6 |  end
7 end
```

6.2 The FP-growth Algorithm

The Apriori solution achieves good performance gains by reducing the size of candidate sets. However, in situations with a large number of frequent patterns, long patterns, or quite low minimum-support thresholds, an Apriori-like algorithm will suffer from two nontrivial costs: generating a huge number of candidate sets, most of which will not be frequent; and scanning the database repeatedly while checking a if a large set of candidates occurs or not in a basket.

The FP-Growth[4] algorithm reduces these costs by making use of a data structure called FP-Tree (an extension of a prefix tree). This method does not rely on candidate generation, and thus does not need to go over the database to count their occurrences. By using a prefix tree, FP-Growth reduces the space needed to represent the baskets, preserving the information needed to solve the problem. It requires only two passes on the entire database; the first one to count frequent items (singles), and the second one to build the tree structure. Once the tree has been built, it is recursively mined, using a divide and conquer approach to generate frequent itemsets.

6.2.1 FP-Tree

The FP-Tree is composed by a root (labelled as null), and nodes. Each node has four fields: *item_name* is the actual name of the item that the node is representing (in our case an integer), *counter* is the number of occurrences of this item, *node-link* a link to the next node in the tree with the same name - if it exists, and *parent* a link to its parent node. To facilitate the tree traversal, an *item header table* is built in which each entry points to its first occurrence (a node) in the tree. The actual procedure to build the tree follows these steps:

1. Scan the transaction database DB once. Collect F, the set of frequent items, and the support of each frequent item. Sort F in support-descending order, the list of frequent items $FList$.

2. Create the root of an FP-tree, T, and label it as "null". For each basket in the DB select only the frequent items in it and sort them according to the order of $FList$. Let this sorted frequent-item list be $\langle p, P\rangle$, where p is the first element and P is the remaining elements. Call insert_tree($\langle p, P\rangle, T$). The function call goes as follows. If T has a child N such that N.item_name = p.item_name, then increment N's count by 1; else create a new node N, with its count initialized to 1, its parent link linked to T , and link to it from the last node with the same item name, which is obtained by following the node-links via the *item header table*. If P is nonempty, call insert $tree(P, N)$ recursively.

In the given structure, transactions containing the same prefix will share some nodes. Since we ordered the baskets based on frequency, more frequent items will be closer to the root, and in this case be shared by more patterns. Since every path in this tree represents a basket, the maximum height of the tree is bounded by the length of the longest basket (after filtering).

Below is an example table of transactions, before and after filtering frequent items and with its corresponding FP-tree:

TID	Items Bought	(Ordered) Frequent Items
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

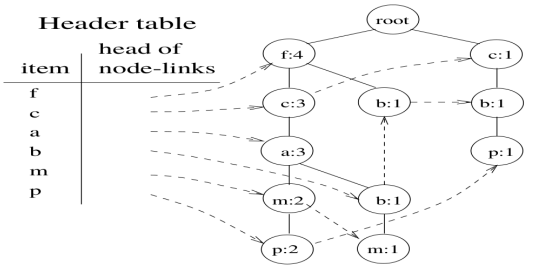


Figure 2: Example FP-Tree

6.2.2 FP-Growth

With this structure, for any frequent item a_i, all the possible patterns containing only frequent items and a_i can be obtained by following a_i's node-links, starting from a_i's head in the FP-tree header. This header list entry also contains a counter which indicates the sum of all counts of a_i in the tree. If the counts are bigger than the support threshold, we simply follow a_i's node-links, and for each node we save the prefix path to the root, and set the count to the same as a_i. This is called the conditional pattern base.

Once we have gathered the conditional pattern bases for a_i, we proceed to build a new FP-tree with them. This tree id called *conditional pattern tree*. Then, a recursive pattern-fragment growth mining method is used. Basically, generating the frequent patterns that contain a_i is equivalent to generating the frequent patterns of a_i's sub-trees and concatenating a_i to them. This corresponds to doing the procedure mentioned in the previous paragraph.

6.3 Apriori SON

These two algorithms form the base of most of the work that has been done in the field of frequent itemset mining. However, in order to scale to truly big datasets one has to find a way to parallelize them. In the case of Apriori, we based our focus on the SON algorithm[8], which makes use of an interesting property to partition the database of transactions into several chunks, which can be executed in parallel[12].

PROPERTY 4. *Let the itemset I be frequent in D (i.e. support(I) $\geq$ min_support). If the set of transactions D is split into n equal blocks, then I will be frequent in at*

least some block b with adjusted support (i.e $support_b(I) \geq min\_support \div n$*).*

On each chunk, some worker thread will run whatever method to find all frequent itemsets limited to it (with an adjusted threshold). The property guarantees that if an itemset is frequent in D, then it will be frequent in at least some chunck, so there will be no false negatives. After every worker has finished, all local itemsets are gathered by a master, and the real support of each one is counted (which can also be done in parallel). This second pass removes any false positives that might have been reported by a worker.

We implemented the SON algorithm on Spark, and let each worker run the apriori algorithm to find local frequent itemsets. The implementation can be summarized in four steps:

1. **Map**: Split the database into chunks and let every worker node execute the Apriori algorithm to find local frequent itemsets with adjusted support threshold. At the end every worker emits a set of value pairs $< item, n >$ where the item is a frequent itemset found, and n is the count (the count is not part of the algorithm).

2. **Reduce**: Gather all the itemsets produced by all workers. The reduce function takes care of merging those that are repeated. This creates the list of global itemset candidates, which might contain false positives.

3. **Map**: Broadcast the list of all itemset candidates to every worker node. Each worker node will then count the number of occurrences for each candidate itemset on baskets in the portion of the database that it was originally assigned. By using Spark's RDD.persist() method on the original data, we instruct Spark and the workers, to keep their chunk in memory, this saves us from re-reading the database and reduces the amount of data transmitted. This map function emits couplec $< item, value >$ where item is a frequent itemset and value is the actual count of its occurrences in this specific chunk of data.

4. **Reduce**: This reduce functions takes as input all values generated from all workers and collects all counts for each frequent itemset, akin to the classical word-count problem. Only then it will filter out itemsets with count smaller than the initial minimum support s (false positives), leaving only the truly frequent itemsets.

6.4 Distributed FP-Growth

The FP-Growth algorithm lends itself to parallelization due to its divide and conquer nature[6]. For this, the key is that every worker node is responsible for finding frequent patterns ending with a particular item. As mentioned in section **6.2.2**, several conditional pattern trees are built during the execution. In this case, each worker will build and mine only *some* of these trees.

It is worth noting that there exists an implementation of this algorithm in Scala inside the Spark Machine Learning Library[5]. But instead of using it we opted for building our own version in Python. This gave us more flexibility in testing and integrating the program with the rest of our code, and due to familiarity, the possibility to quickly adjust the internal data structures and workings of the procedures.

[5]http://spark.apache.org/mllib/

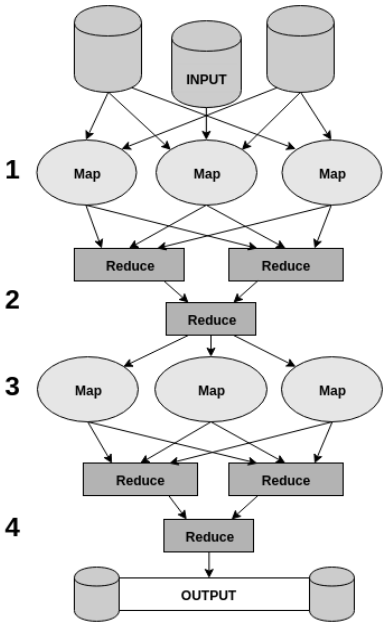


Figure 3: SON Apriori algorithm under the traditional MapReduce paradigm.

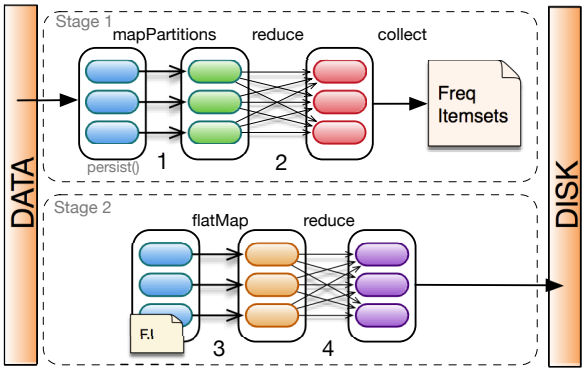


Figure 4: SON Apriori algorithm, expressed with Spark's operations.

However, this said, the major steps (*fig.* 5) are the same, namely:

1. Scan the database and count occurrences of every single item (Steps **A**, **B**). Then filter out the ones that are not above the support threshold.

2. Collect these frequent itemsets at the master and sort them according to their frequency in descending order (Step **C**). This implies a rank where the most frequent item will be assigned 0, the second most frequent a 1, and so forth.

3. For each basket translate the basket's items and order them according to the rank (most frequent items are

at the beginning of the list). Non frequent items are removed (thanks to the apriori property 2).

4. Then extract sub-patterns contained in it and *send* them to the node responsible, this is done by emitting a couple with the partition id as the key, and the sub-pattern as the value (Step **D**). We use a custom partitioning function, which is simply a modulus: $p(x) = x\%\text{numPartitions}$. In case a worker is in charge of many sub-pattern of the same basket, only the longest pattern will be sent (since it contains all others).

5. We call `aggregateByKey`, to gather all of workers sub-patterns in the same node (Step **E**). This effectively builds an FP-tree with all of them. Moreover, since tuples are *keyed* by their worker id, this step guarantees that each worker will have locally the information that he need to build his conditional pattern trees (the recursive step of the algorithm), and therefore eliminates the need to shuffle data during the conditional pattern tree building.

6. Now the algorithm can begin to look for frequent itemsets. From every generated tree we extract all patterns recursively by going over the *item header table* and for each item extracting and emitting frequent patterns from its conditional pattern tree (Step **F**). This is run locally on every worker.

7. Finally the previous encoded items are translated back to their original values (Step **G**).

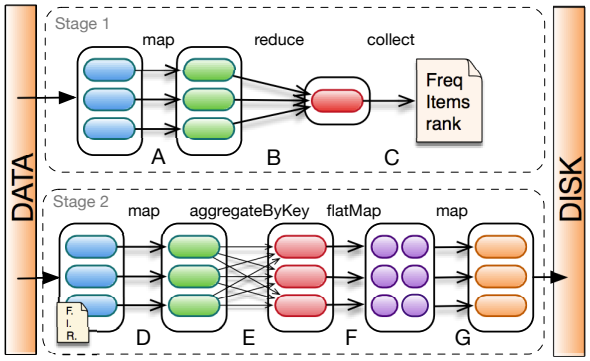


Figure 5: FP-Growth algorithm, expressed with Spark's operations.

6.5 BonGar

Based on some of the implemented algorithms, and some others[9] that we studied, we designed a new algorithm for mining frequent itemsets in huge datasets. Although the results for this algorithm were much worse than expected, we felt fair to include it in this report, along with the justification of its design.

The algorithm is called BonGar, short for *Bonsai Garden*, and this is precisely the idea behind it. It consists of taking several random samples, each of which can be processed by a different worker. Each worker uses the FP-Growth algorithm to find frequent itemsets restricted to that sample, and therefore builds a tree, smaller than if we would have taken the whole dataset.

The principles behind random sampling are two. Mainly, if something is highly frequent in the whole collection of transactions, then it should be frequent in a sample of the collections as well, with a minimum support adjusted to the size of the sample. The second one is that in many applications we are interested in finding most itemsets, but not necessarily all.

Each sample can be processed separately, by a different worker node. In this case we use FP-Growth, since it is the most efficient algorithm of the ones we tested. After each worker finds all his *locally* frequent itemsets, the results are merged at the master. Note that these are candidates, which might contain false positives.

The final step is to count the actual frequency of these candidates over the whole dataset. For this, we broadcast the merged candidates to the workers, and to each one a piece of the original dataset. Afterwords, a simple count and reduce, we can filter out the ones that were not above the minimum support, i.e. the false positives.

The problem with random sampling is that there might be false negatives as well. In order to reduce the number of false negatives, we multiply the minimum support at each worker by a relaxation factor $0 < \delta \leq 1$ which we typically set between 0.8 and 0.9. By doing this, we are reducing the false negatives, but at the cost of increasing false positives. However, these will be filtered out at the final step of the algorithm.Although the algorithm has no guarantee that it will find all itemsets, it guarantees that everything it reports is a frequent itemset.

The structure of the algorithm is as follows:

```
1  Input D:dataset, s:min. support, p:sample size, δ:
   relaxation factor
2  S = take-sample(D, p)
3  foreach π ∈ S do
4  |  I_π = FP-Growth(π, δ · p · s)
5  end
6  C = ⋃_{π∈S} I_π
7  foreach d ∈ D do
8  |  foreach c ∈ C do
9  |  |  if c ⊆ d then
10 |  |  |  count[c] += 1
11 |  |  end
12 |  end
13 end
14 L = {c ∈ C| count[c] ≥ s}
15 Return L
```

7. EXPERIMENTAL RESULTS

Our original idea of mining the almost 300.000 crashes that were collected in the NASS GES dataset seemed to be quite a challenge. Typically, as the problem of frequent itemset mining is presented, there is a very large collection of items and baskets are relatively short in length. Moreover, although a store can have millions of transactions, there is typically not very much in common between two baskets selected at random. In our case, however, after transforming the dataset into items/baskets we were left with only around

1.400 different "items". The average basket length was 65 items. This meant that most of the items were highly frequent, which ultimately made the complexity of the problem explode.

One particularity about this problem is that complexity rises, not so much with the number of transactions, but with the number of frequent items. As the minimum support threshold is set to be lower, the number of frequent single items grows, and this causes a factorial rise in the number of pairs, triplets, and so on.

In order to compare the performance of our algorithms, we tested them with datasets of different domains, some smaller and some bigger. The tables summarize their characteristics.

	num. baskets	num. items
Kosarak	990002	41270
Mushroom	8124	119
Retail	88162	16470
GES14	53030	1272
GES10-14	250978	1416

Table 1: Datasets used for testing

	basket length		item frequency	
	avg.	median	avg.	median
Kosarak	8.09	3	194.3	8
Mushroom	23	23	1570.18	600
Retail	10.30	8	55.16	11
GES14	64.55	65	11427.29	862
GES10-14	64.47	65	11427.29	862

Table 2: Basket length and item frequency

For every one, we tested with different values of minimum support (expressed in this case as a percentage), to see how performance was affected. Tests were done on a Machine with a dual-core Intel i5 processor, with 2 workers, each one with 1Gb of memory.

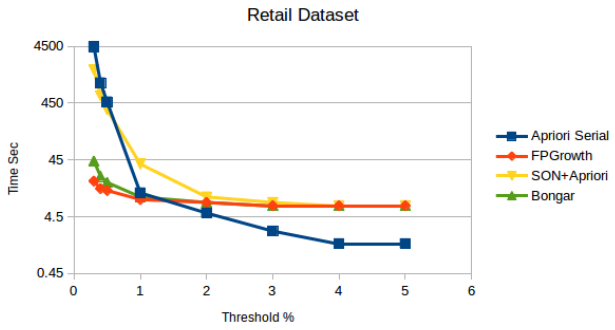


Figure 6: Retail Dataset Tests (Y axis log scale)

As predicted, the performance of the algorithms was much more affected by the ratio between number of different items and number of baskets, and by the average length of baskets,

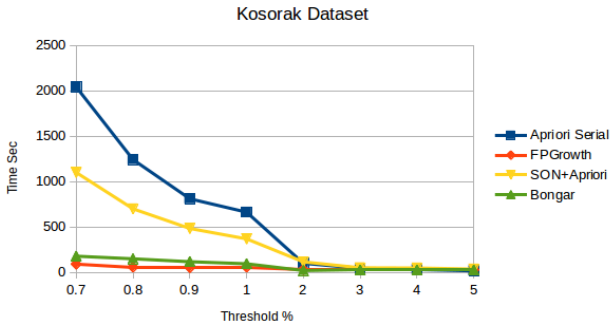


Figure 7: Kosarak Dataset Tests

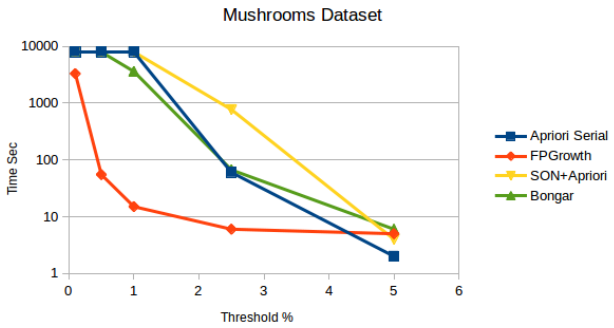


Figure 8: Mushrooms Dataset Tests (Y axis log scale)

than by the actual number of baskets. We can see for example, that *Kosarak* with 990.002 rows was processed by all algorithms fairly quickly, whereas *Mushroom*, with only 8124 rows, took several minutes and in some cases did not finish in reasonable time. This also suggests that some datasets are simply not suited for this solution.

Overall, FP-Growth was the best algorithm. It was able to solve even hard cases like Mushroom with a threshold of 1%. This makes sense, since this algorithm avoids generating candidates explicitly, which is the costliest operation of this problem.

BonGar has bad execution times, but this could be implementation dependent, due to some misunderstanding on our behalf of Spark internals. However, the positive results are that in all cases, very good approximations to the actual number of itemsets were obtained by using random samples. In this case we tested only with 2 samples of 10% of the data, since we run the programs on a two-core processor. However, if more worker nodes were available, we could run more samples, of a smaller size, on each one. However, the positive results are that when it did finish, it was able to find in most cases all of the itemsets by working only with random samples, and when it didn't find them all it came fairly close.

It is worth noting that in these experiments, the algorithms were let run to find all frequent itemsets of any size. In particular, if we look at the results of Mushroom with 1% threshold, there are 9.0751.401 frequent itemsets, which are quite senseless to calculate all, since it contains only 8.124

baskets and 119 items. By going back to the applications of this problem, like finding products bought frequently together, or doing keyword suggestions, it is likely that we are interested in finding itemsets of a limited length, 3 or 4, we might say.

Taking this into consideration, we modified our best algorithm (FP-Growth) slightly to see how it would handle mining the NASS GES dataset for the five years we had originally proposed.

7.1 Re-visiting the GES dataset

Upon an analysis of the GES dataset, it is quite noticeable that it possesses all the characteristics that make it a bad candidate (in terms of efficiency) for the frequent itemset mining algorithms. For starters, it has very few items in regard to the number of baskets.

If we go back and analyse the source of the data, we'll find that it corresponds to tables, for which some attributes have either:

- Very few possible values (e.g. boolean type), so it is expected that at least half of the baskets will have the item that corresponds to this value.
- Values which that are default for most crashes. For example, most crashes are bumper-to-bumper, so it is expected that this item will appear a large number of baskets as well.

So it is expected to have many items with very high frequency. Another particularity, is that since this dataset corresponds to a table, or actually several joined tables, the baskets tend to be fairly long (around 65 items on average). This also severely impacts the performance of our algorithms, since it could potentially generate itemsets of this size.

To confront these obstacles that the data imposed, we took two measures. First, alongside the minimum support threshold, we also set a maximum support threshold. In this way, we would be able to catch the items that are frequent, but discard the ones that are too frequent. This threshold is used in the first phase of the algorithms, when single frequent items are being calculated.

Secondly, we added a new parameter which is the maximum length of desired itemsets. By doing this, we are reducing the number of recursive calls in FP-Growth that correspond to mining the FP-Trees. Also, it makes no sense in this case to extract frequent itemsets of all lengths. In the worst case, we are interested in finding itemsets of 3-4 items, which could later be used to find association rules and prevent traffic accidents. In this case, simple is better. A too complex association rule will be harder to interpret and harder to transform into a public policy. Moreover, if left unbounded, the number of frequent itemsets grows to a point that it would be hard for any agency to process them all.

By observing the histogram of single item frequency in fig. 9, it is noticeable that around 100 items (of possible 1272) have an extremely high frequency. Ideally, we would place the lower and upper support thresholds in a way that it would discard those extremely frequent items, but also discard those that are not frequent at all. In the figure 10, the upper and lower thresholds can be seen in red. The items that have a frequency that falls between the lines are the ones that will be considered for the frequent itemsets.

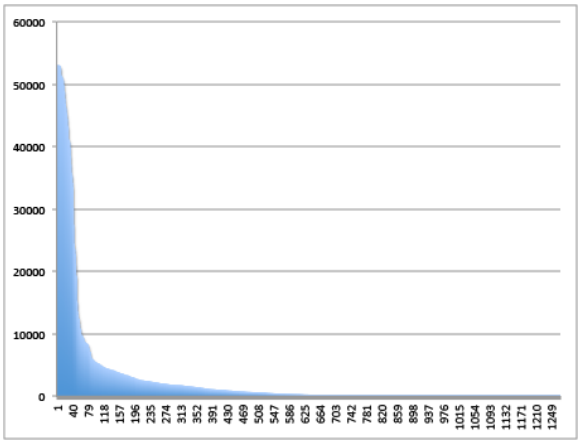


Figure 9: Histogram for the frequency of each item in the GES 2014 data

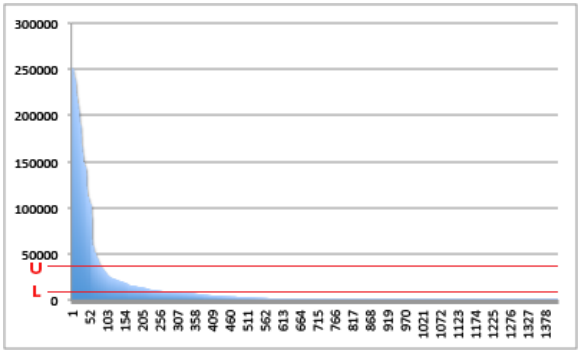


Figure 10: Histogram for the frequency of each item in the GES 2014 data. U corresponds to a maximum, and L to a minimum support threshold

By implementing this modifications, we set a minimum support threshold of 5% and a maximum support threshold of 20%, and then proceeded to test with different volumes of data, and achieved near-lineal scalability. As can be observed in fig.11.

8. CONCLUSIONS

In conclusion we implemented several algorithms both in a serial and parallel versions, we have invested our time on data integration, data analysis, data visualization and by understanding our results with different datasets, and their characteristics. We realized that the nature and structure of the data have a deep impact on performance, in some cases even more than the volume of data. We have also realized that catching hidden information and internal relations in big datasets is not trivial. Finally, we learnt that data integration is a task which requires a good understanding of the data and the domain it represents, and a lot of time.

More than one week was spent reading the NASS GES manuals and data documentation. The dataset was not consistent among the years selected, and we had to take decisions, based on the manuals and our understanding of the data, on how it could be integrated to a common schema. We learnt how to use specific tools for this purpose. But we also learnt that the hard part of data integration is not

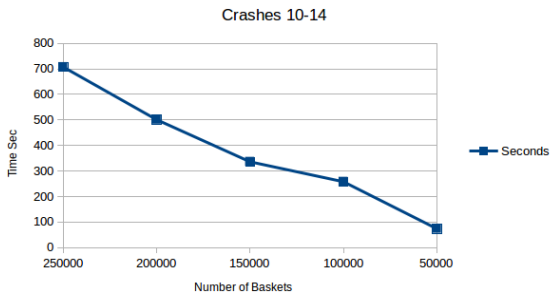


Figure 11: Scalability with Min and Max Support with Bounded Frequent Itemsets Length

necessarily technical, but more conceptual. That is, merging two models of a reality, into the same one.

A fair amount of literature was studied to understand the nature of the frequent itemset mining problem, to determine which solutions could be parallelized, and to develop our own algorithms for this purpose. Moreover, we had to think of ways to implement these parallel algorithms on top of Apache Spark. There is a saying that goes "To the man that has a hammer, everything looks like a nail". In this case, Spark was the hammer, and we had to find a way to transform our algorithms into nails.

The Spark framework clearly facilitates programming of distributed programs. However, by doing so it is hiding underlying implementation details that can be crucial for good performances. Sometimes, especially for the novice, it is possible to write a program that works but is extremely inefficient, due to misunderstanding of the way Spark handles data.

Overall, several versions of each algorithm were implemented, modifying data structures and procedures looking for better/faster implementations. Serial implementations were compared to parallel ones. However, we realized that testing and experimenting takes a lot of time, and that each modification to the algorithms represents a substantial amount of time invested. Also, the choice data structures during problem solving is fundamental, in our case a tree representation is not only useful in terms of saving space, but also reduced the complexity of the problem, since candidate itemsets did not have to be generated explicitly.

Our implementation of a new algorithm (BonGar) didn't achieve the results we expected, but presents some concepts that we believe deserve further studying. In the datasets tested, by taking 2 random samples of 10% of the data, and using a relaxation factor, it was able to consistently returns a good percentage of the frequent itemsets, and in most cases even all of them. However, it needs to be revised to make it scalable.

Going into the field of the data itself, it can be said that it is not always useful to mine very long itemsets (too specific information is provided), and this is the part that takes more time. By setting a limit on itemset length, runtime can be brought down substantially.

The NAS GES dataset doesn't appear to be a good candidate for the FIM problem, since there are very little items with very high frequency, baskets have a fixed length and boolean (or small range of values) columns, which are ultimately all frequent and generate noise on the discovered itemsets.

FIM has many applications, but has to be specifically tuned for each application. Thresholds have to be set based on the characteristics of the data, many parameters influence the problem, such as the number of attributes, baskets length, average of single item frequency and last but not least the number of data records (number of baskets). Once more it is proved that the first step in working with big data is knowing your data.

9. FUTURE WORK

Frequent Itemset Mining is a field with much work to be done. The applications of this problem have travelled far from the original market-basket scenario. Nowadays, as data collections become bigger, the solutions that worked for a supermarket analysing thousands of transactions, are no longer suitable. New methods have to be developed that allow for mining in the millions or millions of millions of "baskets". Moreover, they have to be parallelizable, since the data is huge and most likely distributed.

BonGar's concept of sampling proved interesting, since it was able to catch a good percentage of the frequent itemsets with just a small percentage of the data. However, further studies should be done to understand if its poor scalability is due to a misunderstanding of the inner workings of Spark, or if the algorithm as we proposed it, is simply not suitable for the framework.

Testing of the main algorithms (Bongar and FP-Growth) needs to be done on a bigger cluster and with more data to see how they scale on the number of nodes. In a networked environment, the effects of data shuffling will be much more noticeable than in a local environment, where they can get easily overlooked.

Finally, it would remain to extract the association rules from the different datasets, and see which values of threshold and itemset length make more sense for every type of data. In the particular case of NASS GES, it would be interesting to see if useful information can actually be extracted from these rules, that is, if it shows previously hidden associations that could lead to public policies and accident prevention.

10. REFERENCES

[1] R. C. Agarwal, C. C. Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of parallel and Distributed Computing*, 61(3):350–371, 2001.

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[4] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, pages 1–12. ACM, 2000.

[5] M. Hontsma and A. Swami. Set oriented mining for association rules in relatrend database. Technical report, The technical report RJ9567, IBM Almaden Research Centre, San Jose, California, 1993.

[6] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. Pfp: Parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pages 107–114, New York, NY, USA, 2008. ACM.

[7] J. Malviya, A. Singh, and D. Singh. An fp tree based approach for extracting frequent pattern from large database by applying parallel and partition projection. *International Journal of Computer Applications*, 114(18), 2015.

[8] A. Savasere, E. R. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. 1995.

[9] H. Toivonen et al. Sampling large databases for association rules. In *VLDB*, volume 96, pages 134–145, 1996.

[10] R. Ullman and Leskovec. *Mining of Massive Datasets*. Cambridge University Press, Cambridge, 2014.

[11] N. U.S. Department of Transportation. National automotive sampling system (nass) general estimates system (ges). analytical user's manual 1988-2014. November 2015.

[12] T. Xiao, C. Yuan, and Y. Huang. Pson: A parallelized son algorithm with mapreduce for mining frequent sets. In *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pages 252–257. IEEE, 2011.