



MINING OF MASSIVE DATASETS **FREQUENT ITEMSET MINING**

UNDER THE SUPERVISION OF
DR. SUBRAT DASH

Group Members:

Aakanksha Singh (21UCC002)

Aradhya Singh (21UCS026)

Ashwin Singh (21UCC128)

Shiven Gupta (21UCS195)



Abstract

Frequent itemset mining is a cornerstone in association rule mining, uncovering hidden relationships between items that are often purchased together. In this report we outline our working onto two foundational models: Apriori and PCY. The working of other more efficient, and advanced models are based on these 2 models, and hence it is primary to understand the application of the Apriori and PCY (Park-Chen-Yu) algorithms within market basket and groceries datasets — which is where frequent itemset mining finds its application the most. The datasets employed in this study are real-world transaction data, providing a robust foundation for evaluating the performance and scalability of both algorithms. The findings underscore the importance of algorithmic efficiency and contribute to the development of more effective data mining strategies in the retail sector. This in turn will help companies and industry increase their profits, by learning how the market operates at a subconscious level.

Explanation

A frequent itemset is a set of items that occur together frequently in a dataset. The frequency of an itemset is measured by the support count, which is the number of transactions or records in the dataset that contain the item set. More formally, given a set of items I , we say I is frequent if its support meets or exceeds a specific threshold (s). This threshold ensures that only itemsets occurring frequently enough to be of interest are considered.

Frequent itemsets are crucial for uncovering hidden patterns and help identify relationships between items that frequently occur together, revealing valuable insights about the data. In essence, they help characterize data by revealing associations and correlations among items.

For instance, in a retail context, discovering that customers often buy bread, milk, and eggs together can lead to optimized product placement in stores or targeted advertising. Similarly, in web usage mining, frequent itemsets can highlight common navigation paths, enhancing website structure and user experience.

After understanding the concept of frequent itemsets, we now turn to one of their major applications: the Market-Basket Model.

The Market-Basket Model

The market-basket model of data describes a common form of many-to-many relationships between two kinds of objects: items and baskets (sometimes called “transactions”). Each basket consists of a set of items (or an itemset), and the data is typically represented in a file consisting of a sequence of such baskets.

This model is a foundation in a data mining technique called “Market Basket Analysis”, which is used to uncover purchase patterns in retail settings. Here the “items” represent the different products that a store sells, and the “baskets” are the sets of items in a single market basket. By identifying frequent itemsets, retailers can learn what is commonly bought together. Pairs or larger sets of items that occur much more frequently than would be expected if the items were bought independently are of particular significance. This analysis helps in increasing sales while making the shopping experience more productive and valuable for customers by better understanding customer purchasing patterns.

Taking the instance of **bricks-and-mortar stores**, if analysis showed that magazine purchases often include the purchase of a bookmark, which could be considered an unexpected combination as the consumer did not purchase a book, then the bookstore might place a selection of bookmarks near the magazine rack.

Some Other Applications Of Frequent Itemsets

The original application (analysis of true market baskets) has already been discussed. However, applications of frequent-itemset extend beyond just market baskets. This model can be applied to mine many other kinds of data as well. Some examples are:

- **Related concepts**

In this context, items refer to words, and the baskets are documents. Each basket/document contains those items/words that appear in it. By ignoring all the common words, we aim to find frequent pairs of words that represent a joint concept.

- **Plagiarism Detection**

Items here are documents and the baskets are sentences. An item/document is “in” a basket/sentence if the sentence exists within the document. The objective is to identify pairs of items that frequently appear together in several baskets. Such pairs indicate documents sharing several sentences in common, often suggesting plagiarism. Even a small overlap of one or two sentences can serve as a significant indicator of potential plagiarism.

- **Biomarkers**

Items are of two types – biomarkers such as genes or blood proteins, and diseases. Each basket is the set of data about a patient: their genome and blood-chemistry analysis, as well as their medical history of disease. A frequent itemset that consists of one disease and one or more biomarkers suggests a test for the disease.

In market basket analysis, association rules are used to predict the likelihood of products being purchased together. We shall now explore the concept of Association Rules.

Association Rules

Frequent sets of items from data are presented as a collection of if-then rules, called association rules. Association rules count the frequency of items that occur together, seeking to find associations that occur far more often than expected. Association rule mining aids in finding some interesting relations or associations among the variables of the dataset. It is based on different rules to discover the interesting relations between variables in the database.

This notion of probability can be quantified in terms of confidence of the rule $I \rightarrow j$ which is the ratio of the support for the union of I and $\{j\}$ to the support for I alone. In other words, the confidence of the rule represents the proportion of baskets containing all items in set I that also contain item j .

$$\text{conf}(I \rightarrow j) = \frac{\text{support}(I \cup j)}{\text{support}(I)}$$

The interest of an association rule $I \rightarrow j$ can be defined as the difference between its confidence and the fraction of baskets that contain j . That is, if i has no influence on j , then we would expect that the fraction of baskets including I that contain j would be exactly the same as the fraction of all baskets that contain j .

$$\text{Interest}(I \rightarrow j) = \text{conf}(I \rightarrow j) - \text{Pr}[j]$$

Algorithms like Apriori and Park Chen Yu are vital for efficiently identifying frequent itemsets, even in massive datasets. These algorithms employ diverse techniques to generate and prune itemsets, ensuring computational feasibility while delivering valuable insights

Apriori

The main focus is finding the frequent pairs only. Simple methods fail if there are too many pairs of items to count them all in main memory. The A-Priori Algorithm tackles this issue by reducing the number of pairs that must be counted, at the expense of performing two passes over data.

The key idea utilized here is “monotonicity” which means that if a set of items I appears at least s times, so does every subset J of I .

The First Pass: Count the occurrence of individual items

In the first pass, two tables are created. First one translates item names into integers from 1 to n . The other table is an array of counts; the i th array element counts the occurrences of the item numbered i . As we read baskets, we look at each item in the basket and translate its name into an integer. Next, we use that integer to index into the array of counts, and increment the count of that item by 1.

Note that the items that reach or cross the support threshold are considered frequent and we set the threshold s sufficiently high that we do not get too many frequent sets; **a typical s would be 1% of the baskets.**

Between the Passes

For the second pass of A-Priori, we create a new numbering from 1 to m for just the frequent items. This table is an array indexed 1 to n , and the entry for i is either 0, if item i is not frequent, or a unique integer in the range 1 to m if item i is frequent.

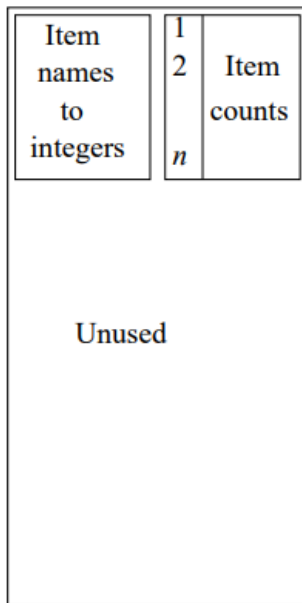
The Second Pass

During the second pass, we count all the pairs that consist of two frequent items (a pair cannot be frequent unless both its members are frequent). Thus, we miss no frequent pairs. Also notice that the benefit of eliminating infrequent items is amplified; if only half the items are frequent we need one quarter of the space to count.

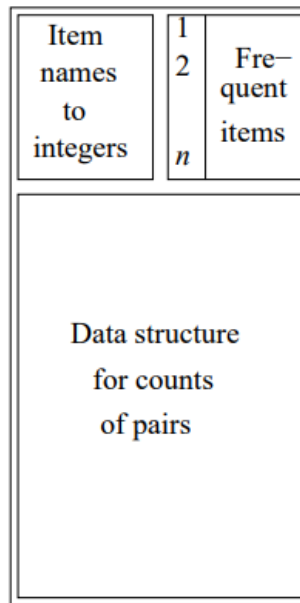
The mechanics of the second pass are as follows:

1. For each basket, look in the frequent-items table to see which of its items are frequent.
2. In a double loop, generate all pairs of frequent items in that basket.
3. For each such pair, add one to its count in the data structure used to store counts.

Finally, at the end of the second pass, examine the structure of counts to determine which pairs are frequent.



Pass 1



Pass 2

Limitation: The A-Priori Algorithm works well when the most memory-intensive step, usually counting candidate pairs, can be executed without thrashing, provided there is enough memory available.

Park Chen Yu

This algorithm takes advantage of the fact that in the first pass of A-Priori there is typically lots of main memory not needed for the counting of single items. Here, in addition to item counts, a hash table with as many buckets as fit in memory is maintained.

The First Pass:

For each bucket where pairs of items are hashed, a count is maintained. We hash each pair, and we add 1 to the bucket into which that pair hashes. The pair itself doesn't enter the bucket; it only influences the single integer within it.

After the first pass, each bucket holds a count, which is the sum of the counts of all the pairs that hash to that bucket. If the count of a bucket is at least as great as the support threshold s , it is called a frequent bucket.

The pairs that hash to a frequent bucket cannot be declared frequent or infrequent from the information available to us. But if the count of the bucket is less than s (an infrequent bucket), we know no pair that hashes to this bucket can be frequent, even if the pair consists of two frequent items.

Between the passes :

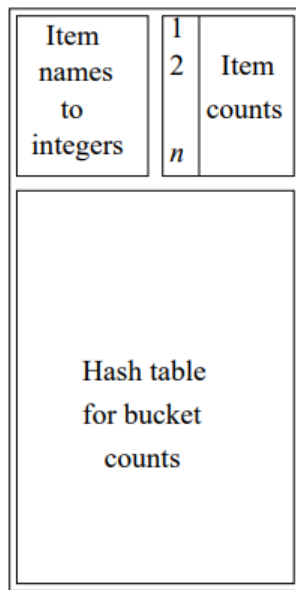
The hash table is summarized as a bitmap, with one bit for each bucket. The bit is 1 if the bucket is frequent and 0 if not. Thus integers of 32 bits are replaced by single bits, and the bitmap takes up only $1/32$ of the space that would otherwise be available to store counts.

The Second Pass:

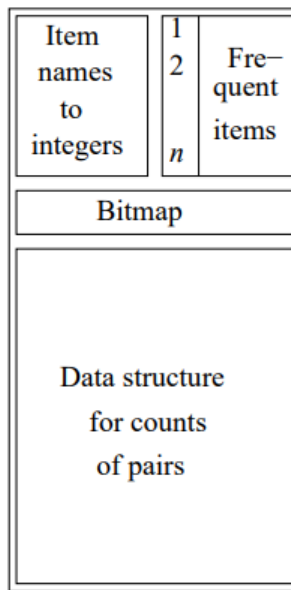
We can define the set of candidate pairs C_2 to be those pairs $\{i, j\}$ such that:

1. i and j are frequent items.
2. $\{i, j\}$ hashes to a frequent bucket.

It is the second condition that distinguishes PCY from A-Priori.



Pass 1



Pass 2

If the majority of buckets are infrequent, we expect that the number of pairs being counted on the second pass will be much smaller than the total number of pairs of frequent items.

Consequently, PCY is capable of managing certain datasets without experiencing thrashing during the second pass, a scenario where A-Priori would exhaust main memory resources and undergo thrashing.

Real World Example

Let's assume that we have the following list of shows that are on viewers' currently watching list on Netflix:

Session ID	Movies/Shows Watched
1	Stranger Things, Narcos
2	Breaking Bad, Narcos
3	Stranger Things, Black Mirror
4	Black Mirror, Narcos
5	Stranger Things, Narcos, Black Mirror

Now, Netflix wants to recommend its users new shows based on their viewing history. To work through this, let's compute the number of times all the shows (items) present in the whole dataset given:

{Stranger Things} appears in 3 sessions

{Narcos} appears in 4 sessions

{Breaking Bad} appears in 1 session

{Black Mirror} appears in 3 sessions

Now, let's define the **minsup** (minimum support) as 40% (of 5 sessions) and **minconf** (minimum confidence) as 70% (of 5 sessions).

Based on our newly defined parameters, it can be seen that only {Stranger Things}, {Narcos} and {Black Mirror} fulfill our **minsup** criteria.

Now we generate the next stage of frequent pairs:

{Stranger Things, Narcos} appears in 2 sessions

{Stranger Things, Black Mirror} appears in 2 sessions

{Narcos, Black Mirror} appears in 3 sessions

From these, it is ascertained that all the itemsets fulfill the **minsup** criteria. Hence we can now move onwards to developing frequent triplets; if possible:

{Stranger Things, Narcos, Black Mirror} appears in 1 session

It can be observed that the only 3 itemset does not enjoy a credible support, hence we can prune this itemset.

Now the second stage of Associate Rule Mining can be employed:

Since any triplet itemset is not frequent, we can only employ ARM for finding frequent pairs and improving recommendation systems:

- From frequent 2-itemset {Stranger Things, Narcos}:

Rule: Stranger Things \rightarrow Narcos

$$\text{Support}(\text{Stranger Things} \cap \text{Narcos}) = 2/5 = 40\%$$

$$\text{Confidence}(\text{Stranger Things} \rightarrow \text{Narcos}) = \text{Support}(\text{Stranger Things} \cap \text{Narcos}) /$$

$$\text{Support}(\text{Stranger Things}) = 40\% / 60\% \approx 66.7\%$$

This rule **does not meet** the minimum confidence threshold (70%).

- From frequent 2-itemset {Narcos, Black Mirror}:

Rule: Narcos \rightarrow Black Mirror

$$\text{Support}(\text{Narcos} \cap \text{Black Mirror}) = 3/5 = 60\%$$

$$\text{Confidence}(\text{Narcos} \rightarrow \text{Black Mirror}) = \text{Support}(\text{Narcos} \cap \text{Black Mirror}) / \text{Support}(\text{Narcos}) =$$

$$60\% / 80\% = 75\%$$

Hence, we can conclude that Netflix can recommend people that are watching “Narcos”, Black Mirror.

Methodology

I. Pseudocode: A priori Algorithm

Join Step: $C_{\{k\}}$ is generated by joining $L_{\{k-1\}}$ with itself

Prune Step: Any $(k-1)$ - itemset that is not frequent cannot be a subset of a frequent k -itemset

Pseudo-code : $C_{\{k\}}$ Candidate itemset of size k

$L_{\{k\}}$: frequent itemset of size k

$L_{\{1\}} = \{\text{frequent items}\};$

for $(k = 1; L_{\{k\}} \neq \phi; k++)$ do begin

$C_{\{k + 1\}} = \text{candidates generated from } L_{\{k\}}$

for each transaction t in database do

increment the count of all candidates in $C_{\{k+1\}}$

that are contained in t

$L_{\{k + 1\}} = \text{candidates in } C_{\{k + 1\}} \text{ with min_support}$

end

return $L_{\{k\}}$ for all k ;

Employing Apriori Algorithm

Dataset Used:

For implementing Apriori Algorithm:

The "Market_Basket_Data.csv" dataset contains all the items purchased in each transaction, with each row representing a single transaction. Besides the items, there are no additional attributes for the transactions, simplifying the preprocessing effort by eliminating irrelevant attributes. First, we'll use `pd.read_csv` to load the CSV file into a pandas DataFrame. The comma-separated items will then be converted from the DataFrame into a list of lists. Let's start by importing the CSV file and examining the dataset.

```
[2] # Load transactions data from the csv file
    # Store into list of strings called groceries

    transactions=[]
    with open('Market_Basket_Data.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
            transactions.append(row)
```

The data is loaded and below show the list of all transactions.

```
[3] # Check the list
    print(transactions)
```

```
... [['shrimp', 'almonds', 'avocado', 'vegetables mix', 'green grapes', 'whole weat flour', 'yams', 'cottage cheese', 'energy drink', 'tomato juice', 'low fat yogurt', 'green tea', 'honey']
```

To get a feel of how it is structured, the code chunk below print the first transaction.

```
[4] # Print the first transaction
    print(transactions[0])
```

```
... ['shrimp', 'almonds', 'avocado', 'vegetables mix', 'green grapes', 'whole weat flour', 'yams', 'cottage cheese', 'energy drink', 'tomato juice', 'low fat yogurt', 'green tea', 'honey',
```

Explore the transaction data:

Before proceeding, it's important to explore and comprehend the structure of the transaction data. Each row corresponds to an individual transaction. The transactions involve 120 different items, which are listed below.

```

from itertools import permutations

# Identify the unique items
flattened = [item for transaction in transactions for item in transaction]
items= list(set(flattened))
#print(items)

# Number of items
# Items from all the transactions
print(f'There are {len(items)} items.\n\nThey are {"", ".join(items)}')

There are 120 items.

They are cauliflower, pasta, soup, light cream, vegetables mix, honey, ketchup, ground beef, hot dogs, sandwich, chili, light mayo, chicken, parmesan cheese, yogurt cake, mint, whole w

```

Generating Association rules with itertools:

```

# Compute the possible rules
rules = list(permutations(items,2))
print(rules)

[('cauliflower', 'pasta'), ('cauliflower', 'soup'), ('cauliflower', 'light cream'), ('cauliflower', 'vegetables mix'), ('cauliflower', 'honey'), ('cauliflower', 'ketchup'), ('cauliflow

# Number of rules
print(f"This yields {len(rules)} of possible rules whereby we only had {len(items)} items and ignored multi-antecedent and multi-consequent rules.")

This yields 14280 of possible rules whereby we only had 120 items and ignored multi-antecedent and multi-consequent rules.

```

Filtering Data : Bought More than 1 items only

Our main goal in market basket analysis is to identify the relationships between two or more items bought together from transaction data. Therefore, transactions involving the purchase of only one item won't provide valuable information for revealing these item associations.

```
onehot = onehot[(onehot>0).sum(axis="columns")>=2].reset_index(drop=True)
print(display(onehot))
```

[16]

Creating and visualising a dataframe for identifying support for the itemsets:

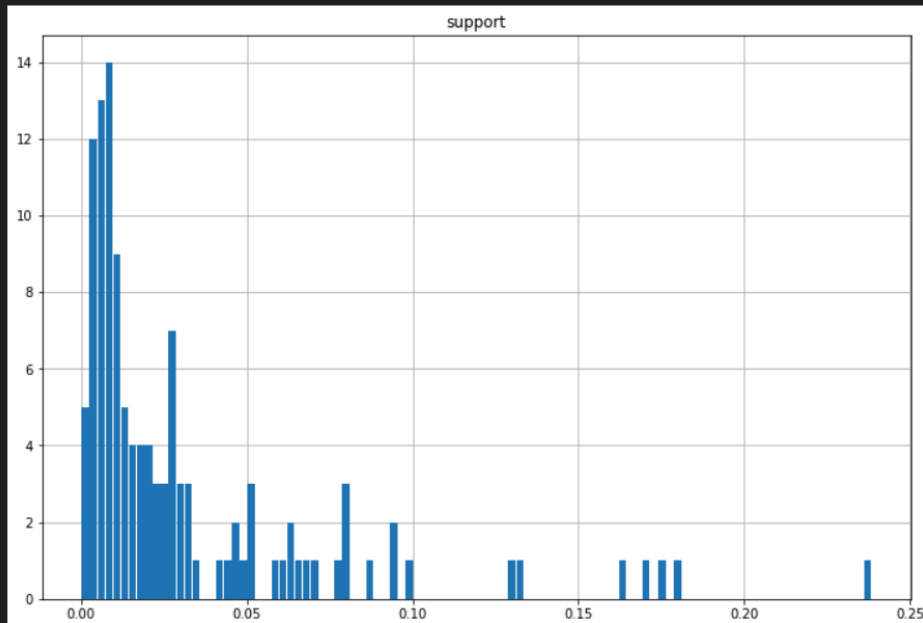
```
import matplotlib.pyplot as plt

support_df=pd.DataFrame(support,columns=["support"])
support_df.hist(bins=100,figsize=(12,8),zorder=2,rwidth=0.9)
```

[17]

```
... array([[<AxesSubplot:title={'center':'support'}>]], dtype=object)
```

...



Implementing Apriori algorithm keeping minimum support to be 0.005:

Apriori Implementation

```
# Compute frequent itemsets
frequent_itemsets = apriori(onehot,
                             min_support=0.005, max_len=3,
                             use_colnames=True).sort_values('support', ascending=False).reset_index(drop=True)

frequent_itemsets['length'] = frequent_itemsets["itemsets"].apply(lambda x: len(x))
# Print number of itemsets
print(len(frequent_itemsets))
print(frequent_itemsets)
```

```
1040
   support      itemsets  length
0  0.294936  (mineral water)      1
1  0.218897      (eggs)          1
2  0.218201  (spaghetti)          1
3  0.201148  (chocolate)          1
4  0.200104  (french fries)        1
...     ...             ...     ...
1035 0.005046  (salmon, cooking oil)  2
1036 0.005046  (spaghetti, eggplant)  2
1037 0.005046  (turkey, eggs, milk)   3
1038 0.005046  (green tea, eggs, milk) 3
1039 0.005046  (turkey, eggs, green tea) 3
```

[1040 rows x 3 columns]

```
rules1 = association_rules(frequent_itemsets, metric="lift", min_threshold=3)
```

```
print(display(rules1))
```

	antecedents	consequents	antecedent support	consequent support	support	confidence
0	(whole wheat pasta)	(olive oil)	0.037237	0.081956	0.010440	0.280374
1	(olive oil)	(whole wheat pasta)	0.081956	0.037237	0.010440	0.127389
2	(herb & pepper, mineral water)	(ground beef)	0.022272	0.124587	0.008700	0.390625
3	(ground beef)	(herb & pepper, mineral water)	0.124587	0.022272	0.008700	0.069832
4	(spaghetti, herb & pepper)	(ground beef)	0.021228	0.124587	0.008352	0.393443
5	(ground beef)	(spaghetti, herb & pepper)	0.124587	0.021228	0.008352	0.067039
6	(pasta)	(escalope)	0.018792	0.085958	0.007656	0.407407
7	(escalope)	(pasta)	0.085958	0.018792	0.007656	0.089069
8	(mushroom cream sauce)	(escalope)	0.024013	0.085958	0.007482	0.311594
9	(escalope)	(mushroom cream sauce)	0.085958	0.024013	0.007482	0.087045
10	(ground beef)	(tomato sauce)	0.124587	0.017922	0.006960	0.055866
11	(tomato sauce)	(ground beef)	0.017922	0.124587	0.006960	0.388350
12	(pasta)	(shrimp)	0.018792	0.090830	0.006612	0.351852
13	(shrimp)	(pasta)	0.090830	0.018792	0.006612	0.072797
14	(chicken)	(light cream)	0.076562	0.020010	0.005916	0.077273
15	(light cream)	(chicken)	0.020010	0.076562	0.005916	0.295652
16	(ground beef, eggs)	(herb & pepper)	0.026101	0.060901	0.005394	0.206667

Conclusion

- The use of the Apriori algorithm and advanced filtering helps to determine potentially useful and interesting rules.
- In terms of item placement, we could suggest the groceries to put Whole wheat pasta, mineral water, and olive oil in a closer place either on the same shelf or any place closer to each other so that it will be easy for people to reach out to the items.
- We could suggest putting Whole wheat pasta, mineral water, and olive oil as a single bundle of products or special packs at a lower price compare which would attract customers to buy and generate more profit in return. We could suggest running discounts and promotions on the items. Whenever customer purchase:
 - pasta, mineral water, they will get a discount in buying olive oil
 - pasta, they will get a discount in buying escalope

Employing PCY

Pseudocode: Park Chen Yu Algorithm

Function PCYAlgorithm(transactions, supportThreshold):

// Step 1: First pass over the data

Initialize item counts

Initialize bucket counts

For each transaction in transactions:

 For each item in transaction:

 Increment item count for this item

 For each pair of items in transaction:

 Compute hash bucket for the pair

 Increment bucket count for this hash bucket

// Step 2: Identify frequent items and create a bitmap for buckets

Frequent items = items with counts \geq supportThreshold

Bitmap = array of Booleans for each bucket

For each bucket in bucket counts:

 If bucket count \geq supportThreshold:

Set corresponding Bitmap entry to True

Else:

Set corresponding Bitmap entry to False

// Step 3: Second pass over the data

Initialize list final-list of frequent pairs

For each pair (i,j) in candidate pairs:

 Compute hash value

 if((i && j in frequent set of size 1) & (corresponding bitmap is True):

 Count occurrences of (i,j) in each transaction

 If count \geq min support : append pair to final-list

Return final-list

Dataset used for PCY:

The 'Groceries_dataset.csv' contains three attributes, Member_number denoting a specific customer, the date of purchase and the item purchased 'itemDescription'. The dataset has 38675 rows in total. The primary resource that is used here is PySpark which enables one to perform real-time, large-scale data processing in a distributed environment using Python.

Since we need a list of transactions that happened on the same date and the same customer, we apply the **groupBy** operation on the columns 'Member_number' and 'date' which gives us all rows with the same Member_number and date values. Now aggregation is performed on the 'itemDescription' which gives a list of values of objects bought by a specific customer on a specific date.

```
[12] from pyspark.sql import functions as F
df_grouped = spark_df.groupBy(['Member_number', 'date']).agg(F.collect_list("itemDescription"))

[14] df_grouped.show(10)
```

Member_number	date	collect_list(itemDescription)
1000	15-03-2015	[sausage, whole m...]
1000	24-06-2014	[whole milk, past...]
1000	24-07-2015	[canned beer, mis...]
1000	25-11-2015	[sausage, hygiene...]
1000	27-05-2015	[soda, pickled ve...]
1001	02-05-2015	[frankfurter, curd]
1001	07-02-2014	[sausage, whole m...]
1001	12-12-2014	[whole milk, soda]
1001	14-04-2015	[beef, white bread]
1001	20-01-2015	[frankfurter, sod...]

only showing top 10 rows

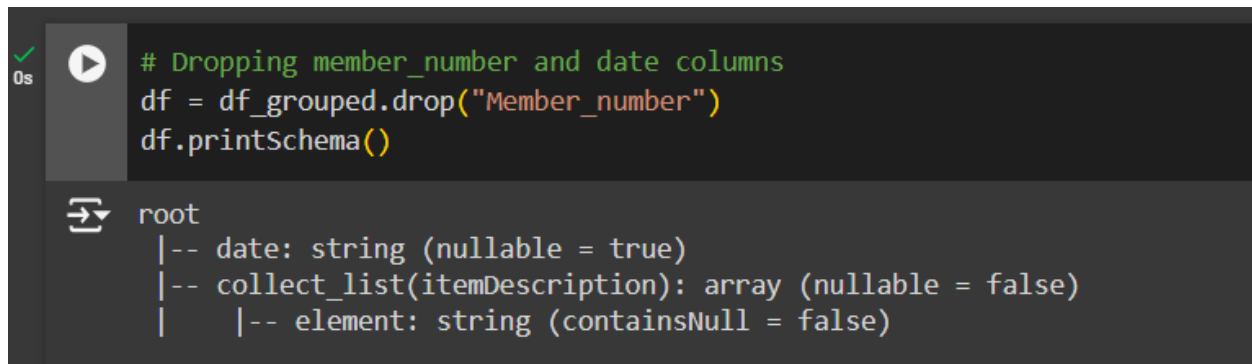
Data analysis reveals that the number of unique customers are : **3898**

```
print("Number of unique customers :")
df_grouped.select('Member_number').distinct().count()
```

Number of unique customers :
3898

Total number of unique transactions by a customer on a day : **14963**

We drop the Member and date columns since we won't be requiring these to calculate frequent itemsets



```
# Dropping member_number and date columns
df = df_grouped.drop("Member_number")
df.printSchema()
```

```
root
 |-- date: string (nullable = true)
 |-- collect_list(itemDescription): array (nullable = false)
 |     |-- element: string (containsNull = false)
```

The new data is saved to a csv file called 'transactions.csv'

Now in order to implement PCY , we define the minimum support threshold as **20** and the number of buckets to hash to as **20** as well.

To simulate an actual environment of big data manipulation, the 'transaction.csv' file is read as an RDD(resilient distributed datasets) which distributes the data across nodes for parallel processing. This is done by using pyspark's sc function. The openfile function distributes the file into buckets.

The first function of the PCY algorithm - size1freqset is then called with the defined parameters.

```
[24] def openfile():
    file_name = "transactions.csv"
    minsupport = 20
    nbuckets=20

    #doc = open(file_name).read()

    #newfile=doc.split("\n")

    data = sc.textFile(file_name)
    # data = data.map(lambda line : ''.join(line.split(',')))
    newfile = data.collect()

    buckets=[]
    for x in newfile:
        buckets.append([y.strip() for y in x.split(',')])

    size1freqset(minsupport,nbuckets,buckets)
```

Size1freqset : Function to filter out the singleton sets which meet the minimum threshold and can be passed on to the next phase.

1. Each item is stored into the '**candidatelist1**', set function is applied to this list to remove any duplicates.
2. For each k item in the candidate list we count the number of occurrences in each transaction and if the count is greater than minimum support we push the item in the **finalcandidate1** list.
3. **Itemdic** is a dictionary that maps each item to a unique id.
4. Calls size2freqset function to find out the frequent pairs.

```
def size1freqset(minsupport,nbuckets,buckets):  
    candidatelist1=[]  
    finalist1=[]  
  
    for i in range(0, len(buckets)):  
        for b in buckets[i]:  
            candidatelist1.append(b)  
  
    candidatelist1=sorted(set(candidatelist1))  
    # print(candidatelist1)  
  
    lala=0  
    for k in candidatelist1:  
        for i in range(0, len(buckets)):  
            for j in buckets[i]:  
                if(k==j):  
                    lala+=1  
            if lala>=minsupport:  
                finalist1.append(k)  
        lala=0  
  
    itemiddic={}  
    counter=1  
    for c in candidatelist1:  
        itemiddic[c]=counter  
        counter+=1  
  
    if finalist1:  
        print("Frequent Itemsets of size 1")  
        with open("output1.txt", "w") as f:  
            f.write("Frequent Itemsets of size 1 \n" + "Total number of itemsets : " + str(len(finalist1)) + '\n'.join(finalist1))  
        for x in finalist1:  
            print(x)  
        size2freqset(minsupport,nbuckets,itemiddic,buckets,finalist1)  
    else:  
        print("That's all folks!")
```

Size2freqset : The next function takes in the minimum hashes the pair of items into buckets

1. **Bucketcount** array to store counts of bucket and a bitmap is initialized and a pair list is initialized to store all the potential candidate pairs.
2. **PCY Pass 1** : The nested loop traverses each bucket and in each bucket iterates through each possible pair. Hashes each pair to a bucket and increments the bucket count. Each unique pair is added to the **'pairs'** list.
3. **Setting the bitmap** : Sets the bitmap values based on the corresponding bucketcount, 1 if more than min support otherwise 0.

```

def size2freqset(minsupport,nbuckets,itemidic,buckets,finalist1):

    k=2

    countofbuckets=[0]*nbuckets
    bitmap=[0]*nbuckets
    pairs=[]

    """PCY Pass 1"""
    for i in range(0,len(buckets)):
        for x in range(0,len(buckets[i])-1):
            for y in range(x+1,len(buckets[i])):
                if(buckets[i][x]<buckets[i][y]):
                    countofbuckets[int(str(itemidic[buckets[i][x]])+str(itemidic[buckets[i][y]]))%nbuckets]+=1
                    if ([buckets[i][x],buckets[i][y]] not in pairs):
                        pairs.append(sorted([buckets[i][x],buckets[i][y]]))
                else:
                    countofbuckets[int(str(itemidic[buckets[i][y]])+str(itemidic[buckets[i][x]]))%nbuckets]+=1
                    if ([buckets[i][y],buckets[i][x]] not in pairs):
                        pairs.append(sorted([buckets[i][y],buckets[i][x]]))

    pairs=sorted(pairs)
    """ Setting the bitmap """
    for x in range(0,len(countofbuckets)):
        if countofbuckets[x]>=minsupport:
            bitmap[x]=1
        else:
            bitmap[x]=0

    prunedpairs=[]

```

1. Now to prune the candidate pairs we check for two conditions :
 - a. If both the items in the pairs are present in the finalist1(if they are individually frequent)
 - b. If the pair has hashed to a frequent bucket
2. **Checking condition 1 of PCY Pass 2** : This loop prunes candidate pairs that don't contain items that meet the first condition. The pairs that satisfy this become a part of the list 'prunedpairs'
3. **Checking condition 2 of PCY Pass 2**: This loop works on the 'prunedpairs' list to decrease the amount of computations. It checks if the bitmap corresponding to the pair is set to 1 or not, if yes the pair is added to the 'candidatelist2' list.
4. **Appending frequent items of size 2 to finalist2** : The nested loop takes in the list of candidate pairs that meet the two conditions and counts the number of the occurrences in the transactions and if it meets the support threshold, adds the pair to the finalist2 containing the frequent pairs.

```

"""Checking condition 1 of PCY Pass 2"""
for i in range(0, len(pairs)):
    for j in range(0, len(pairs[i])-1):
        if (pairs[i][j] in finalist1 and pairs[i][j+1] in finalist1):
            prunedpairs.append(pairs[i])

candidatelist2=[]

"""Checking condition 2 of PCY Pass 2"""
for i in range(0, len(prunedpairs)):
    for j in range(0, len(prunedpairs[i])-1):
        if bitmap[int(str(itemiddic[prunedpairs[i][j]])+str(itemiddic[prunedpairs[i][j+1]])%nbuckets)]==1:
            candidatelist2.append(prunedpairs[i])

"""Appending frequent items of size 2 to finalist2"""
finalist2=[]
p=0
for c in range(0, len(candidatelist2)):
    for b in range(0, len(buckets)):
        if set(candidatelist2[c]).issubset(set(buckets[b])):
            p+=1
    if p>=minsupport and p!=0:
        finalist2.append(sorted(candidatelist2[c]))
    p=0

finalist2=sorted(finalist2)

if finalist2:
    print("\nFrequent Itemsets of size 2")
    with open("output2.txt", "w") as f:
        f.write("Frequent Itemsets of size 2 \n" + "Total number of itemsets : " + str(len(finalist1)) + '\n'.join(finalist2))
    for b in finalist2:
        print(' '.join(b))
    sizekfrequent(minsupport, nbuckets, itemiddic, buckets, finalist2, k)
else:
    print("That's all folks!")

```

Results

Sample of frequent set of size 1 :

```

🎮 openfile()
🔄 Frequent Itemsets of size 1
'Instant food products'
'UHT-milk'
'abrasive cleaner'
'artif. sweetener'
'baking powder'
'beef'
'berries'
'beverages'
'bottled beer'
'bottled water'
'brandy'
'brown bread'
'butter milk'
'butter'
'cake bar'
'candles'
'candy'
'canned beer'
'canned fish'
'canned fruit'
'canned vegetables'

```

Sample of frequent set of size 2 :

```
openfile()
Frequent Itemsets of size 2
'UHT-milk','UHT-milk'
'UHT-milk','other vegetables'
'UHT-milk','rolls/buns'
'UHT-milk','tropical fruit'
'UHT-milk','whole milk'
'baking powder','baking powder'
'beef','beef'
'beef','bottled water'
'beef','brown bread'
'beef','citrus fruit'
'beef','margarine'
'beef','newspapers'
'beef','other vegetables'
'beef','rolls/buns'
'beef','root vegetables'
'beef','soda'
'beef','whipped/sour cream'
'beef','whole milk'
'beef','yogurt'
'berries','berries'
'berries','other vegetables'
'berries','rolls/buns'
'berries','soda'
'berries','whole milk'
'berries','yogurt'
'beverages','beverages'
'beverages','other vegetables'
```

It can be seen that the set of trivial pairs with both items in the pair being the same have also come up in the list of frequent pairs, these can be ignored while drawing inferences.

Conclusion

- The PCY algorithm is an advancement of the A priori algorithm and helps in reducing the number of computations by limiting the number of pairs needed to be counted
- The grocery store can put the items belonging to a frequent pair next to each other because the customers are more likely to buy both items, such as putting beef with yogurt or beverages with vegetables to maximize the number of items sold per customer.
- The profits can also be maximized by discounting the prices of one of the products in the pair and increasing the price of the other, the discounts will lure in more customers, and since the customers are also likely to buy the other item in the pair, the total sales and the profits are maximized. For eg Advertise a discount on beef of

20% and raise the price of yogurt by 10%. The customers will come in the store to buy the beef noticing the discount advertised and will also buy the yogurt most likely not noticing the price increase.

References

- J. Leskovec, A. Rajaraman, J.D. Ullman: "Mining of Massive Datasets," Cambridge University Press, 2nd Edition, 2014
- https://www.researchgate.net/figure/The-pseudocode-of-the-Apriori-algorithm_fig4_227487186
- MMD lectures by Dr Subrat k Dash