Implementation and Optimization of Apriori Algorithm Using Spark Python API

GAO Yuanyuan *20657574*

LIU Xueling *20643822*

GU Yimeng *20634120*

WANG Zhili *20633504*

Abstract

Technology development makes it possible to discover association rules between items in a large database of sales transactions. Apriori algorithm is a classical method for mining association rules from a data set, but for large data set traditional apriori implementation is time-consuming. This article described main idea and key steps of apriori. Then we implement this algorithm using pure python and proposed 3 kinds of Spark parallel optimization implementations. At the end of the article, we conduct experiments on these implementations and discuss how running time changes with datasize, partition number and worker number.

1 Problem Description

We live in a fast-changing digital world. In today's age, customers expect the sellers to tell what they might want to buy, which brings a challenge situation for the retailers. Thanks to the rapid development of modern technology, it has made it possible to easily have access to well-organized database consisting of massive transaction records of customers, where latent purchase pattern and association among items in transactions could be derived later using some data mining algorithms. This kind of information-driven marketing process, managed by database technology, enables marketers to develop and implement customized marketing programs and strategies. Therefore, we've decided to focus on the problem of discovering *association rules* between items in a large database of sales transactions.

There are lots of algorithms that can achieve frequent items mining based on association rules, which can be classified into three main categories, namely Join-Based, Tree-Based, and Pattern Growth [1],etc. However, with the explosive growth of data volume, it has become much more difficult for those traditional algorithms to fit large dataset well, one of the most prominent problem of which is the poor efficiency, or in other words, they might be too time-consuming. Therefore, it is of great necessity to find a breakthrough on the platform of distributed and parallel computing.

Considering Spark platform is a distributed memory-based big data framework suitable for iterative computing, we propose a paralleling scheme of Apriori, a kind of Join-Based algorithm mentioned above, based on Spark so as to improve the mining efficiency of strong association rules [2].

2 Apriori Algorithm

2.1 The Main Idea of Apriori Algorithm

Before applying Apriori to the paralleling computing platform, let's have a brief look at the basic idea of the algorithm itself at first.

As a popular algorithm for extracting frequent itemset with applications in association rule learning, initially Apriori was designed to operate on databases containing transactions, such as purchases by customers of a store. And later on, Apriori has been applied to many other fields, such as recommender system design, and adverse drug reactions (ADR) detection in health care data.

An itemset is considered as *frequent* if it meets a user-specified support threshold [3], which could be an absolute value as well as a percentage. *Support* of the item X is nothing but the ratio of the number of transactions in which the item X appears to the total number of transactions. Another important concept is *Confidence*, measuring the percentage of times that item Y is purchased, given that item X was purchased. The key concept in the Apriori algorithm is that it assumes all subsets of a frequent itemset to be frequent. Similarly, for any infrequent itemset, all its supersets must also be infrequent, which is called *Apriori principle*. Based on this principle, Apriori algorithm finds large itemsets layer by layer: a large k-itemset is used to find possible large $(k+1)$-itemset.

2.2 Key Steps for Apriori

The pseudo code for the algorithm is given below. The *size* of an itemset means the number of items in an itemset. In the pseudo code, C_k represents candidate itemset of size k, L_k represents large itemset of size k.

Algorithm 1 Find large k-itemset

```
$L_1 = \{\text{large 1-itemsets}\}$
for $(k = 0; L_{k-1} \neq \emptyset; k++)$ do
    $C_k$ = candidates generated from $L_{k-1}$
    for transaction $t$ in database $D$ do
        if candidate in $C_k$ is contained in $t$ then
            candidate.$support++$
        end if
    end for
    $L_k$ = candidates in $C_k$ with minimum support
end for
return $\cup_k L_k$
```

Initially, we scan the whole database and find all large itemset of size 1 and denote it as L_1. Then, periodically, we use L_{k-1} to generate candidate itemsets C_k by joining L_{k-1} with itself. After that, the support of each candidate itemset in C_k is counted. Apriori uses Breadth-First Search and a Hash Tree structure to store and count candidate item sets efficiently [4]. For the next step, we apply Apriori principle to delete infrequent itemsets from candidate itemsets to generate L_k. This process is repeated to find large itemsets of size 1 to k, until L_k becomes an empty set.

3 Implementation in of Apriori Algorithm

3.1 Python Implementation

The first implementation is in python. We use python3 list to store our transaction, frequent itemsets and candidates.

```
with open('/dbfs/FileStore/tables/retail.txt') as f:
  for line in f:
    iteration.append(line.strip(' \n').split(' '))
```

Firstly, we use python built-in *map* to generate distinct candidate 1-itemset and then find large 1-itemset.

```
candidate_1 =
     list(set(list(itertools.chain.from_iterable(transactions))))
candidate_1 = list(map(lambda c: [c] ,candidate_1))
```

The process of generating L_k is shown in `prune`. To generate L_k, we scan the transaction using *for* loop to calculate each candidate's support. Furthermore, we use another *for* loop to filter out candidates that have low support.

```
def prune(transactions,Ck,support):
    # CSups candidate and supports
    CSups = {}
    for transaction in transactions:
        for candi in map(frozenset,Ck):
            if candi.issubset(transaction):
```

```
            if not candi in CSups:
                CSups[candi] =1
            else:
                CSups[candi] +=1


    numRecords = len(transactions)

    # FreSups frequent itemset and support
    FreSups = {}
    CurFre = []
    for candi in CSups:
        support_candi = CSups[candi]/numRecords
        if support_candi >= support:
            CurFre.append(candi)
        FreSups[candi] = support_candi

    return CurFre, FreSups
```

And then, we generate (k+1)-candidate set by union itemset in large k-itemsets. While union, we need to make sure that element in candidate set is distinct. This process is shown in `genCk` [5].

```
def genCk(Lk, k):
    Ck = []
    for i in range(len(Lk)):
        for j in range(i+1, len(Lk)):
            L1 = list(set(Lk[i]))[: k-2]
            L2 = list(set(Lk[j]))[: k-2]
            L1.sort()
            L2.sort()
            if L1 == L2:
                Ck.append(frozenset(Lk[i] | Lk[j]))
    return Ck
```

This implementation paves the way for optimization1 and its performance only serves as a reference.

3.2 Spark Parallel Optimization 1

In this implementation, we load transaction data into RDD. We use *mapPartition* to find large itemsets in parallel. For each partition, we apply previously mentioned python implementation to find local frequent itemsets. After finding all the local itemsets, we filter them by their support on the total transaction data in order to find global large itemsets. The layout is shown in Figure 1.

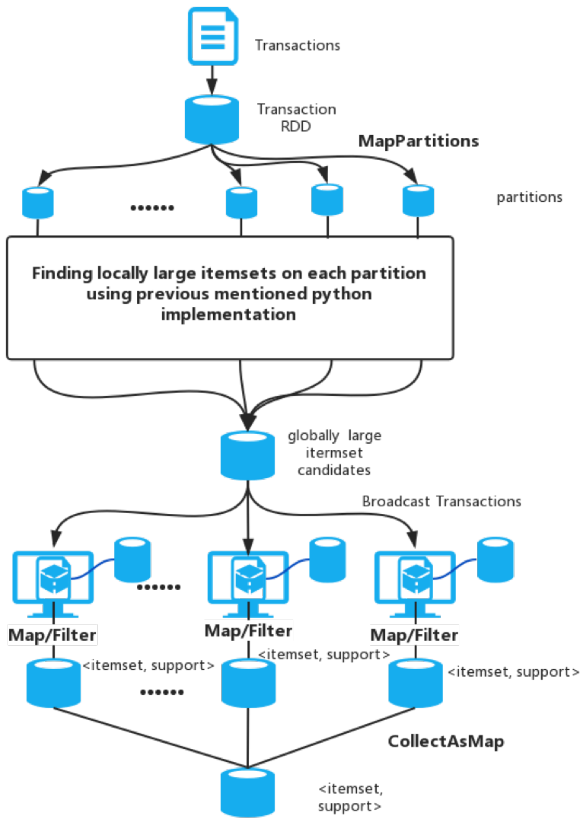

Figure 1: Layout of optimization using *mapPartition*

```
global_candidates = transationss.mapPartitions(lambda x:
     FindFreInPartition(x,support))
global_candidates = global_candidates.flatMap(lambda
     its:[frozenset(it) for it in its]).distinct()
```

If an itemset X is a global large k-itemset (i.e. large itemset in total transaction), then X must be a local k-large itemset (i.e. large itemset in partial transaction) as well [6]. Inversely, large k-itemsets found in each partitions contain large k-itemset of the whole dataset. So after applying *mapPartition* to retrieve local large itemsets in each partition, we combine them using *flatMap* and filter them using *map* and *filter* by minimum support on the complete transaction to find global large k-itemset.

```
original_data=sc.broadcast(transationss.collect())
global_frequent_itemsets=global_candidates.map(lambda
     x:(x,len([transaction for transaction in
     original_data.value if
     x.issubset(frozenset(transaction))])/numRecords))\
    .filter(lambda x: x[1]>=support).collectAsMap()
```

3.3 Spark Parallel Optimization 2

In this implementation, we load transaction data into "transations" RDD, split transatctions into several partitions. And use RDD transformations and actions directly to generate candidate itemsets and frequent itemsets.

```
#Parameter Initialization
K = 200#Number of Iteration
support = 0.1
candidate_sets = []
current_set_let = 1

input_rdd = sc.textFile('/FileStore/tables/retail.txt',8)
print(input_rdd.collect())

def splitlines(x):
    y = ' '.join(x.split())
    return y

splitedline = input_rdd.map(splitlines)
transations = splitedline.map(lambda x: x.split(' '))
```

Firstly, we should discover the large itemset of size 1 (i.e. L_1). We use *flatMap* to get all the items in transactions and use *map* and *reduceByKey* to count the supports of itemsets of size 1. Meanwhile, we prune the infrequent itemsets by predefined support threshold. As thus, we can get L_1.

```
#Discover the large itemset of size 1
candidate_1 =splitedline.flatMap(lambda x: x.split(' '))
FreqSet_1= candidate_1.map(lambda x: (x,1))\
.reduceByKey(lambda x, y: x + y)\
.filter(lambda x: x[1]>freqThre)\
.map(lambda x: (set([x[0]]),x[1]/numRecords))
print ('cur_freqset_1',FreqSet_1.collect())
preFreSets = FreqSet_1.map(lambda x: list(x[0])[0])
```

Secondly, we iteratively use L_k to generate L_{k+1}. The process of each iteration is shown in Figure 2.

For each iteration, we produce candidate itemset of size $k+1$ (i.e. C_{k+1}) using L_k. And then, we broadcast C_{k+1} to each worker. After this step, we use *MapReduce* method to compute L_{k+1}.

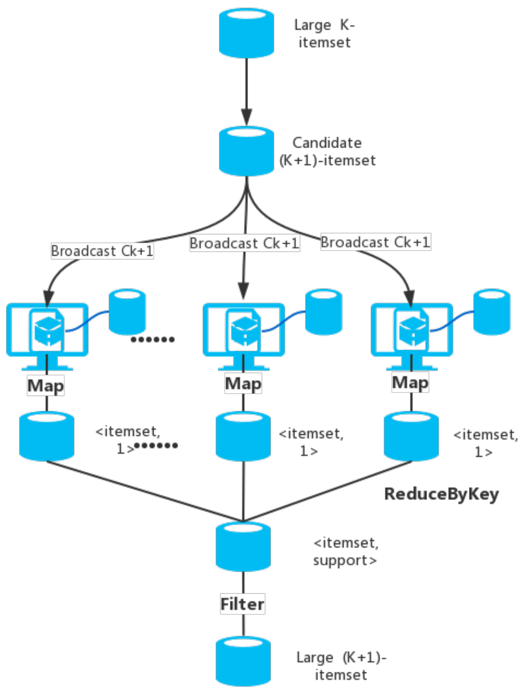

Figure 2: Generating large k-itemset

In the step of generating candidate itemsets C_{k+1}, start by splitting *preFreSets* RDD (i.e. frequent itemsets) in the previous round into elements, and merging them into one list called *elements*. Next, each itemset in the preFreSets is iterated in *flatMap* transformation. If an element in the *elements* does not contained in the itemset, merge this element with the itemset as a candidate itemset. Store these candidate itemsets into *candidates* RDD. Then, use *fileter* method to delete the candidate itemset that have infrequent subset from *candidates* RDD, thus we can get C_{k+1}.

```
elements = list(preFreSets.map(lambda x:
     set(x)).reduce(lambda a,b: a.union(b)))
```

```
canSets = preFreSets.map(lambda x: set(x)).flatMap(lambda
      t: JUDGEt(t,elements))\
         .map(lambda x: frozenset(x)).distinct().map(lambda
               x: list(x))
cantemp = canSets.map(lambda cset
     :ProduceNminus1Subset(cset,r,pre)[-1]).collect()
candidates = canSets.filter(lambda cset
     :ProduceNminus1Subset(cset,r,pre)[-1][-1])#
candidates = candidates.map(lambda x: frozenset(x))
print ('candidates in
     round'+str(r)+':\n'+str(candidates.collect())+'\n')
```

```
def JUDGEt(t,elements):
    for ele in elements:
        s= list(t)
        if (ele not in list(t)):
            s.append(ele)
            yield (set(s))

def ProduceNminus1Subset(cset,cur_round,pre):
    N=cur_round
    tu = []
    csets=list(cset)
    N = len(csets)

    for j in range(N):
        combo = []
        flag = 'TRUE'
        for i in range(N):
            if(i!=j):
                combo.append(csets[i])

        combo2 = set(combo)
        flag = frozenset(combo2).issubset(pre)

        tu.append(combo)
        if flag == 'FALSE':
            return tu

    return tu
```

In the step of deleting infrequent itemsets from candidate itemsets to generate L_{k+1}, because *transations* RDD is splited into several partitions, we have to broadcast C_{k+1} to each worker. Then we use *flatMap* and *reduceByKey* to count the support of all the candidate itemsets in C_{k+1}. And use *filter* to prune candidate itemsets that infrequent.

```
broadcastCandidates =
     sc.broadcast(frozenset(list(candidates.collect())))
curFreqSet = transations.map(lambda x: frozenset(x))\
            .flatMap(lambda transation :
                 verifyCandidates(transation,
                 broadcastCandidates.value))\
            .reduceByKey(lambda a,b: a+b)\
            .map(lambda x: (x[0],x[1]/numRecords))\
            .filter(lambda x :x[1] >= support)
```

```
def verifyCandidates(transation, candidates) :
    for c in candidates :
        if (c.issubset(transation)):
            yield (c, 1)
```

3.4 Spark Parallel Optimization 3

Layout of spark parallel optimization 3 is showed in Figure 3.

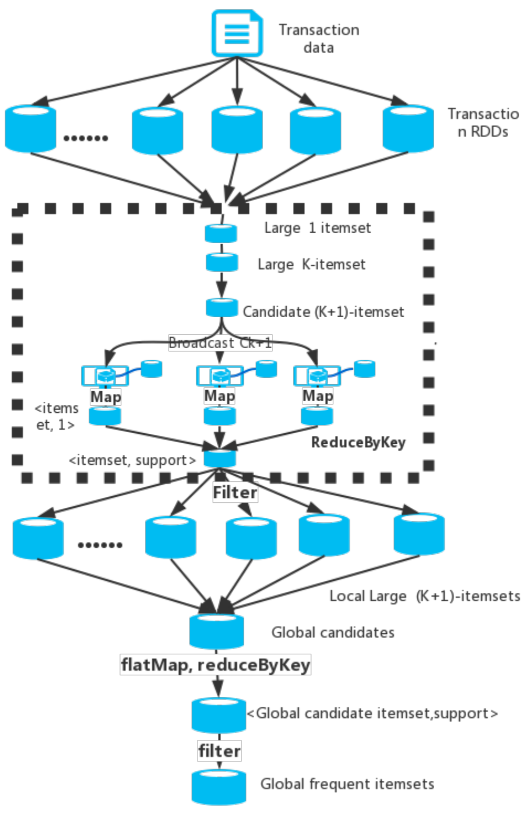

Figure 3: Spark Parallel Optimization 3

In this implementation, we split transaction data into several parts, and load them into different RDDs, and split these RDDs into several partitions. We apply Spark Parallel Optimization 2 on these RDDs sequentially to find local frequent itemsets and then merge them as global candidates.

```
spl_num=4
input_rdd =
     sc.textFile('/FileStore/tables/retail.txt',spl_num)


input_rdd2 = input_rdd.mapPartitionsWithIndex(getindex)
input_rdd2.cache()

frqtemp=[]
for irdd in range(0,spl_num):
  rdd_i = input_rdd2.filter(lambda kv:
       kv[1]==irdd).map(lambda x:x[0])
  print('rddi:\n',rdd_i.collect())
  splitedline = rdd_i.map(splitlines)
  transations = splitedline.map(lambda x: x.split(' '))
```

We broadcast global candidates to each worker node, thus every worker can scan global candidates directly from the broadcast value. Finally, use *flatMap* and *reduceByKey* to count the support of all the global candidate itemsets. And use *filter* to prune candidate itemsets that infrequent.

```
global_candidates_list =
     list(itertools.chain.from_iterable(frqtemp))
global_candidates =
     sc.parallelize(global_candidates_list,8).distinct()
print('global:\n',global_candidates.collect())

transationss = input_rdd.map(splitlines).map(lambda x:
     x.split(' '))
transationss.collect()
transationss.cache()
original_data=sc.broadcast(transationss.collect())
numRecords = float(transationss.count())


global_frequent_itemsets=global_candidates\
.map(lambda x:(x,len([transac for transac in
     original_data.value if
     x.issubset(frozenset(transac))])/len(original_data.value)
     ))\
    .filter(lambda x: x[1]>=support).collectAsMap()
```

4 Experiments

We totally do three kinds of experiments: changing size of datasets, changing rdd's partition number and changing executors' worker number to see the execute time on the above four algorithms. There are two kinds of time in databricks: total process time and total wall time. We choose total process time as our final running time.

4.1 Datasize Changing

We use four datasets to test the above four algorithms and the sample numbers of each dataset is 100,500,1000,10000, and 80000, at the same time, keeping rdd's partition number and executors' worker number unchanged which are both equal to eight. The testing results are in the following table:

algorithm	data size	100	500	1000	10000	80000+
3.1	runtime(s)					1990.525
3.2	runtime(s)	very long time	very long time	very long time	0.830643	2.085631
3.3	runtime(s)	0.833368	2.506395	84.029648	7.674532	1.216038
3.4	runtime(s)					9.979976

For algorithm 3.1, we only test when the data size equals to 80000, and we can see this method takes the longest running time, because this is a python method and doesn't use any parallel algorithm; For algorithm 3.2 which is a spark method and applies the method of algorithm 3.1 to every rdd's partitions, when the data size is smaller than 10000, we run almost one hour still can't get the result, however, When the data size is up to 10000, we can easily get results. As the data size increases from 10000 to 80000, the running time also increases a little. This phenomenon is a little confusing and future we will try to explain it; For algorithm 3.3 which is a standard spark methods, we can know that when the data size is small, the running time is increasing as the data size becomes larger, but when data size becomes large enough which is larger than 10000, the running time becomes shorter. It shows that spark can handle large data size efficiency because when the data size is large, the number of samples each partition should handle almost equal. For algorithm 3.4 which we do some improvements based on spark methods, we only test when the data size equals to 80000, and we can see the running time of it is longer than algorithm3.2 and algorithm3.3, because we split data into several parts, and load them into different RDDs, and split these RDDs into several partitions, then We apply algorithm 3.3 on these RDDs sequentially find local frequent itemsets, so it becomes less efficiency.

4.2 Partiton Number Changing

At this experiment, we just change the number of partitions from four to sixteen of each algorithm, and keep the data size equals to 80000, the number of executors' worker equals to eight. We use table and line chart to show the results:

algorithm	partition	4	8	12	16
3.2	runtime(s)	3.106728	2.108984	1.652818	1.574355
3.3	runtime(s)	1.408123	1.280455	1.207915	1.202925
3.4	runtime(s)	5.951884	9.96085	15.136773	19.86747

In order to see clearly what the trend is, we plot the below two line charts based on the table:

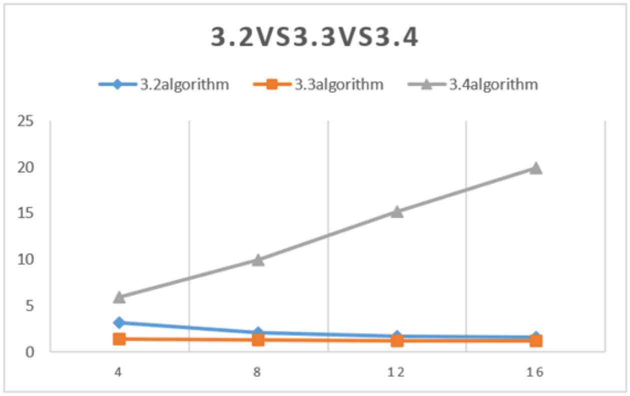

Figure 4: running time via partition number (1)

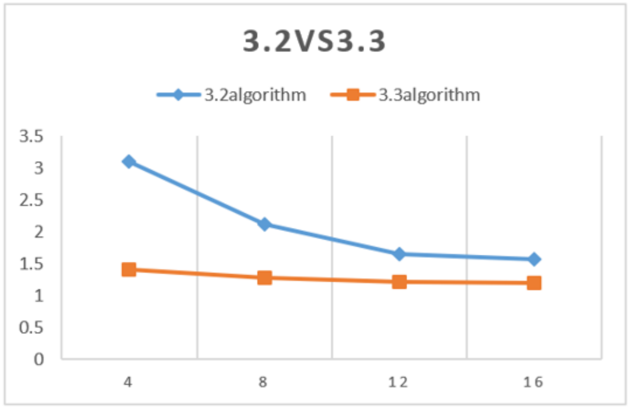

Figure 5: running time via partition number (2)

By analyzing the above two line charts, we find the running time of algorithm 3.4 increases when the number of partitions increases, while algorithm 3.2 and algorithm 3.3 both decrease. This is because algorithm 3.4 includes sequential execute methods and algorithm 3.2 and algorithm 3.3 both run parallel, so we get results that increase number of partitions decrease running time for the parallel methods, for number of data each partition should handle become small and then they run fast.

4.3 Worker Number Changing

In order to find the influence of worker numbers, for each algorithm, we just change the worker number from two to eight, and keep the data size equals to 80000, number of partitions equals to eight. The fianl results are shown follows:

algorithm	worker	2	3	4	5	6	7	8
3.2	runtime(s)	2.009037	2.275706	2.148303	2.06268	2.338704	2.409559	2.085631
3.3	runtime(s)	1.274926	1.231562	1.293001	1.231345	1.27078	1.199098	1.216038
3.4	runtime(s)	9.441765	10.372512	9.767561	9.823093	10.40366	10.29252	9.979976

Based on the above table, we plot another two line charts to show more information:

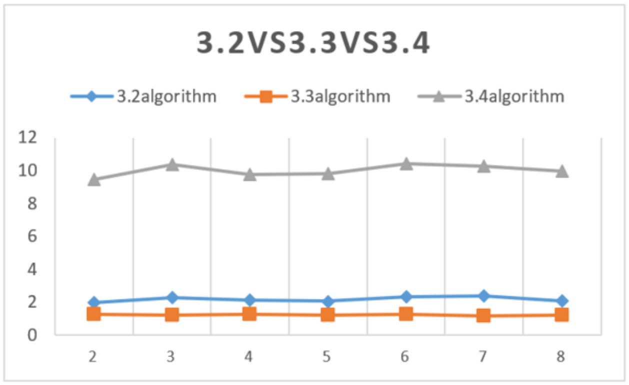

Figure 6: running time via worker number (1)

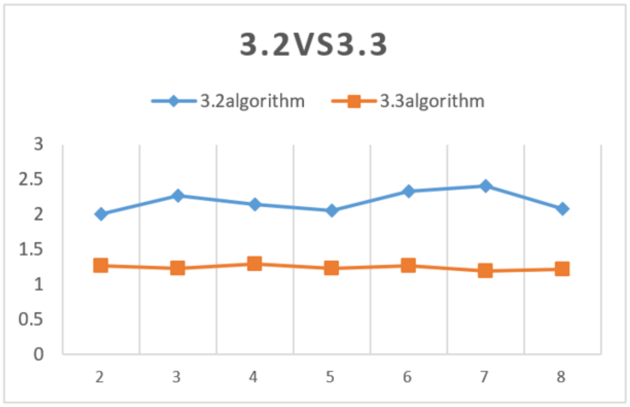

Figure 7: running time via worker number (2)

We can find that for all algorithms the running time almost unchange when the number of workers increase from two to eight.

4.4 Experiments Summary

After analyzing all the above three experiments' results, we can get that for a standard spark algorithm when increasing the data size to a large scale, the running time decreasing. As for parallel spark algorithm, increasing partition numbers can decrease running time. At last, changing worker numbers has little influence on running time.

5 Conclusion

In this report, we implement apriori algorithm in python (as a reference) and optimize it in spark python API in three ways: mapPartion, rdd operations and sequential rdds. After that, we carry out scalability experiments on three variables: dataset's size, number of partitions and number of workers. By comparing the above four implementations' performance on different values of these variables, we conclude that time consumption decreases when dataset's size inflates and when partition number shrinks. However, worker number seems to have no big influence on running time.

References

[1] Itemset mining. https://doi.org/10.1007/s10462-018-9629-z.

[2] Niu Xiaolu Liu Liping, Zhang Xinyou et al. A survey of spark-based parallel association rules mining algorithms. In *Computer Engineering and Applications*, volume 55(09), pages 1–9, 2019.

[3] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[4] Apriori algorithm. https://en.wikipedia.org/wiki/Apriori_algorithm.

[5] Union itemsets. https://bainingchao.github.io/2018/09/27/%E4%B8%80%E6%AD%A5%E6%AD%A5%E6%95%99%E4%BD%A0%E8%BD%BB%E6%9D%BE%E5%AD%A6%E5%85%B3%E8%81%94%E8%A7%84%E5%88%99Apriori%E7%AE%97%E6%B3%95/.

[6] D. W Cheung. A fast distributed algorithm for mining association rules. 1996.