**Apriori algorithm**

1 Jayanth Vikram, 2 S.L Karthik,3 Madhusudan Balajee, Karthik S

(1MS18CS051, 1MS18CS055, 1MS18CS064,1MS18CS054)

**ABSTRACT**

R. Agrawal and R. Srikant proposed the Apriori technique in 1994 to locate common itemsets in a dataset for boolean association rules. Because it relies on prior knowledge about common itemset qualities, the algorithm is called Apriori. We employ an iterative strategy, often known as level-wise search, in which we use k-frequent itemsets to find k+1 item-sets. An important attribute called the Apriori property is utilised to improve the efficiency of level-wise generation of frequent itemsets by minimising the search space. We have executed the apriori algorithm both sequentially and parallelly using threads and have done a performance analysis to compare which method was more efficient. We have executed the sequential part as a java code and the parallel code was executed using c++ and openMP. The time taken for parallel execution was found to be more efficient.

**INTRODUCTION**

Over relational databases, Apriori is an algorithm for frequent item set mining and association rule learning. It works by recognising the most common individual items in the database and expanding them to larger and larger item sets as long as those item sets exist in the database frequently enough. Apriori's frequent item sets can be used to create association rules that highlight general trends in the database, which can be useful in areas like market basket analysis. To effectively count candidate item sets, Apriori uses a Hash tree structure and a breadth-first search. From item sets of length displaystyle k-1k-1, it generates candidate item sets of length displaystyle kk. The candidates with an infrequent sub pattern are then pruned. The candidate set contains all common displaystyle kk-length item sets, according to the downward closure lemma. Following that, it searches the transaction database for common item sets among the candidates. Given below is the algorithm:

**Apriori**(T, ε)

L1 ← {large 1 - itemsets}

k ← 2

**while** Lk−1 **is not** empty

Ck ← Apriori\_gen(Lk−1, k)

**for** transactions t **in** T

Dt ← {c in Ck : c ⊆ t}

**for** candidates c **in** Dt

count[c] ← count[c] + 1

Lk ← {c in Ck : count[c] ≥ ε}

k ← k + 1

**return** Union(Lk)

**Apriori\_gen**(L, k)

result ← list()

**for all** p ⊆ L, q ⊆ L **where** p1 = q1, p2 = q2, ..., pk-2 = qk-2 and pk-1 < qk-1

c = p ∪ {qk-1}

**if** u ⊆ c **for all** u **in** L

result.add(c)

**return** result

The parallel execution pseudocode is given below:

* Pass 1
  + Generate the candidate item-sets in C1
  + Save the frequent item-sets in L1
* Pass k
  + Generate the candidate item-sets in Ck from the frequent itemsets in Lk-1
    - Join Lk-1 p with Lk-1 q, as follows:
      * insert into Ck
      * select p.item1, p.item2, . . . , p.itemk-1, q.itemk-1 from Lk-1 p, Lk-1 q where p.item1 = q.item1, . . . p.itemk-2 = q.itemk-2, p.itemk-1 < q.itemk-1
    - Generate all (k-1) - subsets from the candidate itemsets in Ck
    - Prune all candidate item-sets from Ck where some (k-1) - subset of the candidate itemset is not in the frequent itemset Lk-1
  + Distribute among p threads
  + Scan the transaction database to determine the support for each candidate itemset in Ck
  + Save the frequent item-sets in Lk

**METHOD:**

**Sequential execution:**

Demo.java

import java.util.ArrayList;

import java.util.Arrays;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

public class Demo {

    public static void main(String[] args) {

        AprioriFrequentItemsetGenerator<String> generator =

                new AprioriFrequentItemsetGenerator<>();

        List<Set<String>> itemsetList = new ArrayList<>();

        itemsetList.add(new HashSet<>(Arrays.asList("a", "b")));

        itemsetList.add(new HashSet<>(Arrays.asList("b", "c", "d")));

        itemsetList.add(new HashSet<>(Arrays.asList("a", "c", "d", "e")));

        itemsetList.add(new HashSet<>(Arrays.asList("a", "d", "e")));

        itemsetList.add(new HashSet<>(Arrays.asList("a", "b", "c")));

        itemsetList.add(new HashSet<>(Arrays.asList("a", "b", "c", "d")));

        itemsetList.add(new HashSet<>(Arrays.asList("a")));

        itemsetList.add(new HashSet<>(Arrays.asList("a", "b", "c")));

        itemsetList.add(new HashSet<>(Arrays.asList("a", "b", "d")));

        itemsetList.add(new HashSet<>(Arrays.asList("b", "c", "e")));

        FrequentItemsetData<String> data = generator.generate(itemsetList, 0.2);

        int i = 1;

        for (Set<String> itemset : data.getFrequentItemsetList()) {

            System.out.printf("%2d: %9s, support: %1.1f\n",

                              i++,

                              itemset,

                              data.getSupport(itemset));

        }

    }

}

AprioriFrequentItemsetGenerator.java

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.Objects;

import java.util.Set;

public class AprioriFrequentItemsetGenerator<I> {

    public FrequentItemsetData<I> generate(List<Set<I>> transactionList,

                                           double minimumSupport) {

        Objects.requireNonNull(transactionList, "The itemset list is empty.");

        checkSupport(minimumSupport);

        if (transactionList.isEmpty()) {

            return null;

        }

        Map<Set<I>, Integer> supportCountMap = new HashMap<>();

        // Get the list of 1-itemsets that are frequent.

        List<Set<I>> frequentItemList = findFrequentItems(transactionList,

                                                          supportCountMap,

                                                          minimumSupport);

        // Maps each 'k' to the list of frequent k-itemsets.

        Map<Integer, List<Set<I>>> map = new HashMap<>();

        map.put(1, frequentItemList);

        // 'k' denotes the cardinality of itemsets processed at each iteration

        // of the following loop.

        int k = 1;

        do {

            ++k;

            // First generate the candidates.

            List<Set<I>> candidateList =

                    generateCandidates(map.get(k - 1));

            for (Set<I> transaction : transactionList) {

                List<Set<I>> candidateList2 = subset(candidateList,

                                                     transaction);

                for (Set<I> itemset : candidateList2) {

                    supportCountMap.put(itemset,

                                        supportCountMap.getOrDefault(itemset,

                                                                     0) + 1);

                }

            }

            map.put(k, getNextItemsets(candidateList,

                                       supportCountMap,

                                       minimumSupport,

                                       transactionList.size()));

        } while (!map.get(k).isEmpty());

        return new FrequentItemsetData<>(extractFrequentItemsets(map),

                                         supportCountMap,

                                         minimumSupport,

                                         transactionList.size());

    }

    private List<Set<I>>

        extractFrequentItemsets(Map<Integer, List<Set<I>>> map) {

        List<Set<I>> ret = new ArrayList<>();

        for (List<Set<I>> itemsetList : map.values()) {

            ret.addAll(itemsetList);

        }

        return ret;

    }

    private List<Set<I>> getNextItemsets(List<Set<I>> candidateList,

                                         Map<Set<I>, Integer> supportCountMap,

                                         double minimumSupport,

                                         int transactions) {

        List<Set<I>> ret = new ArrayList<>(candidateList.size());

        for (Set<I> itemset : candidateList) {

            if (supportCountMap.containsKey(itemset)) {

                int supportCount = supportCountMap.get(itemset);

                double support = 1.0 \* supportCount / transactions;

                if (support >= minimumSupport) {

                    ret.add(itemset);

                }

            }

        }

        return ret;

    }

    private List<Set<I>> subset(List<Set<I>> candidateList,

                                Set<I> transaction) {

        List<Set<I>> ret = new ArrayList<>(candidateList.size());

        for (Set<I> candidate : candidateList) {

            if (transaction.containsAll(candidate)) {

                ret.add(candidate);

            }

        }

        return ret;

    }

    private List<Set<I>> generateCandidates(List<Set<I>> itemsetList) {

        List<List<I>> list = new ArrayList<>(itemsetList.size());

        for (Set<I> itemset : itemsetList) {

            List<I> l = new ArrayList<>(itemset);

            Collections.<I>sort(l, ITEM\_COMPARATOR);

            list.add(l);

        }

        int listSize = list.size();

        List<Set<I>> ret = new ArrayList<>(listSize);

        for (int i = 0; i < listSize; ++i) {

            for (int j = i + 1; j < listSize; ++j) {

                Set<I> candidate = tryMergeItemsets(list.get(i), list.get(j));

                if (candidate != null) {

                    ret.add(candidate);

                }

            }

        }

        return ret;

    }

    private Set<I> tryMergeItemsets(List<I> itemset1, List<I> itemset2) {

        int length = itemset1.size();

        for (int i = 0; i < length - 1; ++i) {

            if (!itemset1.get(i).equals(itemset2.get(i))) {

                return null;

            }

        }

        if (itemset1.get(length - 1).equals(itemset2.get(length - 1))) {

            return null;

        }

        Set<I> ret = new HashSet<>(length + 1);

        for (int i = 0; i < length - 1; ++i) {

            ret.add(itemset1.get(i));

        }

        ret.add(itemset1.get(length - 1));

        ret.add(itemset2.get(length - 1));

        return ret;

    }

    private static final Comparator ITEM\_COMPARATOR = new Comparator() {

        @Override

        public int compare(Object o1, Object o2) {

            return ((Comparable) o1).compareTo(o2);

        }

    };

    private List<Set<I>> findFrequentItems(List<Set<I>> itemsetList,

                                           Map<Set<I>, Integer> supportCountMap,

                                           double minimumSupport) {

        Map<I, Integer> map = new HashMap<>();

        // Count the support counts of each item.

        for (Set<I> itemset : itemsetList) {

            for (I item : itemset) {

                Set<I> tmp = new HashSet<>(1);

                tmp.add(item);

                if (supportCountMap.containsKey(tmp)) {

                    supportCountMap.put(tmp, supportCountMap.get(tmp) + 1);

                } else {

                    supportCountMap.put(tmp, 1);

                }

                map.put(item, map.getOrDefault(item, 0) + 1);

            }

        }

        List<Set<I>> frequentItemsetList = new ArrayList<>();

        for (Map.Entry<I, Integer> entry : map.entrySet()) {

            if (1.0 \* entry.getValue() / map.size() >= minimumSupport) {

                Set<I> itemset = new HashSet<>(1);

                itemset.add(entry.getKey());

                frequentItemsetList.add(itemset);

            }

        }

        return frequentItemsetList;

    }

    private void checkSupport(double support) {

        if (Double.isNaN(support)) {

            throw new IllegalArgumentException("The input support is NaN.");

        }

        if (support > 1.0) {

            throw new IllegalArgumentException(

                    "The input support is too large: " + support + ", " +

                    "should be at most 1.0");

        }

        if (support < 0.0) {

            throw new IllegalArgumentException(

                    "The input support is too small: " + support + ", " +

                    "should be at least 0.0");

        }

    }

}

FrequentItemsetData.java

import java.util.List;

import java.util.Map;

import java.util.Set;

public class FrequentItemsetData<I> {

    private final List<Set<I>> frequentItemsetList;

    private final Map<Set<I>, Integer> supportCountMap;

    private final double minimumSupport;

    private final int numberOfTransactions;

    FrequentItemsetData(List<Set<I>> frequentItemsetList,

                        Map<Set<I>, Integer> supportCountMap,

                        double minimumSupport,

                        int transactionNumber) {

        this.frequentItemsetList = frequentItemsetList;

        this.supportCountMap = supportCountMap;

        this.minimumSupport = minimumSupport;

        this.numberOfTransactions = transactionNumber;

    }

    public List<Set<I>> getFrequentItemsetList() {

        return frequentItemsetList;

    }

    public Map<Set<I>, Integer> getSupportCountMap() {

        return supportCountMap;

    }

    public double getMinimumSupport() {

        return minimumSupport;

    }

    public int getTransactionNumber() {

        return numberOfTransactions;

    }

    public double getSupport(Set<I> itemset) {

        return 1.0 \* supportCountMap.get(itemset) / numberOfTransactions;

    }

}

Parallel execution:

Apriorimp.cpp

#include <bits/stdc++.h>

#include <omp.h>

#include <stdio.h>

using namespace std;

//macros

#define structure map<vector<int>, int>

#define FOR\_MAP(ii,T) for(structure::iterator (ii)=(T).begin();(ii)!=(T).end();(ii)++)

#define FOR\_next\_MAP(jj,ii,T) for(structure::iterator (jj)=(ii);(jj)!=(T).end();(jj)++)

#define VI vector<int>

int MIN\_SUP;

structure C;

structure L;

void C1(string);

void L1();

void generate\_C();

void generate\_L();

void output(structure );

void scan\_D(int,int,string);

void prune();

bool check\_compatibility(VI ,VI );

void set\_count(VI , structure\*);

int main(int argc, char const \*argv[])

{

    int l ;

    string file\_name;

    cout<<"Enter the support count: "<<endl;

    cin>>MIN\_SUP;

    cout<<"Enter transactions: "<<endl;

    cin>>l;

    cout<<"Enter the input\_file name"<<endl;

    cin>>file\_name;

    int threads;

    cout<<"Enter the number of threads: "<<endl;

    cin>>threads;

    double start,end,time;

    start = omp\_get\_wtime();

    C.clear();

    L.clear();

    bool mv=true;

    int index=2;

    while(true)

    {

        if (mv)

        {

            C1(file\_name);

            cout<<"Initial Candidates\n";

            output(C);

            L1();

            cout<<"Candidates with sufficient Frequency\n";

            output(L);

            mv=!mv;

        }

        else

        {

            generate\_C();

            if(C.size()==0)

                break;

            prune();

            if (C.size()==0)

                break;

            scan\_D(l,threads,file\_name);

            generate\_L();

            if (L.size()==0)

                break;

            cout<<"\nFrequency List "<<index<<"\n";

            output(L);

            index++;

        }

    }

    end = omp\_get\_wtime();

    time = end - start;

    cout<<"the time required is : "<<time<<"\n";

    cout<<"no of iterations : "<<index<<"\n";

    return 0;

}

void C1(string file\_name)

{

    ifstream fin;

    fin.open(file\_name.c\_str());

    if(!fin)

        {

            cout<<"Input file opening error\n";

            exit(0);

        }

    int n;

    VI v;

    while(fin>>n)

    {

        v.clear();

        if (n==-1)

        {

            continue;

        }

        v.push\_back(n);

        if(C.count(v)>0)

            C[v]++;

        else

            C[v]=1;

    }

    fin.close();

}

void output(structure T)

{

    cout<<"\n";

    VI v;

    FOR\_MAP(ii,T)

    {

        v.clear();

        v=ii->first;

        for (int i = 0; i < v.size(); ++i)

        {

            cout<<v[i]<<" ";

        }

        cout<<" ---(frequency)----->> "<<ii->second;

        cout<<"\n";

    }

}

void L1()

{

    FOR\_MAP(ii,C)

    {

        if (ii->second >= MIN\_SUP)

        {

            L[ii->first]=ii->second;

        }

    }

}

/\* Generating all the cadidates of size k from frequency list of size k-1 \*/

void generate\_C()

{

    C.clear();

    FOR\_MAP(ii,L)

    {

        FOR\_next\_MAP(jj,ii,L)

        {

            if(jj==ii)

                continue;

            VI a,b;

            a.clear();

            b.clear();

            a=ii->first;

            b=jj->first;

            if(check\_compatibility(a,b))

            {

                a.push\_back(b.back());

                sort(a.begin(), a.end());

                C[a]=0;

            }

        }

    }

}

/\* Checking if any two frequency item sets are same or not \*/

bool check\_compatibility(VI a,VI b)

{

    bool compatible=true;

    for (int i = 0; i < a.size()-1; ++i)

    {

        if (a[i]!=b[i])

        {

            compatible=false;

            break;

        }

    }

    return compatible;

}

/\* Removing all the candidates of size k whose subsets of size k-1 are not in the frequency list of size k-1 \*/

void prune()

{

    VI a,b;

    FOR\_MAP(ii,C)

    {

        a.clear();

        b.clear();

        a=ii->first;

        for(int i = 0;i<a.size();i++)

        {

            b.clear();

            for (int j = 0; j < a.size(); ++j)

            {

                if(j==i)

                    continue;

                b.push\_back(a[j]);

            }

            if(L.find(b)==L.end())

                {

                    ii->second=-1;

                    break;

                }

        }

    }

    structure temp;

    temp.clear();

    FOR\_MAP(ii,C)

    {

        if (ii->second != -1)

        {

            temp[ii->first]=ii->second;

        }

    }

    C.clear();

    C=temp;

    temp.clear();

}

/\* For going to a particular line in the file \*/

FILE \* GotoLine(FILE\* file, unsigned int num){

    int count = 0;

    char buf[100];

    do

    {

        if(count == num)

            break;

        fgets(buf,100,file);

        count++;

    }while(1);

    return file;

}

/\* Scanning the database and calling set\_count for each transaction \*/

void scan\_D(int l,int t,string file\_name)

{

    int p = l/t;

    FILE \*f[t];

    int i;

    for(i=0;i<t;i++){

    f[i] = fopen(file\_name.c\_str(), "r");

        if(f[i] == NULL)

                {

                        cout<<"Input file opening error\n";

                        exit(0);

                }

        f[i] = GotoLine(f[i],i\*p);

    }

    int n;

    VI a;

    int count = 0;

    int nthreads, tid, procs, maxt, inpar, dynamic, nested;

    structure temp;

    temp.clear();

    FOR\_MAP(ii,C)

    {

        temp[ii->first]=ii->second;

    }

    int max = omp\_get\_thread\_limit();

    omp\_set\_dynamic(max);

    omp\_set\_num\_threads(t);

    #pragma omp parallel firstprivate(temp) private(count,n,a)

    {

        if(!omp\_in\_parallel())

        {

            cout<<"not parallel"<<endl;

            exit(0);

            }

        #pragma omp master

        {

            procs = omp\_get\_num\_procs();

            nthreads = omp\_get\_num\_threads();

            maxt = omp\_get\_max\_threads();

            inpar = omp\_in\_parallel();

            dynamic = omp\_get\_dynamic();

            nested = omp\_get\_nested();

            printf("Number of processors = %d\n", procs);

            printf("Number of threads = %d\n", nthreads);

            printf("Max threads = %d\n", maxt);

            printf("In parallel? = %d\n", inpar);

            printf("Dynamic threads enabled? = %d\n", dynamic);

            printf("Nested parallelism supported? = %d\n", nested);

        }

    count=0;

    char buf[1000];

    int x;

    int tid = omp\_get\_thread\_num();

    int lev = omp\_get\_level();

        do

        {

        fgets(buf,100,f[tid]);

        string s = buf;

        istringstream iss(s);

        a.clear();

        while (iss)

            {

                string sub;

                iss >> sub;

            stringstream convert(sub);

                convert >> x;

                if(x==-1 && a.size()>0)

                {

                set\_count(a,&temp);

                count++;

                break;

                }

                a.push\_back(x);

         };

        }while(count<p);

    #pragma omp flush(a)

    #pragma omp barrier

    #pragma omp critical

    {

        FOR\_MAP(ii,temp)

        {

            C[ii->first]+=ii->second;

        }

    temp.clear();

    }

    }

for(i=0;i<t;i++)

    fclose(f[i]);

}

void set\_count(VI a,structure\* temp)

{

    FOR\_MAP(ii,(\*temp))

    {

        VI b;

        b.clear();

        b=ii->first;

        int x;

        int true\_count=0;

        int prev=-1;

        if (b.size()<=a.size())

        {

            for (int i = 0; i < b.size(); ++i)

            {

                for (int j = prev+1; j < a.size(); ++j)

                {

                    if(b[i]==a[j])

                    {

                        true\_count++;

                        prev = j;

                        break;

                    }

                }

            }

        }

        if (true\_count==b.size())

        {

            ii->second++;

        }

    }

}

/\* Finding the frequent item-sets of size k by removing all candidate item-sets having their count less than minimum support count \*/

void generate\_L()

{

    L.clear();

    FOR\_MAP(ii,C)

    {

        if(ii->second >= MIN\_SUP)

        {

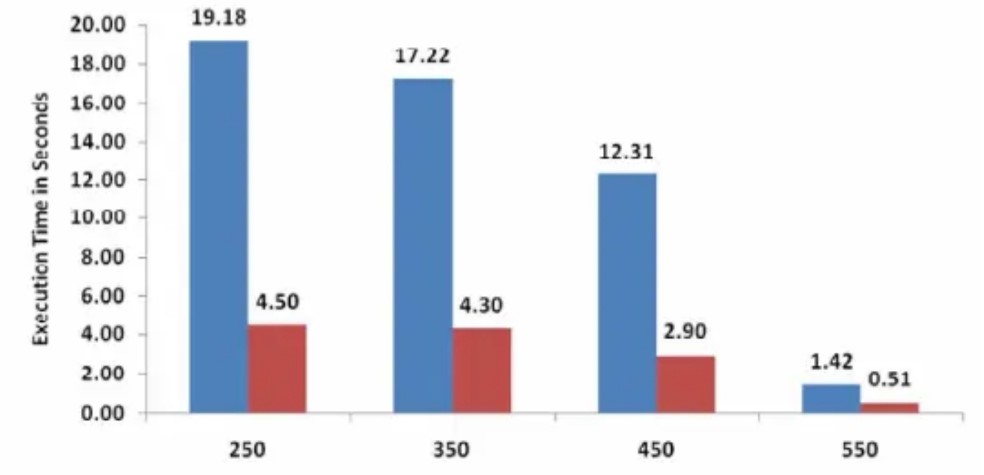
            L[ii->first]=ii->second;

        }

    }

}

Graph:



The above graph shows the time in seconds for the execution of sequential and parallel apriori algorithm. The blue shows the time taken for sequential execution and red bar shows time taken for apriori algorithm.

Limitations:

The Apriori Algorithm is sluggish. The biggest drawback is the amount of time it takes to hold a large number of candidate sets with frequent itemsets, low minimum support, or huge itemsets, making it inefficient for large datasets. For example, if there are 104 common 1-itemets, it must generate more than 107 2-length candidates, who will then be tested and accumulated. Furthermore, to recognise a regular pattern of size 100, i.e. v1, v2,... v100, it must generate 2100 candidate itemsets, which is costly and time consuming. As a result, it will check for several sets from candidate itemsets, as well as scanning the database multiple times to locate candidate itemsets.

REFERENCES:

1. Nilesh.S.Korde, Prof.Shailendra.W.Shende *‘Parallel Implementation of Apriori Algorithm’ .*
2. Mohammed Al-Maolegi, Bassam Arkok *‘An Improved Apriori Algorithm For Association Rules’*