DATA MINING AND DATA WAREHOUSING

NAME : SIBIN K SAMSON..

ROLL NUM :175214136.

Master of computer science.,

INDEX….

1.PROBLEM DESCRIPTION

2.ALGORITHM DESIGN

3.SOURCE CODE

4.TEST CASES

5.PROJECT SUMMARY

1.PROBLEM DESCRIPTION:

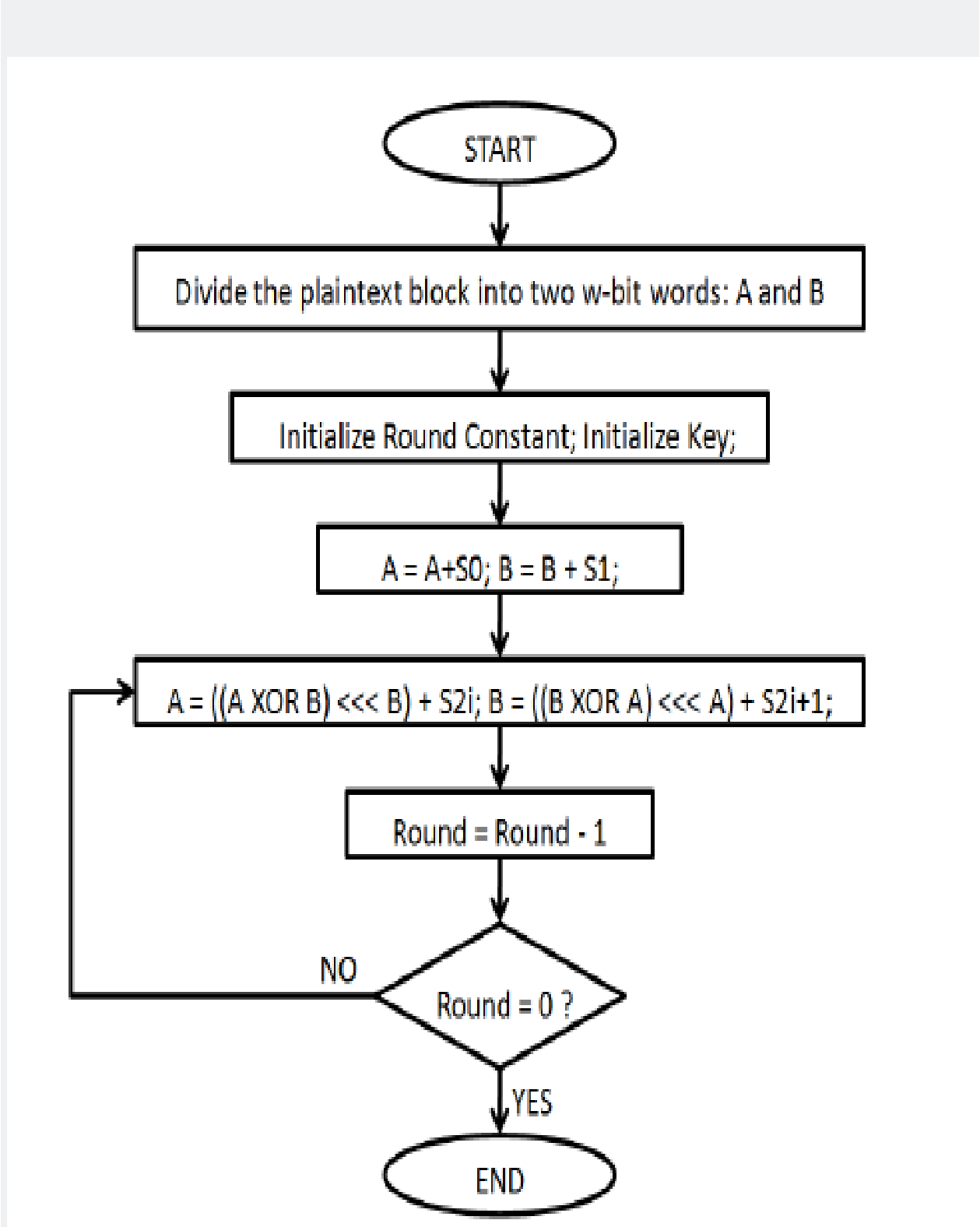
The Apriori algorithm is an influential algorithm for mining frequent item sets for Boolean rules. It uses a Bottom up approach, where frequent subsets are extended one item set at a time.

Apriori uses breadth-first search and hash tree structure to count candidate item set efficiently. In super market stores where the seller intend to observe the correlation of different product in the store.

For example, to observe which item are bought mostly together. This information would enable them to have better arrangement policies of their goods and would help a great deal with effective market products.

The FIM is a very data computation intensive application especially when the frequency threshold of the mining process is small.

2.ALGORITHM DESIGN:



3.SOURCE CODE

```
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;
      }

      // Maps each itemset to its support count. Support count is simply the
      // number of times an itemset appeares in the transaction list.
      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;
         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");
        }
    }
}
```

4.TEST CASES:

OUTPUT:

[0] (0.6666666666666666 6)

[1] (0.7777777777777778 7)

[2] (0.6666666666666666 6)

[3] (0.2222222222222222 2)

[4] (0.2222222222222222 2)

[0, 4] (0.2222222222222222 2)

[0, 1] (0.4444444444444444 4)

[0, 2] (0.4444444444444444 4)

[1, 2] (0.4444444444444444 4)

[1, 3] (0.2222222222222222 2)

[1, 4] (0.2222222222222222 2)

[0, 1, 4] (0.2222222222222222 2)

[0, 1, 2] (0.2222222222222222 2)

5.PROJECT SUMMARY:

For this Apriori algorithm, we developed java implementation. Here we generated Strong association rules from the frequent Item sets.