Improving the performance of Apriori Algorithm

This paper presents an approach, which is a combination to two techniques, to improve the performance of the Apriori Algorithm. The first technique helps in reducing the run-time of the algorithm by calculating support from in-memory representation of the database, thereby resulting in saving the I/O cost to scan the database for finding support of candidate itemsets. The second techniques helps in reducing the number of candidate itemsets generated by pruning the frequent itemsets before generation of next level candidate itemsets.

**The CRS representation of the database:**

The original Apriori Algorithm requires multiple scan of the database to calculate the support of candidate itemsets. The I/O operations involved in scanning the database has impact on the run-time of the algorithm. To overcome this problem, the entire database can be stored in memory. However, we need some efficient representation of database. One approach, which is commonly used, is used is to create a Boolean Matrix representation of the database. The support of candidate itemsets can be directly calculated by using the Boolean Matrix. The Boolean Matrix approach works well if the no. of different items and no. of transactions is less. However, as the no. of items and the no. of transactions increases, the memory requirements for storing the Boolean matrix increases drastically and if the database is sparse, which is usually the case with market basket database, the Boolean matrix representation turns out to be inefficient in terms of memory requirement. The Compressed Row Storage (CRS) format for representing sparse matrix helps in reducing the memory requirement.

For e.g. one database on which test were run contains 16500 different items and 88162 transactions. The memory requirement for storing the Boolean matrix would be:

* (16470\*88162) / (8\*1024\*1024) = 173 MB.

The memory requirement for storing the CRS representation would be (assuming the average size of transaction to be 10):

* For row vector (size: no of transactions +1): [Row vector is stored as vector of integers]

(No of transactions + 1)\*(size of integer)/ (1024\*1024)

88163\*4/ (1024\*1024) = 0.34 MB

* For Column Vector (size : no of items present): [Column vector is stored as vector of shorts]

(Avg. length of transaction)\*(no of transactions)\*(size of short) / (1024\*1024)

(10\*88162\*2) / (1024\*1024) = 1.68 MB

Total: 0.34 + 1.68 = 2.02 MB.

Thus resulting in ((173 – 1.68) / 173) = 99 % reduction in memory requirement in this case.

The CRS representation uses two vectors to represent the transactional database. The following case illustrates the CRS representation of an example database:

E.g. consider the following transactional database:

|  |  |
| --- | --- |
| TID | Items |
| 1 | A, B, E |
| 2 | B, C, D |
| 3 | C, E |
| 4 | A, C, D |
| 5 | A, B, C, D |
| 6 | A, E |
| 7 | A, B |
| 8 | A, B, C |
| 9 | A, C, D |
| 10 | B |

The Boolean Matrix representation of the database:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Transactions/Items | A | B | C | D | E |
| T1 | 1 | 1 | 0 | 0 | 1 |
| T2 | 0 | 1 | 1 | 1 | 0 |
| T3 | 0 | 0 | 1 | 0 | 1 |
| T4 | 1 | 0 | 1 | 1 | 0 |
| T5 | 1 | 1 | 1 | 1 | 0 |
| T6 | 1 | 0 | 0 | 0 | 1 |
| T7 | 1 | 1 | 0 | 0 | 0 |
| T8 | 1 | 1 | 1 | 0 | 0 |
| T9 | 1 | 0 | 1 | 1 | 0 |
| T10 | 0 | 1 | 0 | 0 | 0 |

The CRS representation of the database:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row Vector | 0 | 3 | 6 | 8 | 11 | 15 | 17 | 19 | 22 | 25 | 26 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Column Vector | 0 | 3 | 4 | 5 | 6 | 7 | 8 | 0 | 1 | 4 | 6 | 7 | 9 | 1 | 2 | 3 | 4 | 7 | 8 | 1 | 3 | 4 | 8 | 0 | 2 | 5 |

The CRS representation of the database is created during the initial pass of the database, when C1 is created. The support for each subsequent candidate itemsets can be calculated from the above representation by using column vector in the index range provided by row vector.

**Frequent Itemsets pruning:**

Before generating the next level candidate itemsets (Ck+1) from current level frequent itemsets (Fk), the current level frequent itemsets can be pruned further based on the following logic, which is based on the downward closure property.

* Ck+1 will contain a k+1-itemsets *I* only if all the subsets of length k of*I* belong to frequent itemsets Fk.
* b) There are k+1Ck = k+1subsets of length k of a k+1-itemset, but a transaction item *Ii* , contained in the k+1-itemset, will appear only in k of these subsets. Hence, the count of the item *Ii* , calculated by considering all the k subsets, will be k.
* The k+1 subsets of k+1-itemset in Fk+1 will be found Fk. Hence, an item contained in one of k+1 subsets will appear in at least k itemsets in Fk. Therefore, we derive a condition that if an item count in Fk is less than k, then itemsets containing that item cannot generate Fk+1 itemsets and can be pruned.

Count of the all the items appearing in itemsets in Fkis calculated and if an item *Ii*exists for which the count is less than k then all the itemsets in Fk containing *Ii* can be ignored during generation of Ck+1.

For e.g. consider the database presented above:

The frequent 2-Itemsets F2:

|  |  |
| --- | --- |
| A,B | 4 |
| A,C | 4 |
| A,D | 3 |
| A,E | 2 |
| B,C | 3 |
| B,D | 2 |
| C,D | 4 |

The count of all the items in F2:

|  |  |
| --- | --- |
| A | 4 |
| B | 3 |
| C | 3 |
| D | 3 |
| E | 1 |

Since the count of E is less than 2, itemsets containing E in F2({A, E})can be purged.

This technique results in reduced number of candidate itemsets (Ck+1) and speeds up the process.

**Results:**

For comparing the performance of the modified Apriori Algorithm with the original Apriori Algorithm three market basket database were used. The tests were performed on a machine having Intel I7 processor and 8GB of ram.

* The “T40I10D100K” dataset contains 100,000 transactions on 1000 different items
* The “T10I4D100K” dataset contains 100,000 transactions, containing 1000 different items.
* The “Retail” database contains 88,162 transactions, containing 16470 different items.