Chapter 4

Frequent Itemsets Using Partitioning Approach

Most of the algorithms to find frequent itemsets need multiple passes over the databases. This means that the disk-resident database has to be read several times. So, these algorithms spend a lot of time to perform disk I/O, resulting in maximum burden on the I/O subsystem. Moreover, running these algorithms on OLTP systems, where thousands of transactions are made per second, will increase the query response time to a great extent. If the algorithms are run on network, it may cause network congestion also. One feasible solution is to partition the database and apply the algorithms in the partitions individually. Another advantage of partitioning the database is that it helps parallelize the algorithms.

This chapter presents a useful frequent itemsets finding algorithm using vertical partitioning approach. It also includes an experimental analysis of the horizontal partitioning based frequent itemset finding approach. Then, it has made a comparison between both the approaches: horizontal partitioning and vertical partitioning.

4.1 Partition Algorithm

Partition algorithm [SON95] overcomes difficulties mentioned above to a great
CHAPTER 4. FREQUENT ITEMSETS USING PARTITIONING...

extent for the following reasons.

1. It reads the database only twice to generate the itemsets.

2. CPU overhead is much lower than the other algorithms.

3. The algorithm can generate results with no false negative.

4. The algorithm can be parallelized with minimal communication and synchronization among the processing nodes.

The algorithm uses the concept of partition, local support, local large itemsets, global support and global large itemsets. A partition is defined as any subset of transactions contained in the database \( D \) and partitions are non-overlapping. Local support of an itemset \( X \) in a partition is defined as the fraction of transactions in the partition, which contains the itemset. If the local support of an itemset \( X \) is at least the user-defined minimum support, the itemset is called local large itemset. Global support, global large itemsets, etc. also are defined in the same way, but in the context of global database.

Partition algorithm works in two phases. In phase I, it logically divides the database into some non-overlapping partitions. Next, each partition is considered individually and the large itemsets for each partition are generated. At the end of phase I, large itemsets found in each partition are merged to generate global candidate sets. In phase II, the actual support for these itemsets are computed by scanning the whole database and large itemsets are found.

4.1.1 The Algorithm

The algorithm is reported in Algorithm 4.1 on page 77. The algorithm assumes that transactions are in the form \((TID, i_j, i_k, \ldots)\) and items are kept in lexicographic order. \( TID \), called transaction id, is an unique number associated with each transaction. The algorithm first partitions the database into some non-overlapping logical partitions. Then it reads one partition at a time and finds the local large itemsets of the partition using one function \( \text{Gen\_large\_itemsets} \) (Function 4.1 on page 78). These local large itemsets are potential global large
itemsets. At the end of first phase, these local large itemsets of all the partitions are merged together to form global candidate sets. In the second phase, the algorithm scans the database once to find the actual support of global candidate sets. Global candidate sets whose support is greater than the minimum support becomes global large itemsets. The main strength of the algorithm can be summarized as follows.

- A global large itemset must be locally large in at least one partition.
- It finds all the global large itemsets because global candidate set is the union of all local large itemsets.

4.1.2 Generation of Local Large Itemsets and Global Large Itemsets

Local large itemsets of a partition $p$ are generated by the function $Gen\_large\_itemsets$ (Function 4.1 on page 78). The function takes one partition $p$ and returns large itemsets in the partitions. The function works in the same way as $Apriori$ works. The function maintains $tidlist$ of the items. $tidlist$ of an itemset $X$ is nothing, but an array of transaction ids (TID) of the transactions in the database containing $X$. Support count of an itemset is found by intersection of the $tidlists$ of the items in the itemset. One important step in the function is the pruning step (line 8). A candidate $c$ is pruned away if one of its subsets is not large because any superset of it also cannot be large.

Union of all local large itemsets form the global candidate sets. This global candidate set is the superset of the set of all possible global large itemsets because any global large itemset must be locally large in at least one partition. Then, the algorithm reads one partition at a time and calls the procedure $Gen\_final\_count$ (Procedure 4.1 on page 79) to find support count of all the global candidate sets in the partition. Support count of all the partitions are added to find global support count of the itemsets. Itemsets with support count greater than minimum support are the global large itemsets.
Input: \( D, \text{minsup}, \text{np} \)
Output: All large itemsets \( L^{G} \).

1. \( P = \text{Partition\_database}(D); \)
   //Phase I....................

2. For \( i=1 \) to \( \text{np} \) do begin

3. \( \text{Read\_in\_partition}(p_{i} \in P); \)

4. \( L^{i} = \text{Gen\_large\_itemsets}(p_{i}); \)

5. End
   // Merge phase..............

6. For \( (i=1; L_{i} \neq \phi ; j=1,2,\ldots,\text{np}; i++) \) do begin

7. \( C_{i}^{G} = \bigcup_{j=1,2,\ldots,\text{np}} L_{i}^{j}; \)

8. End

9. \( C^{G} = \bigcup C_{i}^{G} \)
   ; //Phase II....................

10. For \( i=1 \) to \( \text{np} \) do begin

11. \( \text{Read\_in\_partition}(p_{i} \in P); \)

12. \( \text{Gen\_final\_count}(C^{G}, p_{i}); \)

13. End

14. \( L^{G} = \{ c \in C^{G} | c.\text{count} \geq \text{minsup} \}; \)

15. Answer= \( L^{G}; \)

Algorithm 4.1: Partition
CHAPTER 4. FREQUENT ITEMSETS USING PARTITIONING...

Function Gen_large_itemsets(p : a partition)

1. \( L_1^p \) = large 1-itemsets along with their tidlists;
2. For \( (k = 2; L_{k-1}^p \neq \phi; k++) \)
3. For all itemsets \( l_1 \in L_{k-1}^p \)
4. \( L_k^p = \phi; \)
5. For all itemsets \( l_2 \in L_{k-1}^p \)
6. If \( l_1[i] = l_2[i](i=1,2,3...k-2) \) and \( l_1[k-1] < l_2[k-1] \) then
7. \( c = l_1[1].l_1[2]...l_1[k-1].l_2[k-1]; \)
8. If \( c \) cannot be pruned then
9. \( c.tidlist = l_1.tidlist \cap l_2.tidlist; \)
10. If \( |c.tidlist| \geq \text{minsup} \) then
11. \( L_k^p = L_k^p \cup \{c\}; \)
12. Endif
13. Endif
14. Endif
15. Endfor
16. Endfor
17. Endfor
18. Return \( \cup_k L_k^p \);

Function 4.1: Gen_large_itemsets
CHAPTER 4. FREQUENT ITEMSETS USING PARTITIONING...

Procedure Gen_final_count($C^G$: Set of all global candidate sets, $p$: database partition)

1. For $(k=1; \ C_k^G \neq \phi; \ k++)$
2. Forall $k$-itemset $c \in C_k^G$
3. $templist = c[1].tidlist \cup c[2].tidlist \cup c[3].tidlist \ldots \cup c[k].tidlist$;
4. $c.count = c.count + |templist|$;
5. Endfor
6. Endfor

Procedure 4.1: Gen_final_count

4.1.3 An Example

Let us consider a sample database with five items ($A, B, Q, R, S$) and ten transactions as given in Table 4.1 on the next page. Two logical partitions $p_1$ and $p_2$ are created with five transactions in each partition. Partition $p_1$ contains first five transactions and partition $p_2$ contains next five transactions. Minimum support is assumed to be 40%. Now, partition $p_1$ is read and the function Gen_large_itemset is called to find the local large itemsets in $p_1$. Local large itemsets for $p_1$ is $L_1 = \{A, B, Q, S, AB, AQ, AS, BQ, BS, ABQ, ABS\}$. Similarly, local large itemsets for $p_2$ are obtained as $L_2 = \{A, B, Q, S, AB, AQ, AS, BQ, ABQ\}$. After merging, global candidate set is obtained as $C^G = \{A, B, Q, S, AB, AQ, AS, BQ, BS, ABQ, ABS\}$. Now, $p_1$ is read and support count of all the itemsets $c \in C^G$ in $p_1$ are calculated, which are 5, 5, 2, 4, 5, 2, 4, 2, 4 and 5 respectively. Similarly, $p_2$ is read and support count of all the itemsets $c \in C^G$ in $p_2$ are calculated, which are 4, 3, 3, 2, 2, 2, 1, 2 and 1 respectively. Global support count of the itemsets $c \in C^G$ are 9, 8, 5, 6, 8, 4, 6, 4, 5, 4 and 5 respectively. The itemsets $c \in C^G$ with support count greater than minimum support count (i.e. 4 transactions) constitute $L^G$, set of global large itemsets. So, $L^G = \{A, B, Q, S, AB, AQ, AS, BQ, BS, ABQ, ABS\}$. 
4.1.4 Experimental Results

Experiments were carried out to show the execution time of *Partition* algorithm. All the experiments were carried out using *Pentium IV* machine with 256 MB RAM and 40GB disk space. Minimum support was taken as 1% for all the experiments.

**Data set**: Two synthetic data sets and one real data set were used for the experiments. Synthetic data were generated based on the method given in [AMS+96] and the real database (*Connect-4*) was downloaded from UCI machine learning repository (http://www.ics.uci.edu). The Table 4.2 on the following page shows the parameters used in the synthetic data generation, where $|D|$, $|T|$ and $|ML|$ denote the number of transactions, the average size of transactions and the mean size of a potentially large itemset. For synthetic datasets, number of items, potential large itemsets and number of records were taken as 500, 500 and 100K respectively. *Connect-4* database contains around 65K instances with 43 attributes. Each attribute can have one of three values. Here also, each distinct value of the attributes was considered as one item, resulting in total 129 items.

Experiments were carried out to find execution time for different number of partitions. In case of synthetic data sets, number of partitions was increased from 1 to 15 and in case of *Connect 4* number of partitions was increased from

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Q</th>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: A Sample Database - III
1 to 10. Experimental results are given in Figures 4.1 and 4.2 on the following page.

| Data Set  | |T| | |ML| | |D| |
|-----------|---|---|---|---|
| T20I4100K | 20 | 4  | 100K|
| T20I6100K | 20 | 6  | 100K|

Table 4.2: Parameters for Synthetic Databases - III

Figure 4.1: Experimental Results of Horizontal Partition - I
Figure 4.2: Experimental Results of *Horizontal Partition* - II
4.1.5 Discussion

The partition algorithm, which has been discussed above, is basically a horizontal partition algorithm because the database is partitioned horizontally. The algorithm is robust enough to find all the large itemsets in the whole database. It works much faster than Apriori, as has been demonstrated in [SON95] because it needs only two scans of the database. Proper choice of the number of partitions and the partition sizes are crucial for the success of the algorithm. Partition sizes are so chosen that each partition can be accommodated in the main memory and the partitions are read only once in each phase. However, the algorithm is not scalable with number of dimensions and number of partitions as has been shown by the experimental results.

4.2 Vertical Partition Algorithm

As mentioned above, one major factor which greatly affects the performance of the partition algorithm, is the size of the global candidate sets. As the size of the global candidate sets increases, the execution time also increases. Although there are many factors, which affect the size of the global candidate set, one of the main factors is the dimensionality of the records/transactions. It has been observed that with the increase in dimensionality, the size of the global candidate set as well as the execution time also increases. Vertical Partition algorithm, which partitions the database vertically instead of horizontally, overcomes this problem to a great extent.

4.2.1 The Algorithm

Let us consider a customer transaction database $D$, which contains transactions of items. It is assumed that each transaction/record is of the form $<TID, 1, 0, 1, 1, 0...>$ and the items are in lexicographic order. Here, $TID$ is unique transaction identification number; 1 represents that the corresponding item is bought and 0 represents that the item is not bought in the transaction. It is also assumed that the $TIDs$ are kept in monotonically ascending order and the database is
kept in secondary storage. Considering the assumptions, the algorithm works as follows. The algorithm works in two phases. In the first phase, the database is partitioned vertically into some logical partitions and large itemsets for each partition are determined. In the second phase, large itemsets found in each of the partitions are combined to form the global candidate sets. Then, supports of the candidates are also calculated. It should be noted that local large itemsets of a partition are also global large itemsets. The support of the global candidate sets are calculated by intersecting the tidlists of the items of the itemsets. Obviously, the global candidate set is the superset of the actual large itemsets. So, the algorithm is sure to find all possible large itemsets in the database. The algorithm is presented in Algorithm 4.2 on the next page.

4.2.2 Generating Large Itemsets in a Partition

Large itemsets in a partition are generated using the function Gen_large_itemsets (Function 4.2 on page 86). The function takes a partition as input and returns the set of large itemsets in that partition. The function works in the same way as Apriori works. It finds the support count of an itemset by intersection of the tidlists of the items in the itemset.

4.2.3 Combining Local Large Itemsets

The algorithm uses the function Combine_local_large (Function 4.3 on page 87) to find the global candidate itemsets. The function takes two sets of large itemsets; concatenates a large itemset of one set with a large itemset of another set and returns the concatenated sets. Combining the large itemsets in all the partitions at a time will generate too many candidate sets. So, they are combined incrementally. At first, sets of large itemsets of first two partitions are combined to find candidate itemsets. Then, support count of the candidates are found by intersections of tidlists of the items. The itemsets with support count greater than minimum support becomes actual large itemsets. Then, the set of resulting large itemsets is combined with the set of large itemsets of the next partition and the process continues so on. As for example, let the large itemsets in three partitions be \{ AB, A, B \}; \{ Q, R \} and \{ S \}. At first, partitions 1 and 2 will
CHAPTER 4. FREQUENT ITEMSETS USING PARTITIONING... 85

Input: $D$, minsup, $np$
Output: All large itemsets $L^G$.

1. $P = \text{Vertical\_partition\_database}(D)$;

2. For $i=1$ to $np$ do begin

3. $\text{Read\_in\_partition}(p_i \in P)$. create the tidlists for each of the items;

4. $L^i = \text{Gen\_large\_itemsets}(p_i \in P)$;

5. End

6. $L^G = L^1$;

7. For $i = 2$ to $np$ do begin

8. $C^G = \text{Combine\_local\_large}(L^G, L^i)$;

9. For all candidate $c \in C^G$ $\text{gen\_count}(c)$;

10. $L^G = L^G \cup L^i \cup \{c \in C^G | c.\text{count} \geq \text{minsup}\}$;

11. End

12. Answer $= L^G$;

Algorithm 4.2: Vertical Partition
FUNCTION Gen_large_itemsets(p : a partition)

1. $L_1^p$ = large 1-itemsets from the tidlist;

2. For $(k = 2; L_{k-1}^p \neq \phi; k++)$

3. $L_k^p = \phi$;

4. For all itemsets $l_1 \in L_{k-1}^p$

5. For all itemsets $l_2 \in L_{k-1}^p$

6. If $l_1[i] = l_2[i](i=1,2,3...k-2)$ and $l_1[k-1] < l_2[k-1]$ then

7. $c = l_1[1].l_1[2]...l_1[k_1].l_2[k-1]$;

8. If $c$ cannot be pruned then

9. $c.tidlist = l_1.tidlist \cap l_2.tidlist$;

10. If $|c.tidlist| \geq \text{minsup}$ then

11. $L_k^p = L_k^p \cup \{c\}$;

12. Endif

13. Endif

14. Endif

15. Endfor

16. Endfor

17. Endfor

18. Return $\cup_k L_k^p$;

FUNCTION 4.2: Gen_large_itemsets
be combined to find the potential large itemsets { ABQ, ABR, AQ, AR, BQ, BR }. Suppose AQ and BQ are large among them. Then \{AQ, BQ\} will be combined with the partition 3 i.e. \{S\} to generate {AQS,BQS}.

**Function Combine_local_large**(\(X, Y\): Sets of large itemsets)

1. \(Z = \emptyset\);
2. For all \(a \in X\)
3. For all \(b \in Y\)
4. \(c = \text{concatenate } a \text{ and } b\);
5. \(Z = Z \cup c\);
6. Endfor
7. Endfor
8. Return \(Z\);

**Function 4.3: Combine_local_large**

### 4.2.4 Support Count

The supports of the global candidate sets are obtained by the procedure **Gen_count** (Procedure 4.2). The procedure takes one candidate itemset as input and calculates support count of the itemsets as output. The support count is calculated by intersecting the *tidlists* of the items.

**Procedure Gen_count**(\(c : \text{itemset}\))

1. \(k = |c|\);
2. \(\text{templist} = c[1].\text{tidlist} \cap c[2].\text{tidlist} \cap \ldots \cap c[k].\text{tidlist}\);
3. \(c.\text{count} = |\text{templist}|\)

**Procedure 4.2: Gen_count**
4.2.5 An Example

Let us try to understand the Vertical Partition algorithm with an example. Let us consider a Database $D$ with three partitions as given in Table 4.3 and the minimum support be 40% (i.e. $2$ transactions)

<table>
<thead>
<tr>
<th>TID</th>
<th>Partition I</th>
<th>Partition -II</th>
<th>Partition-III</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.3: A Sample Database - IV

The Frequent itemsets in the partition I: $L^1 = \{A, B, AB\}$
The Frequent itemsets in the partition II: $L^2 = \{H\}$
The Frequent itemset in the partition III: $L^3 = \{R, S\}$
Candidate sets generated by combining $L^1$ and $L^2$ are $C = \{AH, BH, ABH\}$.
The Frequent itemsets in $C$ are $L' = \phi$
$L'$ is recalculated as $L' = L' \cup L^1 \cup L^2 = \{A, B, AB, H\}$
Candidate sets generated by combining $L'$ and $L^3$ are $C = \{AR, AS, BR, BS, ABR, ABS, HR, HS\}$
The Frequent sets in $C$ are $L'' = \{AR, BR, ABR, HS\}$
Therefore $L^G = L'' \cup L' \cup L^3 = \{A, B, AB, H, R, S, AR, BR, ABR, HS\}$.

4.2.6 Discussion

The Vertical Partition algorithm finds all possible large itemsets and needs one scan of the database. Another advantage is that it generates less number of global candidate sets because of the fact that an itemset, which is large in a vertical partition, is also large in the whole database. So, it is not required
CHAPTER 4. FREQUENT ITEMSETS USING PARTITIONING...

to test the itemset again for whole database. However, the algorithm requires

to maintain the tidlist of the items. If they cannot be kept in memory, they
can be kept in secondary storage and accessed whenever required. As in the
Horizontal Partition algorithm, determining number of partitions poses a major
consideration. Generally, the size of partition should be chosen in such a way
that it fits in the main memory. However, other factors such as the type of
database, minimum support, etc. should also be taken care of accordingly.

4.2.7 Experimental Results

Experiments were carried out to show comparison of execution time between
Horizontal Partition (HP) algorithm and Vertical Partition (VP) algorithm for
different number of partitions and dimensions. Horizontal Partition algorithm
was implemented using the same approach as it was discussed in [SON95] and
Vertical Partition was implemented using the same approach as was discussed
above. All the experiments were carried out using Pentium IV machine with 256
MB RAM and 40GB disk space.

Data set: Two synthetic databases and one real database were used for the
experiments. Synthetic databases were generated based on the method given in
[AMS+96] and the real database (Connect-4) was downloaded from UCI machine
learning repository (http://www.ics.uci.edu). The Table 4.4 on the following page
shows the parameters used in the synthetic data generation, where $|D|$, $|T|$ and
$|ML|$ denote the number of transactions, the average size of transactions and the
mean size of a potentially large itemset respectively. For all datasets, number of
items, potential large itemsets and number of records were taken as 500, 500 and
100K respectively. Connect-4 database contains around 65K instances with 43
attributes. Each attribute can have one of three values. Here also, each distinct
value of the attributes has been considered as one item, resulting in total 129
items. For all the experiments minimum support was taken as 1%.

Experiments were carried out to compare execution times for different number
of items and different number of partitions. For synthetic data sets, numbers of
items were 100, 300 and 500; for Connect 4 numbers of items were 50, 100 and
129. In case of synthetic data sets, number of partitions was increased from 1
to 15 and in case of Connect 4 number of partitions was increased from 1 to 10. Experimental results are given in Figures 4.3 through 4.7.

| Data Set  | \( |T| \) | \( |ML| \) | \( |D| \) |
|-----------|---------|---------|---------|
| \( T20I4100K \) | 20      | 4       | 100K    |
| \( T20I6100K \) | 20      | 6       | 100K    |

Table 4.4: Parameters for Synthetic Databases - IV

Figure 4.3: Experimental Results of *Horizontal Partition* and *Vertical Partition* - I
Figure 4.4: Experimental Results of Horizontal Partition and Vertical Partition
Figure 4.5: Experimental Results of Horizontal Partition and Vertical Partition - III
Figure 4.6: Experimental Results of Horizontal Partition and Vertical Partition -- IV
Figure 4.7: Experimental Results of Horizontal Partition and Vertical Partition
4.3 \textit{FP-growth} and Vertical Partition

\textit{FP-growth} algorithm \cite{HPY00} finds frequent itemsets without candidate generation. The algorithm is based on a special data structure called FP-tree, which is basically a prefix tree of the transactions of the database such that each path represents a set of transactions that share the same prefix. The algorithm works as follows. The algorithm first scans the database once to find the frequent items in the database. Infrequent items are removed from the database and items in the transactions are rearranged in the descending order of the frequencies of items. Then, all the transactions containing the least frequent item are selected and the item is removed from the transactions, resulting a reduced (projected) database. This projected database is processed to find frequent itemsets. Obviously, the removed item will be a prefix of all the frequent itemsets. Then the item is removed from the database and the above process is repeated with the next least frequent item. It is to be noted that FP-tree contains all the information about the transactions and the frequent itemsets. So, to find any information about the transactions and frequent itemsets, just the tree is to be searched.

\textit{FP-growth} algorithm is one of the efficient algorithms to find frequent itemsets from large databases. However, it suffers from following two problems.

- The algorithm takes much time to construct the FP-tree, specially for higher dimensions.
- The performance of the algorithm degrades with the increase of minimum support \cite{HPY00}.

Vertical partitioning approach helps solve these problems to a great extent by using \textit{FP-growth} in each partition to find local frequent itemsets. The steps can be summarized as follows.

1. Partition the database into some vertical partitions;
2. For each partition do
3. Use \textit{FP-growth} to find frequent itemsets;
4. Generate global candidate sets from frequent itemsets;
5. Find support of the global candidate sets;
6. Return global frequent itemsets.

4.3.1 Experimental Results

Experiments were carried out to evaluate the performance of original \textit{FP-growth} and, that of \textit{FP-growth} with vertical partitions. The \textit{FP-growth} program was downloaded from http://fuzzy.cs.uni-magdeburg.de/ borgelt/software.html. Experiments were carried out on a Pentium IV PC with 256 MB RAM and two synthetic databases (T20.I4.D100K and T20.I6.D100K) were taken for the experiments. The synthetic data were generated using the technique given in [AMS+96] with the values of parameters as given in \textit{Table 4.5} and with dimensionality 500. Here, \(|T|\), \(|ML|\) and \(|D|\) represent the average size of a transaction, mean size of a potentially large itemset and database size (number of transactions) respectively. Each of the datasets contained 100K records. For \textit{FP-growth} with vertical partitions, each database was divided into five logical vertical partitions with 100 items in each partitions. Experimental results are given in the \textit{Figures 4.8} on the following page & 4.9 on the next page.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|}
\hline
Name      & \(|T|\) & \(|ML|\) & \(|D|\) \\
\hline
T20.I4.D100K & 20   & 4      & 100K     \\
T20.I6.D100K & 20   & 6      & 100K     \\
\hline
\end{tabular}
\caption{Parameters for Synthetic Databases - V}
\end{table}

4.4 Discussion

This chapter presents a novel association rule mining algorithm based on vertical partitioning. It also reports a comparative study of the proposed algorithm with the \textit{Partition} algorithm [SON95].

It has been observed from the experimental results that
CHAPTER 4. FREQUENT ITEMSETS USING PARTITIONING... 97

Figure 4.8: Experimental Results of $FP$-growth (Database: T20I4D100K)

Figure 4.9: Experimental Results of $FP$-growth (Database: T20I6D100K)
• In both the algorithms, the execution time increases with the increase in number of items. It can be seen from the Figures that Horizontal Partitions (HP) performs better when number of items is small. However, with the increase in number of items, the performance of the Vertical Partitions (VP) improves over Horizontal Partitions.

• In case of Horizontal Partitions, the execution time increases with the increase of number of partitions. Whereas, in case of Vertical Partitions, the execution time decreases with the increase of number of partitions and becomes almost constant at the end.