The main contribution we propose involves modifying the original FP-Growth algorithm in order to reduce the number of transactional database (TDB) scans required. Thus, creating a more efficient mining process. However, this does come at the expense of using more memory; a time-space trade-off.

The original FP-Growth algorithm consists of four main steps: completing an initial scan of the TDB in order to find the frequent 1-itemsets, sorting the frequent 1-itemsets according to a set of criteria/heuristic, completing a second scan of the TDB to construct an FP-Tree, and passing this tree off to the actual FP-Growth mining algorithm to find the frequent k-itemsets. Our proposal focusses on removing the second TDB scan, thus reducing the transactions scanned in half.

To achieve this, we focus on the TDB representation before passing it off to the FP-Growth algorithm for mining. This will consist of three crucial steps. First, we perform our one and only TDB scan, scanning each transaction and adding each item to a tree structure as a node represented as *<item, occurrence\_count>*. We will call this tree structure REP-Tree, as it will represent the entire TDB as an in-memory data structure. Each path of the REP-Tree will then represent one transaction from the TDB. To save space, we will re-use tree nodes that have the same prefix as the next transaction from the TDB, incrementing the *occurrence\_count* of said node.

Secondly, our algorithm will traverse the REP-Tree, and create a table consisting of *<item, occurrence\_count>* pairs. In this step, any item whose occurrence count does not meet the user specified minimum support threshold with be pruned from the table. The table will then be sorted according to a heuristic. The heuristic we will be using in our implementation and following example will be based on a frequency descending order. Resolving occurrence count ties based on alphabetical order.

Lastly, our algorithm will then convert our REP-Tree into an FP-Tree in preparation for mining. To execute this, we simply need to extract each path in the REP-Tree. Since each path represents a transaction from the TDB, we can remove items that are not present in our header table and add the resulting transaction to an FP-Tree according to the standard FP-Growth algorithm. Once the FP-Tree is constructed it can be passed to FP-Growth for mining. At this point the REP-Tree is no longer required and can be removed from memory.

To illustrate the above description, we will perform a step-by-step example using the data in Table 1. below.

Table 1. Example Transactional Database

|  |  |
| --- | --- |
| **Transaction ID** | **Transaction** |
| t1 | a, c, d |
| t2 | b, c, e |
| t3 | a, b, c, e |
| t4 | b, e |

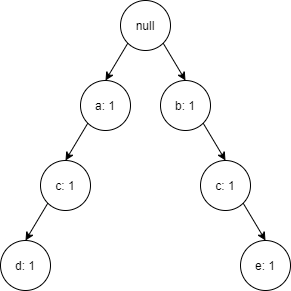
minsup = 2

**Step 1:** Scan the TDB and represent it as a tree.

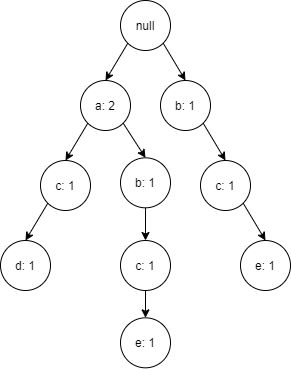
**Scan t1**



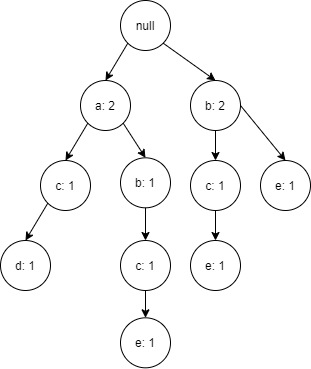
**Scan t2**

****

**Scan t3**

****

**Scan t4**

****

After this first step, the result is a REP-Tree. Following each path of the tree we can observe each transaction from the original TDB.

**Step 2:** Scan the REP-Tree, sum item occurrence count, and re-order based on our heuristic.

|  |  |
| --- | --- |
| **Item** | **Occurrence Count** |
| a | 2 |
| b | 3 |
| c | 3 |
| ~~d~~ | ~~1~~ |
| e | 3 |

**Re-order in frequency descending order**

|  |  |
| --- | --- |
| **Item** | **Occurrence Count** |
| b | 3 |
| c | 3 |
| e | 3 |
| a | 2 |

**Step 3:** For each path in the REP-Tree, sort in header table order, remove item that do not appear in the header table, and construct an FP-Tree.

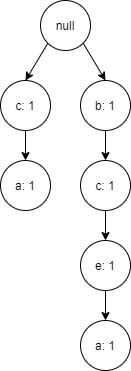
**Path 1 = a – c – d becomes c – a**

FP-Tree



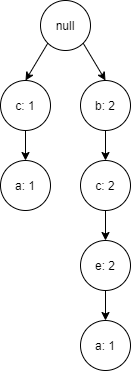
**Path 2 = a – b – c – e becomes b – c – e – a**

FP-Tree



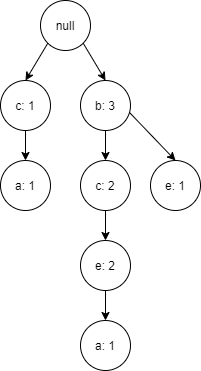
**Path 3 = b – c – e becomes b – c – e**

FP-Tree



**Path 4 = b – e becomes b – e**

FP-Tree



Upon completing this construction process, we can pass the resulting FP-Tree to FP-Growth for mining frequent k-itemsets. This requires only one scan of the transactional database at the cost of using more memory. However, in a practical scenario, if FP-Growth is the algorithm being used for mining, the TDB must fit into memory to be represented as an FP-Tree. If this is possible, representation as a REP-Tree should also be feasible.

**Pseudocode for main mining:**

OPEN database

for each transaction:

    ADD transaction to REP-Tree

PROVIDE REP-Tree to FP-Growth mining algorithm

**Pseudocode for adding a transaction to the REP-Tree:**

ASSIGN the Current Node as the Root Node

for each item in the transaction:

    if the item is in the roots children:

        INCREMENT the nodes count

        ASSIGN this child node as the Next Node

    else:

        CREATE a new node as <item, 1>

        ASSIGN the new node to be a child of the current node

        ASSIGN this new node as the Next Node

    ASSIGN the Current Node as the Next Node

**Pseudocode for getting support values of items in a REP-Tree:**

CREATE a dictionary of the form <item, count>

CREATE a queue

ADD the root node to the queue

while the queue is not empty:

    dequeue node from the queue

    if the node contains an item:

        ADD an entry to the dictionary with the nodes item

        and increment the count by its occurrence count

        ADD each child of the node to the queue

RETURN the generated dictionary

**Comparison of transactions scanned**

**Time comparison**

Comparing the original FP-Growth algorithm to our new proposed FP-Growth algorithm, we noticed some interesting performance differences. We compared these two algorithms using a dataset of 1, 2, 7, and 13 unique items on a database size of 10000, 50000, 100000, 500000, and 1000000. We also experimented with a minimum support threshold of 3, and a much larger minimum support threshold of 10%.

Table 2 and table 3below illustrate the time (in seconds) that it took to execute each algorithm with a minimum support threshold equal to 10% and 3, respectively. These tables are broken down into time it took to complete the first database scan, the time it took to complete the actual mining, and the total execution time for each algorithm.

The time for the first database scan in the original FP-Growth algorithm represents the time required to scan each transaction in the database and construct a simple list of transactions. With respect to our new FP-Growth algorithm, this time represents the time taken to scan each transaction in the database and construct our REP-Tree (a tree representing each transaction in the database). Furthermore, the second column of data represents the time taken to complete the second database scan, determine the frequent 1-itemsets, and complete the mining process to determine the remaining k-itemsets with respect to the original FP-Growth algorithm. This column represents the time taken to iterate over the REP-Tree, determine the frequent 1-itemsets, and complete the mining process to determine the remaining k-itemsets for our new proposed FP-Growth algorithm.

Looking at the results for each of these algorithms, we can see that the original FP-Growth algorithm outperforms our new FP-Growth algorithm with large minimum support thresholds (see table 2). This is largely due to the fact that the overhead to construct and iterate over the REP-Tree when a lot of items will inevitably be pruned due to not meeting the minimum support threshold requirement. It should be noted that our new FP-Growth algorithm performs all mining in a relatively constant time due to the tree traversal process. However, we see the opposite result when using a low minimum support threshold (see table 3). In this scenario, the original FP-Growth algorithm must now iterate over almost all the possible itemsets, reading them off disk to construct the FP-Tree. While FP-Growth still outperforms our new FP-Growth algorithm with on datasets with few unique items, our proposed algorithm significantly outperforms the original FP-Growth algorithm when we begin mining on large datasets with many unique items.

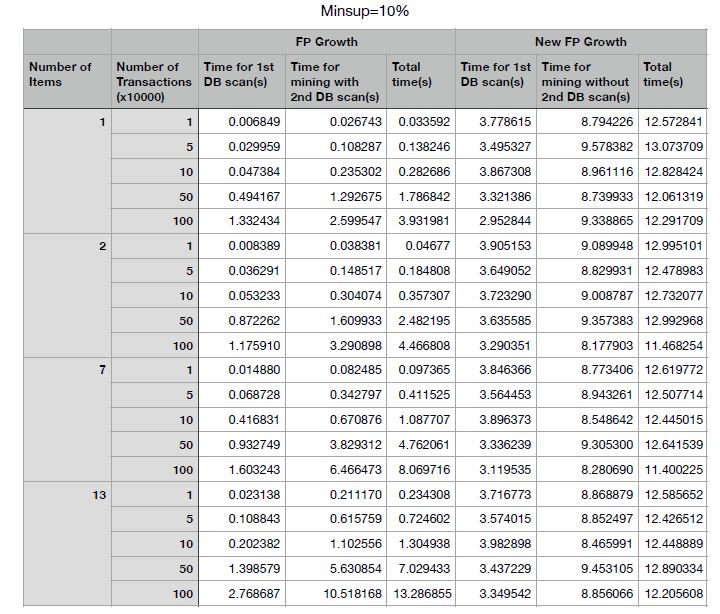


Table 2

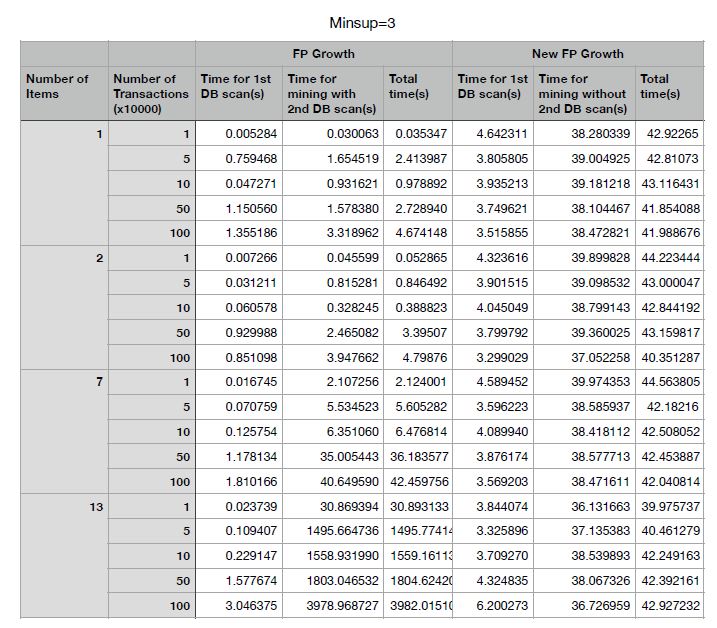
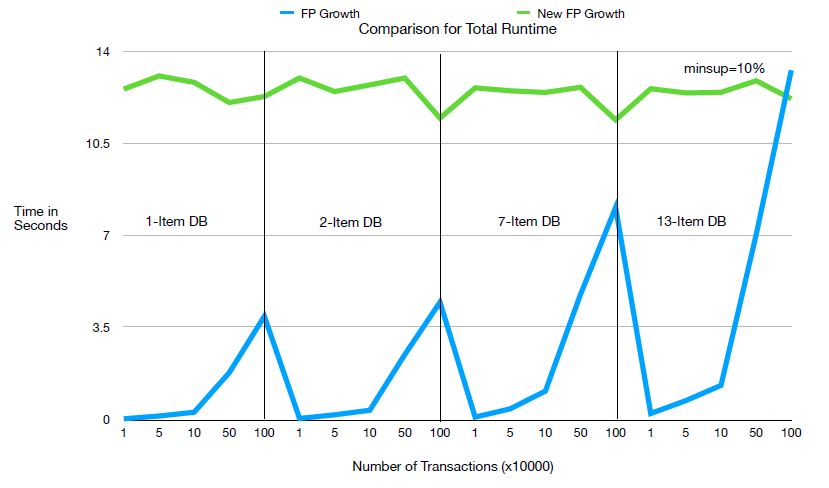
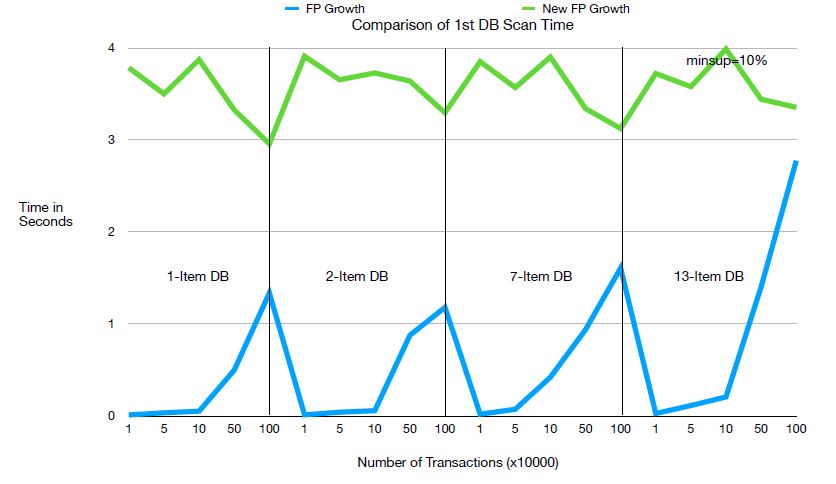
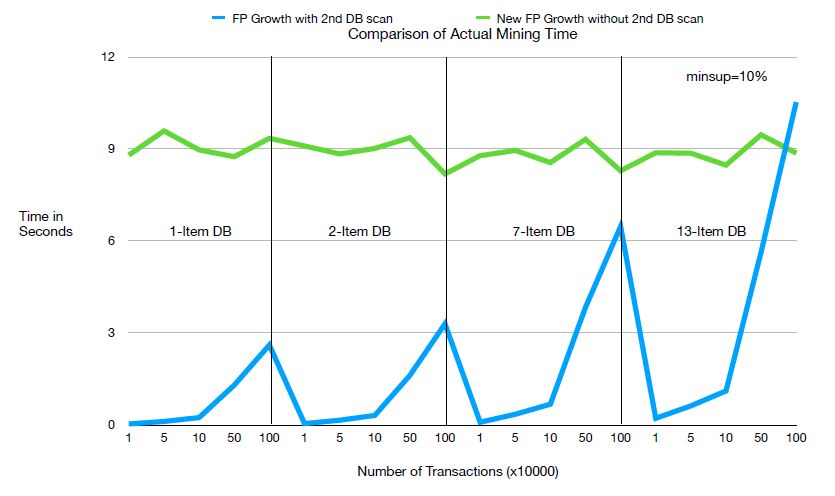


Table 3

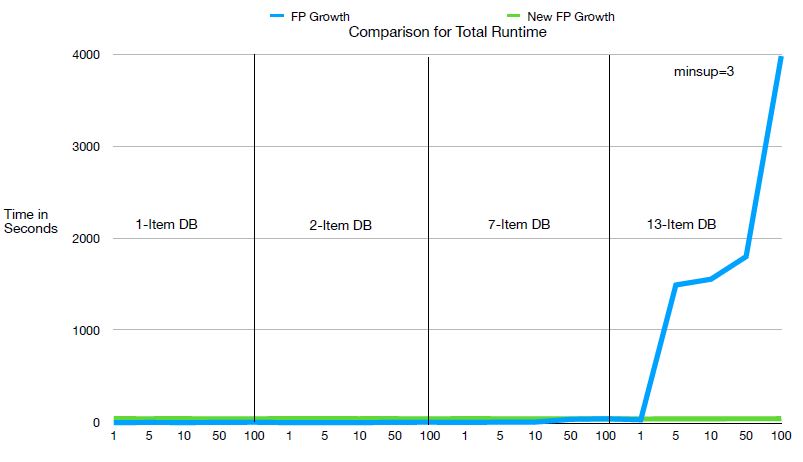
**Graph comparison for minsup = 10%**

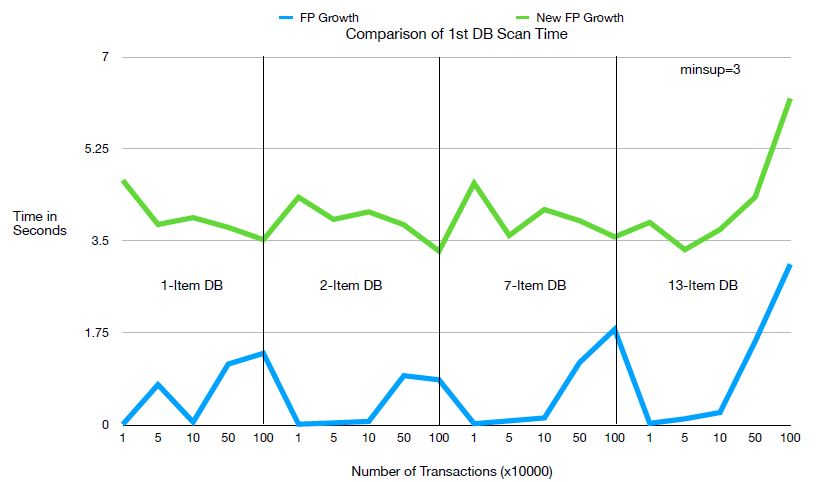


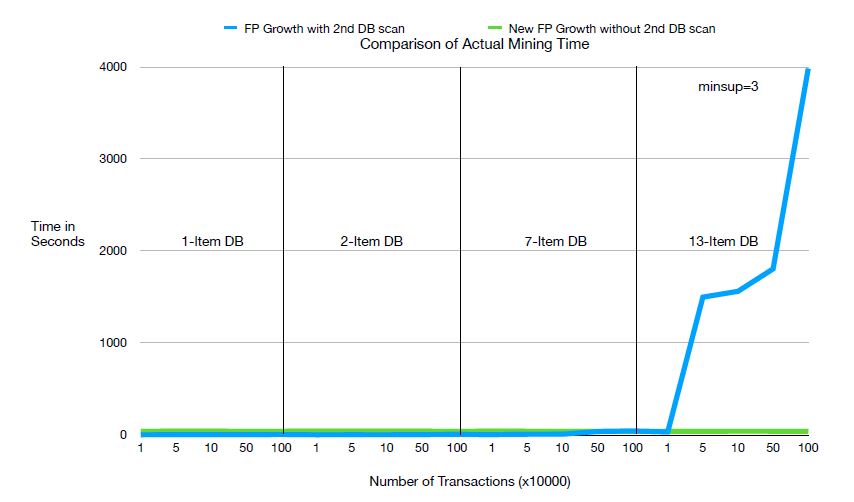




**Graph comparison for minsup = 3**







In terms of the practicality of our proposed changes to FP-Growth, these results are promising. Since most interesting mining will be done using large datasets with many unique itemsets, our new algorithm could pose for more efficient frequent pattern mining in certain scenarios.