An Experimental Study on Dynamic Prefix Tree Algorithm

Dr. Pallam Ravi[1], Beereddy Harshitha[2], Adithya Krishnamurthy[3], Alla Sai Manideep Reddy[4]

1. Assistant professor, Dept. of CSE, Anurag University, Hyderabad, India

2. UG Student, Dept. of CSE, Anurag University, Hyderabad, India

3. UG Student, Dept. of CSE, Anurag University, Hyderabad, India

4. UG Student, Dept. of CSE, Anurag University, Hyderabad, India

ABSTRACT: Frequent itemset mining is a fundamental problem in data mining that has numerous applications, such as recommendation systems and market analysis. Traditional algorithms for frequent itemset mining, while effective, suffer from memory overhead and the need for multiple database scans. To address these limitations, this work proposes a novel approach called the Dynamic Prefix Tree (DPT) algorithm. The DPT algorithm utilizes a single, dynamically adaptive prefix tree to represent datasets and efficiently compute frequent itemsets. This approach significantly reduces memory consumption and processing costs compared to traditional methods. The researchers implemented the DPT algorithm in Java using object-oriented programming and explored various optimization strategies, such as reducing the number of nodes, minimizing DPT calls, and directly generating maximal itemsets. However, these optimization efforts did not yield substantial improvements, indicating that the DPT algorithm is already highly efficient. The results demonstrate that the DPT structure effectively identifies frequently occurring itemsets while minimizing memory requirements and enhancing scalability for large datasets. Unlike traditional methods, the DPT algorithm directly represents all frequently occurring itemsets within a prefix tree, offering a streamlined and memory-efficient approach to frequent itemset mining.

KEYWORDS: Frequent itemset mining, Dynamic Prefix Tree (DPT), Copying and Merging, Memory consumption, Processing speed, Scalability.

I.INTRODUCTION

Frequent pattern mining has been an important subject matter in data mining from numerous times. Frequent itemset mining is a foundational fashion in data mining, particularly within the realm of association rule mining. It involves the birth of sets of particulars that constantly co-occur together in transactional datasets. These sets of particulars, known as frequent itemsets, are pivotal for uncovering hidden patterns and connections within the data, which can give precious perceptivity for decision- timber and analysis. In 1993, Agrawal et al. first introduced the concept of pattern mining as request grounded analysis, which involves analyzing data to randomly select specific items from a request. This idea highlighted associations' accidental architectures, fascinating correlations, or recurring patterns among set of data by using transactional databases and other data depositories. These specifics, substructures, or sequences that duplicate in a database with a predetermined frequency are known as frequent patterns. An itemset will be deemed frequent if its frequency is greater than or equal to the minimum support threshold.

The Apriori algorithm is an effective system for frequent itemset mining. In order to gain all frequent itemsets, Apriori iteratively scans a database. It can find out all frequent itemsets. Generating and testing a large number of candidate itemsets, Apriori and the Apriori like algorithms are called candidate generation- and- test approaches. Although these approaches can mine all frequent itemsets from a database, they're ineluctably brazened with two problems (1) the database is scrutinized numerous times, and the time of database checkup is exactly equal to the length of the longest frequent itemset;(2) the number of candidate itemsets is veritably large, and generating and testing these campaigners is truly expensive, especially when the database or all candidate itemsets cannot be fully loaded in main memory. The issues mentioned over can be avoided and common itemsets from a database can be mined using pattern growth approaches like FP- Growth [10,11]. A database is represented in FP- Growth as an extremely compact prefix tree structure known as an FP- tree. FP- Growth builds tentative FP- trees iteratively to mine common itemsets after creating an original FP- tree from a database.

Problem Illustration

Consider as an example that there are set of transactions T which contains the set of items I= {a, b, c, d, e, f, g, h} as shown in Table 1. Now we have assumed the minimum support threshold (min_sup) as 3. Based on the min_sup we have eliminated the infrequent items. Then the database contains the transactions consisting of frequent items which are sorted in the descending order of min_sup as shown in Table 2. From the Sorted frequent itemsets construct a prefix tree (FP-Tree) as shown in Figure 1.

Table 1. Dataset

Transaction ID	Itemset
1	{a, b, c, e}
2	{b, d}
3	{a, c, e}
4	{a, b, c, f}
5	{b, d, g}
6	{a, b, d, h}
7	{a, h}
8	{b, d, g}

Table 2. Database and Sorted Frequent items

Transaction ID	Itemset	Ordered Frequent itemset
1	{a, b, c, e}	{b, a, c}
2	{b, d}	{b, d}
3	{a, c, e}	{a, c}
4	{a, b, c, f}	{b, a, c}
5	{b, d, g}	{b, d}
6	{a, b, d, h}	{b, a, d}
7	{a, h}	{a}
8	{b, d, g}	{b, d}

In-order to obtain all the frequent patterns related an item we need to construct many conditional trees. For frequent itemset with k items, k conditional prefix trees must be constructed. Prefix tree construction is time-consuming and constantly results in poor data localization. So, to overcome this problem we have used a novel approach known as Dynamic Prefix Tree Algorithm. Unlike traditional methods, DPT offers a direct representation by generating a prefix tree consisting of all frequently occurring itemsets**.**

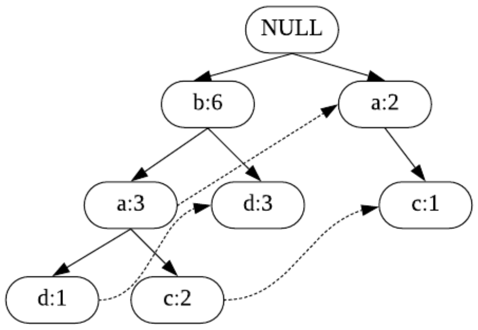

Figure 1. Prefix Tree

II. LITERATURE SURVEY

The foundations of frequent itemset mining were laid by the seminal work of Agrawal, Imieliński, and Swami in 1993 [2]. Their introduction of the concept of "mining association rules between sets of items in large databases" sparked a surge of research in this field, leading to the development of various algorithms and techniques for efficient pattern discovery. The Apriori algorithm [2] is the most widely used algorithm in the history of association rule mining. It utilizes an efficient candidate generation process, where large itemsets generated at level k are used to generate candidates at level k+1. The key insight behind Apriori is the anti-monotone property, which states that if an itemset is frequent, then all of its subsets must also be frequent. This property allows Apriori to avoid the generation of unnecessary candidate itemsets, significantly improving the overall efficiency of the mining process. However, the Apriori algorithm requires multiple scans of the database as long as large frequent itemsets are generated, which can be computationally expensive for large datasets. To address the limitations of the Apriori algorithm, Agrawal and Srikant later introduced the Apriori TID algorithm [8]. Apriori TID generates candidate itemsets before scanning the database using the Apriori-gen function. Instead of scanning the database multiple times, Apriori TID scans the candidate itemsets to count their support. This approach performs better at higher levels of the mining process, as the number of candidate itemsets becomes smaller, while the conventional Apriori performs better at lower levels where the candidate generation is less costly. Building on these ideas, Padillo et al. [9] explored the Apriori Hybrid approach, which combines the strengths of both Apriori and Apriori TID. Apriori Hybrid uses the Apriori TID algorithm in the later passes of the mining process, as it is more efficient at higher levels, while employing the standard Apriori algorithm in the initial passes where the candidate generation is less challenging. Other notable developments in this field include the Direct Hashing and Pruning (DHP) algorithm [12], which aims to maximize efficiency by reducing the number of candidates generated, and the DIC (Dynamic Itemset Counting) algorithm [13], which decreases the number of database scans by dividing the database into intervals of particular sizes. The CARMA (Continuous Association Rule Mining Algorithm) [14] generates more candidate itemsets with less database scanning, while also providing the flexibility to change the minimum support threshold. ECLAT [15], on the other hand, takes a different approach by using a vertical data format to store and process itemsets. In ECLAT, each item is associated with a list of transaction IDs in which it appears, and candidate itemsets are generated by intersecting these lists. This vertical data representation allows for more efficient support counting compared to the traditional horizontal data format. Sampling algorithms [16] address the I/O overhead of scanning the entire database by considering only random samples, while the Rapid Association Rule mining (RARM) [16] approach generates large 1-itemsets and 2-itemsets using a specialized tree structure without the need for database scans. One of the key advancements in this domain was the introduction of the Frequent-Pattern tree (FP-tree) by Han et al. [10]. The FP-tree is a compact, tree-based data structure that enables the mining of frequent patterns without the need for candidate generation, a significant improvement over the Apriori algorithm. This data structure forms the foundation for many subsequent developments in frequent itemset mining. Another notable accomplishment is the FP-Growth algorithm [10], which builds upon the FP-tree concept. FP-Growth overcomes two key limitations of the Apriori algorithm: the complex candidate itemset generation process and the need for expensive and repetitive database scans. FP-Growth achieves efficiency through a divide-and-conquer approach, decomposing the mining problem into smaller problems in conditional databases, and using a highly summarized, compressed FP-tree data structure to avoid repetitive database scans. These foundational works and subsequent advancements have shaped the field of frequent itemset mining, leading to the development of numerous algorithms, techniques, and applications that have significantly improved the efficiency and scalability of pattern discovery in large datasets. Researchers have explored various optimizations, parallelization strategies, and specialized techniques to address the challenges posed by the exponential growth of data in modern applications. The field of frequent itemset mining continues to evolve, with ongoing research focusing on areas such as mining patterns from uncertain data, incorporating user preferences and utility measures, and adapting to the dynamic nature of many real-world datasets. As data volumes and complexity continue to grow, the need for efficient and effective frequent pattern mining algorithms remains a crucial challenge in the data mining and knowledge discovery domains.

III. METHODOLOGY

A) Dynamic Prefix Tree Algorithm

The efficiency of FP-Growth and FP-Growth-like algorithms is higher than that of Apriori and Apriori-like algorithms; however, the performance of these algorithms is lowered when they must spend a lot of time building numerous conditional prefix trees when mining frequently occurring itemsets. The Dynamic Prefix Tree (DPT) algorithm mines all frequent itemsets with their supports using just one prefix tree as opposed to several prefix trees. After constructing the prefix tree from the database, the Dynamic Prefix Tree (DPT) algorithm recursively processes each node within the tree to extract frequent itemsets and manage memory efficiently. For each node (N), DPT initiates by copying the subtree rooted at (N) into its right sibling nodes. This copying process ensures that the algorithm can continue processing the prefix tree while maintaining the original structure intact. Following the copying step, DPT evaluates the support count (N.counter) associated with node (N). If the support count is equal to or greater than the specified minimum support threshold (min_sup), DPT generates an itemset composed of the items along the path from the root node to (N), with (N.counter) representing its support. This step enables the extraction of frequent itemsets directly from the prefix tree.

Subsequently, DPT recursively processes all child nodes of (N), repeating the aforementioned steps for each child node. Once all child nodes have been processed, DPT releases the memory occupied by node (N). If the support count (N.counter) is less than the minimum support threshold (min_sup), indicating that the associated itemset is infrequent, DPT frees the entire subtree rooted at (N) after copying it. This pruning process ensures that only relevant portions of the prefix tree are retained, optimizing memory usage.

DPT Optimization Technique: Efficiently processing small nodes in a prefix tree involves distinguishing between big nodes and small nodes based on their counter values in relation to the minimum support threshold. When the Dynamic Prefix Tree (DPT) algorithm encounters a big node, it proceeds to copy its subtree to the right sibling nodes, generating necessary branches unique to that subtree. However, for small nodes, whose counter values fall below the minimum support threshold, the subtree rooted at them is no longer utilized according to the algorithm. Therefore, an efficient technique involves cutting off the subtree at the small node and merging it with the subtrees rooted at the right sibling nodes of that small node. This approach eliminates the need to generate new branches, streamlining the processing of small nodes within the prefix tree.

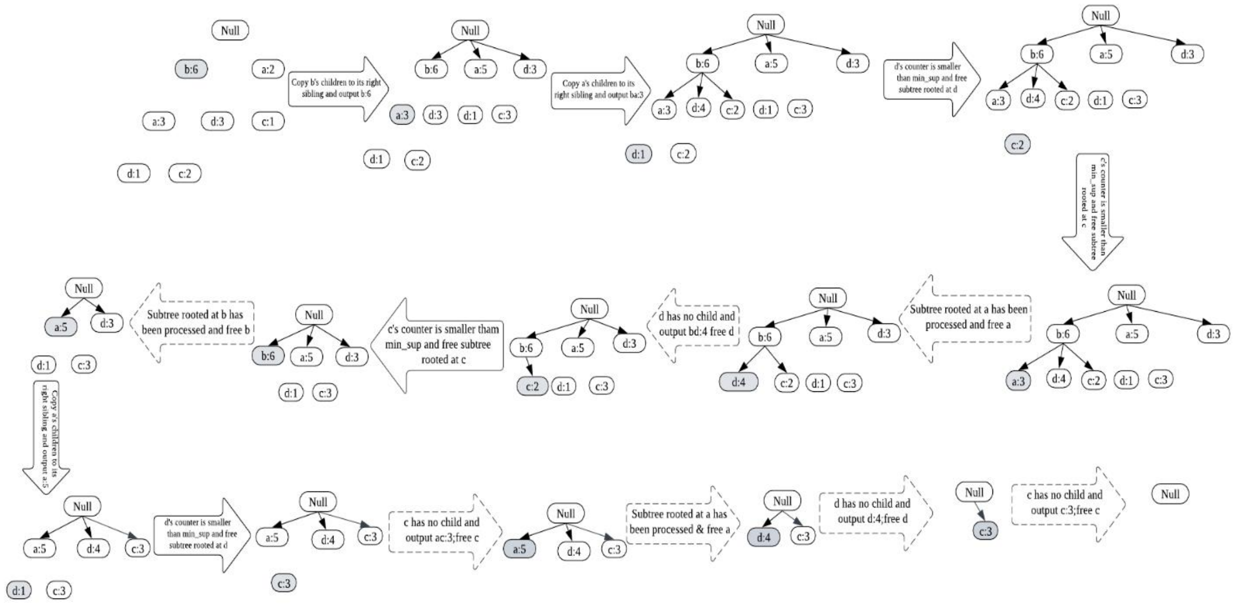
Copy b's children to its right sibling and output b:6
Copy a's children to its right sibling and output ba:3
d's counter is smaller than min_sup and free subtree rooted at d
c's counter is smaller than min_sup and free subtree rooted at c
Subtree rooted at a has been processed and free a
d has no child and output bd:4 free d
c's counter is smaller tham min_sup and free subtree rooted at c
Subtree rooted at b has been processed and free b
Copy a's children to its right sibling and output a:5
d's counter is smaller than min_sup and free subtree rooted at d
c has no child and output ac:3;free c
Subtree rooted at a has been processed & free a
d has no child and output d:4;free d
c has no child and output c:3;free c

Figure 2. DPT Runtime

B) An Experimental Study

a) Reducing the DPT calls

In the attempt to minimize the number of Dynamic Prefix Tree (DPT) calls by introducing a global counter for each item and reducing it upon emitting a pattern with its count, a significant challenge arose due to the interplay between the global counter and the minimum support condition. While the global counter initially seemed like a promising solution to track the occurrence of each item across the dataset, its decrementing nature led to unintended consequences. As patterns are emitted and their counts are subtracted from the global counter, certain items might fall below the minimum support threshold, thus violating the condition and potentially excluding valid patterns from being further explored. This scenario occurs because the global counter's decrementing behavior disregards the context in which an item contributes to the support of a pattern; it treats all occurrences of an item equally, irrespective of their significance within specific patterns or transactions.

Furthermore, the reliance on a single global counter for each item across the entire dataset overlooks the inherent variability in item support levels within different transaction contexts. Items may exhibit varying degrees of importance or relevance across transactions, and a blanket decrementing of their global counters fails to capture this nuanced relationship. Consequently, patterns that may be crucial within certain transaction subsets might be prematurely pruned due to their cumulative counts falling below the minimum support threshold at a global level. This limitation underscores the importance of context-aware support calculations and the need for a more nuanced approach to tracking item occurrences within the DPT algorithm.

For the example described in the Table 1, the process of adding global counter goes as below:

Table 3. Global Counter of each item

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**
5	6	3	4	2	1	2	2

The total number of DPT calls are 14 and the Minimum Support is set to 3. Since the minimum support is 3 the items e, f, g, h are eliminated first during the construction of prefix tree. So, the remaining items are a:5, b:6, c:3, d:4 and the frequent patterns obtained from the prefix tree are b:6, ab:3, bd:4, a:5, ac:3, d:4, c:3.

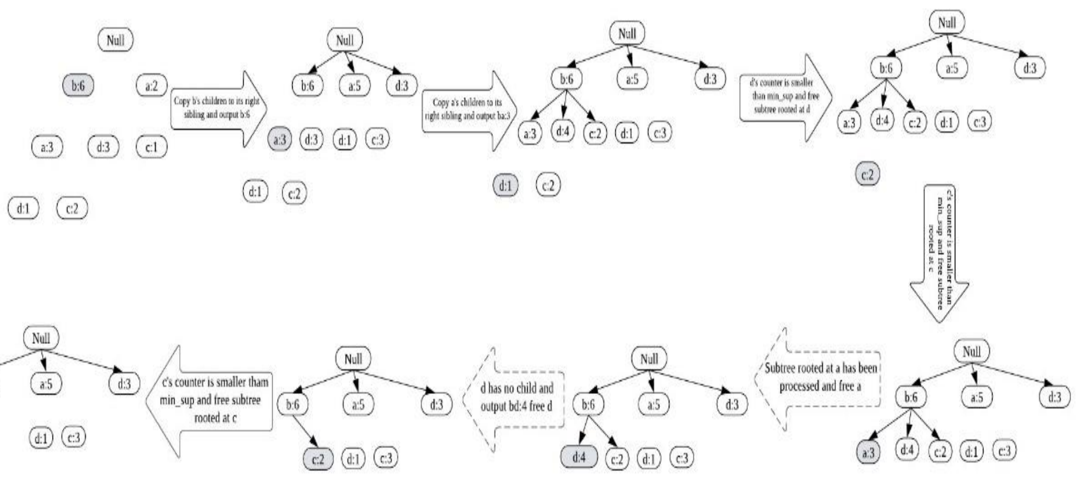
Null
b:6
a:2
a:3
d:3
c:1
d:1
c:2
Copy b's children to its right sibling and output b:6
Copy a's children to its right sibling and output ba:3
d's counter is smaller than min_sup and free subtree rooted at d
c's counter is smaller than min_sup and free subtree rooted at c
Subtree rooted at a has been processed and free a
d has no child and output bd:4 free d
c's counter is smaller tham min_sup and free subtree rooted at c
a:5
d:4
c:3

Figure 3. Failure of DPT Runtime

When the first pattern b:6 is emitted the global counter of b is reduced to 0. Similarily when the second pattern ab:3 is emitted the global counter of a is reduced by 3. When d:1 node is freed the d's counter is reduced by 1 so the global counter of d is 3 after the node is freed. When the pattern bd:4 is emitted, the condition fails as the global counter of d is 3 but the pattern which is to be emitted is bd with counter 4. So, the condition fails and bd:4 will never be emitted.

We attempted to reduce the DPT calls by adding global counter so that when an infrequent node is emitted and global counter is reduced, which doesn't satisfy the condition. So, our attempt to add global counter and reduce the nodes have failed as we thought the items have single occurrence but in reality, the dataset has multiple occurrences of the same item.

b) Reducing the Nodes

In the attempt to refine the Dynamic Prefix Tree (DPT) algorithm by introducing a global counter for each item and adjusting it based on node creation and merging, a critical oversight emerged regarding the interaction between the global counter and the minimum support condition. While the approach sought to dynamically track the support of each item across the dataset, the subsequent adjustment of the global counter after node copying and merging inadvertently undermined the efficacy of the minimum support condition. By incrementing the global counter following each node operation, the condition for node removal based on minimum support became perpetually satisfied. This perpetual satisfaction of the condition rendered it ineffective in accurately identifying and removing nodes that no longer met the minimum support threshold, thus impeding the desired reduction in the number of nodes within the tree structure. This oversight underscores the importance of context-aware support calculations and the need for a more nuanced approach to node management within the DPT algorithm. Ultimately, the attempted refinement, while well-intentioned, highlighted the complexity of balancing global item tracking with contextual support evaluation, necessitating further exploration of alternative strategies to optimize the DPT algorithm effectively.

Table 4. Global counters of each transaction for every DPT call

DPT Calls	a	b	c	d
First DPT call	8	6	4	7
Second DPT call	8	6	7	9
Third DPT call	8	6	7	8
Fourth DPT call	8	6	5	8
Fifth DPT call	5	6	5	8
Sixth DPT call	5	6	5	4
Seventh DPT call	5	6	3	4
Eighth DPT call	5	0	3	4
Ninth DPT call	5	0	6	5
Tenth DPT call	5	0	3	4
Eleventh DPT call	5	0	6	4
Twelfth DPT call	0	0	3	4
Thirteenth DPT call	0	0	3	0
Fourteenth DPT call	0	0	0	0

From the Table 4, we can conclude that the global counter after each DPT call changes because of copying and merging of subtrees. As the copying and merging of nodes increments the counter of each item, the global counter is incremented. In the same way removing of a node decrements the counter of item, then the global counter also decrements. But the global counter of each item changes which always satisfies the minimum support condition even after deleting the counter of freed node. This attempt also fails because of multiple occurrences of items in the dataset.

Finding the Maximal Frequent Itemsets using Dynamic Prefix Tree

The attempt to integrate the Maximum Frequent Itemset (MFI) list algorithm with the Dynamic Prefix Tree (DPT) algorithm for finding maximal itemsets encountered challenges, ultimately resulting in failure. The failure can be attributed to several factors, primarily stemming from the complexity and intricacies involved in synchronizing the two algorithms seamlessly. One of the critical points of failure lies in the process of traversing the MFI list and simplifying the FP-tree. While the MFI list aims to identify maximal frequent itemsets efficiently, the integration with the DPT algorithm introduces additional complexities in managing the tree structure and support counts, leading to discrepancies in the traversal process.

Moreover, the process of copying and merging nodes within the DPT algorithm further complicates the traversal of the MFI tree. The intricate relationships between nodes in the FP-tree, coupled with the dynamic adjustments made during node copying and merging, pose significant challenges in maintaining the integrity of the MFI list. Additionally, the failure to precisely identify the reason behind the integration failure underscores the inherent complexities and nuances involved in combining distinct algorithms to achieve a unified objective.

IV. EXPERIMENTAL SETUP

Eclipse is a robust IDE for Java, streamlining object-oriented development. It simplifies tasks like code management and debugging. The FIMI repository offers structured text datasets, enhancing Java's data processing capabilities.

Understanding time and space complexity helps optimize algorithms for efficiency and scalability, crucial for handling large datasets.

Parameters

a) Time complexity

Time complexity in programming is a key metric for assessing algorithmic efficiency, focusing on how an algorithm's execution time grows relative to its input size. It's often represented using Big O notation (O()), which estimates the worst-case scenario for runtime. By understanding time complexity, developers can anticipate how algorithms will perform with larger datasets and pinpoint potential performance bottlenecks.

Execution Time = End Time – Start Time

b) Space complexity

Space complexity evaluates the memory space an algorithm requires concerning its input size. It's denoted using Big O notation (O()) and provides an upper bound on the worst-case memory usage. Analyzing space complexity helps developers comprehend how memory usage scales with input size. This involves assessing various elements such as variables, data structures, additional resources allocated during execution, and memory implications of inheritance hierarchies and polymorphic behavior.

Memory = Total Runtime Memory –Free Runtime Memory

V. RESULTS

The Dynamic Prefix Tree (DPT) algorithm with optimization demonstrates superior performance compared to other algorithms, showcasing the efficiency and effectiveness of the optimization techniques employed. The reduced memory consumption and faster execution time of DPT with optimization highlight its capability to handle large datasets more adeptly. Additionally, the comparison with alternative algorithms such as Apriori and FP-Growth underscores the competitive advantage of DPT in terms of both time and space complexity. These findings emphasize the importance of optimizing algorithms for enhanced performance in real-world applications, particularly when dealing with substantial datasets where efficiency is paramount.

Table 5. Chess Dataset

Method	Memory Allocated	Execution Time
Apriori	118310344 bytes	1388430400 nanoseconds
FPGrowth	192346616 bytes	391595800 nanoseconds
DPT Optimization	96062464 bytes	2741550900 nanoseconds
DPT	101010432 bytes	2939057900 nanoseconds

From the above table, we can observe out of the four algorithms tested - Apriori, FP Growth, Dynamic Prefix Tree (DPT), and DPT with optimization - the latter proved to be the most efficient, consuming significantly less memory and time. With optimization techniques, DPT outperformed the others by consuming only 96,062,464 bytes of memory and completing the task in 2,741,550,900 nanoseconds. Consequently, DPT with optimization emerges as the preferred choice for handling large datasets and complex processing tasks.

Table 6. Accidents Dataset

Method	Memory Allocated	Execution Time
Apriori	592447488 bytes	2423542600 nanoseconds
FPGrowth	72571368 bytes	27753853500 nanoseconds
DPT Optimization	716702440 bytes	2013028000 nanoseconds
DPT	1274542592 bytes	418222068800 nanoseconds

From the above table, we can observe out of the four algorithms tested - Apriori, FP Growth, Dynamic Prefix Tree (DPT), and DPT with optimization - the latter proved to be the most efficient, consuming significantly less memory and time. With optimization techniques, DPT outperformed the others by consuming only 716702440 bytes of memory and completing the task in 2013028000 nanoseconds. Consequently, DPT with optimization emerges as the preferred choice for handling large datasets and complex processing tasks.

Table 7. T10L4D100K Dataset

Method	Memory Allocated	Execution Time
Apriori	103286784 bytes	866487400 nanoseconds
FPGrowth	17153736 bytes	10858916200 nanoseconds
DPT Optimization	200369184 bytes	2005798800 nanoseconds
DPT	245338112 bytes	2754644900 nanoseconds

From the above table, we can observe out of the four algorithms tested - Apriori, FP-Growth, Dynamic Prefix Tree (DPT), and DPT with optimization - the latter proved to be the most efficient, consuming significantly less memory and time. With optimization techniques, DPT outperformed the others by consuming only 200369184 bytes of memory and completing the task in 2005798800 nanoseconds. Consequently, DPT with optimization emerges as the preferred choice for handling large datasets and complex processing tasks.

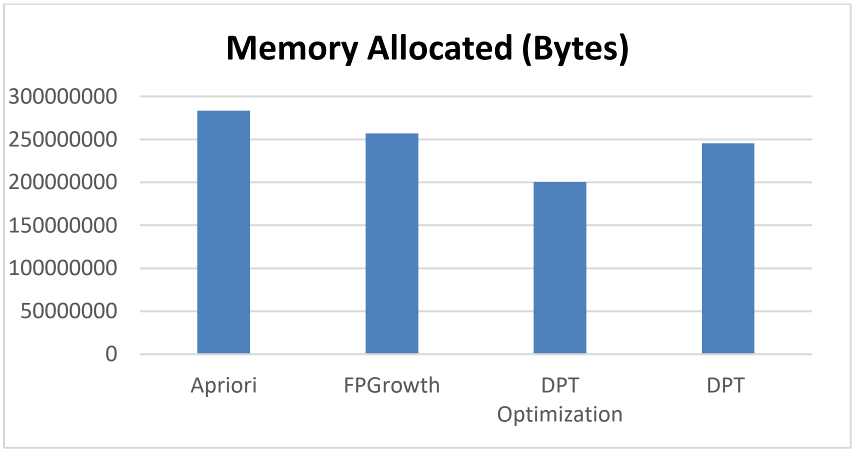

Figure 4. Memory Consumed by T10I4D100K dataset

From the above diagram we can draw the conclusions such as both DPT and FP-Growth can be optimized for efficiency, the provided diagram suggests DPT with optimization shines in terms of memory usage. This is evident because the line representing DPT with optimization consistently stays below the line for FP-Growth with optimization, signifying DPT's ability to handle the data while requiring less memory throughout the process.

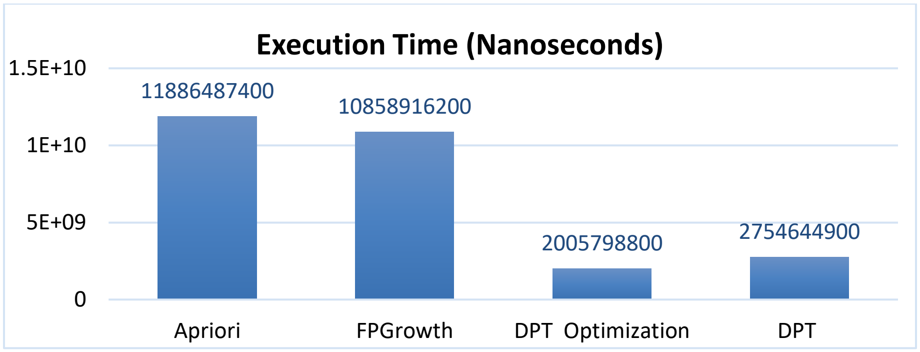

Figure 5. Time Consumed by T10I4D100K dataset

From the above diagram, we can obtain the conclusions as DPT with Optimization stands out as the undisputed leader in terms of speed. Its optimized execution time, depicted by the shortest line on the graph, approximately at 200 billion nanoseconds, surpasses Apriori (1.2 trillion nanoseconds), FPGrowth (1.1 trillion nanoseconds), and even the non-optimized DPT (2.7 trillion nanoseconds).

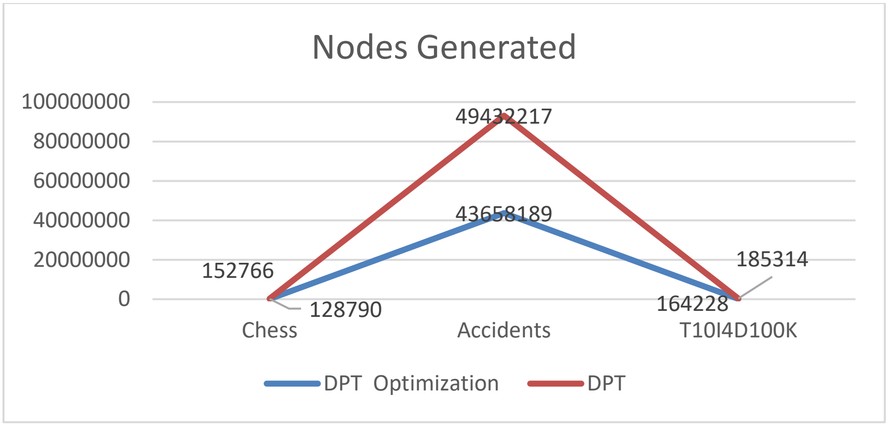

Figure 6. Nodes Generated by three different datasets

From the above diagram, we can obtain the conclusions as the graph compares the number of nodes generated by three datasets Chess, Accidents, and T10I4D100K under two conditions, DPT Optimization and DPT. Overall, all the datasets generate significantly more nodes with the DPT condition. Chess, Accidents and T10I4D100K datasets DPT Optimization generates fewer nodes than DPT.

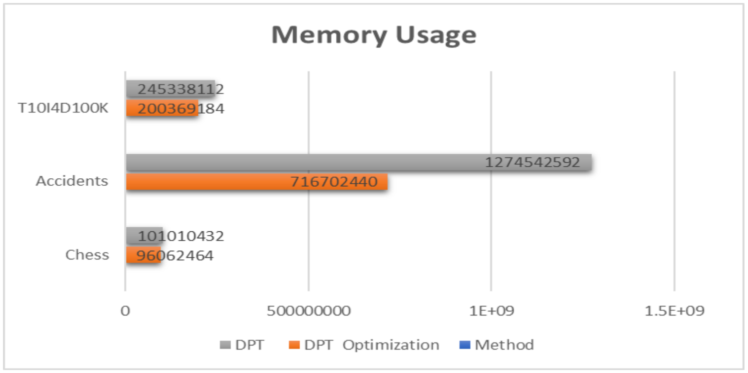

Figure 7. Memory Usage of three different datasets

From the above diagram, we can obtain the conclusions that it compares memory usage between two algorithms, likely DPT (Dynamic Prefix Tree) with and without optimization. The y-axis shows memory usage and the x-axis likely represents different datasets. For each dataset, the memory usage of DPT with optimization is lower than DPT without optimization.

Findings

1) Dynamic prefix tree is easily implemented in object-oriented programming. There are many advantages like:
Firstly, OOP facilitates modularity by organizing the DPT implementation into smaller, manageable classes, simplifying development and maintenance. Secondly, encapsulation hides the internal details of the tree structure, exposing only the necessary interfaces for interaction, thus reducing complexity. Thirdly, inheritance enables the creation of specialized versions of the DPT, promoting code reuse and extensibility. Additionally, polymorphism allows flexibility by treating objects of different classes as instances of a common superclass, enabling seamless integration of various functionalities.

2) The drawback of assuming single occurrences of items in the dataset undermines the reliability of support counts and may lead to inaccurate pattern identification and premature pruning, ultimately compromising the effectiveness of the DPT algorithm in analyzing the dataset.
3) The complexity introduced by merging and copying subtrees within the Dynamic Prefix Tree (DPT) algorithm, coupled with the complexity of managing multiple algorithms within a unified framework, poses significant challenges in finding maximal itemsets. The process of synchronizing these algorithms becomes difficult to deal, especially considering the dynamic adjustments made during subtree copying and merging. The complexity inherent in managing these operations within a single framework renders the task of finding maximal itemsets using the DPT algorithm unfeasible. This underscores the need for alternative approaches or simpler algorithms to achieve this objective effectively.

VI. CONCLUSION AND FUTURE ENHANCEMENT

In conclusion we have explored various optimization approaches for the Dynamic Prefix Tree (DPT) method, including attempts to reduce DPT calls, minimize node count, and extract maximal frequent itemsets directly using DPT. However, these efforts encountered challenges: the first two approaches were impeded by the dataset's multiple occurrences of itemsets, contrary to the assumption of single occurrences. Additionally, merging DPT with the maximal frequent itemset algorithm led to increased complexity without notable performance gains. These challenges underscore the DPT's existing optimization, as it stands as one of the best-performing algorithms to date.

VII. REFERENCES

[1] Jun-Feng Qu, Bo Hang, Zhao Wu, Zhong Bo Wu, Qiong Gu and Bo Tang, "Efficient Mining of Frequent Itemsets Using Only One Dynamic Prefix Tree", January 2020, IEEE, Access 8:183722183735, DOI:10.1109/ACCESS.2020.3029302.
[2] R. Agrawal, T. Imieliński, and A. Swami, ''Mining association rules between sets of items in large databases,'' in Proc. ACM SIGMOD Int. Conf. Manage. Data, 1993, pp. 207–216.
[3] Y. Djenouri, D. Djenouri, A. Belhadi, and A. Cano, ''Exploiting GPU and cluster parallelism in single scan frequent itemset mining,'' Inf. Sci., vol. 496, pp. 363–377, Sep. 2019.
[4] F. Guil, ''Associative classification based on the transferable belief model,'' Knowl.-Based Syst., vol. 182, Oct. 2019, Art. no. 104800.
[5] S. Gao, M. Zhou, Y. Wang, J. Cheng, H. Yachi, and J. Wang, ''Dendritic neuron model with effective learning algorithms for classification, approximation, and prediction,'' IEEE Trans. Neural Netw. Learn. Syst., vol. 30, no. 2, pp. 601–614, Feb. 2019.
[6] X. Luo, M. Zhou, Y. Xia, Q. Zhu, A. C. Ammari, and A. Alabdulwahab, ''Generating highly accurate predictions for missing QoS data via aggregating nonnegative latent factor models,'' IEEE Trans. Neural Netw. Learn. Syst., vol. 27, no. 3, pp. 524–537, Mar. 2016.
[7] J. Huang, S. Li, and Q. Duan, ''Constructing multicast routing tree for inter-cloud data transmission: An approximation algorithmic perspective,'' IEEE/CAA J. Automatica Sinica, vol. 5, no. 2, pp. 514–522, Mar. 2018.
[8] R. Aggrawal and R. Srikant, ''Fast algorithm for mining association rules in large databases,'' in Proc. VLDB, 1994, pp. 487–499.
[9] J. Han, J. Pei, and Y. Yin, ''Mining frequent patterns without candidate generation,'' in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2000, pp. 1–12.
[10] B. Li, Z. Pei, K. Qin, and M. Kong, ''TT-miner: Topology-transaction miner for mining closed itemset,'' IEEE Access, vol. 7, pp. 10798–10810, 2019.
[11] F. Padillo, J. M. Luna, F. Herrera, and S. Ventura, ''Mining association rules on big data through map reduce genetic programming,'' Integr. Comput.- Aided Eng., vol. 25, no. 1, pp. 31–48, 2018.
[12] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, ''A tree projection algorithm for generation of frequent item sets,'' J. Parallel Distrib. Comput., vol. 61, no. 3, pp. 350–371, Mar. 2001.
[13] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, ''New algorithms for fast discovery of association rules,'' in Proc. KDD, 1997, pp. 283–286.
[14] J. Han, J. Pei, Y. Yin, and R. Mao, ''Mining frequent patterns without candidate generation: A frequent-pattern tree approach,'' Data Mining Knowl. Discovery, vol. 8, no. 1, pp. 53–87, Jan. 2004.
[15] G. Gatuha and T. Jiang, ''Smart frequent itemsets mining algorithm based on FP-tree and DIFFset data structures,'' TURKISH J. Electr. Eng. Comput. Sci., vol. 25, pp. 2096–2107, 2017.
[16] N. Shahbazi, R. Soltani, J. Gryz, and A. An, ''Building FP-tree on the fly: Single-pass frequent itemset mining,'' in Proc. Int. Conf. Mach. Learn. Data Mining Pattern Recognit. Cham, Switzerland: Springer, 2016, pp. 387–400.