IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 3, 2020, accepted July 26, 2020, date of publication July 29, 2020, date of current version August 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3012817

Mining Productive Itemsets in Dynamic Databases

XIANG LI[1], JIAXUAN LI[1], PHILIPPE FOURNIER-VIGER[2], M. SAQIB NAWAZ[2], JIE YAO[2], AND JERRY CHUN-WEI LIN[3], (Senior Member, IEEE)

[1]Department of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China
[2]School of Humanities and Social Sciences, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China
[3]Department of Computing, Mathematics and Physics, Western Norway University of Applied Sciences (HVL), 5063 Bergen, Norway

Corresponding author: Philippe Fournier-Viger (philfv8@yahoo.com)

This work was supported in part by the National Science Foundation of China; and in part by the Harbin Institute of Technology.

ABSTRACT Discovering frequent itemsets is a data analysis task used in numerous domains. It consists of finding sets of items (itemsets) that frequently appear in a set of database records (also called transactions). Though discovering frequent itemsets is useful, it can produce a large amount of spurious patterns. As a result, the user may spend a great amount of time to analyze the itemsets found by a frequent itemset mining algorithm to find truly interesting patterns. Hence, in recent years, a key research topic has emerged which is to discover statistically significant patterns in databases. The most popular model for identifying itemsets that are statistically significant is to discover non-redundant productive itemsets. The state-of-the-art algorithm to extract this set of patterns is OPUS-Miner. A key drawback of that algorithm is that it is designed to be applied to a static database. Moreover, a second drawback of OPUS-Miner is that it discovers all patterns in a database. In other words, the user cannot search for itemsets containing some specific items. This paper addresses these issues by defining the novel problem of discovering targeted non redundant productive itemsets in dynamic databases. An algorithm named IDPI+ (Interactive Discovery of Productive Itemsets) is presented, storing transactions in a tree structure, which can then be interactively queried to identify productive and non redundant itemsets containing specific items. A structure named Query-Tree is also introduced to process many queries at the same time. Moreover, to handle dynamic databases, efficient transaction insertion and deletion algorithms are provided to update the tree. It was observed in an experimental evaluation on benchmark datasets containing various types of data that IDPI+ can handle thousands of queries per second on a desktop computer. Moreover, it was found that IPDI+ is more than an order of magnitude faster than a baseline algorithm.

INDEX TERMS Dynamic database, itemset mining, nonredundant itemsets, productive itemsets, query-tree, significant patterns.

I. INTRODUCTION

Frequent pattern mining [1] is a traditional data mining task, used to find frequently appearing patterns in data. The assumption is that values that appear frequently together are of interest to users. However, numerous frequent patterns found in real-life applications are weakly correlated and uninteresting [37]. For example, assume that some items {*bread*} and {*cola*} are frequently purchased by customers of a retail store. Although these items are weakly correlated, the pattern {*bread*, *cola*} may be frequent just because both items appear frequently in transactions. Finding too many frequent but weakly correlated patterns is a major problem in frequent pattern mining as it is inconvenient for users to look at a large set of patterns. Moreover, spurious patterns can be misleading for decision-making.

To avoid finding spurious patterns, discovering statistically significant patterns has become a key research topic in data mining. Several significant pattern models were put forward to handle various pattern types such as frequent itemsets [24], [37], sequential patterns [30] and periodic patterns [27], [28]. The most well-known model for discovering statistically significant frequent itemsets is that of *non-redundant productive itemsets*. It consists of applying the Fisher exact test to assess if an itemset is significantly correlated when considering all of its bipartitions. In other words, an itemset is considered as significant if its frequency is considerably higher than if all its bipartitions were assumed to be independent. For example, an itemset {*alcool*, *diabetes*} may be considered as significant because the support of {*alcool*, *diabetes*} is much higher than

The associate editor coordinating the review of this manuscript and approving it for publication was Huiling Chen.

if assuming that {*alcool*} and {*diabetes*} are independent. On the other hand, an itemset {*bread*, *cola*} is not significant because its frequency may not be much higher than when assuming that {*bread*} and {*cola*} are independent. The state-of-the-art algorithm to extract the non-redundant productive itemsets from a dataset is OPUS-Miner (Optimized Pruning for Unordered Search Miner) [37]. However, it has several important limitations.

The first one is that to use OPUS-Miner, one must set a parameter k. Then, the algorithm finds the top-k productive itemsets having the highest leverage or lift in the whole database. This is a problem because if k is set too low, patterns that are not in the set of top-k patterns may still be interesting but are not shown to the user. On the other hand, if k is set to a large value, more patterns are shown to the user but the runtime and memory consumption of OPUS-Miner can greatly increase. Thus, a user may be frustrated that he/she cannot guide the algorithm to search for some specific items or itemsets to avoid considering all possibilities. For example, using OPUS-Miner to analyze retail data, a user cannot restrict the search to find significant patterns containing specific items such as *bread*. In fact, OPUS-Miner does not support interactive data mining. If a user would like to determine if some itemset X is productive, he/she has to run OPUS-Miner while setting k to a large value, in the hope that X will appear in the top-k patterns. In case X is not a top-k patterns, the user has to run the algorithm again with a different value of k. This process of trial and error to find patterns is time consuming, cumbersome as well as inconvenient. And in some cases, even if a large k value is used, the desired pattern X may still not be found. Thus, a major limitation of OPUS-Miner is its inability to process queries to verify if a set of itemsets of interest are productive and non-redundant.

For many applications, processing targeted queries is important as users want to find patterns containing a small set of items contained in the database rather than finding patterns containing any combinations of items. In FIM (Frequent Itemset Mining), tree structures have been developed to efficiently process targeted queries on static or dynamic database such as the IT (Itemset Tree) data structure [19], and variations of the IT such as the Min-Max IT [20], Flagged IT [21] and the Memory Efficient IT [14]. These structures can be used to quickly answer queries about specific itemsets such as (1) identifying all frequent itemsets that contain some items, and (2) obtaining the frequency of a specific itemset. Thus, these structures can be used to interactively explore patterns in a database by performing several targeted queries, and refining the queries after obtaining the result of each query. The IT can be updated incrementally by adding new transactions, and can be used to efficiently answer queries.[1] Although the IT structure and its variations have several applications, such as real-time prediction of missing items in shopping carts [12], it is not designed to find patterns that are statistically significant, and hence tends to find many spurious patterns. Moreover, although the IT can be updated by inserting new transactions to learn new trends in the data, no mechanisms are provided to remove old transactions, and thus forget old trends from the data. Thus, as an IT is used over time, its size continuously grows, which decreases its performance for answering queries.

Lastly, another limitation of the OPUS-Miner algorithm is that some patterns that are interesting to the user may not have a high p value for the Fisher exact test. But to apply OPUS-Miner, the user must set a maximum p value. This p value is defined for single items and is then reduced for itemsets containing more than one item by applying the Bonferroni correction. As a result, many itemsets may be discarded by the maximum threshold. But in some cases, it is desirable to show patterns to the user with their p values to let the user decide if the patterns are interesting even if these patterns are not top-k patterns in terms of p values.

In this paper, aforementioned limitations of OPUS-Miner are addressed by proposing a novel approach called IDPI+ (Interactive Discovery of Productive Itemsets). It is designed to efficiently answer user queries such as to check if some itemsets are productive and non redundant. The proposed approach is designed to work on a dynamic database where new transactions may be inserted. Moreover, unlike previous work on the IT structure, the proposed approach allows the deletion of old transactions to forget old trends. The proposed approach efficiently answers queries by relying on a novel query processing algorithm. Queries are handled using a novel structure named Query-Tree (QT), which can also process multiple queries at once to reduce its runtime. The proposed approach was evaluated on several benchmark real-life datasets used in the FIM literature. Experimental results show that IDPI+ can process up-to 1,000 queries per second on a desktop computer, and that IDPI+ can be from 2 to up-to 27 times faster than a baseline IT-based approach. The baseline approach determines whether an itemset in a query is productive and non redundant by computing the support of an itemset and all its subsets using the standard IT structure. Whereas, IDPI+ uses the novel QT structure and MEIT (Memory Efficient Itemset-Tree) to store and compute the support of itemset and all its subsets.

The following sections are organized as follows. Related work is reviewed in Section II. Section III presents preliminaries and the problem definition. Then, Section IV describes the proposed algorithm. Lastly, Section V presents the experimental evaluation, and Section VI draws a conclusion.

II. RELATED WORK

The problem of discovering frequent itemsets in transaction databases was formalized by Agrawal and Srikant [1]. It consists of discovering sets of items that are frequently purchased together in a set of customer transactions. It was shown that this problem is computationally expensive because it has a

[1] It is important to note that the IT is different from the popular FP-tree (Frequent Pattern tree) structure used by the FP-Growth [16] algorithm, which is not designed for answering targeted queries or interactive data mining.

very large search space. For example, if a retail store has 10,000 distinct items on sale, then there are $2^{10000}-1$ possible itemsets. To reduce the search space, the first algorithm, named Apriori [1], applies a property called the Apriori property or downward-closure property to reduce the search space. This property states that the support (occurrence frequency) of an itemset cannot be greater than the support of its subsets. Hence, if an itemset is infrequent, it is unnecessary to consider its supersets, and a large part of the search space can be eliminated to speed up the discovery of frequent itemsets. Although the Apriori algorithm can find all frequent itemsets in a database, it was shown to be quite inefficient. The reasons are that it performs a breadth-first search and can generate a very large number of candidate itemsets that do not exist in the database. As a result, Apriori can have long execution times [13]. To more efficiently find frequent itemsets, several frequent itemsets mining algorithms have been proposed such as ECLAT (Equivalence Class Transformation) [39], FP-Growth (Frequent Pattern-Growth) [16], H-Mine [31] and LCM (Linear time Closed itemset Miner) [36]. To avoid generating candidates, the FP-Growth algorithm introduced a structure named FP-tree (Frequent Pattern tree), which compresses a database in a tree-like structure consisting of a tree and a header table with pointers linking nodes representing the same items in the tree. Then, FP-Growth recursively discovers all frequent itemsets in a database by recursively creating projected FP-Trees from the original FP-Tree, and enumerating itemsets in each tree. H-Mine and LCM also relies on the concept of database projection but adopt different database representations, that is a hyper-structure and an horizontal database, respectively. The ECLAT algorithm is different from the aforementioned algorithms as it uses a vertical database representation to avoid repeatedly scanning the database. Although frequent itemset mining has been initially used for analyzing customer transactions, it has been applied in numerous other domains. Moreover, besides the above algorithms, many other frequent itemset mining algorithms have been developed in recent years [13].

Even though FIM is popular, an important limitation is that frequent itemsets are not always interesting to the user as a pattern may be frequent but contain items that are weakly correlated. In the literature on pattern mining, numerous measures have been proposed to measure the interestingness of patterns. For example, the *occupancy* [35] measure assesses if an itemset represents a large part of the transactions where it occurs. To find itemsets that are correlated, some simple correlation measures have first been considered [2], [11], [33]. A measure called *bond* [11] is defined as the number of transactions that contain an itemset X, divided by the number of transactions that contain at least one item of X. In that case, a large bond value means that items in that itemset may be positively correlated. Other measures similar to the bond include the affinity [38], all-confidence [29], coherence and mean [2], [33], among others [15]. All these measures are useful in some situations but have poor performance in other situations. A major drawback of these measures is that they are not based on statistical tests. Thus, they cannot be used to determine if items in an itemset are correlated in a statistically significant way. Thus, discovering itemsets having a high bond, affinity or coherence value can be highly misleading. Moreover, measures like the bond and affinity can be viewed as rough correlation measures since they do not systematically check that subsets of an itemset are also correlated.

Recently, researchers have addressed these limitations of early work on mining correlated patterns by proposing models to mine correlated patterns that are statistically significant. Different ways of integrating statistical tests in the pattern mining process have been proposed [24], [37]. Among those, the model of discovering *productive* patterns has attracted the most attention. The main idea, which will be explained in more details in the following section, is to apply the Fisher exact test on an itemset to assess if all its bipartitions are significantly correlated. It was shown that the model of mining productive patterns can eliminate many spurious patterns to discover statistically significant itemsets [37].

In that study, the Fisher Exact test was selected because it is non-parametric test for testing independence, which works well even on small sample sizes, provides an exact p value, and is designed for testing if the proportions of two nominal variables are independent (here, the support values of two bipartitions). Compared to other tests such as the Chi-Square test and G-test, the Fisher test is said to be more accurate for small sample sizes (e.g. less than 1000) but may be less for larger sample sizes [25]. For itemset mining, the support values of itemsets can vary depending on the datasets, but may sometimes be very small. Thus, the Fisher test is deemed particularly appropriate for this context. In accordance with this, for mining positive and negative dependency rules, the Fisher test was found to produce better patterns than the Chi-Square test [17]. However, a limitation of the Fisher test is that it is somewhat conservative [25], [37]. Also as for other tests, there is an assumption that each observation is independent from others. To address the problem of multiple-testing, the OPUS-Miner algorithm adds a correction to the Fisher test [37].

In recent years, the productive itemset mining model has been extended from frequent itemsets [24], [37] to other types of patterns such as sequential patterns [30] and periodic patterns [27]. To our knowledge, the state-of-the-art algorithm for identifying productive itemsets is OPUS-Miner [37]. It is a top-k algorithm where the user must specify a parameter k and the algorithm returns the k most statistically significant correlated patterns. Some of the major limitations of OPUS-Miner are that it does not let the user guide the search for patterns. In other words, the user cannot search for targeted patterns such as to quickly determine if a given itemset $\{bread, milk\}$ is productive. Moreover, another limitation is that OPUS-Miner can only be applied on static databases (it is a batch algorithm).

To find patterns in dynamic database, several solutions have been proposed in the field of FIM. First, several

incremental mining algorithms have been designed to find and update the set of frequent itemsets when transactions are deleted, modified or inserted [18], [22], [23], [26] in a transaction database. Several of these algorithms maintain itemsets that are almost-frequent in memory to avoid rescanning the database when it is updated. Second, some *stream mining algorithms* [3]–[5], [32], [34] have been proposed to update frequent itemsets for potentially infinite transaction streams, that is when the database cannot be read more than once. Popular algorithms of this type include estDec [3] and estDec+ [32], which use structures to maintain patterns and calculate upper-bounds on calculation errors of itemset support values. However, a drawback of these algorithms is that they are designed to maintain all frequent itemsets in memory over time rather than for processing targeted queries made by the user on some specific itemsets. Thus, these algorithms maintain many itemsets in memory that are uninteresting to the user, which can results in long runtimes.

As an alternative to incremental and stream mining algorithms, algorithms have been designed to efficiently answer targeted queries made on a dynamic database [14], [19], [20]. The idea is to adopt a lazy approach by maintaining a special structure called Itemset-Tree and then only extracting frequent itemsets when a query is made by a user. This approach has the advantage that not all frequent itemsets must be maintained in memory. The first approach of this type, to our best knowledge, is the Itemset-Tree [19]. Note that the Itemset-Tree structure is fundamentally different from the FP-Tree structure, as in the former, each node represents the intersection of one or more transactions, and can contain multiple items, while in the latter, each node is an item and transactions are represented by paths in the tree. Using the Itemset-Tree structure, targeted queries made by users can be very efficiently processed such as (1) finding all itemsets that contain some given items and are frequent, and (2) calculating the support of an itemset. The Itemset-Tree structure can be viewed as an important component to build interactive pattern mining systems where a user may explore a database by executing multiple queries, while refining queries after obtaining the result of each query. Some improved versions of the Itemset-Tree structure were proposed such as the Memory Efficient Itemset-Tree [14], the Flagged Itemset-Tree [21] and the Min-Max Itemset-Tree [20], which are designed to reduce memory usage, find all frequent itemsets more efficiently and improve the speed for processing queries, respectively. An Itemset-Tree can be incrementally updated by inserting new transactions. However, a major drawback of the Itemset-Tree is that no transaction deletion operation is provided. Thus, it is impossible to remove old transactions. But deleting transactions is a necessary operation for real-time applications where the user wants to discover recent patterns and forget old trends in the data. Because of this limitation, an Itemset-Tree may continuously grow over time, and one may need to rebuild an Itemset-Tree from scratch to remove old transactions, which can be very costly. Another major problem of the Itemset-Tree structure is that it is not designed to find patterns that are statistically significant. Hence, it can yield several spurious patterns and telling apart spurious patterns from interesting patterns may become a difficult task for users.

To address these limitations of previous work, the following section presents the novel problem of discovering targeted statistically significant itemsets in a dynamic database where both transaction insertions and deletions are performed. Then, Section IV presents the proposed approach.

III. PRELIMINARIES AND PROBLEM STATEMENT

The following paragraphs first presents preliminaries about frequent itemset mining and important definitions about productive and non redundant patterns. Then, based on these definitions, the proposed problem is defined. Then, the following section presents the novel IDPI+ approach to solve this problem.

Definition 1 (Transaction Database): Let I represents a set of items (symbols), denoted as $I = \{i_1, i_2, \ldots, i_n\}$. A set of transactions denoted as $D = \{T_1, T_2, \ldots, T_m\}$ is called a *transaction database*. Each transaction T_d of D is a set of items ($T_d \in I$), and d is an integer called the TID (transaction identifier) of T_d.

Example 1: The transaction database D of Table 1 contains five transactions $D = \{T_1, T_2, T_3, T_4, T_5\}$ and five items $I = \{a, b, c, d, e\}$. The first transaction T_1 consists of items a and b. In the context of market basket analysis, T_1 can be interpreted as the purchase of items a and b by a customer. Table 1 is used as running example in the rest of this paper to illustrate definitions and how the proposed approach is applied.

TABLE 1. A transaction database.

TID	Items
T_1	{a, b}
T_2	{b, e}
T_3	{a, c, d}
T_4	{a, b}
T_5	{a}

Definition 2 (Itemset): An itemset Y is a set of items $Y \in I$ that is unordered and where each item cannot appear more than once. An itemset having a cardinality of k is also called an *itemset* of length k.

Definition 3 (Suffix of an Itemset): Consider an itemset $Y = \{a_1, a_2, \ldots, a_k\}$, which is sorted according to a total order $a_1 \succ a_2 \ldots a_k$ (e.g. the alphabetical order), for the convenience of processing itemsets. Furthermore, let an integer r ($1 \leq r \leq k$) be the position of an item in the itemset Y. Then, the suffix of Y for a position r is the subset of Y defined as $suf(Y, r) = \{a_{r+1}, a_{r+2}, \ldots, a_k\}$.

Example 2: Consider that $Y = \{b, c, d\}$ and that $b \succ c \succ d$. It is found that $suf(Y, 1) = \{c, d\}$.

Definition 4 (Cover of an Itemset): Each itemset Y has a cover, defined as the set of transactions that contains Y. It is formally defined as $cov(Y, D) = \{T_i | T_i \in D \wedge Y \subseteq T\}$. In the

following, $cov(Y, D)$ will be denoted as $cov(Y)$ for the sake of brevity.

Example 3: The set of transactions $\{T_1, T_2, T_4\}$ is the cover of itemset $\{b\}$, that is $cov(b) = \{T_1, T_2, T_4\}$. The cover of itemset $\{a, b, d\}$ is $cov(a, b, d) = \emptyset$ as this itemset does not appear in the database.

Most studies on FIM are designed based on the assumption that frequent itemsets are useful or interesting. A measure called support is used to evaluate the occurrence frequency of an itemset.

Definition 5 (Support of an Itemset): Let there be a set of items $Y \subseteq I$. Its support is the number of transactions where it appears, and is written as $sup(Y)$. In other words, $sup(Y) = |cov(Y)|$.

Example 4: Because $Y = \{a, b\}$ appears in transactions T_1 and T_4, its cover is $cov(Y) = \{T_1, T_4\}$. Furthermore, its support is $sup(Y) = 2$.

The traditional task of FIM is applied to a transaction database and requires that the user sets a *minsup* threshold. Then, all the frequent itemsets of that transaction database are enumerated, that each itemset having a support greater or equal to *minsup* [1], [16].

Definition 6 (Mining Frequent Itemsets): Let there be a minimum support threshold (*minsup*) specified by the user (a positive integer) and a database D containing a set of transactions. Mining frequent itemsets in D consists of enumerating all its frequent itemsets. The set of frequent itemsets in D is defined as $FI = \{X | sup(X) \geq minsup \wedge X \subseteq I\}$.

A drawback of FIM is that multiple spurious patterns may be discovered. For this reason, recent studies were done on devising techniques to discover patterns that are statistically significant. In particular, the Opus-Miner algorithm was designed [37] to find a set of statistically significant itemsets in a database, called the *productive itemsets*.

Definition 7 (Bipartition): Let there be three non empty itemsets A, B and C. It is said that $\{B, C\}$ is a bipartition of A if and only if $B \cap C = \emptyset$ and $B \cup C = A$. An itemset A may have multiple bipartitions. The set of all these bipartitions is denoted as $bipart(A)$.

Example 5: Consider that $A = \{a, b, c\}$. The set of all bipartitions of A is $bipart(A) = \{\{\{a, b\}, \{c\}\}, \{\{a, c\}, \{b\}\}, \{\{b, c\}, \{a\}\}\}$.

Definition 8 (Productive Itemset): An itemset A is called a productive itemset in a database D if it contains more than one item and $P(A \subseteq R) > \max_{\{B,C\} \in bipart(A)} P(B \subseteq R) \times P(C \subseteq R)$, where the probability that B is drawn from the same distribution as D is denoted as $P(A \subseteq R)$.

Example 6: Consider the *Mushroom* database, often used in frequent itemset mining. This database contains data about different mushroom species and their attributes. The database is available at `www.philippe-fournier-viger.com/spmf/`. Consider the item a = *stalk color above the ring is white* and the item b = *stalk color below the ring is white*. The itemset $\{a, b\}$ has a support of $sup(\{a, b\}) = 3688$. The itemset $\{a\}$ has a support of $sup(\{a\}) = 4640$. The itemset $\{b\}$ has a support of $sup(\{b\}) = 4744$. Thus, the *p-value* of the itemset $\{a, b\}$ is 0.137891. Hence, this itemset is productive if we consider a maximum *p-value* of 0.05.

A productive itemset is considered as interesting because each of its bi-partitions contributes to its support. For instance, if the itemset $\{high\_sugar\_consumption, diabetes\}$ is found to be productive, this indicates that the support (probability) of having a high sugar consumption and diabetes is greater than if we assume that they are independent. Now, consider an itemset $Y = \{high\_sugar\_consumption, diabetes, right\_handed\}$ indicating that a right handed individual having a high sugar consumption has diabetes. That itemset Y may be frequent in a population but it may not be productive since the bipartition $\{\{high\_sugar\_consumption, diabetes\}, \{right\_handed\}\}$ does not have a strong correlation with the support of Y. In other words, the support of that bipartition does not explain the support of Y, when assuming independence. By discovering only productive itemsets instead of all frequent itemsets, one can thus eliminate many spurious patterns. To check the statistical significance of itemsets and find productive itemsets, Opus-Miner applies the Fisher exact test. Besides, after mining productive itemsets, the concept of redundancy can be applied to filter redundant itemsets before presenting the result to the user. This allows selecting a smaller set of itemsets called the non redundant productive itemsets [37].

Definition 9: Let there be an itemset Y. This itemset is non redundant if there is no subset $X \subset Y$ such that X and Y have the same support ($sup(X) = sup(Y)$). Otherwise, Y is called redundant.

Example 7: Consider the transaction database D of Table 1. The itemset $\{b, e\}$ is redundant because $\{e\} \subset \{b, e\}$ and $sup(\{b, e\}) = sup(\{e\}) = 2$. The itemset $\{b, c\}$ is non-redundant since none of its proper subsets has the same support, that is $sup(\{b, c\}) = 2$, while $sup(\{b\}) = 5$, $sup(\{c\}) = 3$ and $sup(\emptyset) = 8$.

Eliminating redundant itemsets is useful to filter unecessary information. For example, consider the itemset $\{woman, pregnant, heart\_disease\}$. This itemset obviously has the same support as $\{pregnant, heart\_disease\}$ and hence is redundant. In the FIM literature, that notion of non redundant itemsets has also been studied under the names of *key itemsets* [12] and *generator itemsets*. The motivation for discovering key itemsets is that they are the smallest itemsets that can be used to describe a set of transactions based on the MDL (Minimum Description Length) principle. For instance, in the context of customer behavior analysis, a key itemset may describe the smallest set of products purchased by some group of persons.

Property 1: Let there be a non redundant itemset X. All subsets of X are also non redundant itemsets [12].

Based on the above discussion, it is desirable to find productive itemsets that are non redundant. However, it is not an easy task from a computational perspective. For an itemset Y having r items, checking it is productive and non

redundant requires to calculate the support of each bipartition in $bipart(Y)$. This means that the occurrence frequency of each non empty subset of Y must be computed. If Y has r items, then there are $2^r - 1$ subsets of Y that are non empty. For instance, if we assume that $r = 8$, then the support of $2^8 - 1 = 255$ itemsets needs to checked to determine if Y is productive. A second problem of OPUS-miner is that it is designed to find the top-k productive itemsets in a static database. Because it is a batch algorithm, it cannot be applied to a dynamic database without running the algorithm again from scratch if the database is updated, which is inefficient. Moreover, choosing an appropriate value of k to find interesting patterns is not easy. If k is small, then interesting patterns may be missed, while a large k value can yield too many itemsets and runtime and memory consumption may increase. To verify whether an itemset Y is productive or not, a user hence needs to run the algorithm with a value of k that is large enough so that Y will be among the top-k itemsets. Since Y may not be in the top-k itemsets, a user may have to run the algorithm numerous times, which is cumbersome as it may require a lot of time. The solution to this problem presented in this paper is to design a novel data structure and algorithm for processing queries to check if some itemset(s) are productive and non-redundant.

Definition 10 (Problem Definition): The problem studied in this paper is called the interactive discovery of non-redundant productive itemsets. It consists of answering queries about a dynamic database that have the form "Is an itemset Y non redundant and productive?". The problem should be solved using a data structure that can be efficiently updated by inserting transactions and deleting transactions to cope with the dynamic aspect of the database.

IV. THE DESIGNED APPROACH

The approach presented in this paper for processing queries about non redundant and productive itemsets is named IDPI+. It is composed of three main parts: (1) a variation of the Itemset-Tree structure [19] to compress the dynamic database, which is adapted to support both transaction insertions and deletions, (2) a novel structure called Query Tree is used to gather information related to the support of itemsets to answer queries, and (3) an algorithm that efficiently answer queries by comparing the two aforementioned structure.

An overview of how the IDPI+ approach is used and its internal processes is presented in Figure 1. The first operation that a user can do is to insert transactions to be able to perform queries on these transactions. The IDPI+ approach inserts these transactions into a compressed structure called Memory Efficient Itemset-Tree (MEIT). A MEIT can be updated by deleting transactions or inserting novel transactions. This process is explained in section IV-A. The second operation that a user can do is to submit queries to determine if one or more itemsets are non redundant and productive. The first step to answer queries is to build a novel structure called Query Tree (QT) to store the queries. This process is explained in section IV-B. Then, the next step is to compare the MEIT with the QT to compute the support of X and its subsets. This process is described in section IV-C. Finally, the last step is to use the information about the support of itemsets stored in the QT to determine if an itemset is productive and non redundant using the Fisher test. This process is described in section IV-D. Finally, the result of the queries are shown to the user.

The main algorithm of the proposed IDPI+ approach is shown in Algorithm 1. It takes a transaction database D and a set of queries as input. The output is the result of the queries. Each steps of the algorithm are described in the following subsections.

Algorithm 1 IDPI+

input: D: a transaction database, QD: a set of queries

1 $MEIT$ = buildMEIT(D); `// Build MEIT (if not previously built) - Section IV-A`
2 QT = buildQueryTree(QD); `// Build the Query Tree - Section IV-B`
3 `GetSupportUsingQueryTree` ({$QT.root$},$MEIT.root$); `// Get support of itemsets - Section IV-C`
4 `CheckRedundantAndProductive`($QT.root$); `// - Section IV-D`

A. USING AN ITEMSET-TREE TO COMPRESS THE DATABASE

The approach presented in this paper consists of first storing the input database in an IT to compress it. However, as it is known that an IT can still be quite large, it is desirable to further reduce memory usage. For this reason, this paper relies on an improved structure called MEIT (Memory Efficient IT) [14], designed to remove redundant information from an IT. The IT is a well-known structure in FIM that was proposed to handle the interactive mining of frequent itemsets in a dynamic transaction database. This section first describes the IT structure and how it is constructed. Then, the section presents the MEIT variation of the IT structure, which is deemed more appropriate for the problem of interest. Finally, the section proposes a novel delete operation for the MEIT structure, which allows to update a MEIT by deleting transactions. The delete operation is especially useful in practice to remove old transactions, and thus forget old trends from the data.

1) THE ITEMSET-TREE DATA STRUCTURE

An IT is a tree-based data structure containing a set of nodes. The process of creating an IT for a database D consists of first creating an empty IT and then to insert each transaction of that database into the tree. Each node f of an IT is described using three fields. First, a value $f.it$ stores an itemset Y that is a transaction or the intersection of one or more transactions from D. It is assumed that items in $f.it$ and all other nodes are

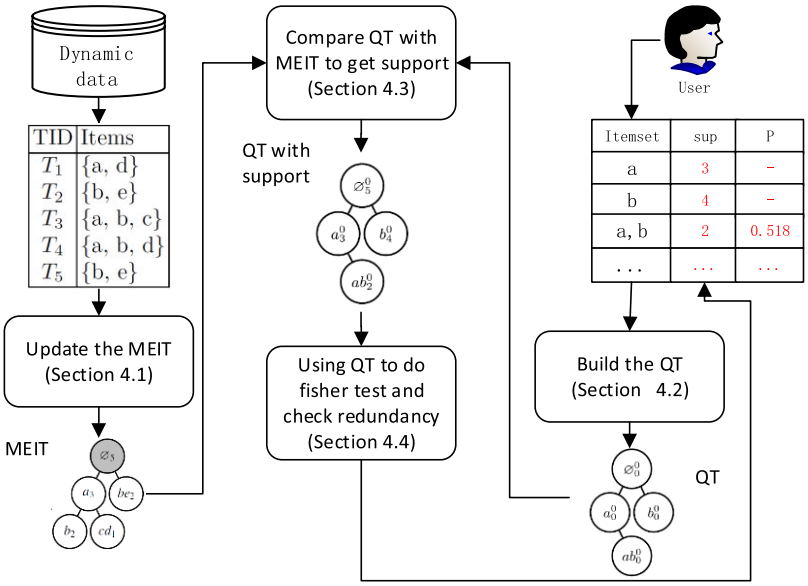

FIGURE 1. Overview of the proposed IDPI+ approach.

sorted according to a total order such as the lexicographical order. Second, a field $f.sup$ stores $sup(Y)$. Third, pointers to child nodes of f are stored in a field $f.childs$. Note that this latter field may be empty if a node has no child. When an IT is first created, it contains only a node called its root, which is denoted as $IT.root$, representing the empty itemset ($IT.root.it = \emptyset$). The total ordering of items in itemsets allow to define a key concept of leading items. Let there be two itemsets $X = \{c_1, c_2, \ldots c_k\}$ and $Y = \{d_1, d_2, \ldots d_l\}$. These itemsets are said to be *sharing the same leading items* if there exists an integer $1 \leq v \leq argmin(\{k, l\})$ such that $c_1 = d_1, c_2 = d_2, \ldots c_v = d_v$.

2) INSERTING A TRANSACTION IN AN ITEMSET-TREE

An itemset-tree is initially empty, only containing a root node. Thereafter, the *Add-Transaction* (Algorithm 7) algorithm is called to insert each transaction of a database D in the itemset-tree. This algorithm was proposed in previous work [19] and is hence only briefly presented in this paper. The algorithm receives as input an itemset-tree IT and a transaction T. The algorithm starts from the root and traverses a branch by recursively calling itself to locate the node that represents the transaction in the tree. At the same time, the *sup* value of each node in that branch is increased by 1. Moreover, if no node is found for the transaction T, a new node is created. For each subtree IT rooted in the visited branch, five cases are considered. In the first case, the transaction T is equal to the itemset stored in the root of IT. This means that the node representing T is found and its count is increased by 1 (line 1 to 2). In the other cases, the algorithm checks each sub-tree ITT that is a children of $IT.root$ and share some leading items with T. In the second case, if there does not exist such sub-tree ITT, a node representing T is inserted as a child of IT with a count of 1 (line 4). In the third case, if $ITT.root.itemset$ is a proper subset of T, it means that the node representing T may be in the subtree ITT. Thus, the algorithm is recursively called to explore ITT (line 5). In the fourth case, if the transaction T is a proper subset of $ITT.root.itemset$, then a new node g representing T is inserted between the nodes $IT.root$ and $ITT.root$. The new node has a support of $ITT.root.sup + 1$ (line 6). In the fifth case, a new node g representing $T \cap ITT.root.itemset$ is inserted between the nodes $IT.root$ and $ITT.root$, with a support of $ITT.root.sup + 1$. Then, a node h is added as a child of g such that $h.sup = 1$ and $h.itemset = T$ (line 7). The process of inserting a transactions is illustrated next with an example.

Algorithm 2 Add-Transaction-IT

input: IT: an itemset-tree, T: a transaction

1 $IT.root.sup \leftarrow 1 + IT.root.sup$;
2 **if** $IT.root.itemset = T$ **then return**;
3 Find a sub-tree ITT of $IT.root.itemset$ such that T and $ITT.root.it$ share some leading items;
4 **if** *there exists no such* ITT **then** Add a node g such that g is a child of $IT.root$, $g.sup = 1$ and $g.itemset = T$;
5 **else if** $ITT.root.itemset \subset T$ **then** `Add-Transaction-IT`(ITT, T);
6 **else if** $T \subset ITT.root.itemset$ **then** Add a node g such that g is a parent of $ITT.root$, g is a child of $IT.root$ $g.sup = ITT.root.sup + 1$ and $g.itemset = T$;
7 **else** Add a node g as a parent of $ITT.root$ such that $g.sup = ITT.root.sup + 1$, $g.itemset = T \cap ITT.root.itemset$. In addition, add a node h such that h is a child of g, $h.sup = 1$ and $h.itemset = T$;

Example 8: Consider the database depicted in Table 1. Figure 2 illustrates the process of creating the corresponding itemset-tree by inserting these transactions, one by one. The tree after inserting transaction $\{a, b\}$ is presented in Figure 2(a). It can be observed that a node was added as child of the root, containing the itemset $\{a, b\}$ and its support value of 1. The tree after inserting the transaction $\{b, e\}$ is depicted in Figure 2(b). Another child node has been added under the root. This node represents the itemset $\{b, e\}$ and indicates that its support is 1. Then, the transaction $\{a, c, d\}$ is inserted. The tree after this update is presented in Figure 2(c). Then, Figure 2(d) shows the tree after inserting transactions $\{a\}$,$\{a, b\}$, $\{a, c\}$ and $\{b, e\}$. Finally, the tree after inserting the last transaction ($\{b, c\}$) is presented in Figure 2(e). The corresponding MEIT for the IT in Figure 2(e) is shown in Figure 2(f).

The cost of this algorithm is linear in terms of time [19]. In particular, to insert a transaction, the algorithm traverses one branch of the tree, which contains no more than z nodes, where z is the number of items in the longest transaction stored in the tree. Moreover, the algorithm increments the counts in the branch by one, and inserts at most one or two nodes.

3) MEMORY EFFICIENT ITEMSET-TREE

The designed IDPI+ approach relies on a variation of the itemset-tree, which is called the memory efficient itemset-tree (MEIT) [14]. It is briefly described in the next paragraphs. Then, the paper further explain how the MEIT structure is improved in IDPI+ by proposing a novel delete operation.

The motivation for proposing the MEIT was to reduce the memory requirement of the itemset-tree. The main differences between these structures are the following. Each node of an itemset-tree is either a transaction or the intersection of some transactions. A problem with this representation is that a lot of memory is wasted because each IT node contains the items of its parent node. For instance, it can be observed in the IT of Figure 2(e) that the IT node of the itemset *acd* contains the items *ab* of its parent, which is redundant. Similarly, the node *ab* contains the item *a* of its parent node. The MEIT addresses this issue by removing the redundancy between a node and its parent by not storing a parent's items in its child nodes. For instance, the MEIT corresponding to the IT of Figure 2(e) is shown in Figure 2(f). In an experimental evaluation on several benchmark databases, it was found that a MEIT can in some cases utilizes 50% less memory than an itemset-tree [14]. After building a MEIT, it can be utilized to quickly determine what is the support of any itemset Y. This information is very useful in the context of this paper since it is used to check if an itemset is non-redundant and productive. Furthermore, another interesting property of a MEIT, is that new transactions can be added in real-time. This allows to use an MEIT for interactive data mining.

a: INSERTING A TRANSACTION IN A MEIT

To insert a transaction in an MEIT, Algorithm 9 is applied. This latter is very similar to the algorithm for adding a transaction to an itemset-tree and has a similar complexity [14].

Algorithm 3 Add-Transaction-MEIT

input: ITN: an itemset-tree node, T: a transaction

1 $TTprefix = \text{suf}(T, ITN.itemset.length)$;
2 $TTsuffix = TT$ -$TTprefix$;
3 $ITN.sup \leftarrow ITN.sup + 1$;
4 **if** $T = ITN.itemset$ **then** exit;
5 Let $ITNC$ be a sub-tree of ITN such that $ITNC.itemset$ and T share some leading items;
6 **if** *there does not exist such node ITNC* **then** Add a child node g to ITN where $g.sup = 1$ and $g.itemset = T$;
7 **else if** $ITNC.itemset \subset T$ **then** $Construct(T, ITNC)$;
8 **else if** $T \subset ITT.root.it$ **then** Create a new node g as a son of $ITT.root$ and a father of $ITT.root$ where $g.itemset = T$ and $g.sup = ITT.root.sup + 1$;
9 **else** Insert a node g such that g is a parent of $ITT.root$, $g.sup = ITT.root.sup + 1$ and $g.itemset = T \cap ITT.root$. In addition, insert a new node h such that $h.sup = 1$, $h.itemset = T$, and h is a child of g.;

b: DELETING A TRANSACTION IN A MEIT

The original IT and MEIT were designed to support transaction insertions. This is useful to update a tree with new transactions. For example, consider transactions made in a retail store. If new transactions are made by customers, they can be added to the tree to learn new trends about customer behavior. However, the IT and MEIT do not provide any algorithm to remove transactions, and thus are unable to forget old transactions. Because of this, the original MEIT and IT may continuously grow over time and as a result their performance can deteriorate. This section addresses this issue by proposing an efficient algorithm to delete transactions from a MEIT. That algorithm can be used to remove old transactions from a MEIT. A benefit of doing this is to forget old trends, while keeping the tree small over time and avoiding rebuilding the tree from scratch to remove transactions, which can be time-consuming.

The proposed algorithm for deleting transactions is named *Delete-Transactions*. The pseudocode is shown in Algorithm 19. It takes as input a transaction to be deleted DT, a MEIT node named ITN (initially set as the root node), the parent node of ITN called $ITNP$ (initially set to *null*), and k, the number of occurrences of DT to be deleted. The parameter k is used because a MEIT can contain the same transaction multiple times. If the parameter k is set to 1, it means to remove one occurrence of DT from the tree. If the parameter k is set to ∞, then all occurrences of DT are removed from the tree. The output of the *Delete-Transactions* algorithm is *num*, the number of occurrences of T that were

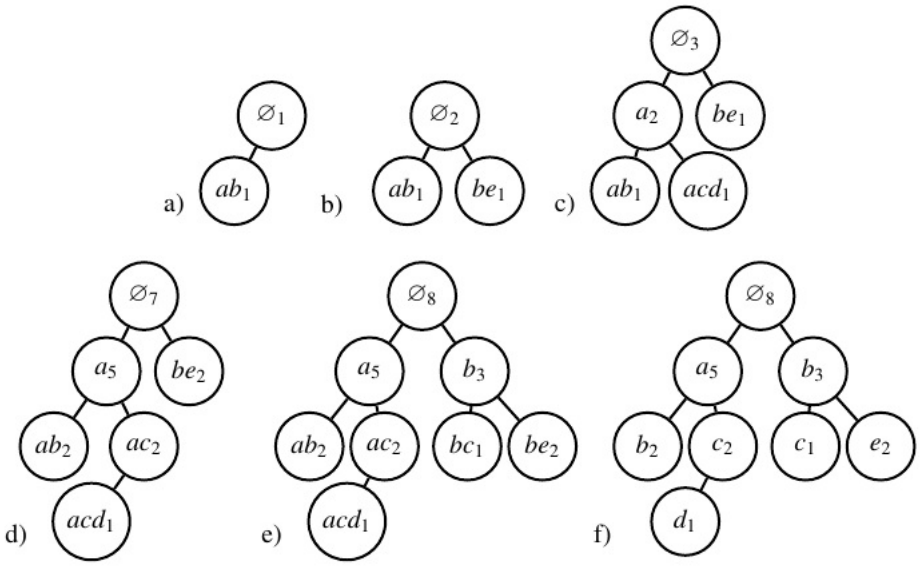

FIGURE 2. Itemset-Tree construction for the database of Table 1.

deleted. The variable *num* is used to handle the case where a tree contains less than k occurrences of the transaction DT to be deleted. For example, assume that DT is $\{a, b, c\}$, its support is $sup(\{a, b, c\}) = 2$, and $k = 4$ (the number of occurrences of DT to be deleted is 4). Then, $num = \min(k, \sup(DT)) = \min(4, 2) = 2$. If $num = 0$ is returned, it means that the tree did not contain the transaction DT and thus that no transaction was deleted. Table 2 provides a summary of the notation used in Algorithm 19. Additionally, a flowchart of the deletion process is shown in Figure 3.

TABLE 2. Notation used in the *Delete-Transactions* algorithm.

Notation	Description
D	Dynamic transaction database
DT	A transaction to be deleted
$DT[i]$	The i_{th} item of transaction DT
ITN	A node of the MEIT tree
$ITN.it$	The itemset of ITN
$ITN.child$	A child node of ITN
$ITN.parent$	The parent node of ITN
$LNOT$	A node that represents the last item of DT
k	A user-defined number of occurrences of DT to be deleted
num	The total number of deleted transactions
True transaction	The transaction exists in the MEIT, i.e. $sup(LNOT) > \sum \sup(LNOT.child)$.
False transaction	The transaction does not exist in the MEIT, i.e. $sup(LNOT) = \sum \sup(LNOT.child)$.

The *Delete-Transactions* algorithm first traverses the tree to identify the node *LNOT* that represents the last item of the transaction DT in the MEIT. Then, the algorithm starts from that node and recursively deletes each item of the transaction by going upward in the tree. To identify *LNOT*, the algorithm starts from *ITN* (initially set to *ITR*) and checks if it matches $DT[i]$, where $(1 \leq i \leq DT.length)$ (e.g. for $DT = \{a, b, c\}, DT[1] = a, DT[2] = b.DT[3] = c$). If yes, childs of *ITN* and its descendants are recursively considered by checking if $ITN = ITN.child$ and i is incremented until *LNOT* is found. Otherwise, if there exits a node *ITN* that does not match $DT[i]$, it means that DT is not in the MEIT and the algorithm returns 0, indicating that no transaction was removed from the MEIT.

Otherwise, the *LNOT* (node that represent the last item of DT) is found. Then, the algorithm checks if the *LNOT* represents a true transaction (Figure 3(C.b)). A *true transaction* is a transaction that exists in the MEIT. For example, assume that $DT = \{b\}$, and consider the tree of Figure 4(g). There are three transactions that exist in the subtree rooted at b: an occurrence of transaction $\{b, c\}$ and two occurrences of transaction $\{b, e\}$. When the algorithm searches for the *LNOT* of $\{b\}$, it will consider that the nodes representing $\{b, c\}$ or $\{b, e\}$ may be the *LNOT* of DT, which is incorrect. The transaction $\{b\}$ does not exist in these subtrees. Thus, the algorithm will return 0 because the *LNOT* represents a *false transaction* (Figure 3(C.b)), i.e. a transaction that does not exist in the tree. Otherwise, if *LNOT* represents a true transaction (Figure 3(3 C.a)), the algorithm calculates the number of transactions that will be deleted from the tree, that is $num = min(k, \sup(DT))$. Then, the algorithm checks if all occurrences of the transaction DT represented by the node LNOT must be deleted. If $k \geq sup(DT)$ then all transactions must be deleted (Figure 3(C.a.a)). Otherwise, only some occurrences of transactions DT are deleted from the MEIT (Figure 3(C.a.b)). In the case where all occurrences of DT are removed (Figure 3(C.a.a)), the algorithm checks if the MEIT must be restructured. There are three cases.

- **Case 1.** If $ITN = LNOT$ has no child and its parent has only one child, the node *ITN* is directly deleted, and *ITN*'s parent will be merged.

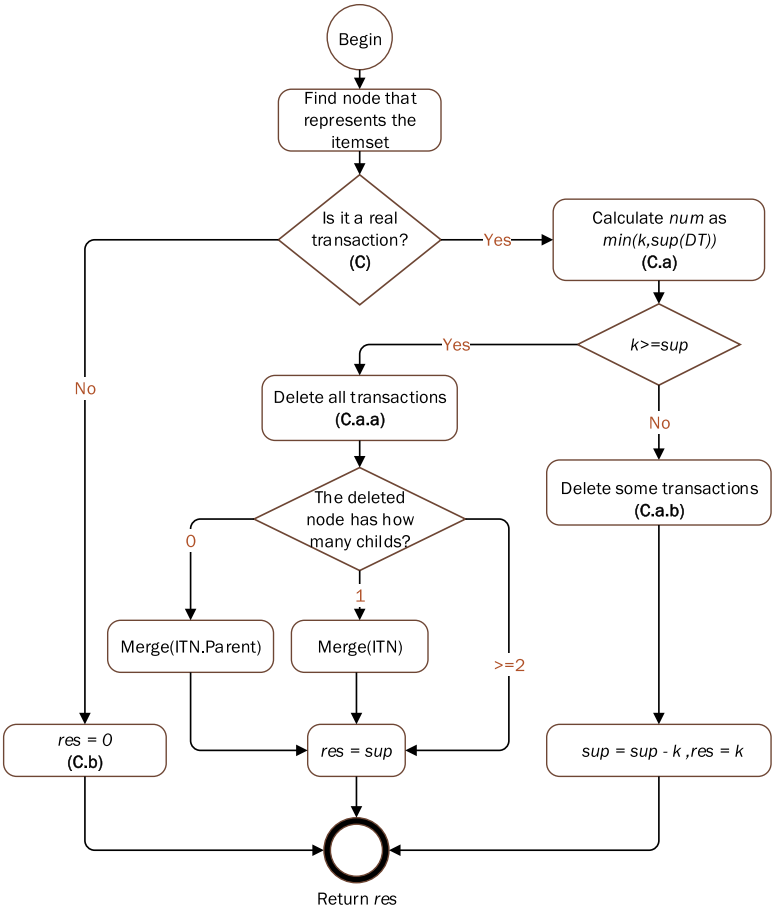

FIGURE 3. Flowchart of the transaction deletion process.

- **Case 2.** If *ITN* only has one child and *ITN* represents a false transaction, then *ITN*.*child* is merged with *ITN*. The itemset of *ITN* becomes ($ITN.it = ITN.it \cup ITN.child.it$). Then, the algorithm checks if *ITN*'s parent needs to be merged.
- **Case 3.** If *ITN* has two or more childs and represents a false transaction, it is not necessary to restructure the tree.

The process of deleting transactions using the *Delete-Transactions* algorithm is illustrated using three examples representing these three cases, respectively.

Example 9 (Case 1): Consider the MEIT of Figure 4(a) and that the *Delete-Transactions* algorithm is called with $DT = \{b, c\}$ and $k = 1$. Then, *ITN* is initially set to be the MEIT's root. The algorithm first calls itself recursively to find the node *LNOT*. This node $ITN = LNOT = \{c_1\}$ is highlighted in gray color in Figure 4(a). Then, *num* is calculated as $min(sup(ITN) = 1, k = 1) = 1$. This means that all occurrences of transaction *DT* need to be deleted. Thus, *ITN* is removed. The result is shown in Figure 4(b). Thereafter, because *ITN*.*parent* (node b_3) has only one child and *ITN*.*parent* represents a false transaction, the node *ITN*.*parent* is merged with its child *ITN* (node e_2). The itemset of *ITN*.*parent* becomes $ITN.parent.it = (ITN.parent.it \cup ITN.it) = be_3$, as shown in Figure 4(c). The current call to the *Delete-Transactions* algorithm returns $num = 1$ and $ITN = ITN.parent = be_3$. Then, the algorithm updates the support value of ancestor nodes by recursively moving upward from $ITN = ITN.parent$ until reaching the root. For each node *ITN* encountered, the support is updated as $sup(ITN) = sup(ITN) - 1$ because one transaction has been deleted. The tree obtained after deleting the transaction $\{b, c\}$ is shown in Figure 4(d).

Example 10 (Case 3): Consider the MEIT of Figure 4(e) and that the *Delete-Transactions* algorithm is called with $DT = \{a\}$ and $k = 1$. Initially, *ITN* is the MEIT's root. The algorithm first calls itself recursively to find *LNOT*. The node $LNOT = \{a_5\}$ is colored in gray in Figure 4(e). The node *LNOT* represents a true transaction (the transaction *a* exists in the tree). Then, *num* is calculated as $min(sup(DT) = 1, k = 1) = 1$. This means that all occurrences of transaction *DT* need to be deleted. This is done by updating the support of *LNOT* as $sup(LNOT) = sup(LNOT) - 1$. Then, because *LNOT* has more than one child, the MEIT does not need to be restructured. The result after this step is shown in Figure 4(f). Then, the algorithm updates the support value

Algorithm 4 Delete-Transaction

Input : T: a transaction to be deleted, *ITN*: a MEIT node, *ITNP*: the parent node of *ITN*, k: the number of occurrences of T to be deleted
Output: *num*: the number of ocurrences of T that were deleted (e.g. 0 means that no transaction was deleted).

— *Find the node representing the transaction T* —
if $ITN.itemset \not\subset T$ **then return** *0*;
if $ITN.itemset \subset T$ **then**
  $Tsuffix = \text{suf}(T, |ITN.itemset|)$;
  foreach *child ITNC of ITN* **do**
    res = `Delete-Transaction` (*Tsuffix*,*ITNC*,*ITN*,*k*);
    if $res \neq 0$ **then** $ITN.sup = ITN.sup - res$;
    return *res*;
  end
end
childSup = the sum of the *sup* values of *ITN*'s children;
if $ITN.itemset = T$ **then**
  if $childSup = ITN.sup$ **then return** 0;
end
— *Update the node ITN representing transaction T* —
if *(ITN has more than one child)* $\vee(k + childsup < ITN.sup)$ **then**
  $ITN.sup = ITN.sup$ - min(k,$ITN.sup - childsup$);
  return k;
end
else if *ITN has only one child* **then** `Merge` (*ITN*, $ITN.child$);
else
  Remove *ITN* from $ITNP.child.list$;
  if *ITNP has only one child (after the deletion) and* $ITNP.sup = ITNP.child.sup + ITN.sup$ **then** `Merge` (*ITNP*, $ITNP.child$);
end
return $ITN.sup - childSup$;

Algorithm 5 Merge

input: *ITN*: a MEIT node, *ITNC*: the child of *ITN*

$ITN.itemset = ITN.itemset \cup ITNC.itemset$;
$ITN.child.list = ITNC.child.list$;

of ancestor nodes of *LNOT* by recursively moving upward from $ITN = ITN.parent$ until reaching the root. For each node *ITN* encountered, the support is updated as $sup(ITN) = sup(ITN) - 1$ because one transaction has been deleted. The tree obtained after deleting the transaction $\{a\}$ is shown in Figure 4(g).

Example 11 (Case 2): Consider the MEIT of Figure 4(h) and that the *Delete-Transactions* algorithm is applied with $DT = \{a, c\}$ and $k = 1$. Initially, *ITN* is the MEIT's root. The algorithm first calls itself recursively to find the node representing transaction *LNOT*. This node $LNOT = \{c_2\}$ is colored in gray in Figure 4(h). Then, *DT* is deleted and $LNOT = c_1$. Because *LNOT* represents a false transaction and has only one child, the node *LNOT* is merged with its child. This is done by calling the *Merge* procedure (Algorithm 2) with node $LNOT = \{c_2\}$ and its child node $ITNC = \{d_1\}$. This procedure merges the itemset of $LNOT.child$ with that of *LNOT* to obtain $LNOT.it = LNOT.it \cup LNOT.child.it$. As a result, the node $LNOT.child = \{d_1\}$ is merged with $\{c_1\}$ to obtain a node $\{cd_1\}$. Then, the algorithm updates the support value of ancestor nodes of *LNOT* by recursively moving upward from $ITN = ITN.parent$ until reaching the root. For each node *ITN* encountered, the support is updated as $sup(ITN) = sup(ITN) - 1$ because one transaction has been deleted. The tree obtained after deleting the transaction $\{a, c\}$ is shown in Figure 4(j).

c: DELETION COST

The cost of deleting a transaction T from the MEIT using the *Delete-Transactions* algorithm is the cost of finding the node representing the transaction T in the tree, and then to delete the node and/or merge a node in the tree. Finding the node requires to traverse a branch from the tree, which contains no more than z nodes, where z is the number of items in the longest transaction stored in the tree. When traversing a node, the algorithm may search for a child node. This is implemented using a binary search, which has $\mathcal{O}(log(w))$ complexity, where w is the child count. In the worst case, w is equal to the number of transactions but is usually much smaller since transactions often share many items. Then, to update the tree, at most one node is deleted and at most one node is merged by updating the parent node. Then, the counts of all nodes from the current node to the root are updated. The number of updated node is the transaction T's length.

B. REPRESENTING QUERIES USING THE QUERY-TREE STRUCTURE

Another component of the proposed IDPI+ approach is a tree-like structure, named the *Query Tree (QT)*. This structure is introduced to decrease the time for counting the support of itemsets using a MEIT. Consider that one wishes to evaluate a query to determine if an itemset Y is productive and non-redundant. Answering the query requires to find out the support of the itemset Y and all its (non empty) subsets. The traditional approach for deriving the support of itemsets using a MEIT is to perform a distinct query to count the support of each itemset. But that approach is inefficient because each node of the MEIT may be traversed multiple times.

To address this problem, this paper proposes an alternative approach, which consists of first building a QT to store the itemset Y and its subsets. Afterward, the itemsets' support is efficiently calculated by traversing the MEIT only once to

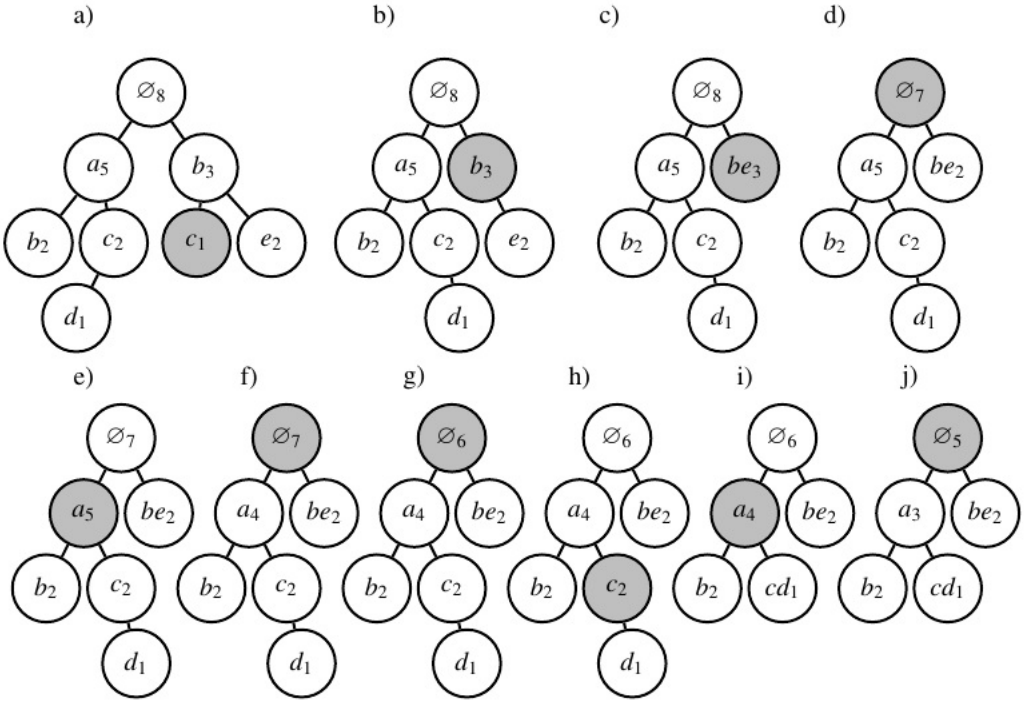

FIGURE 4. Deleting transactions {*b*, *c*} (from a) to d)), {*a*} (from e) to g)) and {*a*, *c*} (from h) to j)) from a MEIT.

compare it with the QT. This process is performed by a query processing algorithm, described in the following section.

The Query Tree structure. A Query Tree is a tree-based data structure, which initially contains only a root node representing the empty set. Then, other nodes can be added. The following fields are used to describe each node h from a QT. First, $h.itemset$ contains an itemset Y. Second, $h.sup$ is used to store its support $sup(Y)$. Third, $h.childs$ contains an ordered list of pointers to the child nodes of h if h has childs. It is important to note that pointers are sorted by a total order $\succ$, which can be any total order such as the alphabetical order. Finally, $h.pos$ contains a position (an integer initialized to zero). The purpose of $h.pos$ will be explained later.

Constructing a Query Tree. The QT of an X is built by adding X and each non-empty subset of X in a QT. This is achieved by using a modified version of Algorithm 7 for each non empty subset of X. In the modified algorithm, the support field of each node is not updated (it remains equal to 0) and the algorithm sorts the child nodes of each node according to the $\succ$ order. The structure of a QT is different in four aspects from that of an IT: (1) a QT stores itemsets rather than transactions, (2) the *sup* field is used differently, (3) the *childs* field is sorted, and (4) the *pos* field is introduced to match a query to an MEIT for answering queries (the next subsection describes the matching algorithm).

Example 12: For example, Figure 5 shows a QT that is constructed to determine if the itemsets $\{a, b, d\}$ and $\{a, c\}$ are non redundant and productive. In Figure 5, each node g contains an item, and the subscript and superscript in a node indicate $g.sup$ and $g.pos$, respectively.

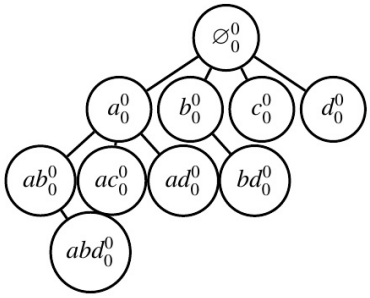

FIGURE 5. The Query Tree of {*a*, *b*, *d*} and {*a*, *c*}.

Construction cost. The cost of constructing a Query Tree for a set of queries is the cost of inserting each itemset from the queries and their subsets into the tree. For inserting an itemset, the cost is similar to that of applying the *Add-Transactions-IT* algorithm (Algorithm 7). A branch of the tree must be traversed to update the *sup* field of nodes, and insert zero, one or two nodes. For each itemset X in a query, $2^{|X|}$ subsets of X are inserted in the QT resulting in the insertion of at most $2^{|X|}$ nodes. However, if multiple itemsets representing different queries are inserted in the Query Tree, some subsets may be shared by multiple itemsets. The nodes representing these subsets are only created once, thus reducing the size of the Query Tree. Moreover, although the number of subsets exponentially increases with the size of an itemset, users typically do not perform queries for very large itemsets, which ensures that the Query Tree remains small. For example, if a user performs a query for an itemset with five items, only $2^5 = 32$ nodes are created in the corresponding Query Tree.

C. CALCULATING THE SUPPORT OF ITEMSETS AND THEIR SUBSETS USING A QUERY-TREE

A Query Tree can be used to verify if an itemset Y is non redundant and productive. The first step to evaluate such query is to create the Query Tree of that itemset, as presented in the previous sub-section. Thereafter, the *GetSupportUsingQueryTree* procedure is invoked to compare the Query Tree with the MEIT to calculate the support of Y and that of its subsets (line 3). The Query Tree stores this information in the Query Tree's nodes.

The procedure *GetSupportUsingQueryTree* is presented in Algorithm 25. It receives as input a MEIT node (initially, the root), and the list of Query Tree nodes QL ordered by $\succ$ (initially, the root node). The procedure compares the QT with the MEIT by performing a depth-first search on the MEIT to update the support values of all itemsets in the QT. In the MEIT, each node is visited at most once. To compare a MEIT node ITN with a QT node QTN, the main challenge is that an itemset stored in a MEIT node is not completely stored (for example, Figure 2(f) left-most node represents the itemset $\{a, b, c\}$ but only $\{c\}$ is stored in that node), while itemsets in QT nodes are fully stored. To explain the difference between these two itemset representations, the concept of *suffix* of an itemset is used (presented in Definition 3). For QTN, the *pos* field indicates that only the items $suf(QTN.itemset, QTN.pos)$ should be compared with the items in ITN. For instance, if $QTN.itemset = \{a, b, c\}$, $QTN.pos = 2$, and $ITN = \{b\}$, it indicates that only items $\{b, c\}$ of $QTN.itemset$ should be compared with ITN. In the following, the notation $QTNsuffix$ denotes $suf(QTN.itemset, QTN.pos)$. When comparing ITN and QTN, five distinct cases are encountered:

Case 1. The first case is that $suf(QTN.itemset, QTN.pos) \subseteq ITN.itemset$, that is the itemset represented by ITN is a superset of $QTN.itemset$. The support of ITN is then added to the support of QTN. Moreover, all child nodes of QTN are inserted into the list QL (while preserving the $\succ$ order) with *pos* equal to the number of items in QTN, so that these nodes will be processed later. Lastly, QTN is removed from QL. **Case 2.** If $ITN.itemset$ and $QTNSuffix$ have items in common such that all other items of $QTNSuffix$ are greater than the largest item in $ITN.itemset$ when considering the $\succ$ order, it indicates that $QTN.itemset$ is not a subset of the itemset represented by ITN but that it may be included in those represented by ITN's childs. In this case, the *pos* value of QTN is stored in a map, and *pos* is incremented by the number of items that QT and ITN have in common. **Case 3.** The third case is that an item i of $QTNsuffix$ does not appear in $ITN.itemset$ and i is smaller than the last item in $ITN.itemset$ when considering the $\succ$ order. This case indicates that QTN is not a subset of the itemset represented by ITN and those represented by its childs. Hence, QTN is removed from the QL list.

Case 4. If the first item in $QTN.itemset$ is greater than the last item in ITN according to the $\succ$ order, it means that the itemsets represented by QTN and its siblings in QL may be included in those represented by ITN and its childs. In this case, QTN and its siblings must remain in QL to be processed next when considering ITN's childs.

Case 5. Otherwise, it is necessary to compare the siblings that succeed to QTN according to the $\succ$ order, with ITN.

Example 13: We illustrate how to calculate the support of itemsets using a QT in this example. Consider the itemset $\{a, b\}$. Figure 6 shows how the QT of that itemset is updated by traversing the MEIT with a depth-first search. Five steps (A), B), C), D), E)) are used to illustrate the comparison of the QT with the MEIT's five nodes respectively. The content of the QL and QT before the comparison is shown in the first and second lines respectively, whereas, the MEIT is shown in the third line. The node colored in light gray is the current node ITN that is used for comparison. Now initially (Figure 6(a)), all QT values are set to zero. The list QL only contains the root of the Query Tree that represents the empty itemset. This current node (QTN) is compared with the root of the MEIT (ITN). Because it is found that $QTN.itemset \subseteq ITN.itemset$ ($\varnothing \subseteq \varnothing$), the procedure to handle case 1 is applied. Hence, the support of ITN is added to that of QTN. Moreover, QTN's childs are inserted in QL with *pos* equal to the number of items in QTN, and QTN is removed from QL. QL now contains two nodes: a_0^0 and b_0^0. The QL next node a_0^0 is considered as QTN. The reason is that the first item of $\{a\}$ is greater than the last item of ITN. For this reason, case 4 is used, where the main loop is stopped. The states of QL and QT after this step are shown in Figure 6(b). Next, *GetSupportUsingQueryTree* is recursively called for comparing nodes in QL with the first child of ITN. This makes a_3 the node ITN. This QL current node, $QTN = a_0^0$, is compared with ITN. Since $QTN.itemset \subseteq ITN.itemset$ ($\{a\} \subseteq \{a\}$), the procedure to handle case 1 is applied again. Therefore, ITN's support is added to that of QTN. Moreover, QTN childs are inserted in QL with *pos* equal to the number of items in QTN, and QTN is removed from QL. Now QL contains two nodes: ab_0^1 and b_0^0. Then, QL's next node ab_0^1 is considered as QTN. Case 5 is used to process the QL's next node b_0^0. As the first item of $\{b\}$ is greater than the last item of $ITN = \{a\}$, case 4 is used, where the main loop is stopped. The current states of QL and QT are depicted in Figure 6(c). This depth-first search process is repeated for other nodes of the MEIT. The final QT is shown in Figure 6(f). It is found in the final tree that the support of itemsets $\{a\}$, $\{b\}$ and $\{a, b\}$ are 3, 4, and 2, respectively. This makes the itemset $\{a, b\}$ non redundant. Now the Fisher test can be used with these support values to examine whether the itemset $\{a, b\}$ is productive or not. This step is not explained in this example because the considered database is too small to check if an itemset is productive.

1) CORRECTNESS

The *GetSupportUsingQueryTree* procedure calculates the support of all itemsets stored in a QT. To show that this procedure is correct, we must show that the support of

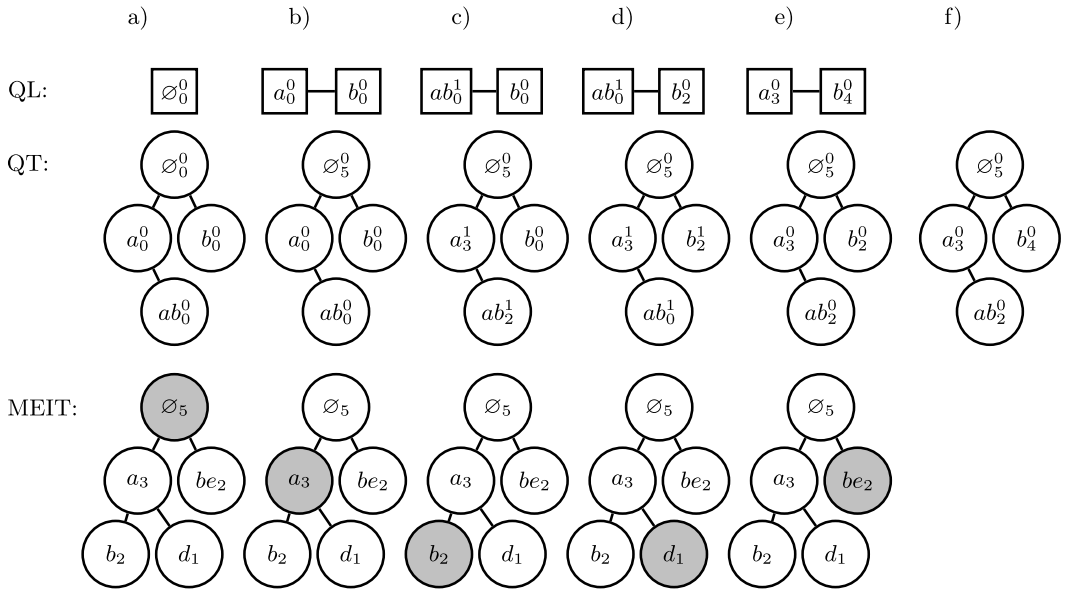

FIGURE 6. Updating the query tree of {*a*, *b*} by traversing the MEIT.

each itemset is calculated correctly. Initially, the support of all itemsets in the QT are initialized to zero. Then, each itemset (QT Node) is compared with the database transactions to calculate its support. The transactions are stored in a compact form in the MEIT. The naive approach would be to compare each QT node with each transaction of the MEIT. But this would be inefficient because an itemset may only appear in a few transactions. Hence, it would result in performing many unnecessary comparisons. The *GetSupportUsingQueryTree* procedure is thus designed to avoid comparing a transaction with an itemset if it does not contain the itemset, when possible. The procedure performs a depth-first search on the MEIT to use each MEIT node once to update the support of itemsets in the QT. Each MEIT node represents a database transaction or the intersection of some transactions.

To make sure that the support of an itemset is correctly calculated, we must show that the itemset is not compared with a transaction (or MEIT node) more than once to update its support (*condition1*), and that it is compared with all the transactions (or MEIT nodes) containing the itemset (*condition2*). The next paragraphs show that this is true for the five cases of the algorithm.

In Case 1, when a QT node is found to be contained in the current MEIT node, the current QT node is replaced by its childs in the QueryList. The reason for doing this is to ensure that all transactions represented by the MEIT node are compared with all itemsets appearing in those transactions. Thus, for a QT node, representing an itemset X, if X is included in the transactions represented by the MEIT node, then its supersets will be added to the QueryList, and will also be compared with that node next (*condition1*). This also does not influence *condition2*.

In Case 3, it is found that the query itemset contains an item that is not contained in the transactions represented by the current MEIT node, and that the item is smaller than the last item in the current MEIT node. This means that the MEIT node and its childs do not contain the current query itemset. Thus, the query itemset does not need to be compared with these nodes. For this reason, the query itemset is deleted from the QueryList to reduce the number of comparisons, and this does not influence *condition1* and *condition2*.

In Case 4, it is found that the first item of a query itemset is greater than the last item of the itemset represented by the current MEIT node. This means that the query itemset is not included in the itemset represented by the MEIT node. Moreover, all itemsets following the query itemset in the QueryList are also not included in the itemset represented by the MEIT node. For this reason, all these itemsets do not need to be compared with the MEIT node. But the supersets (childs) of the MEIT node may contain these itemsets. For this reason, the algorithm only skips the comparisons with the current MEIT node. This does not influence *condition1* and *condition2*.

In Case 2 and Case 5, the algorithm does not perform any operations that influence how itemsets of the QT are compared with MEIT nodes. In particular, no itemsets are added or deleted from the QueryList. Thus, Case 2 and Case 5 do not influence *condition1* and *condition2*.

2) SPACE COMPLEXITY

The space complexity of this algorithm is analyzed as follows. Three data structures are used: the MEIT, the QT and the QueryList. Let N be the number of nodes in the MEIT. To build the MEIT, transactions are inserted one by one. For each insertion, at most two nodes are created in the MEIT. Thus,

Algorithm 6 GetSupportUsingQueryTree

input: QL: a QT node list (initially containing only the QT root node), ITN: a MEIT node (initially the root node)

```
1  if QL is empty then exit;
2  QTNsuffix = suf(QTN.itemset, QTN.pos);
3  foreach QTN ∈ QL do
4      if QTNsuffix ⊆ ITN.itemset                    // Case 1
5      then
6          QTN.sup += ITN.sup; foreach child QTNC of
           QTN do
7              QL.add(QTNC);                         // while
               preserving the ≻ order in QL
8              tjmap[QTNC] = QTNC.pos;
9              QTNC.pos = |QTN.itemset|;
10         end
11         QL.delete(QTN);
12     end
13     else if QTNsuffix ∩ ITN.itemset ≠ ∅ ∧ ∀i ∈
       QTNsuffix \ ITN.itemset, i > ITN.itemset.last
       // Case 2
14     then
15         tjmap[QTNC.itemset] = QTNC.pos;
16         QTN.pos += |QTNsuffix ∩ ITN.itemset|;
17     end
18     else if
       ∃i ∈ QTNsuffix \ ITN.itemset ∧ i < ITN.itemset.last
       then
19         QL.delete(QTN);                           // Case 3
20     end
21     else if QTN.itemset.first > ITN.itemset.last then
       break;                                        // Case 4
22     else continue;                                // Case 5
23 end
24 foreach ITNC ∈ ITN do
   GetSupportUsingQueryTree(QL, ITNC);
25 foreach QTN ∈ tjmap do QTN.pos = tjmap[QTN];
```

the number of node in the MEIT is no more than $2 \times |D|$, and each node stores an itemset and a support count. The size of the QT depends on the number of queries and their size. For a query to determine if an itemset X is productive, $2^{|}X|$ nodes are inserted in the tree, that is one for each subset of X. However, if multiple queries are inserted in the QT, some nodes in the QT may be shared by multiple queries. Thus, an upper-bound on the size of the QT is $Z \times 2^{|}L|$ where L is the maximum query length. In practice, many queries may share the same subsets, as it will be shown in the experiments. Moreover, not all items from the database are inserted in the Query Tree but only those appearing in the queries. The size of the QueryList must be less than the size of the QT, since each QT node cannot appear more than once in the QueryList at any time.

3) TIME COMPLEXITY

The time complexity of the algorithm is analyzed as follows. The algorithm processes each node of the MEIT once. It compares the node with each node in the QueryList in linear time. Let M be the average QueryList length M for each MEIT node that is processed. The time complexity of the algorithm is thus $\mathcal{O}(MN)$.

D. CHECKING IF QUERIED ITEMSETS ARE PRODUCTIVE AND NON-REDUNDANT USING THE QUERY-TREE WITH SUPPORT COUNTS

Let there be a query to check if an itemset X is productive and non redundant. After building the Query Tree (as described in section IV-B) and collecting the support of X and its subsets in the QT (as described in section IV-C), the next step is to check if X is productive and non-redundant. This is performed by calling the algorithm *CheckRedundantAndProductive* with the root of the QT as parameter.

The algorithm *CheckRedundantAndProductive* (Algorithm 7) takes as input a QT node QTN. The algorithm performs a depth-first search on the QT to evaluate if each itemset stored in a QT node is non redundant and productive. To determine if an itemset is non redundant and productive, it is necessary to know its support and that of its subsets. To have this information, the algorithm traverses the tree by considering the childs of each node in reverse $\succ$ order. This ensures that when a node representing an itemset is visited, all the nodes representing its subsets have already been visited.

When visiting a node QTN, the algorithm first saves its support $QTN.sup$ in a map called *MapItemsetSupport* (line 1). Then, the algorithm checks if $QTN.itemset$ is non redundant. An itemset is non redundant if it contains a single item or if it has no subsets having the same support according to *MapItemsetSupport* (line 2). If $QTN.itemset$ is non redundant and contains more than one item, then the itemset may be productive. To verify this, the Fisher test is applied by calling the *Fisher* procedure with $sup(X)$, $sup(Y)$ and $QTN.sup$ for each bipartition $\{X, Y\}$ of $QTN.itemset$. The values of $sup(X)$ and $sup(Y)$ are obtained from *MapItemsetSupport*. The *Fisher* procedure performs the Fisher exact test and is the same as in the Opus-Miner algorithm [37]. For this reason, it is not described here. This procedure returns a p value that is compared with a critical value $\alpha < 0.05$ to determine if the test is significant. As suggested in Opus-Miner [37], the value of α is corrected for multiple testing based on the length of $QTN.itemset$. This value is calculated according to Equation 1 [37], where $s = a_{|QTN.itemset|}$.

$$a_{|s|} = \max_{x \subseteq s} \left(\sum_{i=0}^{\omega} \frac{\binom{\#(x, D)}{\#(x, s \backslash x, D)} \binom{\#(\neg x, D)}{\#(\neg x, s \backslash x, D) + i}}{\binom{n}{\#(s \backslash x, D)}} \right) \tag{1}$$

If the p value is not greater than this α value, then $QTN.itemset$ is a productive itemset and is output (line 3).

Then, a loop is performed to continue the depth-first search for each child of *QTN* (line 4 to 6). Note that only the childs of non redundant itemsets are explored since an itemset cannot be non redundant if it has a redundant subset (by Property 1). When the algorithm terminates all the non redundant and productive itemsets have been output. The time cost of this algorithm is proportional to the number of nodes in the QT since each node in the QT is visited once.

Note that the algorithm can be modified to also output the non productive itemsets and their p values to the user. This is useful to understand why an itemset was considered non productive.

Algorithm 7 CheckRedundantAndProductive

input: *QTN*: a QT node

1 Insert ($QTN.itemset$, $QTN.sup$) in a map *MapItemsetSupport*;
2 **if** $|QTN.itemset| = 1$ $\vee$ $\nexists(X, sup(X)) \in MapItemsetSupport$ *such that* $sup(X) = QTN.sup$ **then**
3 **if** $|QTN.itemset| > 1 \wedge$ `Fisher`$(sup(X), sup(Y), QTN.sup) \leq \alpha_{|QTN.itemset|} \forall \{X, Y\} \in bipart(QTN.itemset)$ **then** Output $QTN.itemset$ as a non redundant productive itemset;
4 **foreach** *child ITNC of QTN in reverse* $\succ$ *order* **do**
5 `CheckRedundantAndProductive`(*INTC*);
6 **end**
7 **end**

V. EXPERIMENTAL EVALUATION

To evaluate the performance of IDPI+, we performed several experiments. Various parameters are varied in the experiments to investigate their influence on the proposed approach's overall performance. Experiments were performed on a computer with a Xeon E3-1270 processor, running Windows 10 and 64 GB of RAM.

Since IDPI+ is the first algorithm for processing queries to determine if itemsets are non redundant and productive, it is not possible to directly compare its performance with prior work. For this reason, a baseline algorithm is used for comparison. Let there be a query to determine if an itemset X is productive and non redundant. The baseline computes the support of X and all its subsets using a standard IT (as presented in Section IV-A). Thus, the baseline performs a distinct query on the IT to calculate the support of each itemset. Differently from the baseline, the proposed IDPI+ approach uses the proposed Query-Tree structure to store X and all its subsets. Then, IDPI+ compute the support of X and all its subsets at the same time by comparing the Query-Tree with a MEIT. Both the baseline and the IDPI+ approaches are implemented in C++.

Experiments were carried on real-life and synthetic datasets commonly used in the association rule mining literature, namely Mushroom, Connect, Accidents, Pumsb, Retail and Chess. They represent the main types of data typically encountered in real-life scenarios (dense, sparse, and long transactions). Let $|D|$, $|I|$ and A represents the number of transactions, distinct items and average transaction length. Datasets' characteristics are presented in Table 3.

TABLE 3. Characteristics of the datasets.

Dataset	D	I	A	Type
Mushroom	8,416	119	23	dense, short transactions
Connect	67,557	129	43	dense with long transactions
Accidents	340,183	468	33	sparse with short transactions
Pumsb	49,046	2113	74	dense, very long transactions
Retail	88,162	16,470	10.3	sparse, many items, short transactions
Chess	3,196	75	37	dense with long transactions

A. COMPARISON OF THE TIME TO PROCESS DIFFERENT NUMBER OF RANDOM QUERIES

The first experiment aims to evaluate the time required by the proposed approach to process queries. The proposed approach and the baseline were applied on each database to process sets of random queries of various size ranging from 500 to 5,000 queries. The time required to process the queries was measured. For this experiment, the number of items per query is randomly selected between two and four. The reason for using itemsets with two to four items is that in real-life, long itemsets may not be useful and are also not likely to be productive because all bipartitions of a productive itemset must be positively correlated. Note that Subsection V-B presents an experiment where the influence of query length on runtime is evaluated with longer queries. Besides, note that itemsets containing a single item cannot be productive and thus are not evaluated in the experiment.

The results are shown in Figure 7. For each dataset, a chart is presented, indicating the time to apply the baseline (denoted as *IT*) and the IDPI+ algorithm (denoted as *IDPI+*). Moreover, to better understand the cost of operations performed by IDPI+, the time required by its two most costly operations is also illustrated. These operations are building a Query Tree (denoted as *BuildQueryTree*) and calculating the support of itemsets using the *GetSupportUsingQueryTree* procedure by comparing a Query Tree with the MEIT (denoted as *GetSup*).

Several observations are made based on these results. First, it can be seen that as the number of queries is increased, the runtime also increases in a more or less linear way, both for IDPI+ and the baseline. Second, it can be seen that IDPI+ is always faster than the baseline. For the Mushroom, Connect, Accidents, Pumsb, Retail and Chess datasets, IDPI+ is up to about 10, 10, 27, 10, 2, and 6 times faster than the baseline, respectively. This shows that the proposed Query Tree structure is useful to reduce the runtime when processing

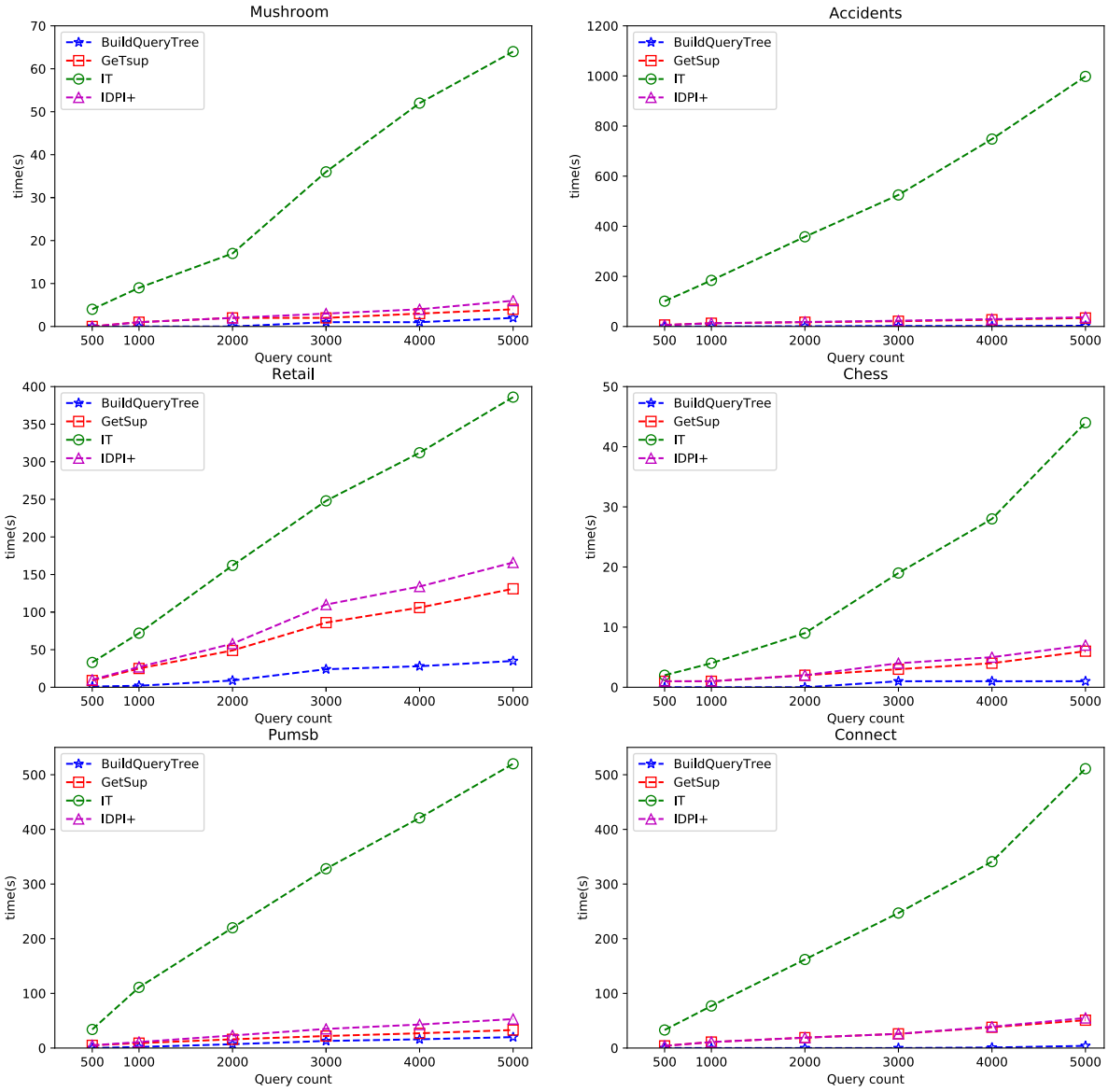

FIGURE 7. Time to process *n* random queries of two to four items.

multiple queries. Third, it can be observed that for some datasets, the time to build query trees is greater than for other datasets. For example, the time required to build query trees on the Retail dataset is longer than on the Chess dataset. The main reason is that for Retail, itemsets inserted in a query tree may not share many subsets because there is many items. Thus, many nodes may need to be created in the Query Tree for each query, resulting in long execution times. On the other hand, for datasets like Chess containing few items, itemsets in queries share many subsets and thus the size of a Query Tree is smaller. This explains why building a Query Tree for a dataset containing fewer different items is faster than for datasets having more items. The influence of the number of items on the time for building query trees will be further investigated in Section V-C using large sets and random synthetic queries. Fourth, it is observed that on a desktop computer, the IDPI+ approach can process about 100 to 1000 queries per second, which is considered as satisfying, as it can support the interactive exploration of patterns by one or more users performing multiple queries.

B. COMPARISON OF THE TIME TO PROCESS RANDOM QUERIES OF DIFFERENT ITEMSET LENGTH

In the previous section, the runtime of the proposed IDPI+ approach was compared with that of the baseline for random queries of two to four items on each real-life dataset. To evaluate the influence of query length on the runtime, two additional sets of experiments were performed using random queries of 5 to 7 items and 8 to 10 items, respectively. As in the previous experiment, the runtime of IDPI+ and the

baseline were recorded for processing n random queries for each dataset, where n was varied from 500 to 5,000 queries.

Results are shown in Figure 8 and Figure 9, respectively for queries of 5 to 7 items, and 8 to 10 items. Moreover, for the convenience of the reader, Table 4 provides a summary of all results from the previous subsection and this section. Overall it is observed that IDPI+ was faster than the baseline IT model for processing random queries with varied items. For example, IDPI+ was up to 11 times faster than the baseline on the Mushroom dataset, 24 times faster on Connect, 14 times faster on Accidents, 3 times faster on Pumsb and 4 times faster on the Chess dataset. For the Retail dataset, the processing time of IDPI+ was up to twice faster than the IT but sometimes the difference was negligible.

From the results of Figure 8 and Figure 9, several observations are made. First, as the size of queries is increased, the cost of answering queries often greatly increases. The reason is that larger itemsets have more subsets, and thus more bi-partitions needs to be checked, which is reasonable for the problem of mining productive and non redundant patterns. Second, it can be observed that generally IDPI+ is faster than the baseline. However for large queries and on the Retail dataset, the baseline approach is faster for queries of 8 to 10 items, while it is not the case for shorter queries. This is considered good since most of the time IDPI+ is faster than the baseline and for real-life applications, very large itemsets are rarely productive (since all their bipartitions must be positively correlated), and less likely to be useful for users. The reason why IDPI+ performs less well on Retail is that it is a sparse dataset with many different items. As a result, it is less likely that two itemsets will share the same subsets in the Query Tree, which thus reduces the performance improvement obtained by IDPI+ compared to the baseline.

The speed of processing queries is generally influenced by (1) the length of itemsets, (2) whether they share subsets, and (3) the database's nature. It is important to point out here that if a QT would be reused multiple times to query the same database (e.g. to perform queries on different days), the query processing time would be reduced as it would not required to rebuilt the QT.

TABLE 4. Time for processing n queries using IDPI+ and the baseline approach.

Dataset	Itemset length	Time for processing n queries(s)											
		IDPI+						Baseline (IT)					
		500	1000	2000	3000	4000	5000	500	1000	2000	3000	4000	5000
Mushroom	2-4	0	1	2	3	4	6	4	9	17	36	52	64
	5-7	1	4	8	14	22	28	14	33	66	121	158	201
	8-10	9	18	35	57	77	96	41	86	194	260	364	475
Connect	2-4	4	11	19	26	39	55	33	77	162	247	341	511
	5-7	26	75	107	150	159	263	137	374	743	1192	2011	3114
	8-10	128	297	698	926	1210	1490	846	1501	3361	5503	6760	8343
Accidents	2-4	6	13	18	23	29	37	101	184	358	525	748	998
	5-7	20	35	59	82	116	161	268	511	1063	1507	2445	3144
	8-10	48	83	160	230	352	531	547	1007	2240	3345	4687	6421
Pumsb	2-4	5	11	23	35	43	53	34	111	220	328	433	580
	5-7	16	35	86	139	174	215	114	262	522	777	981	1205
	8-10	80	168	448	782	965	1204	247	461	922	1398	1735	2126
Retail	2-4	10	27	58	110	134	166	33	72	162	248	312	386
	5-7	35	71	217	434	525	659	73	213	451	704	893	1106
	8-10	120	244	861	1402	2195	3722	241	435	968	1214	1833	2046
Chess	2-4	1	1	2	4	5	7	2	4	9	19	28	44
	5-7	3	7	15	20	16	31	14	29	56	86	107	131
	8-10	26	50	93	132	165	200	77	159	307	450	565	691

C. EVALUATION OF THE TIME TO BUILD A QT

The previous subsections compared the time for processing queries using the proposed IDPI+ algorithm and a baseline algorithm, for various number of queries and query lengths. This subsection presents an experiment where we evaluate the time that IDPI+ spends for building a QT for different number of items and queries. The goal of this experiment is to determine if building the QT is costly and how the performance is influenced by the number of items and queries. In this experiment, queries are generated randomly and are thus not specific to any datasets.

Figure 10 shows the time for building a QT when the number of queries is varied from 5,000 to 40,000, and the number of items is varied from 500 to 5,000. Generally, it is observed that the time for building a QT increases linearly with the number of queries and items. The time to build the QT increases with the number of queries because more itemsets are inserted in the QT and more nodes are added to the QT. It is reasonable that the time increases with the number of items. The reason is that to insert a query in a QT to find productive itemsets, it is required to insert all the subsets of the query to also calculate their support values. If the number of items is increased, it is less likely that two queries will share the same subsets. If two queries share some subsets then the time to build the QT is reduced because nodes representing these subsets are created only once. It is to be noted that if the query tree is not used for finding productive itemsets, then the subsets of each query would not need to be inserted in the QT. In that case, the number of items would not influence the runtime much.

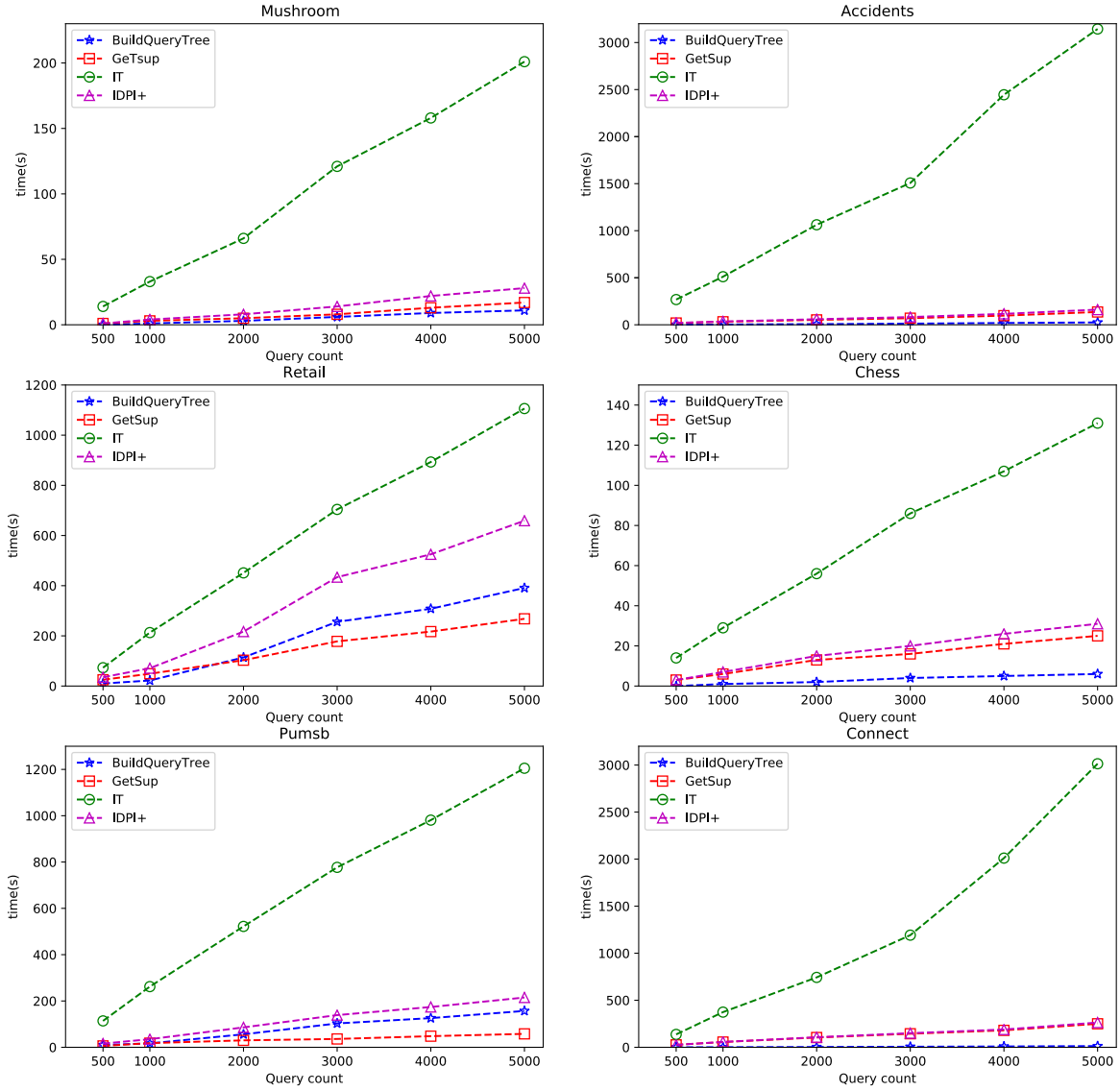

FIGURE 8. Time to process *n* random queries of five to seven items.

D. EVALUATION OF THE TIME TO DELETE A TRANSACTION FROM THE MEIT

Another experiment was performed to evaluate the cost of deleting transactions from the MEIT using the novel transaction deletion algorithm. Deleting transactions is important to be able to perform queries on a database that is frequently updated, as it allows removing old transactions to forget old trends. To evaluate the cost of deleting transactions, the time for inserting all transactions and removing all transactions was measured for each dataset. Table 5 shows the total time for inserting and removing all transactions for each dataset, as well as the average number of transactions removed per second.

It is first observed that removing transactions from the MEIT is very fast, compared to inserting transactions. For the Mushroom, Connect, Accidents, Pumsb, Retail and Chess datasets, deleting transactions is respectively 3.84, 3, 2.22, 2.18, 2.31 and 2.34 times faster than inserting all transactions. The reason is that the delete operation is more simple than the insertion operation, since no new nodes are created. It is also observed that several hundreds of transactions can be deleted per second for each dataset, which can be considered as satisfying performance for applications where a database is frequently updated. Moreover, since the time for deleting transactions is generally much shorter than the time for inserting transactions, the delete operation is useful, as it is faster to remove transactions from a MEIT than to rebuild it from scratch after some updates.

Globally, the experiments described in Section 5 have shown that the proposed IDPI+ approach is more efficient

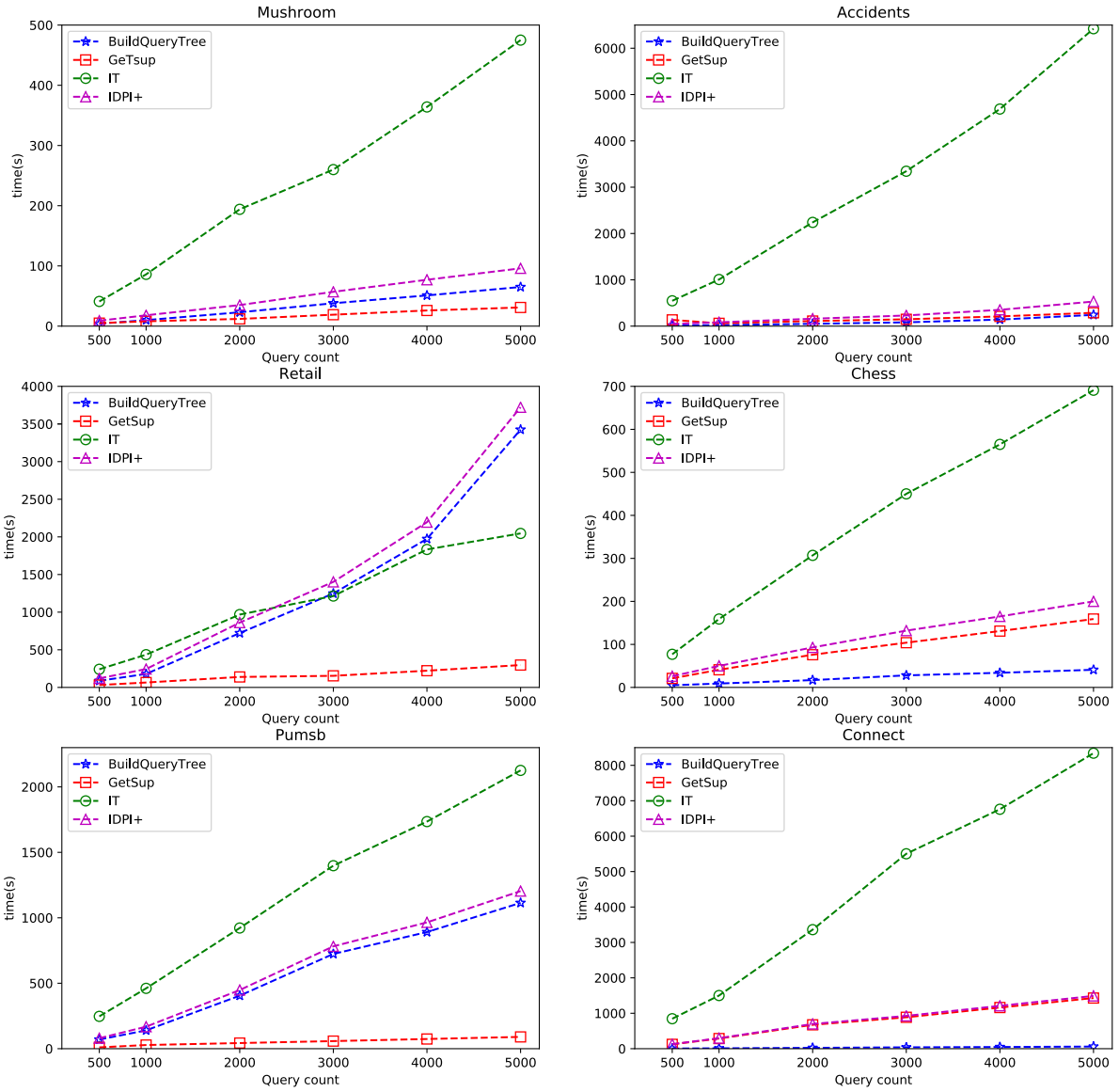

FIGURE 9. Time to process *n* random queries of eigth to ten items.

than the baseline approach for answering queries about productive and non redundant itemsets. It was also shown that the cost of inserting and removing transactions is very small, and thus that the approach is suitable for dynamic databases.

TABLE 5. Time to insert and delete transactions, and average number of transactions removed per second on various datasets.

Dataset	Time insert (ms)	Time remove (ms)	TRPS
Mushroom	254	66	127,515
Connect	2,356	786	85,950
Accidents	16,153	7,262	46,844
Pumsb	2,947	1,346	36,438
Retail	32,991	14,261	6,182
Chess	96	41	77,951

TRPS: Transactions removed per second.

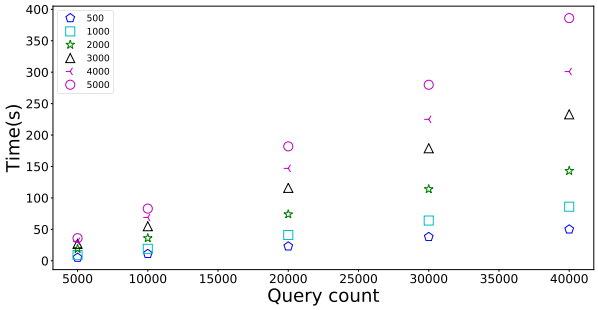

FIGURE 10. Time to build a QT for different number of queries and items.

E. MEMORY COST

Another experiment was performed to specifically evaluate memory consumption. The memory usage of IDPI+ and the

baseline was compared on several real datasets. Results are shown in Table 6 (measured in MB). The column *IT/MEIT node count* indicates the number of node in the IT and MEIT. The column *IT* shows the memory usage of the IT structure, used by the baseline to store transactions. The column *MEIT* shows the memory usage of the MEIT structure, used by the IDPI+ approach to store transactions. Lastly, the columns 5K, 10K, ... 30K depict the memory usage of the Query-Tree structure used by the IDPI+ approach to store various number of random queries.

TABLE 6. Memory usage of IT, MEIT and QT (MB).

Dataset	IC	IT	MEIT	Query Tree Memory			
				5K	10K	20K	30K
CO	390	51.3	39.0	49.58	115.8	245.3	390
MU	119	1.1	0.35	43.85	114.3	242.2	388.1
CH	75	0.27	0.11	35	75.99	167.7	267.3
AC	468	296.5	256.2	138	310.27	677.3	1034
PU	116	21.3	11.2	48.6	107.9	230.6	366.7
RE	16470	656.4	643.3	1052	2790	4676	7013.5

CO:Connect, MU: Mushroom, : CH: Chess, AC: Accidents, PU: Pumsb, RE: Retail, IC: Item count

It is observed that the MEIT can reduce memory usage compared to the IT, which is in accordance with a previous study [14]. This shows the benefit of using the MEIT instead of the IT in the proposed approach. Second, it is observed that using the Query Tree to store multiple queries requires a moderate amount of memory. In some cases, it requires less memory than the MEIT and in some cases more. This is reasonable because the number of queries is very large in this experiment (from 5K to 30K), and all subsets of queried itemsets must be stored in the Query Tree. Also, because queries were randomly generated, itemsets are less likely to share common subsets, which increases the size of the QT.

Overall, memory usage is considered as reasonable since the amount of memory of IDPI+ remains small compared to the memory available on a modern desktop computer, even for large number of queries. Moreover, although additional memory is required to store the Query Tree in the proposed approach, using this structure can greatly decrease the runtime of query processing, as shown in previous experiments.

VI. CONCLUSION

In this paper, the problem of discovering non redundant and productive itemsets in dynamic transaction databases was defined first. IDPI+ approach was proposed to efficiently process targeted queries by users to examine if some patterns are non redundant and productive. In the proposed approach, the MEIT structure and a novel Query Tree structure was used to efficiently process queries. Unlike previous Itemset-Tree structure based approaches, which process queries one by one, the novel Query Tree structure allows to process multiple queries at the same time to improve the query answering performance. Moreover, a novel delete operation has been introduced to remove transactions. Thus, the proposed approach is suitable for real-life dynamic databases where transactions are inserted and removed, to learn new trends and forget old trends appearing in the data. An extensive experimental evaluation was performed. It was shown that the IDPI+ algorithm can be up to 27 times faster than a baseline approach, and that the insert and delete operation are efficient. It can also be used on a modern desktop computers to process up to thousands of queries per second using a reasonable amount of memory.

The solution developed in this work can be seen as a building block to develop a pattern mining software that can find statistically significant patterns in an interactive way. In fact, a user may performs multiple queries and refine queries based on the results of previous queries. Thus, this process can be viewed as interactive.

The proposed IDPI+ approach has several advantages as outlined above but also several limitations that could be addressed in future work. First, time and memory efficiency could still be improved, especially to deal with very large databases. This could require designing a parallel or distributed version of IDPI+ or some alternative algorithms. Second, the query type supported by IDPI+ is useful but limited. It would be interesting to extend IDPI+ to support additional types of queries such as finding all productive itemsets not containing some items. Third, IDPI+ is designed to find positive correlations rather than negative correlations but negative correlations can also reveal interesting information [17]. Thus, adapting IDPI+ for negative correlations or finding negative patterns is an interesting possibility. Fourth, IDPI+ could be extended to handle other statistical tests. Fifth, the type of patterns considered in this paper (itemsets) is simple and not suitable for all applications. For instance, some more complex pattern types such as episodes [6], [7] or sequential patterns could be considered [30] to handle datasets with temporal or sequential information. Besides, variations of the itemset mining problem could be studied to include additional information such as weights or contextual information. Sixth, the algorithm aims to always find an exact solution, which can be costly. To address this issue, designing structures and algorithms that can provide approximate answers with an upper bound on the error in shorter time could be interesting.

In future work, we will aim to address some of these limitations. And in particular, we will focus on developing a user interface to support the interactive exploration of patterns in databases.

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. 20th Intl. Conf. Very Large Databases*, 1994, pp. 487–499.

[2] M. Barsky, S. Kim, T. Weninger, and J. Han, "Mining flipping correlations from large datasets with taxonomies," *Proc. VLDB Endowment*, vol. 5, no. 4, pp. 370–381, Dec. 2011.

[3] J. H. Chang and W. S. Lee, "Finding recent frequent itemsets adaptively over online data streams," in *Proc. 9th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2003, pp. 487–492.

[4] Y. Chi, H. Wang, S. Y. Philip, and R. R. Muntz, "Catch the moment: Maintaining closed frequent itemsets over a data stream sliding window," *Knowl. Inf. Syst.*, vol. 10, no. 3, pp. 265–294, 2006.

[5] Z. Farzanyar, M. Kangavari, and N. Cercone, "Max-FISM: Mining (recently) maximal frequent itemsets over data streams using the sliding window model," *Comput. Math. Appl.*, vol. 64, no. 6, pp. 1706–1718, Sep. 2012.
[6] P. Fournier-Viger, Y. Wang, P. Yang, J. C.-W. Lin, and U. Yun, "TKE: Mining top-k frequent episodes," in *Proc. 33rd Int. Conf. Ind., Eng. Other Appl. Appl. Intell. Syst.*, 2020, pp. 22–25.
[7] P. Fournier-Viger, P. Yang, J. C.-W. Lin, and U. Yun, "HUE-SPAN: Fast high utility episode mining," in *Proc. 14th Int. Conf. Adv. Data Mining Appl.*, 2019, pp. 169–184.
[8] P. Fournier-Viger, C. Cheng, J. C.-W. Lin, U. Yun, and R. U. Kiran, "TKG: Efficient mining of top-k frequent subgraphs," in *Proc. 7th Int. Conf. Big Data Anal.*, 2019, pp. 209–226.
[9] P. Fournier-Viger, J. Li, J. C.-W. Lin, T. T. Chi, and R. U. Kiran, "Mining cost-effective patterns in event logs," *Knowl.-Based Syst.*, vol. 191, Mar. 2019, Art. no. 105241.
[10] P. Fournier-Viger, J. Li, J. C.-W. Lin, and T. T. Chi, "Discovering and visualizing patterns in cost/utility sequences," in *Proc. 21st Int. Conf. Data Warehousing Knowl. Discovery*, 2019, pp. 73–88.
[11] P. Fournier-Viger, J. C. W. Lin, T. Dinh, and H. B. Le, "Mining correlated high-utility itemsets using the bond measure," in *Proc. Int. Conf. Hybrid Artif. Intell. Syst.*, 2016, pp. 53–65.
[12] P. Fournier-Viger, C. W. Wu, and V. S. Tseng, "Novel concise representations of high utility itemsets using generator patterns," in *Proc. Int. Conf. Adv. Data Mining Appl.*, 2014, pp. 30–43.
[13] P. Fournier-Viger, J. C.-W. Lin, B. Vo, T. T. Chi, J. Zhang, and H. B. Le, "A survey of itemset mining," *Wiley Interdiscipl. Rev., Data Mining Knowl. Discovery*, vol. 7, no. 4, Jul. 2017, Art. no. e1207.
[14] P. Fournier-Viger, E. Mwamikazi, T. Gueniche, and U. Faghihi, "Memory efficient itemset tree for targeted association rule mining," in *Proc. 9th Intl. Conf. Adv. Data Mining Appl.*, 2013, pp. 95–106.
[15] L. Geng and H. J. Hamilton, "Interestingness measures for data mining: A survey," *ACM Comput. Surv.*, vol. 38, no. 3, p. 9, Sep. 2006.
[16] 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.
[17] W. Hämäläinen, "Kingfisher: An efficient algorithm for searching for both positive and negative dependency rules with statistical significance measures," *Knowl. Inf. Syst.*, vol. 32, no. 2, pp. 383–414, Aug. 2012.
[18] J.-L. Koh and S. F. Shieh, "An efficient approach for maintaining association rules based on adjusting FP-tree structures," in *Proc. 9th Int. Conf. Database Syst. Adv. Appl.*, 2004, pp. 417–424.
[19] M. Kubat, A. Hafez, V. V. Raghavan, J. R. Lekkala, and W. K. Chen, "Itemset trees for targeted association querying," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 6, pp. 1522–1534, Nov. 2003.
[20] J. Lavergne, R. Benton, and V. V. Raghavan, "Min-max itemset trees for dense and categorical datasets," in *Proc. 20th Int. Symp. Methodologies Intell. Syst.*, 2012, pp. 51–60.
[21] Y. Li and M. Kubat, "Searching for high-support itemsets in itemset trees," *Intell. Data Anal.*, vol. 10, no. 2, pp. 105–120, Apr. 2006.
[22] C.-W. Lin, T.-P. Hong, and W.-H. Lu, "The pre-FUFP algorithm for incremental mining," *Expert Syst. Appl.*, vol. 36, no. 5, pp. 9498–9505, Jul. 2009.
[23] C. K.-S. Leung, Q. I. Khan, Z. Li, and T. Hoque, "CanTree: A canonical-order tree for incremental frequent-pattern mining," *Knowl. Inf. Syst.*, vol. 11, no. 3, pp. 287–311, Apr. 2007.
[24] F. Llinares-López, M. Sugiyama, L. Papaxanthos, and K. Borgwardt, "Fast and memory-efficient significant pattern mining via permutation testing," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2015, pp. 725–734.
[25] J. H. McDonald, *Handbook of Biological Statistics*, 3rd ed. Baltimore, MD, USA: Sparky House, 2014.
[26] B. Nath, D. K. Bhattacharyya, and A. Ghosh, "Incremental association rule mining: A survey," *Wiley Interdiscipl. Rev., Data Mining Knowl. Discovery*, vol. 3, no. 3, pp. 157–169, May 2013.
[27] V. M. Nofong, "Discovering productive periodic frequent patterns in transactional databases," *Ann. Data Sci.*, vol. 3, no. 3, pp. 235–249, Sep. 2016.
[28] W. Ismail, M. M. Hassan, and G. Fortino, "Productive-associated periodic high-utility itemsets mining," in *Proc. IEEE 14th Int. Conf. Netw., Sens. Control (ICNSC)*, May 2017, pp. 637–642.
[29] E. R. Omiecinski, "Alternative interest measures for mining associations in databases," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 1, pp. 57–69, Jan. 2003.
[30] F. Petitjean, T. Li, N. Tatti, and G. I. Webb, "Skopus: Mining top-k sequential patterns under leverage," *Data Mining Knowl. Discovery*, vol. 30, no. 5, pp. 1086–1111, Sep. 2016.
[31] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-mine: Hyper-structure mining of frequent patterns in large databases," in *Proc. IEEE Int. Conf. Data Mining*, 2001, pp. 441–448.
[32] S. J. Shin, D. S. Lee, and W. S. Lee, "CP-tree: An adaptive synopsis structure for compressing frequent itemsets over online data streams," *Inf. Sci.*, vol. 278, pp. 559–576, Sep. 2014.
[33] A. Soulet, C. Raïssi, M. Plantevit, and B. Cremilleux, "Mining dominant patterns in the sky," in *Proc. IEEE 11th Int. Conf. Data Mining*, Dec. 2011, pp. 655–664.
[34] J. Sun, Y. Xun, J. Zhang, and J. Li, "Incremental frequent itemsets mining with FCFP tree," *IEEE Access*, vol. 7, pp. 136511–136524, 2019.
[35] L. Tang, L. Zhang, P. Luo, and M. Wang, "Incorporating occupancy into frequent pattern mining for high quality pattern recommendation," in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2012, pp. 75–84.
[36] T. Uno, M. Kiyomi, H. Arimura, "LCMver.2: Effcient mining algorithms for frequent/closed/maximal itemsets," in *Proc. Workshop Frequent Itemset Mining Implement. (ICDM)*, 2004, pp. 1–11.
[37] G. I. Webb and J. Vreeken, "Efficient discovery of the most interesting associations," *ACM Trans. Knowl. Discovery from Data*, vol. 8, no. 3, pp. 15:1–15:31, Jun. 2014.
[38] H. Xiong, P.-N. Tan, and V. Kumar, "Mining strong affinity association patterns in data sets with skewed support distribution," in *Proc. 3rd IEEE Int. Conf. Data Mining*, Nov. 2003, pp. 387–394.
[39] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 3, pp. 372–390, May 2000.

XIANG LI received the M.Sc. degree from the Harbin Institute of Technology, Shenzhen, in 2019. His research interests include data mining, pattern mining, and machine learning.

JIAXUAN LI received the M.Sc. degree from the Harbin Institute of Technology, Shenzhen, in 2019. Her research interests include data mining, pattern mining, and machine learning.

PHILIPPE FOURNIER-VIGER received the Ph.D. degree. He is currently a Full Professor with the Harbin Institute of Technology, Shenzhen, China. He is also the Founder with the Popular SPMF Open-Source Data Mining Library, which provides more than 170 algorithms for identifying various types of patterns in data. His SPMF software has been used in more than 800 articles for many applications from chemistry, smartphone usage analysis restaurant recommendation to malware detection, since 2010. He has published more than 280 research papers in refereed international conferences and journals, which received more than 6000 citations. He was a Co-Organizer with the Utility Driven Mining and Learning Workshop, KDD in 2018, and ICDM in 2019 and 2020. His research interests include data mining, frequent pattern mining, sequence analysis and prediction, and big data and applications. He received the Title of National Talent from the National Science Foundation of China. He is the Editor of the Book *High Utility Pattern Mining: Theory, Algorithms and Applications* (Springer), in 2019.

Dr. Fournier-Viger is an Associate Editor-in-Chief of the Applied Intelligence journal.

M. SAQIB NAWAZ received the B.S. degree in computer systems engineering from the University of Engineering and Technology, Peshawar, Pakistan, in 2011, the M.S. degree in computer science from the University of Sargodha, Pakistan, in 2014, and the Ph.D. degree from Peking University, Beijing, China, in 2019. He is currently a Postdoctoral Fellow with the Harbin Institute of Technology, Shenzhen, China. His research interests include formal methods (theorem proving and model checkers), evolutionary computation, use of machine learning, and data in software engineering.

JIE YAO received the Ph.D. degree in educational psychology from The Ohio State University. She is currently an Associate Professor with the Harbin Institute of Technology, Shenzhen. Her research interests include design psychology and quantitative research methods, digital media and user experience assessment, data science, and statistical models.

JERRY CHUN-WEI LIN (Senior Member, IEEE) received the Ph.D. degree from the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan. He is currently a Full Professor with the Department of Computer Science, Electrical Engineering and Mathematical Sciences, Western Norway University of Applied Sciences, Bergen, Norway. He is also a Project Co-Leader of wellknown SPMF: An Open-Source Data Mining Library, which is a toolkit offering multiple types of data mining algorithms. He has published more than 300 research articles in refereed journals, such as the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, the IEEE TRANSACTIONS ON CYBERNETICS, *ACM TKDD*, *ACM TDS*, and international conferences, such as the IEEE ICDE, the IEEE ICDM, PKDD, and PAKDD. His research interests include data mining, soft computing, artificial intelligence, machine learning, and privacy-preserving and security technologies. He is a Fellow of IET and ACM. He serves as the Editor-in-Chief for the *International Journal of Data Science* and *Pattern Recognition*.