

Article

Recursive Queried Frequent Patterns Algorithm: Determining Frequent Pattern Sets from Database

Ishtiyaq Ahmad Khan ¹, Hsin-Yuan Chen ^{2,*}, Shamneesh Sharma ³ and Chetan Sharma ⁴

¹ Academic Delivery and Student Success, upGrad Education Private Limited, Bangalore 560071, Karnataka, India

² Center for Digital Technology Innovation and Entrepreneurship, Institute of Wenzhou, Zhejiang University, Wenzhou 325000, China

³ Customer Success and Quality Control, byteXL TechEd Private Limited, Hyderabad 500081, Telangana, India; shamneesh.sharma@gmail.com

⁴ PW-Institute of Innovation, PhysicsWallah Limited, Lucknow 226030, Uttar Pradesh, India; chetanshekhu@gmail.com

* Correspondence: hychen0925@gmail.com or l3115@ms34.hinet.net

Abstract

Frequent pattern mining is a fundamental method for Data Mining, applicable in market basket analysis, recommendation systems, and academic analytics. Widely adopted and foundational algorithms such as Apriori and FP-Growth, which represent the standard approaches in frequent pattern mining, face limitations related to candidate set generation and memory usage, especially when applied to extensive relational datasets. This work presents the Recursive Queried Frequent Patterns (RQFP) algorithm, an SQL-based approach that utilizes recursive queries on relational Mining Tables to detect frequent itemsets without the need for explicit candidate development. The algorithm was implemented using a Microsoft SQL Server and demonstrated through a custom-developed C# web application interface. RQFP facilitates easy integration with database systems and enhances result interpretability. Comparative analyses of Apriori and FP-Growth on an academic dataset reveal competitive efficacy, accompanied with diminished memory requirements and enhanced clarity in pattern extraction. The paper further contextualizes RQFP using benchmark datasets from the previous literature and delineates a roadmap for future evaluations in healthcare and retail data. The existing implementation is educational, although the technique demonstrates the potential for scalable, database-native pattern mining.

Keywords: frequent pattern; apriori; recursive queried frequent patterns; data mining; pattern mining



Academic Editors: Leandros Maglaras and Ognjen Arandjelović

Received: 10 May 2025

Revised: 26 July 2025

Accepted: 17 August 2025

Published: 28 August 2025

Citation: Khan, I.A.; Chen, H.-Y.; Sharma, S.; Sharma, C. Recursive Queried Frequent Patterns Algorithm: Determining Frequent Pattern Sets from Database. *Information* **2025**, *16*, 746. <https://doi.org/10.3390/info16090746>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Data Mining is an old concept that dates back to the inception of computing. The idea dates back to the beginning of the 20th century, but became widely known in the 1930s. Data Mining was first employed in 1936 by Alan Turing, who designed a machine capable of performing the calculations that modern computers can perform. Thanks to technology, we have come a long way since then. Data Mining and machine learning are now being used by businesses to boost sales processes and aid investment analysis [1]. This is because data scientists are now widely employed, and companies worldwide seek to realize more ambitious goals using data science than before. Data Mining is analyzing large volumes of data to uncover business intelligence that assists businesses in solving problems,

mitigating risks, and expanding their horizons [2]. Data science is known because it can be compared to looking for valuable information in an extensive database or finding precious minerals in a mountain. There needs to be a tremendous amount of material that needs to be sifted through to find hidden value. Data Mining can answer questions that previously required excessive time to work through manually [3]. Using various statistical methods to examine data allows researchers to see patterns, trends, and connections they may have missed otherwise.

As a consequence of their findings, they may make predictions and utilize that knowledge to improve business performance. As a result, Data Mining is frequently used in many fields, such as business and research, product development, healthcare, and education [4]. However, incorrect Data Mining can put you at a significant disadvantage compared to competitors, as it may generate misleading insights, drive wrong strategic decisions, and cause financial losses through wasted resources and ineffective targeting. Inaccurate mining can also damage customer trust, raise compliance and ethical concerns, create operational inefficiencies, and ultimately weaken a firm's competitive position in the market.

The Frequent Pattern Mining Technique [5] is the central point of research in Data Mining; therefore, this field is overstocked in terms of the literature devoted to this field of study. Agrawal et al. (1993) first proposed frequent pattern mining for market basket analysis in the form of association rule mining [6]. Remarkable progress has been seen in inefficient and scalable algorithms in frequent pattern Data Mining, associated with frequent itemset mining techniques such as correlation mining, associative classification mining, sequential patterns mining, and frequent pattern-based clustering [7]. This research is limited to their functioning and elaborates on their broad applications. When the occurrence of patterns becomes more than the threshold value set by the user, they are termed frequent patterns. The frequency with which the items often occurred makes them a common itemset in the transaction data. If we take an example of a data store, milk and bread are a customary or ordinary itemset, as they are often bought together. The other sample which makes the recurring structural database more relevant in the field of graph databases can be taken from the ledger of a PC (personal computer) shop. The sequential pattern a user follows in buying a camera after buying a PC, and then buying a memory card, is the best example of a recurring structural database. Various concepts like subgraphs, subtrees, or sublattices may be deliberated as substructures accompanying the itemset. While working with the graph database, a recurring structural trend can be noticed.

The need to inspect the associations, correlations, and other captivating data communications arises by identifying traditional data patterns, categorizing them, and indexing them [8]. Accordingly, the field of frequent pattern mining has developed as a focal point of research in Data Mining. The research in this field has extended the boundaries of data analysis and created an enduring impression on various Data Mining tools, techniques, and applications. As a result, many complex research issues have been solved. In addition, many application-based roadblocks have been overcome to reach the field beyond the point of disillusionment and move into the plateau of productivity on the hype-cycle curve of technology. In contrast to conventional algorithms like FP-Growth or ECLAT, which depend significantly on memory-resident tree structures or depth-first traversal methods, the suggested RQFP algorithm employs an innovative database-native methodology. It incorporates SQL-based recursion and a mining-specific tabular structure (Mining Table) that facilitates dynamic pattern extraction directly from the database. This method diminishes dependence on preprocessing and memory-demanding structures, rendering it appropriate for extensive and real-time transactional systems.

Pattern Mining—A Road Map

Users' demands and applications for a wide range of data and patterns have led to the development of a wide range of Data Mining techniques [9]. Given the wealth of information available in this sector, it is critical to create a road map to assist us in going through it all and picking the best pattern mining techniques, as shown in Figure 1 [10].

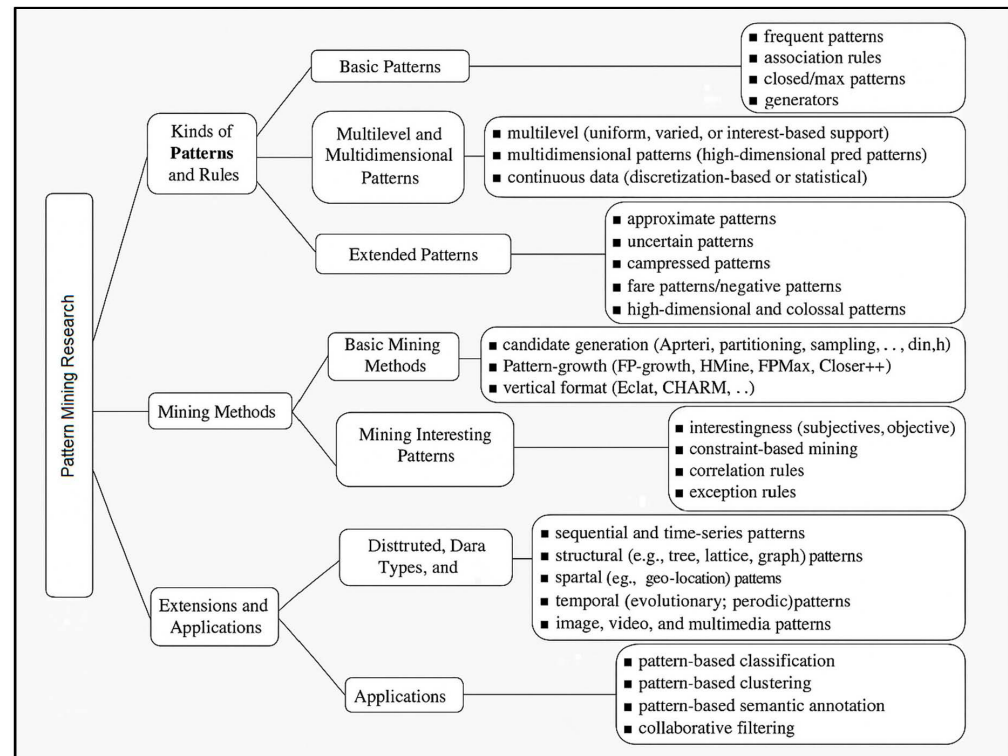


Figure 1. Road map for pattern mining [10].

The basic terms used in frequent pattern mining are:

- Item: An item is a single article in the dataset or the content of a data cell of the dataset.
- Transaction: A transaction is a set of items, usually a dataset row.
- K-Item Set: A K-Item Set is the itemset of size k, i.e., the set contains k items.
- Support: The support is the number of transactions containing an itemset (XUY).

$$\text{Support} = \frac{(X \cup Y).count}{n}$$

- Frequent Set: This is the set of items satisfying the minimum support threshold value.
- Confidence: The confidence of a rule is defined as follows:

$$\text{Confidence}(X \rightarrow Y) = \frac{\text{Support}(X \cup Y)}{\text{Support}(X)}$$

Typically, pattern mining research focuses on three things: the types of patterns mined, mining methods, and application use. Others combine a variety of factors; for example, different applications may need the extraction of data in various ways, necessitating the development of new mining methods.

The key contributions of this study are as follows:

- The proposal of RQFP, a novel SQL-based frequent pattern mining algorithm that utilizes recursive queries over a Mining Table to avoid candidate generation and reduce memory overhead.

- The implementation of RQFP as a visual web-based tool for step-by-step algorithmic understanding, especially in educational and instructional contexts.
- A comparative performance analysis with Apriori and FP-Growth using an academic dataset and literature-based benchmarks, with a roadmap for direct benchmarking against vertical- and closed-itemset algorithms.

2. Related Work

We have had various algorithms that deal with frequent pattern mining; some algorithm types are briefly discussed with their advantages and limitations.

2.1. Apriori-Based Algorithms

According to [11], Apriori was the first algorithm to find frequently occurring itemsets in a data collection. As a starting point, Apriori performs a breadth-first search (BFS). Apriori takes advantage of the downward closure feature at every stage [12]. Candidates are generated using this approach. The first candidate to be produced is one whose superset is either a rare or most frequent pattern (MFP) [13]. This candidate can or cannot be MFP. Candidate itemsets have their database support counted during the database scan. Candidate sets C_k are produced from standard K -Item Set $(k - 1)$ combinations, which causes Apriori to be less efficient and more expensive because of the high number of candidate generations it must perform and necessitates many database searches [14]. Dynamic itemset counting (DIC) is utilized to address the drawbacks of Apriori Dynamic Itemset Counting (ADIC) [15]. The number of data scans needed by Apriori-based algorithms was significantly decreased by employing such techniques.

2.2. Partition-Based Algorithms

Some researchers have directed their focus on the correlation between frequent pattern mining and partition-based algorithms. Although Apriori-based algorithms are ubiquitous in Data Mining these days, their shortcoming of inefficient handling of a high number of database searches has caused researchers to instead move towards partition-based algorithms [16]. The partition-based approach uses the double search method to uncover the standard itemsets [17]. Using a divide-and-conquer strategy, the partition-based process divides the more extensive database into smaller ones, and each sub-database is kept in the system's memory. These algorithms are based on four simple steps. The first step deals with generating the database's partitions (P), and the second step involves the computation of two bounds, i.e., P_Lower and P_Upper , for each section. The third part of the algorithm identifies the candidate partitions, which contain the outliers. In the last step, outlier points are computed regarding candidate partitions. However, the partition-based approach can also be inaccurate in some cases. For example, once the database is alienated haphazardly among the sections, this approach might stipulate a specious astronomical candidate from the smaller database partitions. In worst-case scenarios, when itemsets are pruned and cannot be used often, scans are diminished to $(2k - 1)/k$.

2.3. Depth-First Search (DFS) and Hybrid Algorithms

Eclat and Clique [18] combine depth-first search (DFS) with intersection counting to provide a more efficient algorithm. This method does not require a complex data structure. When using these hybrid methods, which decrease the amount of memory needed, only the transactional item datasets (TID) for the route itemsets from the root to the leaves must be maintained in the memory simultaneously, reducing the amount of memory required [19]. When the remaining length of the shortest TID set is less than the necessary support minus the counted support value, the TID set intersection is complete. It is costly to intersect TID sets of a single itemset to create frequent sets of two objects. Maximal hypergraph clique

clustering is used for two frequently occurring itemsets to produce a more precise collection of maximum itemsets. Ref. [20] proposes a BFS/DFS hybrid method. When you possess only a few standard itemsets, it is much less expensive to utilize BFS's itemset counting to find the supports than the traditional method. In cases when the number of potential frequent itemsets is high, the hybrid method favors TID set intersection with DFS over occurrence counting since the former is more efficient. Consequently, the cost of creating TID sets has increased significantly to make things as simple as possible; a hash tree-like structure is suggested [21].

2.4. Incremental Update with Apriori-Based Algorithms

When working with an extensive dataset, minor changes do not have much of an impact. While transactions are being added, a complete Data Mining procedure is usually impossible. Incremental Data Mining techniques can decrease computational and I/O costs to the absolute minimum [22]. Transactions may be added and deleted using the total mining technique Fast Update 2 (FUP2) [23]. The primary aim of FUP2 is to reduce the cost of generating frequent candidate itemsets. Incremental datasets are scanned, and their common patterns are compared to the previously existing frequent itemsets in the entire dataset.

As a consequence of the incremental data entry or deletion, itemsets that had been previously calculated as frequent are no longer frequent and are removed. Additions and deletions have impacted the calculated frequency of itemsets; thus, the support for previously computed frequent itemsets is adjusted. These methods remove the requirement to keep checking for new instances of already calculated frequent itemsets. Frequent m -item sets generate new $(m + 1)$ candidate-item sets that are often seen. We search until the list of often added candidate itemsets is empty to ensure that newly added itemsets are common in the dataset. When compared to the conventional Apriori approach, FUP2 offers many advantages. However, numerous database searches are still essential.

A further incremental Apriori-based method is Sliding Window Filtering (SWF) [15]. SWF allows for total mining due to the Partition Algorithm and Apriori. SWF divides the dataset into a multitude of smaller ones. A filtering threshold is applied to each partition to produce candidate frequent 2-item sets during the partition scan. If a particular group of elements turns out to be expected, that set's partition number and frequency are recorded. Subsequent partition scans are carried out using a well-curated collection of information on possibly frequent 2-item groupings gathered over time. Prior candidate frequent 2-item sets are used to condense the scanning frequency of fresh partitions. When the time comes to eliminate the candidates, those who have received less support than a certain level are disqualified. Scan reduction methods are used once the incremental segment has been scanned to generate all subsequent frequent candidate itemsets [24]. Finally, the data is checked again to verify that the collection of objects is consistent. Even though SWF is quicker than Apriori methods, its performance is still dependent on partition size and the data pruning method.

2.5. SQL-Based Algorithms

DBMS should allow for existing queries and built-in processes to make Data Mining an online, concurrent activity. The SETM algorithm was the first to experiment with frequent itemset mining [25]. The Apriori algorithm provided an important new direction for frequent pattern mining (FPM) research and development [11]. The Apriori algorithm's database-coupled variants were thoroughly studied. The performance of SQL-92 implementations was improved by object-relational extensions (SQL-OR), but they remained sluggish. Implementing frequent mining with an FP-tree is more efficient than other

SQL-based techniques [26]. Although the FP-tree is generally acknowledged as compact, building an extensive database using a main memory-based FP-tree is unfeasible [27]. The advantage of using RDBMSs is that they contain buffer management systems intended to alleviate data capacity constraints in user applications.

2.6. Pattern-Growth Algorithms

The two highest costs of Apriori-based algorithms are building a candidate frequent itemset I and performing I/O operations. The I/O difficulties have been addressed; however, the problems with creating the frequent candidate itemset I persist. Apriori methods need the development of approximately $I_2/2$ candidates if the n -item set I_n is standard. In addition, the amount of memory required to store the often-occurring candidate itemset I and the data it contains may be considerable [28]. Jiawei Han, Guozhu Dong and Yiwen Yin have developed the frequent pattern tree (FP-tree) [21]. The FP-Growth technique allows it to extract frequent itemsets from the FP-tree without creating new candidates for the current frequent itemset (I). The prefix tree structure has been extended with the creation of FP-tree. Nodes in a tree are used to hold frequently used objects [22]. An item's frequency in a collection is recorded for each node. The whole path from zero to the last level is arranged in decreasing frequency depending on the object's support value, which implies that the total number of nodes under each parent is more than or equal to the node frequency of 25. In Apriori-based algorithms, the two main expenditures are the creation of a candidate frequent itemset I and execution of I/O operations. This problem has since been addressed; however, problems with building the candidate frequent item collection are still present. In Apriori-based algorithms, the two main expenditures are the creation of a candidate frequent itemset I and execution of I/O operations.

2.7. Frequent Pattern Mining Challenges

The following are the most critical challenges to consider while conducting frequent pattern mining:

- There are numerous real-world situations where it is impossible to enumerate all potential subsets of a given pattern length.
- Many patterns meeting a low minimum support criterion are generated due to repeated pattern mining on vast amounts of data.
- The cost of creating candidate itemsets is high (large candidate sets). For example, it takes 2,100,103 candidates to find a typical size 100, $a_1, a_2, \dots, a_{100}$.
- It becomes tiresome to keep scanning the database and comparing many possibilities if you are conducting lengthy pattern mining.
- Multimedia, geographical, temporal, and other data types may be stored in the database. One system will never be able to harvest all of these data types.
- Multiple LAN and wide-area network (WAN) data sources are accessible for use. Structured, semi-structured, or unstructured data sources are all possible. However, Data Mining is more difficult due to the need to mine their knowledge.

2.8. Apriori Algorithm

The Apriori algorithm is an essential and well-known method for finding frequently recurring itemsets. In a particular database (DB), Apriori is used to discover all frequent itemsets. This algorithm's central concept is to perform many iterations through the database. It iteratively searches the search space using K-Item Sets to learn about $(k + 1)$ -item sets using a breadth-first search (also known as a level-wise search). The algorithm's operation heavily depends on the Apriori property, stating, "All nonempty subsets of a frequent itemset must be frequent." A system's supersets will fail if the minimum support test, as

specified in the anti-monotone characteristics, is not met, which means that if one set is rare, then all of its supersets are rare. This attribute is used to exclude candidate components that are only considered seldom. In the end, the Apriori algorithm successfully identified the database's most common items without question. More search space and higher I/O expenses are required when the database's dimensionality grows with the number of entries. Because the number of database searches has risen, the computing cost of generating candidates has also increased. As a result, the Apriori algorithm has undergone many modifications to reduce the constraints mentioned above as the database grows in size. While the Apriori algorithm uses a similar database scanning approach, the techniques for candidate generation and pruning, support counting, and candidate representation may differ in the later presented algorithms. Apriori algorithms are improved in the following ways thanks to these new algorithms:

- A decreased frequency of scanning a transaction database;
- A reduced number of potential applicants;
- Support for the counting of candidates by making it easier to facilitate.

Stepwise execution of the Apriori algorithm is represented in Figure 2.

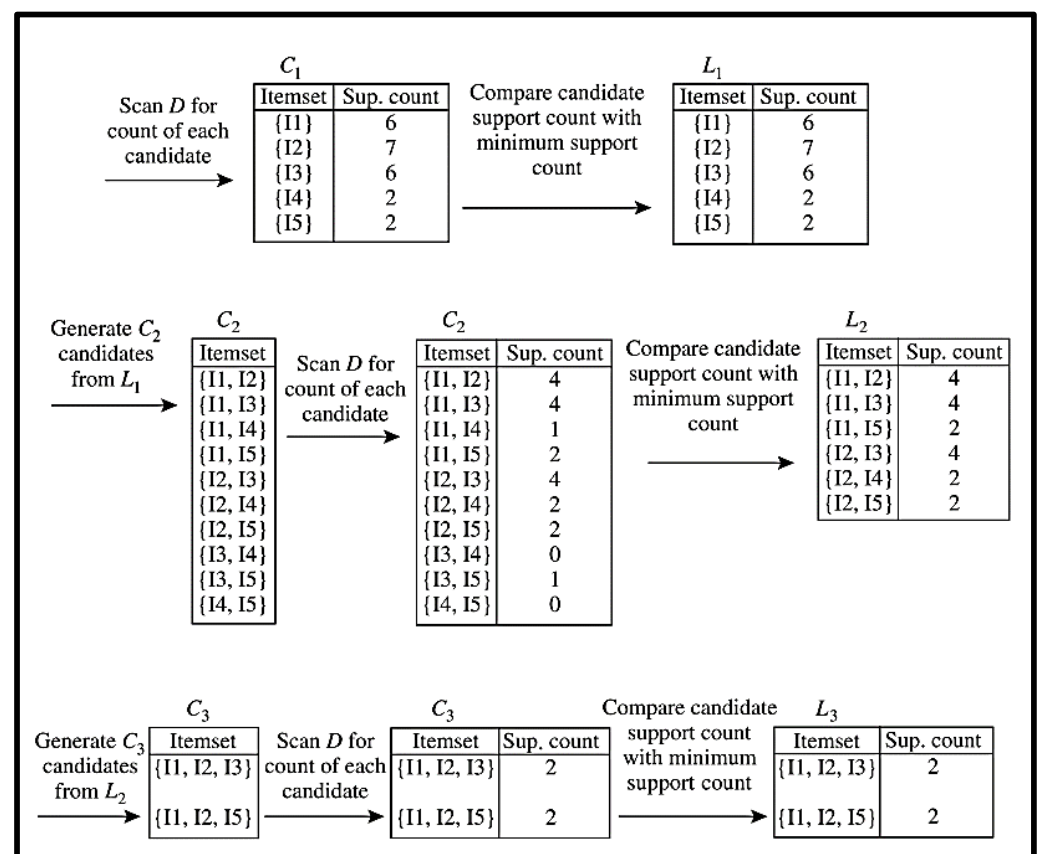


Figure 2. Stepwise execution of Apriori algorithm.

2.9. Frequent Pattern Growth

The Apriori and the frequent pattern (FP) growth algorithms uncover valuable patterns from transactional databases using a single minimum support value. As a result of often recurring items, they produce powerful association rules. A prior knowledge of frequently occurring item characteristics is used in the Apriori method. There are a lot of common patterns of length 1 in the Apriori algorithm that meets the minimal support. To discover L_2 , the collection of standard designs of length L_2 to n , and so on, we utilize L_1 as a starting point. The whole database is checked each time. It also produces a large number

of possible candidate groups. Since the Apriori algorithm has flaws, researchers have developed a new technique known as Frequent Pattern Growth, or FP-Growth, that utilizes the divide-and-conquer approach. There are common itemsets in FP-Growth. However, there are no candidate generations for these items. An FP-tree is built and then used to mine for common patterns in the data. It finds the most common patterns using a single framework with the bare minimum of support. The FP method is much more efficient than the Apriori algorithm. Extended prefix trees (FP-trees) contain compressed and essential information about frequent patterns, while FP-Growth methods use the FP-tree structure to find common patterns in a data collection. The divide-and-conquer rule is the foundation of the FP-Growth algorithm. Often, patterns that recur may be identified by utilizing the FP-Growth technique without creating candidates. As a result, it beats the Apriori algorithm efficiently and produces common patterns more quickly.

Because it depends on a single minimum support structure, the downward closure property is followed by FP-Growth. The tree contains a single null root, item prefix subtrees as children, and a frequent-item header table as a root structure. The item ID, support count, and node-link data are present on every node in the prefix subtree. The number of transactions in the database is stored in a node's count, and a node's link links to the next node in the FP-tree with the same item ID as the present node. Frequent items contain two fields: the item name and the head of the node link, which refers to a node in an FP-tree with this item name as its first node in the table's header. It also demands that items be shown in decreasing order of support counts, and that only often occurring items be included. Therefore, it is essential to build the FP-tree before utilizing it to generate the conditional pattern base and the conditions FP-tree for each frequent item in the FP-tree. There are two stages to FP-Growth. First, we construct the FP-tree, which is a tiny data structure. One method is to utilize the FP-tree's itemsets as a data source, followed by mining the FP-tree starting with each typical length-1 pattern, creating the conditional pattern base, and then recursively mining the dependent FP-trees. FP-Growth working steps are shown in Figure 3.

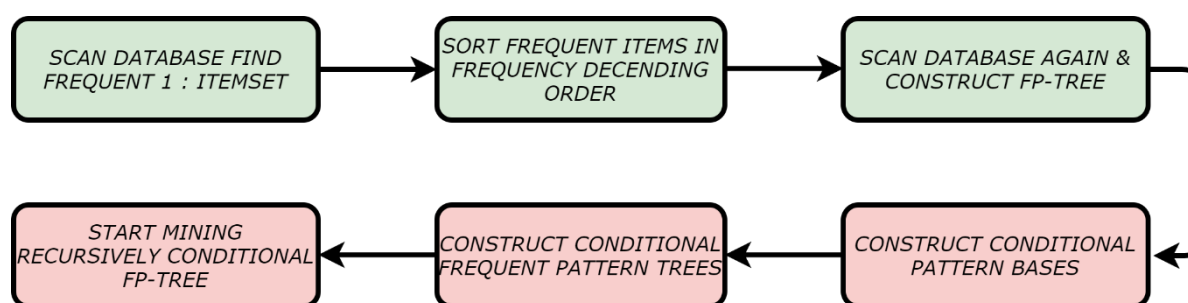


Figure 3. Steps of FP-Growth.

2.10. Theoretical Analysis

The validity of the RQFP algorithm relies on its compliance with the downward closure property: if an itemset is frequent, then all of its subsets are likewise frequent. The recursive query mechanism guarantees that combinations are examined and documented solely when all prior subsets satisfy the minimal support requirement. The Mining Table exclusively comprises frequent items, as indicated by the sorted list L , and recursive searches are conducted just on items with valid support, ensuring that all reported itemsets meet the requisite support criteria. Every recursive step signifies a conditional projection similar to conditional FP-trees in the FP-Growth methodology. Consequently, the output of the RQFP algorithm is certain to encompass exclusively the frequent itemsets.

Time Complexity

Let the following conditions be true:

- n is the number of transactions.
- m is the average number of items per transaction.
- $|I|$ is the number of frequent items.
- Mining Table Construction: This takes $O(n \cdot m)$ time for a single scan to collect support counts and reorder transactions based on the list L .
- Recursive Query Processing: In the worst case (where all items are frequent and appear in all combinations), the number of recursive calls could reach $O(2^{|I|})$, akin to Apriori. However, in practice, early pruning based on the support threshold reduces this significantly.

Expected time complexity of the RQFP algorithm is calculated as follows:

$$O(n \cdot m + R)$$

where $R \ll 2^{|I|}$ due to pruning and dataset sparsity.

Space Complexity

- Memory Storage: The algorithm avoids loading the entire dataset into memory. The Mining Table is disk-resident and indexed, which means main memory is used only for temporary recursive call states and result aggregation.
- Worst-case space usage (for recursive state and caching $O(2^{|I|})$ where d is the maximum depth of recursion) is practically bound.
- Compared to FP-Growth's in-memory FP-tree $O(n \cdot m)$ and Apriori's candidate generation space $O(|I|^k)$, RQFP offers significant savings, particularly on systems with limited memory and efficient SQL execution engines.

3. Research Methodology

The Recursive Queried Frequent Patterns algorithm (RQFP), based on the divide-and-conquer strategy, is presented in this research. It utilizes recursive database queries. The proposed approach is an amalgamation of partition-based and SQL-based techniques.

In the FP-Growth algorithm, to solve real-time data insertion into database issues, we had two options: either reflect those changes into an FP-tree, which is possible until the limiting threshold is reached for the item, or reconstruct the FP-tree from the start. Suppose the first option is selected, and then data items are inserted into the database. Then, if any data item now satisfies the minimum threshold criteria or a data item now loses its minimum threshold limit, the FP-tree needs to be reconstructed. In either case, authors have to rebuild the FP-tree. Reconstructing an FP-tree, again and again, is time-consuming and takes a considerable number of resources for such an organization. Also, it is unrealistic to construct an FP-tree for large databases [29]. The proposed approach uses a Mining Table that resides on a disk instead of in memory. Thus, the problem of determining all frequent patterns is handled in two phases. First, create a Mining Table with the help of List L . This list L consists of key-value pairs of data items and their occurrences throughout the database until its most recent updated state. Secondly, use a recursive queried mining algorithm that takes the Mining Table and List L as an input and determines frequent patterns. The schema of the Mining Table is the same as the Transaction Table, so the Mining Table may be used as a Transaction Table if specific properties are maintained. Despite its advantages, RQFP has specific limitations. The efficacy is contingent upon the SQL engine and the fundamental database architecture's efficiency. Delays may result from the maintenance and updating of the Mining Table in instances where the information is

sparse or particularly dynamic. Additionally, disk-based systems may result in decreased performance unless indexing or parallel query execution methods are implemented to enhance them, despite their scalability. In order to enhance the real-time capabilities of RQFP, future research may concentrate on the integration of GPU-accelerated SQL engines and multi-threading.

The proposed Recursive Queried Frequent Patterns (RQFP) algorithm is a hybrid method integrating SQL-based recursive querying and partitioning strategies. Unlike FP-Growth and Apriori, it relies on a disk-resident structure called the Mining Table and uses SQL recursion to explore frequent patterns efficiently.

3.1. Formal Definition of RQFP Algorithm

The RQFP algorithm operates in two phases:

- **Mining Table Construction:** This involves generating a frequency-based list of items (List L) and organizing the database transactions accordingly:
Let $I = \{ a_1, a_2, \dots, a_m \}$ be a set of items to be stored in a transaction database $DB = [T_1, T_2, \dots, T_n]$, where T_i ($i \in [1..n]$) in Fields $F_1, F_2, \dots, F_m$ and T_i transaction contains a set of items in I .
- **Recursive Pattern Mining:** Using SQL recursion, the algorithm queries item combinations directly from the Mining Table, reducing memory consumption and pre-processing overhead.

The RQFP algorithm is a technique for deriving a collection of frequent patterns from a transaction database. It employs a recursive pattern-finding method, executed through SQL Common Table Expressions (CTEs), utilizing recursive joins and simulating depth-first traversal. The algorithm utilizes a Mining Table, created by scanning the database, filtering things with support $\geq \theta$, and inserting the sorted items into the Mining Table. The recursive SQL query has a worst-case time complexity of $O(n \cdot m)$, while memory usage is minimized by relying on disk-resident tables and SQL query execution mechanisms instead of memory-intensive data structures. The technique is juxtaposed with alternative methods such as Apriori, characterized by substantial candidate generation expenses and multiple database scans, and FP-Growth, which depends on an in-memory FP-tree and significant RAM consumption. To guarantee reproducibility and platform neutrality, the recursive logic of RQFP has been formulated using standard SQL-99 components compatible with PostgreSQL, MySQL 8+, and contemporary cloud SQL engines. Future endeavors will execute RQFP on PostgreSQL and MySQL to evaluate cross-platform performance, facilitating wider use in open source and enterprise settings. The algorithm's recursive logic employs conventional SQL-99 constructs, compatible with PostgreSQL, MySQL 8+, and contemporary cloud SQL engines.

3.2. Mining Table Design

To avoid multiple database scans and memory-intensive candidate generation, the Mining Table stores transactions using only frequent items. Each transaction is sorted based on descending support (List L), and identical frequent itemsets can be merged using a frequency counter (Fsup). This structure enables streamlined querying and supports real-time updates. Moreover, it is costly to handle a vast number of candidate keys and then test those keys with a transactional database to check whether those candidates satisfy the minimum support threshold S_t which most of the previous studies, such as [30–33], adopt as an Apriori-like approach, and is based on an anti-monotone Apriori heuristic [31]. But approaching it like [29] eliminates such problems. Therefore, in our study, we have adopted such a concept. Still, instead of making a data structure, i.e., FP-tree, a table known as a Mining Table is constructed, designed using the following considerations:

1. The Mining Table stores the frequent items of every transaction in some order, avoiding repeatedly scanning DB.
2. Since only frequent items will play a role in the frequent pattern mining, it is necessary to perform one scan of DB to identify the set of frequent items and store them in list L .
3. According to list L , if two transactions share some everyday items, the shared objects can be merged by running a query on that field and calculating the count for that item.
4. If multiple transactions share an identical frequent itemset, they can be merged into one with its frequency registered in a separate field F_{m+1} , also known as F_{sup} .

We have designed the following Mining Table construction algorithm based on the above considerations:

3.3. Algorithmic Workflow

This algorithm can be used only when there is a need for data cleansing, data transformation, data reduction, or discretization. Moreover, this algorithm can be applied from the start, i.e., when there are no rows or transactions in the table on which frequent patterns are generated. So, if the organization stores data according to this algorithm, the transactional table is ready for Data Mining, eliminating Data Mining pre-processing requirements. After the Mining Table has some rows, we can start mining frequent items immediately. For organizations having an online/offline business of shopping stores or any other transaction-related industry, this will provide accurate-time information for the whole DB. We have introduced a query-based algorithm that operates on table fields $F_0 \dots F_m$, where F_0 represents the first field and F_m denotes the last lot. This algorithm recursively traverses from F_0 to F_m using a particular condition. When a recursive call of a mining function reaches F_m , it returns combinations of Field itemsets traversed so far with a support count for each variety satisfying the minimum support threshold St .

To update the Mining Table, we need to maintain its property, i.e., the order of itemsets in each transaction according to list L previously generated by Algorithm 1. Some transactions may effectively hold until the order of elements in the list L does not change. Still, when the order in the list changes, the Mining Table is no longer suitable for frequent mining elements, so we need to order each transaction according to the new changed list L . Maintaining list L comes in handy here, as we know exactly where the change has occurred. We can issue a few simple queries to the database and swap particular items to solve this issue.

Algorithm 1 Mining Table Creation

Input: A transaction database DB.

Output: List L , Mining Table.

Method: The Mining Table is constructed in the following steps.

1. Generate the key-value list L of items with several occurrences “support” by scanning database DB.
2. Sort the list as support descending order or their frequencies.
3. Scan the transaction database DB. Sort T_i ($i \in [1 \dots n]$) as list L , obtained in step 2.
4. Scan DB once again to merge multiple transactions containing identical frequent itemset by creating another Field F_{m+1} or F_{sup} with support count as number occurrences of that transaction. ^{*1}

^{*1} This step is optional.

3.4. Illustrative Example

Consider the following Transactional DB. The steps included in the RQFP algorithm are shown in Figures 4–13.

Transaction	F1	F2	F3	F4	F5
T1	<i>f</i>	<i>a</i>	<i>c</i>	<i>d</i>	<i>g</i>
T2	<i>a</i>	<i>i</i>	<i>c</i>	<i>f</i>	<i>l</i>
T3	<i>b</i>	<i>f</i>	<i>j</i>		
T4	<i>b</i>	<i>c</i>	<i>k</i>	<i>e</i>	
T5	<i>a</i>	<i>f</i>	<i>c</i>	<i>i</i>	

Figure 4. Transactional DB.

Step 1 and 2.

Generate List L and Sort it according to support descending order.

Item Support	
<i>a</i>	3
<i>b</i>	2
<i>c</i>	4
<i>d</i>	1
<i>e</i>	1
<i>f</i>	4
<i>g</i>	1
<i>i</i>	2
<i>j</i>	1
<i>k</i>	1

(a) List L

Item Support	
<i>f</i>	4
<i>c</i>	4
<i>a</i>	3
<i>b</i>	2
<i>d</i>	1
<i>e</i>	1
<i>g</i>	1
<i>j</i>	1
<i>k</i>	1
<i>i</i>	0

(b) Sorted L

Figure 5. (a) List L; (b) Sorted List.

Step 3.

Sort every transaction according to List L (b).

Transaction	F1	F2	F3	F4	F5
T1	<i>f</i>	<i>c</i>	<i>a</i>	<i>d</i>	<i>g</i>
T2	<i>f</i>	<i>c</i>	<i>a</i>	<i>i</i>	<i>l</i>
T3	<i>f</i>	<i>b</i>	<i>j</i>		
T4	<i>c</i>	<i>b</i>	<i>e</i>	<i>k</i>	
T5	<i>f</i>	<i>c</i>	<i>a</i>	<i>i</i>	

Figure 6. Mining DB.

Now let us see how the mining algorithm works.

The first frequent pattern, Set FP, is created, which initially is NULL. Then, query for the first field is executed, which selects specific items from the first field with support count.

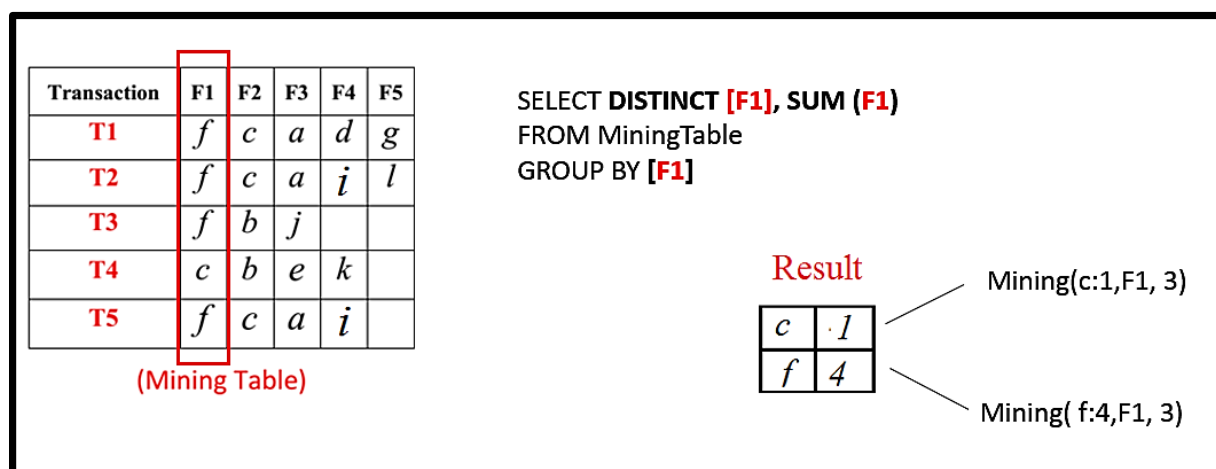


Figure 7. RQFP algorithm step.

For every result item, a Mining () procedure is called, so when the Mining (.) procedure for [c] is called, it will return the FP set combination because the FP set is NULL, so it will return NULL.

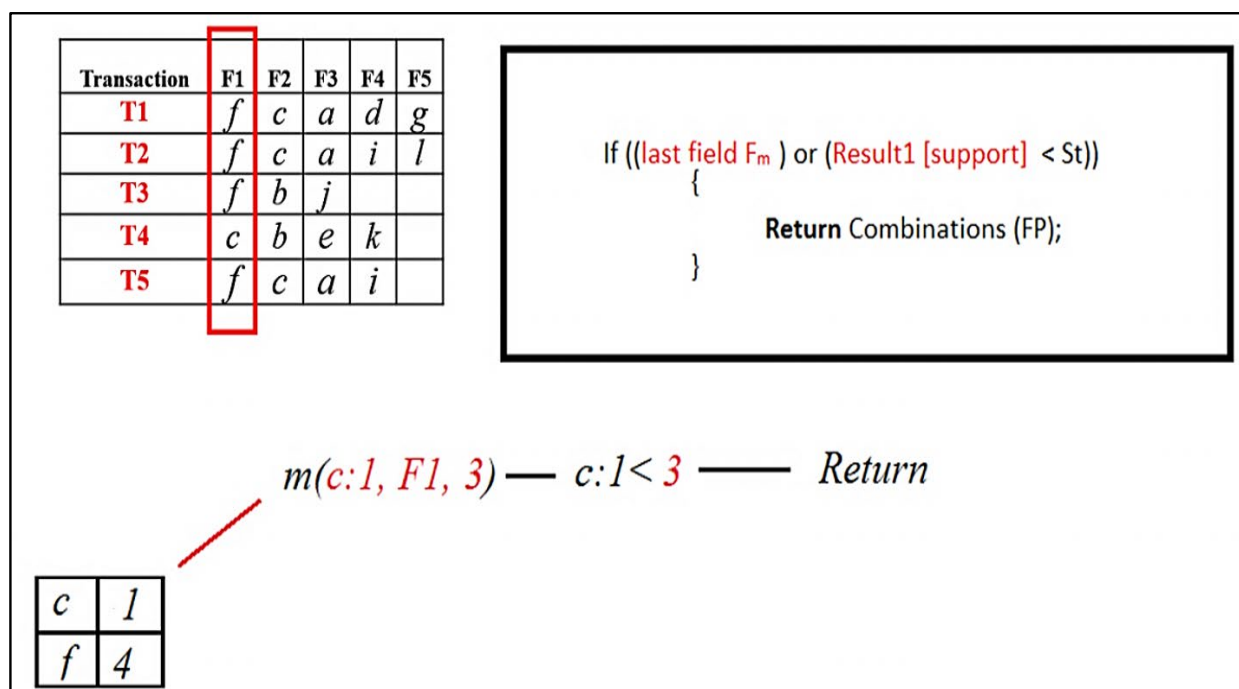


Figure 8. RQFP algorithm step.

When the Mining () procedure for [f] is called, it will be added to the FP set first because its support is greater than the minimum support threshold. Then, it will execute a query that will select particular items from the following field, F2 with support count, where the previous field, F1, is [f].

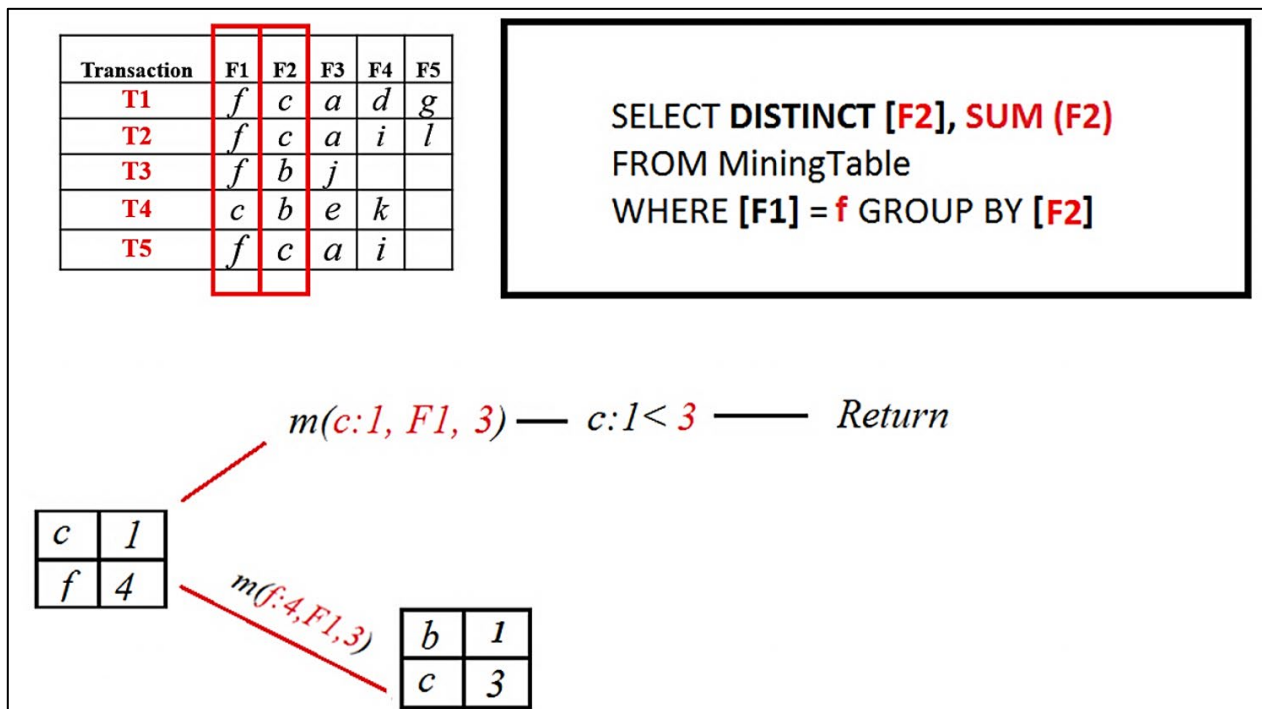


Figure 9. RQFP algorithm step.

The result set now contains [b, c]. Now, when the Mining () procedure for [b] is called, it will return combinations of the FP set, which contains [f] only; therefore, it will return [f] only.

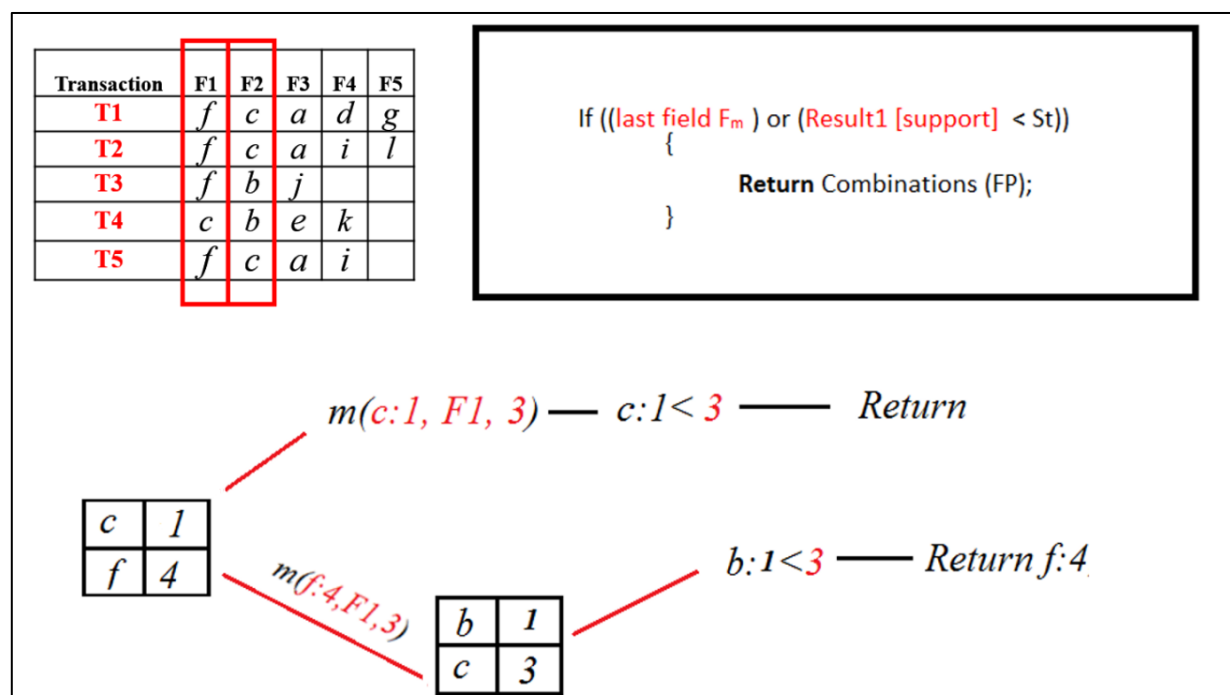


Figure 10. RQFP algorithm step.

The Mining () procedure for [c] will be called and added to the FP set because its support is greater than the minimum support threshold. Then, it will execute a query that will select particular items from the following field, F3 with support count, where the previous field, F2, is [c].

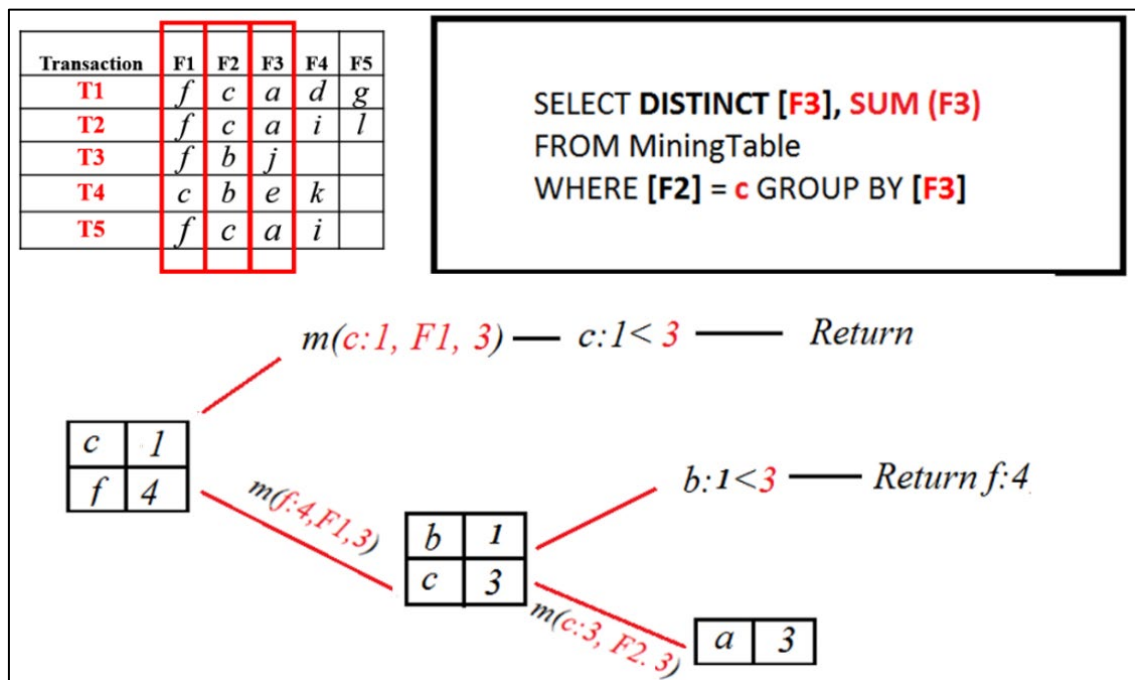


Figure 11. RQFP algorithm step.

The result set now contains [a]. The Mining () procedure for [a] will be called and added to the FP set because its support is greater than the minimum support threshold. Then it will execute a query that will select particular items from the next field, F4 with support count, where the previous field, F3, is [a].

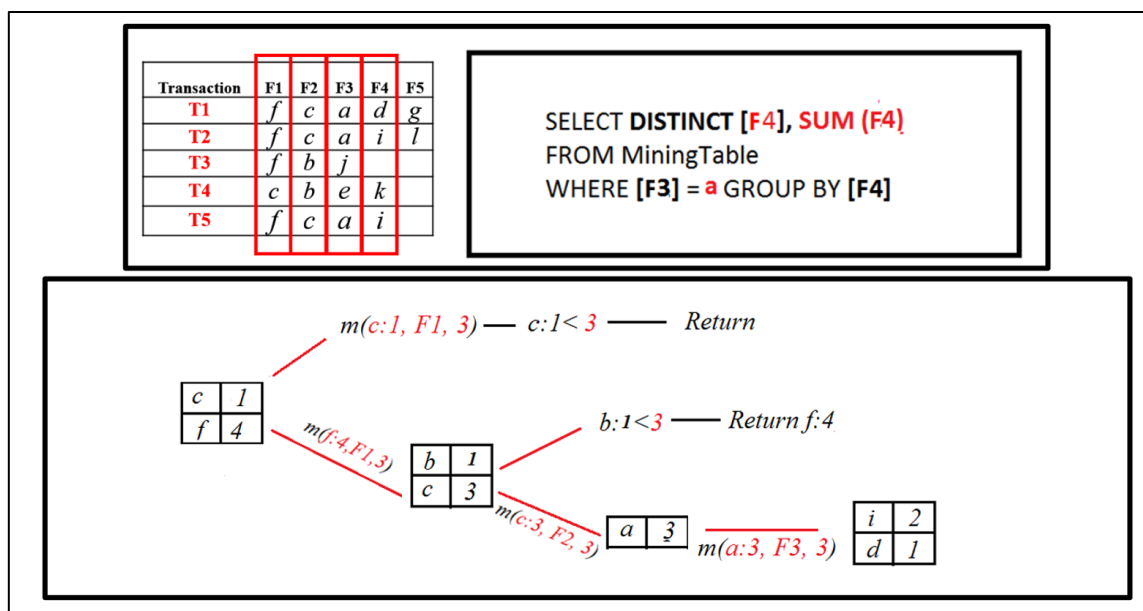


Figure 12. RQFP algorithm step.

The result set now contains [i, d]. Now, when the Mining () procedure for both [i, d] is called, it will return combinations of the FP set, which contains [f, c, a]; therefore, it will return [f, a, c, fa, ac, fc, fac], which are frequent patterns for our DB.

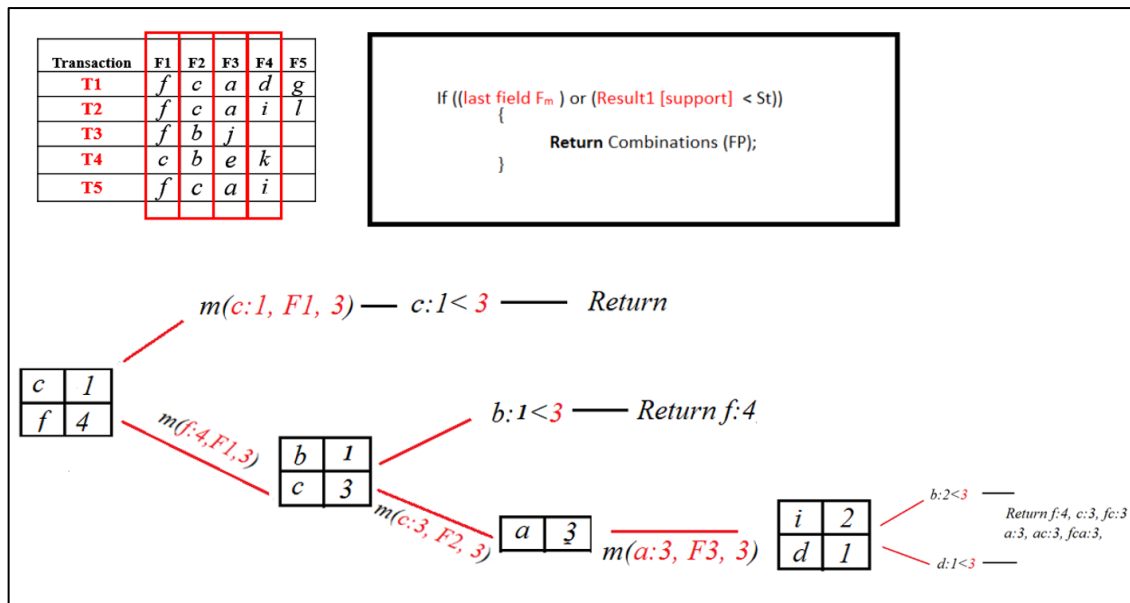


Figure 13. RQFP algorithm step.

The proposed Recursive Queried Frequent Patterns algorithm efficiently mines frequent patterns without candidate generation or pre-processing data before Data Mining. Instead, the organization implements an approach like Algorithm 1 and updates it to Algorithm 2, making data always ready for mining. So, in real time, organizations will be familiar with market trends, which will be helpful for decision making and, in turn, will increase revenues, cut costs, and reduce risks. Aside from that, it utilizes a divide-and-conquer strategy based on partitioning to minimize the problem's size as the function call results come in. Of course, the performance of this algorithm will largely depend upon the SQL engine. Several optimization techniques like parallel programming or multi-threading may also increase the performance.

Algorithm 2 Recursive Queried Frequent Patterns Algorithm

Input: Mining Table created by Algorithm 1, Minimum support threshold S_t .

Output: The complete set of frequent patterns.

Method:

Var FP = null;

Var Result1 [item] [support] = "SELECT DISTINCT [F₁], SUM (F_{m+1} or F₁) FROM MiningTable WHERE [F₁] = L_i GROUP BY [F₁]" ; *1

While (Result [item] ≠ NULL)

{
 Mining_procedure (Result, F₁, S_t)
 }

Mining_procedure (I[item] [support], F_i, S_t)

{
 If (I [Support] > S_t)
 {
 FP = FP + I [item] [Support];
 }
 }

```

Else If ((last field Fm) || (I[Support] < St))
{
    Return Combinations (FP); *2
}
Else
{
    Result[item][support] = "SELECT DISTINCT [Fi+1], SUM (Fm+1 or Fi+1) FROM
MiningTable WHERE [Fi] = Ii GROUP BY [Fi+1]";
    While (Result[item] ≠ NULL)
    {
        Mining_procedure (Result, Fi+1, St)
    } } }

```

*¹ If step 4 of Algorithm 1 is executed, then it is necessary to use SUM (F_{m+1}) else SUM (F₁). *² Combinations take the FP Set (Frequent Patterns Set) as input and return combinations of items traversed so far. Algorithm 3 defines the process of updating the mining table.

Algorithm 3 Updating Mining Table

Input: Mining Table created by Algorithm 1, Itemset I_s or T_{n+1}.

Output: Updated Mining Table, updated list L.

Method:

Update (Mining_Table, I_s)

```

{
    INSERT INTO Mining_Table values Is into Fm.
    Loop (I0...s)
    {
        Update item Support value of list L.
    }
    If (L changed)
    {
        Sort L.
        Swap Changed items.
    }
}

```

It is important to note that, while RQFP does not introduce a fundamentally new theoretical complexity class, it presents a novel algorithmic design by leveraging SQL recursion and disk-resident mining structures to implement frequent pattern mining. This differs significantly from both FP-Growth (tree-based mining) and ECLAT (TID-set intersections). The transformation of recursive frequent pattern discovery into a sequence of relational queries allows for a seamless integration with modern database systems, enabling native, real-time, and scalable pattern discovery without external data structures or engines. This hybridization of pattern mining and relational processing introduces a new design space for future database-integrated Data Mining techniques.

3.5. Theoretical Foundation and Complexity Analysis

The Recursive Queried Frequent Patterns (RQFP) algorithm is grounded in two primary principles: the downward closure property of frequent itemsets and the recursive depth-first traversal implemented using SQL common table expressions (CTEs). This sec-

tion formally defines the core notations, establishes algorithmic correctness, and analyzes its time and space complexity.

3.5.1. Formal Definitions

- $DB = \{T_1, T_2, \dots, T_n\}$ and is a transactional database, where each transaction T_i is a subset of a universal itemset $I = \{i_1, i_2, \dots, i_m\}$.
- List L is a frequency-ordered list of frequent items (i.e., items satisfying the minimum support threshold σ).
- A Mining Table is a disk-resident structure that stores transactions using only frequent items, sorted in a descending order based on List L .
- $F_0 \dots F_k$ represent positional fields in the Mining Table corresponding to item positions within a transaction.
- FP denotes the set of frequent item combinations generated recursively.
- σ is the user-defined minimum support threshold.
- d is the maximum recursion depth (equal to the maximum transaction length among frequent items).

The RQFP algorithm proceeds by recursively querying item combinations from the Mining Table, extending partial frequent itemsets (prefixes) at each depth only if all prefix subsets satisfy the support threshold σ .

3.5.2. Correctness and Downward Closure

The algorithm adheres to the anti-monotonicity (downward closure) property of support as follows:

If an itemset X is frequent, then all subsets of X are also frequent.

By construction, the recursive SQL query evaluates item extensions only if their prefixes have previously met the threshold. The Mining Table only contains frequent items, and all recursive branches are pruned as soon as the support condition is violated. Thus, the algorithm does not report any spurious patterns.

3.5.3. Time Complexity

Let the following conditions be true:

- n is the number of transactions in the database.
- m is the average number of items per transaction.
- $|I|$ is the number of frequent items after pruning with σ .

Phase 1: Mining Table Construction

- Scan DB once to compute support counts $\rightarrow O(n \times m)$.
- Construct List L and reorder transactions accordingly $\rightarrow O(n \times \log m)$.

Phase 2: Recursive Pattern Mining

- In the worst case (where all items are frequent), the number of recursive branches is exponential: $O(2^{|I|})$. However, early pruning drastically limits actual expansion.
- Each recursive step performs an indexed SQL SELECT DISTINCT ... GROUP BY operation on the Mining Table, bounded by $O(n)$.

Thus, the expected total time complexity is as follows:

$T(n, m, |I|) = O(n \times m) + O(R)$, where $R \ll 2^{|I|}$ due to pruning and dataset sparsity.

3.5.4. Space Complexity

Unlike FP-Growth, which requires constructing an in-memory prefix tree, RQFP stores all intermediate data on the disk using SQL tables. This results in the following:

- The main memory is used only for recursive call states and result aggregation.
- The Mining Table remains disk-resident and can be indexed for efficient access.

Memory usage:

- Recursive call stack: $O(d)$, where d is the depth of recursion.
- Disk storage: $O(n \times m)$ for the Mining Table (shared with transactional data).

Hence, the space complexity is as follows:

$S(n, m) = O(d)$ in memory, $O(n \times m)$ on disk.

3.5.5. Advantages over Existing Approaches

Unlike Apriori and FP-Growth, RQFP is fully database-integrated, with no external data structures or in-memory FP-trees required. It offers a practical trade-off by minimizing memory usage at the expense of minor overhead in maintaining sorted transactional data. Table 1 defines the advantages of the proposed method over the existing approaches used by other researchers.

Table 1. Advantages of proposed method over existing approaches.

Feature	Apriori	FP-Growth	RQFP
Candidate Generation	Yes	No	No
Memory Requirement	High	Medium	Low (disk-resident)
Number of DB Scans	Multiple	2	1–2
Tree Construction	No	Yes	No
SQL Native Execution	No	No	Yes (Standard SQL-99)

4. Experiments, Results, and Discussions

4.1. Dataset Description

In this study, these algorithms were implemented for a university dataset to determine frequent courses chosen by students in various departments of the university. The dataset was collected from the students at the university. The data of 434 students were composed of four subjects chosen by the student. The collected dataset contained a unique number of students whose subject choice was compiled for study. Students had to choose four subjects; the dataset sample is shown below. In the dataset, GE refers to general English, CH refers to chemistry, BO represents biology, and ZO refers to zoology. Table 2 illustrates a sample or screenshot of the dataset formulated for the experiment.

Table 2. Dataset.

Sr. No.	Unique Number	SUB1	SUB2	SUB3	SUB4
1	2021-P-01	GE	CH	BO	ZO
2	2021-P-02	GE	CH	BO	ZO
3	2021-P-03	GE	CH	BO	ZO
4	2021-P-04	GE	CH	BO	ZO
⋮	⋮	⋮	⋮	⋮	⋮
431	2021-P-431	GE	PH	MA	CH
432	2021-P-432	GE	PH	MA	CH
433	2021-P-433	GE	PH	MA	CH
434	2021-P-434	GE	PH	MA	CH

4.2. System Configuration and Software Environment

In this study, the authors used a high configuration system. The implementation of the proposed algorithm was tested on a stand-alone machine with a processing unit of Intel i5 with four cores and a frequency of 2.5 GHz. The system consists of 4 GB of RAM. The RQFP algorithm was executed and evaluated utilizing the Microsoft SQL Server 2012 on a Windows 10 system. Indexes were established on frequently searched fields (F1 to Fm), and the system utilized the default caching and query optimization functionalities offered by the SQL Server 2012 without any manual adjustments. Recursive CTE syntax conformed to SQL:1999 and was portable to MySQL 8+, PostgreSQL. Considering that the efficacy of RQFP is affected by the internal mechanisms of SQL engines, such as recursive query support, indexing methodologies, and memory allocation, subsequent research will entail evaluating the algorithm on additional prominent relational database systems, including MySQL, PostgreSQL, and cloud-based SQL services. This will evaluate the algorithm's portability and consistency across many data settings.

4.3. Execution and Results

To facilitate the clear presentation and interpretation of the intermediate results generated by the FP-Growth and RQFP algorithms, a graphical user interface (GUI) was developed using C#. While these algorithms can be implemented using common programming languages such as C, C++, Java, or C#, console-based applications tend to limit the readability and visual structuring of complex outputs, especially when dealing with large datasets and multiple derived data structures. In this study, the implementation was carried out using Java (version 17, LTS), which was the stable and widely supported release available. For instance, FP-Growth involves several stages: these include generating the frequency list L, sorting transactions, constructing the FP-tree, deriving conditional pattern bases, and building conditional FP-trees, before arriving at the final set of frequent patterns. Displaying these multi-level outputs in a console environment can hinder comprehension. Therefore, a visual application was preferred to enhance usability, allow for better debugging, and support the step-by-step execution of the algorithms in a user-friendly interface.

A web-app-based application was created using the C# programming language for both algorithms. It was made ready to be published online for testing various datasets by uploading them onto a server space and then executing these algorithms with just a click.

Step 1:

Generate List L, which contains items as course code and support as the frequency for each item. Then, sort the list according to descending order of support count. As shown in Figure 14, general English (GE) has the highest frequency among all subjects, justifying its consistent appearance in frequent itemsets. This confirms the RQFP algorithm's ability to dynamically adapt to item frequencies when building List L.

Step 2:

When the support threshold is 100%, only GE (0) is the frequent item, since GE is mandatory for each student. When the support threshold is lowered to 80%, a few more combinations make it to the frequent pattern set. More and more combinations will reduce the support threshold to the frequent pattern set. When the support threshold reaches 0%, the whole dataset is a frequent pattern set.

4.4. Comparative Analysis

Frequent itemset mining methods, such as Apriori, FP-Growth, and RQFP, are based on a transaction database and a minimal level of support. These algorithms produce outputs using a level-wise method, scanning the database frequently to determine the level of support for each pattern. FP-Growth, an alternative to Apriori, only considers patterns

already present in the database. RQFP, on the other hand, uses SQL-based recursive algorithms to divide-and-conquer databases. The algorithms are compared in terms of their method, memory use, number of scans, and time consumed. Apriori uses Apriori properties like join and pure property for frequent mining patterns, while FP-Growth constructs conditional pattern-free and dependent patterns based on the database, satisfying the minimum support. RQFP uses disk-resident tables and an SQL procedure to generate frequent patterns from the dataset. The algorithms use different search types and memory utilizations. Apriori performs multiple scans for generating candidate sets, while FP-Growth scans the database only twice. RQFP uses two scans to create a mining dataset, which can be eliminated if the mining dataset and transaction dataset are the same. In terms of execution time, the Apriori algorithm wastes more time in producing candidates, while the FP-Growth and RQFP algorithms have lower execution times compared to Apriori.

Item	Support	Item	Support
GE	434	GE	434
BO	421	BO	421
CH	355	ZO	366
PH	13	CH	355
ZO	366	SD	79
FP	42	FP	42
GG	13	GG	13
MA	13	MA	13
SD	79	PH	13

Figure 14. List L and Sorted List L with support count for student dataset.

The findings from the academic dataset indicate that the suggested RQFP algorithm surpasses Apriori and somewhat enhances performance compared to FP-Growth regarding execution time, despite their comparable pattern counts. This is accomplished by utilizing disk-based structures and SQL recursion, which enhance memory efficiency and minimize the preprocessing burden. In comparison to normal benchmark datasets, the execution time for Apriori and FP-Growth markedly escalates with high-dimensional or sparse data, such as retail data. While RQFP has not been assessed using these benchmark datasets, its architecture indicates possible benefits, particularly in real-time or memory-limited contexts.

Researchers recognize that subsequent research needs include the direct benchmarking of RQFP against Retail, Chess, and Mushroom datasets to validate its generalizability and scalability. Nevertheless, the existing academic dataset functions as a regulated environment to verify the accuracy and efficacy of the RQFP's logic. Among the three algorithms, RQFP demonstrates the lowest execution time (00:00:03.6337641), as shown in Table 3, thereby outperforming both Apriori and FP-Growth. Furthermore, the results summarized in Table 4 indicate that while all three algorithms (RQFP, Apriori, and FP-Growth) yield identical patterns for the academic dataset, they differ notably in terms of execution efficiency across both academic and benchmark datasets.

Table 3. Comparison based on parameters.

Algorithm → Parameters ↓	Apriori Algorithm	FP-Growth	RQFP Algorithm
Technique	Apriori Property	FP-Tree Generation	Disc Resident Table
Search Strategy	Breadth-First Search	Divide and Conquer	Divide and Conquer
Space Utilization	$O(2^d)$ Where d is the horizontal width of DB	$O(n^2)$ where n is the item from the database	$O(n)$ where n is the item from the database
Number of Scans	1 for each candidate	2	2
Execution Time	00:00:06.7341631	00:00:03.7641633	00:00:03.6337641

Table 4. Comparative performance analysis of frequent pattern mining algorithms across an academic dataset (434 university transactions) and standard benchmark datasets (Retail, Chess).

Dataset	Algorithm	Support (%)	Patterns Found	Exec. Time (s)	Source/Environment
Academic (434)	RQFP	75	12	3.63	Current Study (SQL Server 2012, C#, 4GB RAM)
Academic (434)	Apriori	75	12	6.73	Current Study
Academic (434)	FP-Growth	75	12	3.76	Current Study
Retail (Dunnhumby)	Apriori	0.5	3205	18.23	[18]
Retail	FP-Growth	0.5	3205	10.87	[18]
Chess	ECLAT	60	1689	8.2	[25]

5. Conclusions

This research presents the Recursive Queried Frequent Patterns (RQFP) method, a database-integrated approach for frequent itemset mining. It diminishes memory reliance and enhances compatibility with relational database systems. RQFP surpasses Apriori and FP-Growth in terms of output interpretability and memory efficiency. Subsequent study will concentrate on extensive implementation and high-dimensional domains. The collection of common patterns created by the bulk of current pattern mining methods is too extensive for practical use. A wide range of research associations in areas like closed patterns, maximum patterns, approximation patterns, condensed pattern bases, representative patterns, clustered patterns, and frequent discriminative patterns has made the field of frequent pattern Data Mining more and more popular. However, the compactness of the techniques in terms of time and space complexity has been a significant area of concern in the cases of program or algorithm designing. Much research is needed to reduce the size of derived pattern collections while simultaneously enhancing the quality of the patterns maintained. While there are many fast techniques for mining an entire array of frequent patterns, frequent approximation patterns may be the best option in some instances. In this research, the researchers have designed a technique that integrates recursive and frequent pattern techniques better than the existing techniques in terms of efficiency and complexity. Although the execution time enhancements over FP-Growth were modest (~0.13 s) in the current configuration, RQFP provides architectural advantages in terms of memory efficiency and database integration. The RQFP algorithm was formally defined using mathematical notation and pseudocode. SQL recursion, implemented through standard

constructs, ensures compatibility across database engines. Theoretical complexity analysis confirms its efficiency under memory constraints.

6. Limitations and Future Work

The behavior of RQFP in real-world deployments with more complex operational constraints will be evaluated through systematic performance evaluations under diverse data scales, hardware configurations, and system loads in future work.

The Recursive Queried Frequent Patterns (RQFP) algorithm exhibits a commendable memory economy and integration with databases; yet, this work has specific limitations. The present assessment relies on a singular academic dataset of moderate dimensions. Consequently, statistical significance testing, including t-tests or Wilcoxon signed-rank tests, was not conducted owing to the restricted number of experimental runs and sample size. The lack of extensive and high-dimensional datasets constrains the generalizability of the reported performance improvements. To mitigate these constraints, further research will involve evaluating the RQFP algorithm on synthetic datasets expanded to 10x–100x the original size, facilitating a comprehensive investigation of recursion depth, indexing overhead, and execution time over diverse support thresholds. Additionally, cross-platform benchmarking will be performed on various relational database systems, including MySQL 8+, PostgreSQL, and the SQL Server, to evaluate the effects of engine-specific optimizations such as recursive CTE execution, caching methods, and index management. These improvements will yield more robust evidence of the scalability, resilience, and portability of the RQFP framework in real-world settings.

Author Contributions: I.A.K.: conceptualization, data collection, writing, and analysis; H.-Y.C.: conceptualization, supervision, and organization; S.S.: writing, project administration, and referencing; C.S.: proofreading, validation, writing, and reviewing. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data on the measured system variables indicating system functions that support the findings of this study are included within this paper.

Conflicts of Interest: Author Ishtiyah Ahmad Khan is employed by upGrad Education Private Limited. Author Shamneesh Sharma is employed by byteXL TechEd Private Limited, Hyderabad. Author Chetan Sharma is employed by PW-Institute of Innovation, PhysicsWallah Limited. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Sai Prasad, D.S.P.K. A Theoretical Review on Data Mining and Machine Learning Techniques for Data Analysis. *Int. J. Adv. Sci. Technol.* **2020**, *29*, 1220–1226.
2. Chakraborty, M.; Biswas, S.K.; Purkayastha, B. Data Mining Using Neural Networks in the form of Classification Rules: A Review. In Proceedings of the 2020 4th International Conference on Computational Intelligence and Networks (CINE), Kolkata, India, 27–29 February 2020; pp. 1–6.
3. de Sousa, L.R.; de Carvalho, V.O.; Pentead, B.E.; Affonso, F.J. A Systematic Mapping on the Use of Data Mining for the Face-to-Face School Dropout Problem. In Proceedings of the 13th International Conference on Computer Supported Education (CSEDU 2021), Online Streaming, 23–25 April 2021; Volume 1, pp. 36–47.
4. Dridi, A.; Gaber, M.M.; Azad, R.M.A.; Bhogal, J. Scholarly data mining: A systematic review of its applications. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **2021**, *11*, e1395. [[CrossRef](#)]

5. Li, L.; Ding, P.; Chen, H.; Wu, X. Frequent Pattern Mining in Big Social Graphs. *IEEE Trans. Emerg. Top. Comput. Intell.* **2022**, *6*, 638–648. [\[CrossRef\]](#)
6. Agrawal, R.; Imieliński, T.; Swami, A. Mining association rules between sets of items in large databases. In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, 25–28 May 1993; pp. 207–216.
7. Almoqbily, R.S.; Rauf, A.; Quradaa, F.H. A Survey of Correlated High Utility Pattern Mining. *IEEE Access* **2021**, *9*, 42786–42800. [\[CrossRef\]](#)
8. Jazayeri, A.; Yang, C.C. Frequent Pattern Mining in Continuous-time Temporal Networks. *arXiv* **2021**, arXiv:2105.06399. [\[CrossRef\]](#) [\[PubMed\]](#)
9. Djenouri, Y.; Lin, J.C.-W.; Nørnvåg, K.; Ramampiaro, H.; Yu, P.S. Exploring decomposition for solving pattern mining problems. *ACM Trans. Manag. Inf. Syst. (TMIS)* **2021**, *12*, 1–36. [\[CrossRef\]](#)
10. Han, J.; Kamber, M.; Pei, J. Data mining trends and research frontiers. In *Data Mining*; University of Illinois at Urbana–Champaign: Urbana, IL, USA, 2012; pp. 585–631.
11. Chi, Y.; Wang, H.; Yu, P.S.; Muntz, R.R. Moment: Maintaining closed frequent itemsets over a stream sliding window. In Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM'04), Brighton, UK, 1–4 November 2004; pp. 59–66.
12. Fard, M.J.S.; Namin, P.A. Review of Apriori based Frequent Itemset Mining Solutions on Big Data. In Proceedings of the 2020 6th International Conference on Web Research (ICWR), Tehran, Iran, 22–23 April 2020; pp. 157–164.
13. Liu, C.; Li, X. Mining Method Based on Semantic Trajectory Frequent Pattern. In *Advanced Information Networking and Applications, Proceedings of the International Conference on Advanced Information Networking and Applications, Toronto, ON, Canada, 12–14 May 2021*; Springer: Cham, Switzerland, 2021; pp. 146–159.
14. Sornalakshmi, M.; Balamurali, S.; Venkatesulu, M.; Krishnan, M.N.; Ramasamy, L.K.; Kadry, S.; Lim, S. An efficient apriori algorithm for frequent pattern mining using mapreduce in healthcare data. *Bull. Electr. Eng. Inform.* **2021**, *10*, 390–403. [\[CrossRef\]](#)
15. Han, J.; Fu, Y. Discovery of multiple-level association rules from large databases. In Proceedings of the 21st VLDB Conference, Zurich, Switzerland, 11–15 September 1995; Volume 95, pp. 420–431.
16. Han, J.; Pei, J.; Dong, G.; Wang, K. Efficient computation of iceberg cubes with complex measures. In Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA, 21–24 May 2001; pp. 1–12.
17. Clustering, P.-B. Grouping of Questions From a Question Bank Using Partition-Based Clustering. In *Developing a Keyword Extractor and Document Classifier: Emerging Research and Opportunities*; Engineering Science Reference: Hershey, PA, USA, 2021.
18. Han, J.; Pei, J.; Yin, Y. Mining frequent patterns without candidate generation. *ACM Sigmod Rec.* **2000**, *29*, 1–12. [\[CrossRef\]](#)
19. Chormunge, S.; Mehta, R. Comparison Analysis of Extracting Frequent Itemsets Algorithms Using MapReduce. In *Intelligent Data Communication Technologies and Internet of Things, Proceedings of the ICICI 2020, Coimbatore, India, 27–28 August 2020*; Springer: Singapore, 2021; pp. 199–210.
20. Agrawal, R.; Shafer, J.C. Parallel mining of association rules. *IEEE Trans. Knowl. Data Eng.* **1996**, *8*, 962–969. [\[CrossRef\]](#)
21. Han, J.; Dong, G.; Yin, Y. Efficient mining of partial periodic patterns in time series database. In Proceedings of the 15th International Conference on Data Engineering (Cat. No. 99CB36337), Sydney, NSW, Australia, 23–26 March 1999; pp. 106–115.
22. Thurachon, W.; Kreesuradej, W. Incremental Association Rule Mining With a Fast Incremental Updating Frequent Pattern Growth Algorithm. *IEEE Access* **2021**, *9*, 55726–55741. [\[CrossRef\]](#)
23. Cong, S.; Han, J.; Padua, D. Parallel mining of closed sequential patterns. In Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, Chicago, IL, USA, 21–24 August 2005; pp. 562–567.
24. Sun, J.; Xun, Y.; Zhang, J.; Li, J. Incremental frequent itemsets mining with FCFP tree. *IEEE Access* **2019**, *7*, 136511–136524. [\[CrossRef\]](#)
25. Zaki, M.J. Efficient enumeration of frequent sequences. In Proceedings of the Seventh International Conference on Information and Knowledge Management, Bethesda, MD, USA, 3–7 November 1998; pp. 68–75.
26. Bernal, J.N.; Rodriguez, J.P.; Portella, J. DBMS and Oracle Datamining. *Preprints* **2021**. [\[CrossRef\]](#)
27. Teng, Y. A Critical Review of SQL-Based Mining Relational Database. *Int. J. Comput. Commun. Eng.* **2021**, *10*, 68–74. [\[CrossRef\]](#)
28. Nasyuha, A.H.; Jama, J.; Abdullah, R.; Syahra, Y.; Azhar, Z.; Hutagalung, J.; Hasugian, B.S. Frequent pattern growth algorithm for maximizing display items. *Telkonnika* **2021**, *19*, 390–396. [\[CrossRef\]](#)
29. Afrati, F.; Gionis, A.; Mannila, H. Approximating a collection of frequent sets. In Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, WA, USA, 22–25 August 2004; pp. 12–19.
30. Agarwal, R.C.; Aggarwal, C.C.; Prasad, V.V.V. A tree projection algorithm for generation of frequent item sets. *J. Parallel Distrib. Comput.* **2001**, *61*, 350–371. [\[CrossRef\]](#)
31. Aggarwal, C.C.; Yu, P.S. A new framework for itemset generation. In Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Seattle, WA, USA, 1–4 June 1998; pp. 18–24.

32. Beil, F.; Ester, M.; Xu, X. Frequent term-based text clustering. In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, AB, Canada, 23–26 July 2002; pp. 436–442.
33. Agrawal, R.; Srikant, R. Mining sequential patterns. In Proceedings of the Eleventh International Conference on Data Engineering, Taipei, Taiwan, 6–10 March 1995; pp. 3–14.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.