Introduction to Association Rule Mining (ARM)

Association Rule Mining (ARM) is the process of identifying significant relationships, patterns, and associations among sets of items or objects within large datasets. These datasets are typically found in transaction databases, relational databases, and other information repositories. The goal is to uncover frequent patterns and associations that can provide valuable insights for decision-making and strategic planning.

- **Item Collection and Itemset**
  - Item Collection (X): A set of all items under consideration. Example: $X = milk, bread, butter, cheese$.
  - Itemset: A subset of items from the item collection. Example: $milk, bread$ is a 2-itemset.
  - Transaction (T): A subset of items from the item collection, associated with a unique transaction ID. Example: Transaction $T_1 = milk, bread$ with transaction ID 1.
  - Rule Form ($\alpha \Rightarrow \beta$): An implication expression where α and β are itemsets and $\alpha \cap \beta = \emptyset$. Example: $milk \Rightarrow bread$ suggests that if milk is purchased, bread is also likely to be purchased.
- **Support and Confidence**
  - Support (s):
    - Definition: The probability that a transaction contains both α and β.
    - Formula: $\text{support}(\alpha \Rightarrow \beta) = P(\alpha \cap \beta)$.
    - Example: If 3 out of 5 transactions contain both milk and bread, the support for $milk \Rightarrow bread$ is 0.6 (60%).
  - Frequent Itemset:
    - Definition: An itemset whose support is greater than or equal to a specified minimum support threshold.
    - Example: If the minimum support threshold is 50%, then an itemset occurring in 60% of the transactions is a frequent itemset.
  - Confidence (c):
    - Definition: The conditional probability that a transaction containing α also contains β.
    - Formula: $\text{Confidence}(\alpha \Rightarrow \beta) = P(\beta|\alpha) = \frac{P(\alpha \cap \beta)}{P(\alpha)}$.
    - Example: If 4 out of 5 transactions containing milk also contain bread, the confidence for $milk \Rightarrow bread$ is 0.8 (80%).
- **Interestingness Measure: Correlations (Lift)**
  - Definition: A measure of how much more likely α and β are to occur together than if they were independent.
  - Formula: $\text{lift}(A, B) = \frac{P(A \cap B)}{P(A)P(B)}$.
  - Interpretation: A lift value greater than 1 indicates a positive correlation between α and β, meaning they occur together more frequently than would be expected by chance.

- Example: If the probability of buying milk is 0.6, the probability of buying bread is 0.5, and the probability of buying both is 0.4, then the lift for $milk, bread$ is $\frac{0.4}{0.6 \times 0.5} = 1.33$. This indicates that milk and bread are 33% more likely to be bought together than expected by random chance.

Frequent Pattern Mining

Association rule mining, particularly when applied to large datasets, faces significant resource constraints that can impact performance and scalability. One of the primary challenges is the number of database scans required. Traditional algorithms like Apriori necessitate multiple full scans of the database, which can be computationally expensive and time-consuming. Each iteration of the Apriori algorithm, which involves generating candidate itemsets of increasing length, requires scanning the entire database to count the support for each candidate. As the size of the dataset grows, the number of scans and the volume of data to be processed increase, leading to substantial I/O overhead and prolonged execution times.

In addition to the number of scans, the computational complexity of handling large numbers of candidate itemsets presents another significant resource constraint. As the length of itemsets increases, the number of potential candidate itemsets grows exponentially. This combinatorial explosion results in an extensive list of candidates that need to be generated, stored, and tested against the database. Managing this large set of candidates requires considerable memory and processing power, often pushing the limits of available computational resources.

Memory usage is another critical constraint in association rule mining. Algorithms must store intermediate data, such as candidate itemsets, frequency counts, and support values, during their execution. When dealing with large datasets, the amount of memory required to store these intermediate results can become prohibitive. Insufficient memory can lead to frequent paging and swapping, further degrading performance and increasing the time required to complete the mining process.

To address these constraints, advanced algorithms like FP-growth have been developed. FP-growth reduces the number of database scans to just two: one to determine the frequent 1-itemsets and another to construct the FP-tree. This significantly cuts down on the I/O overhead compared to Apriori. Furthermore, FP-growth's use of a compact data structure (the FP-tree) helps to mitigate memory usage issues by providing a more efficient way to represent the dataset and its frequent patterns. This divide-and-conquer approach, which breaks down the mining process into smaller subproblems, also reduces the computational load and makes better use of available resources.

Overall, addressing resource constraints in association rule mining involves balancing the trade-offs between computational complexity, memory usage, and I/O operations. By optimizing these factors, advanced algorithms can achieve more efficient and scalable solutions for extracting meaningful patterns from large datasets.

Apriori

Why Use Apriori?

Handling Exponential Complexity

The challenge of finding frequent itemsets in large datasets is formidable due to the exponential growth in the number of possible itemsets as the number of items increases. In a dataset with nnn unique items, there are $2^n - 1$ potential itemsets. This combinatorial explosion makes it computationally infeasible to evaluate all possible itemsets directly.

Systematic Reduction of Candidate Itemsets: Apriori addresses this complexity by systematically reducing the number of candidate itemsets that need to be considered at each iteration. It does this by leveraging the observation that an itemset can only be frequent if all of its subsets are also frequent. This allows the algorithm to eliminate many candidate itemsets early in the process, significantly reducing the computational burden.

Efficiency Gains: By focusing only on those itemsets that have a reasonable chance of being frequent, Apriori minimizes the amount of work required to count the supports of itemsets. This targeted approach helps in managing the computational load more effectively and makes it feasible to work with large datasets that would otherwise be impractical to analyze.

Efficient Pruning

One of the key strengths of the Apriori algorithm is its ability to prune large portions of the search space early in the process. The pruning strategy is based on the simple yet powerful insight that any subset of a frequent itemset must itself be frequent.

Pruning the Search Space: As the algorithm progresses through its iterative levels, it uses the frequent itemsets identified at one level to generate candidate itemsets for the next level. Before generating these candidates, Apriori prunes the search space by eliminating any itemset that contains an infrequent subset. This ensures that the algorithm does not waste time evaluating itemsets that cannot possibly be frequent.

Example of Pruning: Consider a scenario where the algorithm identifies that the itemset {A,B}{A, B}{A,B} is infrequent. Based on the Apriori principle, any itemset that includes {A,B}{A, B}{A,B} (such as {A,B,C}{A, B, C}{A,B,C}) will also be infrequent. By pruning {A,B}{A, B}{A,B} early, the algorithm avoids generating and evaluating {A,B,C}{A, B, C}{A,B,C} and other larger supersets, thereby saving computational resources.

Design Philosophy

Apriori Principle

The Apriori principle is central to the algorithm's design and efficiency. It states that for an itemset to be frequent, all of its subsets must also be frequent. Conversely, if an itemset is infrequent, then all of its supersets must be infrequent as well.

Fundamental Insight: This principle provides a foundation for the algorithm's pruning strategy. By ensuring that only those itemsets whose subsets are all frequent are considered, the Apriori algorithm effectively narrows down the search space to the most promising candidates. This insight dramatically reduces the number of itemsets that need to be evaluated, particularly in the later stages of the algorithm where the itemsets become longer and more numerous.

Implementation: The Apriori principle is implemented by generating candidate itemsets from the frequent itemsets identified in the previous iteration. Any candidate itemset that includes an infrequent subset is immediately discarded. This not only streamlines the generation of candidates but also ensures that the algorithm maintains a high level of efficiency as it progresses.

Level-wise Search

The Apriori algorithm employs a level-wise search strategy, which involves generating candidate itemsets of increasing length iteratively. This structured approach ensures a systematic exploration of the itemset space. This methodical, level-wise approach ensures that the algorithm only considers itemsets that have a realistic chance of being frequent, based on the frequent itemsets identified in previous iterations. It also allows for incremental and manageable increases in computational complexity, making the algorithm more scalable and efficient.

The Apriori Algorithm: Candidate Generation-and-Test Approach

Initial Scan: The database is scanned to find all frequent 1-itemsets. These are the itemsets that appear in at least a minimum number of transactions, known as the minimum support threshold.

Candidate Generation: For each level k, the algorithm generates candidate itemsets of length $k+1$ by joining frequent itemsets of length k with each other. This is done by combining itemsets that share a common prefix.

Testing Candidates: The candidate itemsets generated in the previous step are tested against the database to determine their support. Itemsets that meet or exceed the minimum support threshold are deemed frequent.

Pruning: Any candidate itemset that includes an infrequent subset is pruned, as it cannot be frequent by the Apriori principle.

Termination: The process repeats until no new frequent itemsets are found.

Steps of Apriori Algorithm

1. **Initial Scan:**
    - Scan the database to find frequent 1-itemsets.
    - Example: If the database contains transactions such as {A, B, C}, {A, C}, {B, C}, {A, B}, and the minimum support is 2, the frequent 1-itemsets are {A}, {B}, {C}.
2. **Candidate Generation:**
    - Generate candidate 2-itemsets from the frequent 1-itemsets.
    - Example: From {A}, {B}, {C}, generate {A, B}, {A, C}, {B, C}.
3. **Testing Candidates:**
    - Scan the database to find the support for each candidate 2-itemset.
    - Example: If {A, B} appears in 2 transactions, {A, C} appears in 2 transactions, and {B, C} appears in 3 transactions, and the minimum support is 2, all these 2-itemsets are frequent.
4. **Pruning:**
    - Use the Apriori principle to prune any candidate that contains an infrequent subset.
    - Example: If {A, B} was infrequent, any candidate containing {A, B} would be pruned in the next iteration.
5. **Repeat:**
    - Continue generating and testing candidate itemsets of increasing length until no new frequent itemsets are found.
    - Example: Generate candidate 3-itemsets from {A, B}, {A, C}, {B, C}.

Complexity

The Apriori algorithm, while foundational in the field of association rule mining, is well-known for its computational complexity and resource demands. At its core, Apriori operates on the principle of generating and testing candidate itemsets. This iterative process involves multiple passes over the dataset, where each pass increases the length of the itemsets by one.

- **Combinatorial Explosion:** One of the main sources of complexity in Apriori is the combinatorial explosion of candidate itemsets. As the number of items in the dataset increases, the number of potential itemsets grows exponentially. For instance, a dataset with n items can generate up to $2^n - 1$ possible itemsets. In each iteration, the algorithm generates new candidate itemsets by joining frequent itemsets from the previous iteration. This not only increases the computational burden but also requires substantial memory to store and process these candidates.
- **Multiple Database Scans:** Another significant factor contributing to the complexity is the need for multiple database scans. For each iteration k, the Apriori algorithm scans the entire database to count the support of the candidate k-itemsets. This repeated scanning can be highly inefficient, especially for large datasets, leading to increased I/O operations and longer execution times. Each scan involves reading the entire dataset, which can be a substantial overhead when dealing with large volumes of data.
- **Support Counting:** Counting the support for each candidate itemset is also computationally intensive. As the length of the itemsets increases, the number of candidates to be evaluated grows, requiring more comparisons and more memory to maintain frequency counts. This process becomes particularly burdensome as the dataset size and the number of items increase.

Parallelism

To mitigate the inherent complexity and improve the performance of Apriori, parallelism can be leveraged. By distributing the computational load across multiple processors or machines, the algorithm can significantly reduce execution time and resource usage.

- **Data Partitioning:** One approach to parallelism in Apriori is data partitioning. The dataset is divided into smaller, non-overlapping partitions, and each partition is processed independently in parallel. In the first phase, local frequent itemsets are identified within each partition. These local results are then combined in a second phase to determine the global frequent itemsets. This method reduces the number of candidate itemsets generated and the number of database scans required, as each partition can be processed concurrently.
- **Task Parallelism:** Task parallelism involves dividing the candidate generation and support counting tasks among multiple processors. For example, in each iteration, different processors can handle different subsets of candidate itemsets, performing support counting in parallel. This approach helps to balance the computational load and reduces the time required for each iteration.
- **MapReduce Framework:** Using distributed computing frameworks like MapReduce can further enhance the scalability and efficiency of Apriori. In a MapReduce implementation, the Map function processes individual transactions to generate local counts of itemsets, while the Reduce function aggregates these local counts to determine global supports. This framework is particularly effective for handling large-scale data distributed across multiple nodes, providing fault tolerance and scalability.

- **Hybrid Approaches:** Combining data and task parallelism can yield even greater performance improvements. For instance, data can be partitioned across nodes in a distributed system, while within each node, multiple processors can parallelize the support counting and candidate generation tasks. This hybrid approach maximizes resource utilization and minimizes the bottlenecks associated with each individual method.

Improving Apriori: Partition, DIC, DHP

Partition Method

The Partition Method addresses the resource-intensive nature of the Apriori algorithm by breaking down the dataset into smaller, more manageable segments. This approach not only reduces computational complexity but also enhances scalability and efficiency, especially in large datasets.

Partition Data into Small Partitions:

- **Process:**
  - The entire dataset is divided into multiple smaller partitions, each of which can be processed independently. This segmentation is crucial for reducing the overall problem complexity by allowing localized analysis before global integration.
  - Each partition is treated as a separate dataset, enabling parallel processing and reducing the need for repetitive database scans.
- **Phase 1: Find Local Frequent Itemsets, Local Analysis**
  - Within each partition, the algorithm identifies frequent itemsets that meet the minimum support threshold specific to that partition. This localized approach simplifies the computation by focusing on smaller subsets of data.
  - Example: If a partition contains transactions {A,B,C}{A, B, C}{A,B,C}, {A,C}{A, C}{A,C}, and {B,C}{B, C}{B,C}, the algorithm will determine which itemsets are frequent within just this partition.
- **Phase 2: Integrate Local Frequent Itemsets, Global Integration**
  - After identifying local frequent itemsets in each partition, these itemsets are combined to form a candidate set of global frequent itemsets.
  - A second scan of the entire database is then performed to verify and count the support of these combined itemsets across all partitions. Itemsets that meet the global minimum support threshold are considered globally frequent.
  - Example: Local frequent itemsets from partitions are {A,B}{A, B}{A,B} and {A,C}{A, C} {A,C}. The global integration phase confirms which of these are frequent across the entire dataset.

Advantages:

- Scans Database Only Twice: This method reduces I/O operations by limiting the number of full database scans to just two. The first scan identifies local frequent itemsets, and the second scan validates global frequent itemsets. This is a significant improvement over the traditional Apriori algorithm, which may require multiple scans for each iteration.
- Suitable for Parallel/Distributed Systems:

- The independence of partition processing makes this method ideal for parallel and distributed computing environments. Each partition can be analyzed concurrently, leveraging multiple processors or distributed nodes, thus enhancing performance and scalability.
- In a distributed system, each node can process a partition independently, and results can be aggregated to form the final global frequent itemsets.

Direct Hashing and Pruning (DHP)

Direct Hashing and Pruning (DHP) optimizes the Apriori algorithm by incorporating a hash-based technique to reduce the number of candidate itemsets early in the process.

Hash-Based Algorithm:

- Hash $(k + 1)$-Itemsets into Buckets During Scanning:
  - As candidate (k+1)(k+1)(k+1)-itemsets are generated, they are hashed into buckets. Each bucket maintains a count of the itemsets that hash into it, providing an efficient way to track potential candidates.
  - Example: If itemsets {A,B}{A, B}{A,B}, {A,C}{A, C}{A,C}, and {B,C}{B, C}{B,C} are hashed, their respective counts are stored in corresponding buckets.
- Remove Candidates Whose Corresponding Bucket Count is Below the Threshold:
  - If the count in any bucket is below the minimum support threshold, all (k+1)(k+1)(k+1)-itemsets hashing to this bucket are pruned from consideration. This early pruning step eliminates a significant number of non-promising candidates, streamlining the process.
  - Example: If a bucket for {A,B,C}{A, B, C}{A,B,C} has a count less than the threshold, all itemsets hashing to this bucket are discarded.

Advantages:

- Reduces the Number of Candidates:
  - By hashing and pruning candidates early, the number of itemsets that need to be tested and counted is significantly reduced. This reduction in candidates decreases computational overhead and improves algorithm efficiency.
  - Example: Instead of testing all possible k+1k+1k+1-itemsets, only those in buckets meeting the threshold are considered.
- Execution Time Scales Linearly with Data Size:
  - The use of hashing ensures that execution time grows linearly with the size of the dataset, enhancing scalability. As the dataset size increases, the hashing mechanism efficiently manages the growing number of itemsets.

Transaction Reduction

Transaction Reduction is a technique that improves the efficiency of the Apriori algorithm by reducing the number of transactions considered in each iteration.

Technique:

- Mark Transactions Containing No kkk-Candidates:
  - During each iteration, the algorithm identifies and marks transactions that do not contain any frequent kkk-itemsets. These transactions are irrelevant for finding k+1k+1k+1-itemsets and can be excluded from further analysis.

    - Example: If a transaction {D,E}{D, E}{D,E} contains no frequent kkk-itemsets, it is marked for exclusion.
- Remove Marked Transactions:
    - Marked transactions are removed from the dataset for subsequent iterations. This reduction in the number of transactions helps in speeding up the computation by focusing only on relevant data.
    - Example: Transactions {D,E}{D, E}{D,E} and similar are removed from the dataset, reducing the size of the dataset for the next iteration.

Advantages:

- Reduces Dataset Size:
    - By eliminating transactions that do not contribute to finding frequent itemsets, the effective size of the dataset is reduced. This reduction directly decreases the computational load and memory usage.
    - Example: A dataset reduced from 1000 to 800 transactions means fewer comparisons and support counts.
- Increases Efficiency:
    - With fewer transactions to process, the number of comparisons and support calculations is reduced, thereby increasing the overall efficiency of the algorithm. This leads to faster iterations and quicker convergence to the final set of frequent itemsets.
    - Example: Faster processing times due to fewer transactions needing analysis.

Dynamic Itemset Counting (DIC)

Dynamic Itemset Counting (DIC) enhances the flexibility and efficiency of the Apriori algorithm by allowing dynamic adjustments during the mining process.

Technique:

- Add New Candidates at Any Starting Point:
    - Unlike traditional methods where candidates are fixed per iteration, DIC allows new candidate itemsets to be introduced at any point in the database scan. This flexibility ensures that new promising candidates can be tested as soon as their subsets are found to be frequent.
    - Example: If during the scan, {A,B}{A, B}{A,B} is found frequent, {A,B,C}{A, B, C}{A,B,C} can be tested immediately without waiting for the next iteration.
- Reduces the Number of Database Scans:
    - By dynamically introducing and counting new candidates, the total number of full database scans is reduced. Multiple itemset counts can be maintained simultaneously, making the process more efficient.
    - Example: Instead of multiple scans for each level, fewer scans suffice to count multiple itemsets concurrently.

Advantages:

- Fewer Database Scans:
    - The dynamic introduction of candidates reduces the total number of passes over the database. This significant reduction in I/O operations lowers overall computational costs and improves performance.

    - Example: Instead of 5 scans, only 3 scans may be required with DIC.
- Flexible and Dynamic:
    - DIC's ability to adapt to the data distribution and introduce new candidates dynamically makes the algorithm more responsive to variations in the dataset. This flexibility can lead to earlier and more efficient discovery of frequent itemsets.
    - Example: Immediate testing of promising itemsets leads to quicker identification of frequent patterns.

Projection-Based (FP-Growth)

FP-tree Construction

FP-growth (Frequent Pattern Growth) is a highly efficient algorithm for mining frequent patterns, addressing the limitations of the Apriori algorithm. It employs a compressed representation of the database called an FP-tree (Frequent Pattern Tree) to avoid the combinatorial explosion of candidate generation. Here's a detailed explanation of how FP-trees are constructed and their benefits:

- **Scan Database to Find Frequent 1-itemset:**
    - **Initial Scan:** The first step involves scanning the entire database to count the frequency of each individual item. This initial scan helps to identify the items that meet or exceed the minimum support threshold.
    - **Example:** For transactions {A,B,C},{A,C},{B,C},{A,B}{A, B, C}, {A, C}, {B, C}, {A, B}{A,B,C}, {A,C},{B,C},{A,B}, if the minimum support threshold is 2, the frequent 1-itemsets are {A}, {B},{C}{A}, {B}, {C}{A},{B},{C}.
- **Sort Frequent Items in Descending Order of Frequency:**
    - **Ordering Items:** After identifying the frequent 1-itemsets, the items in each transaction are sorted by their overall frequency in descending order. This ensures that more frequently occurring items are processed first, leading to a more compact tree structure.
    - **Example:** If item A appears most frequently, followed by B and C, then the sorted order of items in each transaction will prioritize A first, then B, then C.
- **Create Tree Branches for Transactions:**
    - **Path Creation:** For each transaction, a path is created in the FP-tree. If a part of the transaction's path already exists in the tree, the count of existing nodes is incremented. Otherwise, new nodes are created to represent the items in the transaction.
    - **Example:** For transactions {A,B}{A, B}{A,B}, {A,C}{A, C}{A,C}, and {B,C}{B, C}{B,C}, the tree branches will share common prefixes where possible, such as nodes representing A and B being shared between the first two transactions.
- **Build Header Table Linking Each Item in the Tree:**
    - **Header Table:** A header table is created to keep track of the first occurrence of each item in the FP-tree. This table links all occurrences of the same item together via node-links, facilitating efficient traversal of the tree for mining frequent patterns.
    - **Example:** The header table will link all nodes representing item A, then B, and so on, allowing quick access to all nodes containing the same item.

Benefits of FP-tree Structure

- **Preserves Complete Information for Frequent Pattern Mining:**

    - **Comprehensive Representation:** The FP-tree maintains all necessary information to mine frequent patterns without the need for multiple database scans. Each path in the tree represents a set of transactions sharing common prefixes, ensuring that no information is lost.
    - **Efficiency:** This comprehensive representation allows for efficient pattern mining, as the structure of the tree facilitates quick access to frequent itemsets.
- **Compact Representation Reduces Irrelevant Information:**
    - **Size Reduction:** The FP-tree only includes frequent items, significantly reducing the size of the tree compared to the original dataset. Irrelevant (infrequent) items are pruned early, focusing the mining process on significant data.
    - **Example:** If items D and E are infrequent, they are excluded from the FP-tree, reducing its complexity and size.
- **More Frequently Occurring Items Are More Likely to Be Shared:**
    - **Shared Structure:** Common prefixes in transactions are shared in the tree structure, leading to a more compact and efficient representation. This shared structure allows for efficient traversal and pattern discovery.
    - **Example:** Transactions sharing items A and B will have a common path in the tree, reducing redundancy and improving efficiency.

Conditional Pattern Bases and FP-trees

To mine the frequent patterns efficiently, FP-growth uses conditional pattern bases and conditional FP-trees, further breaking down the problem into smaller, manageable subproblems.

- **Constructing Conditional Pattern Bases**, Traverse FP-tree to Accumulate Transformed Prefix Paths:
    - For each item in the header table, traverse the FP-tree to collect all paths (prefixes) leading to nodes containing that item. These paths constitute the conditional pattern base for that item.
    - **Example:** If constructing the conditional pattern base for item B, collect all paths ending in B. If the paths are {A,B}{A, B}{A,B} and {C,B}{C, B}{C,B}, they form the conditional pattern base for B.
- **Building Conditional FP-trees,** Construct FP-tree for Frequent Items in the Pattern Base:
    - Use the conditional pattern base to construct a new, smaller FP-tree (conditional FP-tree) for each frequent item. This conditional FP-tree is built similarly to the original FP-tree, focusing only on the items in the conditional pattern base.
    - **Example:** The conditional FP-tree for item B will be constructed from the collected paths (prefixes) of B, maintaining the same counting and linking principles, ensuring efficient mining of frequent patterns involving B.

Advantages of FP-growth over Apriori

FP-growth provides several advantages over the traditional Apriori algorithm, making it a more efficient and scalable solution for frequent pattern mining.

- **Divide-and-Conquer Approach:**

    - **Subproblem Decomposition:** FP-growth divides the mining process into smaller, more manageable subproblems by creating conditional pattern bases and conditional FP-trees. This divide-and-conquer approach allows for efficient handling of large datasets and complex patterns.
    - **Example:** By breaking down the problem into conditional FP-trees, the algorithm can focus on smaller portions of the dataset, improving overall efficiency.
- **No Candidate Generation or Testing:**
    - **Efficiency:** Unlike Apriori, FP-growth does not generate candidate itemsets and then test them against the database. This eliminates the need for multiple scans and extensive candidate generation, significantly reducing computational overhead.
    - **Example:** Directly using the FP-tree for mining frequent patterns avoids the combinatorial explosion of candidate itemsets.
- **Compressed Database Representation:**
    - **Compact Form:** The FP-tree structure provides a highly compressed representation of the original dataset. This compact form facilitates efficient traversal and frequent pattern mining.
    - **Example:** The FP-tree condenses a large number of transactions into a tree structure, making it easier to mine frequent patterns without redundant processing.
- **Only Two Scans of the Entire Database:**
    - **Reduced I/O Operations:** FP-growth requires only two full scans of the database: one to determine the frequent 1-itemsets and another to construct the FP-tree. This is a substantial improvement over Apriori, which may require multiple scans for each level of candidate generation.
    - **Example:** The reduction in database scans minimizes I/O operations, leading to faster and more efficient mining.