Computing and Informatics, Vol. 38, 2019, 473–496, doi: 10.31577/cai_2019_2_473

MINING QUERY PLANS FOR FINDING CANDIDATE QUERIES AND SUB-QUERIES FOR MATERIALIZED VIEWS IN BI SYSTEMS WITHOUT CUBE GENERATION

Atul Thakare, Srijay Deshpande
Amit Kshirsagar, Parag Deshpande

Department of Computer Science and Engineering
Visvesvaraya National Institute of Technology
South Ambazari Road, Nagpur, Maharashtra
India - 440010
e-mail: {aothakare, deshpandesrijay, amit.kshirsagar185}@gmail.com,
psdeshpande@cse.vnit.ac.in

Abstract. Materialized views are important for optimizing Business Intelligence (BI) systems when they are designed without data cubes. Selecting candidate queries from large number of queries for materialized views is a challenging task. Most of the work done in the past involves finding out frequent queries from the past workload and creating materialized views from such queries by either manually analyzing workload or using approximate string matching algorithms using query text. Most of the existing methods suggest complete queries but ignore query components such as sub queries for creation of materialized views. This paper presents a novel method to determine on which queries and query components materialized views can be created to optimize aggregate and join queries by mining database of query execution plans which are in the form of binary trees. The proposed algorithm showed significant improvement in terms of more number of optimized queries because it is using the execution plan tree of the query as a basis of selection of query to be optimized using materialized views rather than choosing query text which is used by traditional methods. For selecting a correct set of queries to be optimized using materialized views, the paper proposes efficient specialized frequent tree component mining algorithm with novel heuristics to prune search space. These frequent components are used to determine the possible set of candidate queries for creation of materialized views. Experimentation on standard, real and synthetic data sets, and also the theoretical basis, proved that the proposed method is able to optimize a large number of queries with less number of materialized views and showed a significant improvement in performance compared to traditional methods.

Keywords: Query optimization, view selection, tree mining, query plans, materialized views, query response time

Mathematics Subject Classification 2010: 68Uxx

1 INTRODUCTION

Most of the BI systems are implemented using data cube architecture. Generation of data cubes for processing user queries helps in reducing time, but has storage and data synchronization overheads. In some cases, storage overheads are so large that it becomes extremely prohibitive to generate a cube. If it is not feasible to generate a cube, the user queries are processed using on the fly aggregation. The "on the fly" aggregation demands very good query optimization, which otherwise would lead to high response time for user queries. Query optimization [31] plays a vital role in such BIS. Most of the time, the query optimization is done using techniques like indexing, but unfortunately this technique is not able to optimize aggregate queries. Hence, the idea of materialized views has been proposed to optimize such queries. A materialized view [17] is like a normal view with storage used for storing results of a view query. When a materialized view is referred, rows are directly retrieved from the storage rather than the execution of the query again, thereby reducing the processing time of the query. The stored rows of materialized view are refreshed when base tables of materialized views are updated to keep the data synchronized. Thus, materialized views have data synchronization costs which may reduce the overall advantage of improving query response time [29]. If the system has many materialized views, then the performance of the system deteriorates due to high data synchronization overheads. Therefore, it is necessary to have a minimum number of materialized views to improve the queries, which otherwise cannot be optimized by conventional methods.

The aim of this paper is to create a set of materialized views with minimum cardinality, which can optimize most of the queries. Our approach is to find frequent components in queries and create materialized views on them to optimize them. Such frequent components may represent frequent subqueries. The traditional approaches like approximate string matching algorithm [2] will not be useful here because similar queries may not have the same text. For example, if two different queries have the same subquery, then by using normal string matching technique, the queries are treated differently. Therefore, we propose a new approach of finding frequent components in queries by analyzing the query execution plan rather than query text using data mining techniques. Most of the database management systems construct query execution plan in the form of a binary tree. This paper uses this property to build a recursive tree mining algorithm to find the frequent components of a query. In most of the applications, the query workload follows 80–20 rule, i.e., almost 20 % queries form 80 % of the work load. As the number of queries increases, the probability of a query component to repeat also increases. The creation of materialized views on these frequent components will result in the overall optimization of queries that reuse the same components. Generalized graph mining algorithms like G-Span are already developed to obtain frequent graphs from a memory dump of graphs [30]. However, this is a highly generalized algorithm and such conventional graph mining algorithms are not useful for mining the "execution plan tree components" because of the specialized nature of these trees. Node in the tree represents an operation, whereas the level of the node indicates the order in which the operation is performed. In this paper, we have proposed a new tree mining algorithm for identifying the frequent tree components in a set of such specialized trees.

The algorithm proposed in this paper analyzes multiple queries and recommends queries as well as query components. Creation of materialized views on these components will result in the optimization of all the queries having these components. Many database systems contain hundreds and even thousands of tables. Such database applications may have millions of queries [1]. These queries may have many frequent components. Mining of frequent components that can be translated as candidate queries is a challenging task [4]. The proposed tree mining algorithm for finding these components should be able to handle the work load by pruning the search space efficiently.

For a given workload, we have found that the candidate queries using our method are more in number as compared to the queries resolved using the conventional method. We have also shown that the materialized views created using candidate queries used by our algorithm show a considerable improvement in terms of "reduction of the logical block reads (GAIN MEASURE)" as a performance measure [12]. The contributions of the proposed work are as follows:

1. We have introduced the creation of a materialized view based on query components and subqueries rather than creating it only on the full query.
2. We have proposed a method of finding query components by analyzing execution plans of past query workload rather than analyzing query texts. This is a fundamental change in approach as compared to traditional methods because it helps in getting a larger set of queries which can be optimized with a lesser number of materialized views and thus improving system performance to a very large extent.
3. We have proposed new tree mining algorithms for specialized trees, which represent query execution plans to handle large query loads by providing efficient pruning techniques to reduce search space. We have also provided correctness proof of our newly designed algorithm and proved that it will mine all necessary frequent subcomponents from the given set.

4. We have done exhaustive experimentation on standard, real and synthetic workload to show that the number of candidate queries reported by our algorithm is expected to be higher than the number of candidate queries reported by the state of the art algorithms by the traditional methods.

The rest of the paper is organized as follows: Section 2 describes the related work, followed by Section 3 which describes the proposed work and the algorithm in detail. Section 4 describes how the output of the tree mining algorithm helps to create materialized views, Section 4 describes the experimentation and results and Section 5 contains the conclusion and future work.

2 RELATED WORK

View materialization is a widely-used strategy employed in data warehouses of the Business Information System to improve the performance of decision support queries. Decision support queries are highly complex in nature and make heavy use of joins and aggregations. Moreover, solving decision support queries involves computations on huge volumes of historical data, as these queries are more inclined to find trends rather than individual facts. Historical data is continuously generated by multiple fast operational OLTP (Online Transaction Processing) systems and gets accumulated in warehousing systems. Since access to a materialized view is faster than computing the views on demand, using the materialized view can speed up the analytical query processing in a data warehouse. Hence, naturally it is desirable to materialize all the possible views in a data warehouse, but this is not feasible because of resource constraints such as disk space, computation time and maintenance cost. Especially, creation of materialized views incurs an overhead after update of the base database objects which demands refreshing of all the affected materialized views.

Hence, to acquire a quick response to analytical queries within the system's resource constraints, selection of a proper set of views to materialize is an important decision while designing the warehousing system. The most commonly used technique is materializing frequent queries, which are obtained by text matching [13].

Gong in his paper [13] proposed clustering based dynamic materialized view selection algorithm (CBDMVS). It finds a cluster of SQL queries using a similarity threshold τ and if a new query's similarity is below τ for all the existing clusters, then a new cluster is formed. Similarity between queries is measured based on certain parameters like base table sets, equivalence connectivity conditions, scope connectivity conditions and output column sets. These queries are mined using text mining. CBDMVS dynamically adjusts the materialized view set, by replacing views with lowest gains where the system lacks storage space for the new query. Basically, it not only improves the overall query response time, but also reduces the computational cost that is spent while updating materialized view. Rajyalakshmi in paper [26] proposed association rule mining based materialized view selection algorithm (ARMMVVM) for improving the performance of materialized view selection and materialized view maintenance using association rule mining. It integrates the technique of improving query response time by using frequent mining algorithm along with adjustments of the view set.

Sohrabi and Ghods in their paper [28] explored the view selection problem as a two-step process where, the first step is finding the candidate views and the second step decides the final view set from a set of candidate views under the system resource constraints such as storage space and view maintenance cost. This paper discussed the usage of Directed Acyclic Graphs (DAGs) and data cube lattice in candidate generation step, and various heuristic algorithms in the view selection step. The authors also proposed a novel algorithm based on frequent item set mining technique which aims at minimizing the view creation and maintenance cost. Paper [14] proposed a systematic review of various view selection techniques in which various techniques are compared in terms of memory storage space, cost, and query processing time. By means of this review of available literature, the authors have drawn several conclusions about the status quo of materialized view selection and a future outlook is predicted on bridging the large gaps that were found in the existing methods.

In paper [27], the authors proposed a greedy materialized view selection algorithm, which extracts query-processing and view maintenance cost related information from multiple query processing plans into a table-like structure and the algorithm also computes the optimal view set. In paper [21], the authors proposed a similarity interaction operator-based particle swarm optimization (SIPSO), in which materializing an appropriate subset of views was suggested for achieving acceptable response times for analytical queries. The proposed SIPSO-based view selection algorithm (SIPSOVSA) selects the Top-K views from a multidimensional lattice. Paper [5] proposed a game theory based framework for the materialized view selection. In the proposed framework, query processing and view maintenance costs play a game against each other as two players and continue the game until they reach the equilibrium.

The authors in paper [20] talk about a uniform query framework that can be used for traditional relational databases and NOSQL databases. This query framework can also perform joins, aggregates, filter on the data from various data sources in a single query. Hung et al. in their paper [18] proposed a cost model, having well-defined gain and loss metrics used for deciding the member views in a view set. For candidate generation, data cube is represented as lattice, and lattice is expressed in the vector form. This vector is then used to search for other dependent views. Afrati in the paper [1] addresses the problem of view selection for aggregate queries considering rewritings with multiple view sub-goals and multi-aggregate views. This paper explains how to answer aggregate queries using aggregate views by constructing equivalent rewritings and how to optimally select aggregate views to materialize, for use in those rewritings.

The authors in the paper [15] present a greedy view selection algorithm in AND/OR view graphs, which describes all possible ways to generate warehouse views. It describes different approaches to address the view selection problem, selects the best query path which can be maximally utilized to optimize the response time of most of the queries, under the maintenance cost constraints. In the paper [25] the authors present the heuristics to determine the additional set of views to materialize under given storage constraints to reduce the overall maintenance cost of all the views. The algorithms aim at minimizing the query response time and view maintenance overheads under the given storage constraints. The paper [19] proposes a greedy-repaired genetic algorithm which selects a set of materialized cubes from the data cubes under storage space constraints, in order to reduce the amount of query cost as well as the cube maintenance cost.

In a paper [22] authors gave importance on integration of computational methods for design optimization based on data mining and knowledge discovery. This paper proposes to use radial basis function neural networks to analyze the large database generated from evolutionary algorithms and to extract the cause-effect relationship, between the objective functions and the input design variables. Gupta and Mumick in the paper [16] developed a method to deliver the optimal set of views to optimize the total query response time for a given workload under constraints that the selected set of views should incur less collective maintenance overheads than the specified amount of maintenance time. The paper proposed approximation greedy algorithm for query load having view OR graphs and A* heuristics algorithm for query load with general AND-OR view graphs. This paper has kept the storage constraints out of all the equations.

The authors in the paper [3] proposed a framework for materialized view selection that exploits data mining technique (clustering), to determine clusters of similar queries. The paper also proposed a view merging algorithm that builds a set of candidate views, by iteratively building the lattice of views. To determine the final view set, a greedy process was used where the selection criteria considered cost of storing and accessing data from views. Ashadevi in her paper [4] presented a critical survey of the past and present methodologies and solutions for the view selection problem.

Mohania and Kambayashi in the paper [24] showed that the warehouse views could be made self-maintainable if additional auxiliary relations were derived from the intermediate results of view computation in the warehouse. This paper proposed an algorithm for determining what auxiliary relations needed to be materialized to make a materialized view self-maintainable, i.e., maintainability could be viewed as an incremental process that computes the updates to both the materialized view and the additional relations.

Daneshpour and Barfourosh in their paper [9] proposed a dynamic view management system to select materialized views with new and improved architecture, which could predict incoming queries through association rule mining and three probabilistic reasoning approaches: Conditional probability, Bayes' rule, and Naïve Bayes' rule.

3 PROPOSED WORK

3.1 Problem Statement

A materialized view is a proven technique to optimize queries such as aggregate queries which otherwise cannot be optimized using conventional optimization techniques like indexing or clustering. Since materialized views have additional storage and data synchronization overheads it is better to create less number of materialized views. If a query load consists of millions of queries, it is very costly to create materialized view. It is reasonable to use less number of materialized views for optimizing computationally intensive queries.

3.2 Theoretical Analysis and Motivation

Matching query text can be used to find out frequent queries, but this approach may not work if the queries are written differently or, if several queries are doing similar operations. For example, the following queries, i.e. queries Q1 and Q2, are computationally similar to query Q3, but their textual representations are different.

Q1: select avg (salary) from emp_company group by cname;

Q2: select max (salary) from emp_company group by cname;

Q3: select max (salary), avg (salary) from emp_company group by cname.

Unlike the tree mining algorithm, the string matching algorithm will not be able to find any correlation between these queries. But, if we create materialized view for the query Q3, both the queries Q1 and Q2 will get optimized. Another example could be queries that are different, but they use the same subqueries. For example, the following queries are different, but use the same subquery.

Q4: select ename from emp_company where salary > all (select avg (salary) from emp_company);

Q5: select cname from emp_company where salary > all (select avg (salary) from emp_company).

Both the queries are using the same subquery, but they are not textually similar. Both can be optimized by creating materialized view of the subquery. The text matching algorithm which is used by traditional researchers will fail to detect such kind of commonality amongst the queries Q4 and Q5. To overcome this problem we have used the execution plan tree as a basis for finding similarity between trees and detecting similar subqueries because of the following facts.

1. If the two queries are similar, then their execution plans trees are also the same.
2. If the two queries are having the same subquery, then their execution plan trees are having the same subtree components.

With tree mining, we will be able to find a common subtree in the query plan trees of Q4 and Q5. Therefore, the problem statement is defined as "To find a set of materialized view queries" [4] from the existing query load by mining database of query execution plans such that:

1. The set of materialized views should optimize a large number of time consuming queries.
2. The set should have low cardinality to avoid storage overheads and data synchronization costs.

Once "execution plan tree" is decided as the basis of similarity then the next challenge is to design algorithm which efficiently mines frequent tree components from the set of millions of trees. General frequent itemset mining or graph mining algorithms cannot be applied because of specialized nature of plan trees. So the specialized frequent tree algorithm is designed by providing heuristics and correctness proof of the algorithm is provided.

3.3 Terminologies and Examples

In this section, we first define a few terminologies used for explaining our tree mining algorithm which is used to mine database of query execution plans which are in the form of binary trees.

View: A view is a derived relation, defined by a query in terms of base relations and/or other views.

Materialized view: A view is said to be materialized if its query result is persistently stored, otherwise it is said to be virtual. We refer to a set of selected views to materialize as a set of materialized views.

Workload: A workload or a query workload is a given set of queries, $Q = \{Q_1, Q_2, \dots, Q_n\}$. Each query in the query workload can be described using its frequency and cost of execution.

View selection: Given a database schema and a query workload, the objective is to select an appropriate set of materialized views to improve performance of database in processing the workload, i.e. in executing queries in the workload. The ideal view set can comprise queries which are useful in optimizing the performance of a large number of queries in the workload.

Tree: Tree is a directed acyclic graph denoted as $T(R, V, L, E)$, where V is the set of nodes in T; R is one of the node of T and is the root of T; L is the set of the labels of the nodes; and E is the set of directed edges in T. All trees considered in this paper are rooted labeled trees.

Query and query plans: Every query has a query plan associated with it. A query plan shows the execution path of the query [12]. Most of the Database Management Systems (DBMS) use binary trees to represent the query execution plans where leaves indicate the data source and node indicates the type of operation such as join. Though we have designed an algorithm based on the assumption that the query plan is in the form of binary tree, the algorithm can be easily extended for generalized plan tree. Figure 1 indicates query plan for the following query.

Select e.ename, e.city from employee e where e.ename in (select c.ename from emp_company c where c.cname = 'ACC' and c.salary > (select avg (salary) from emp_company)).

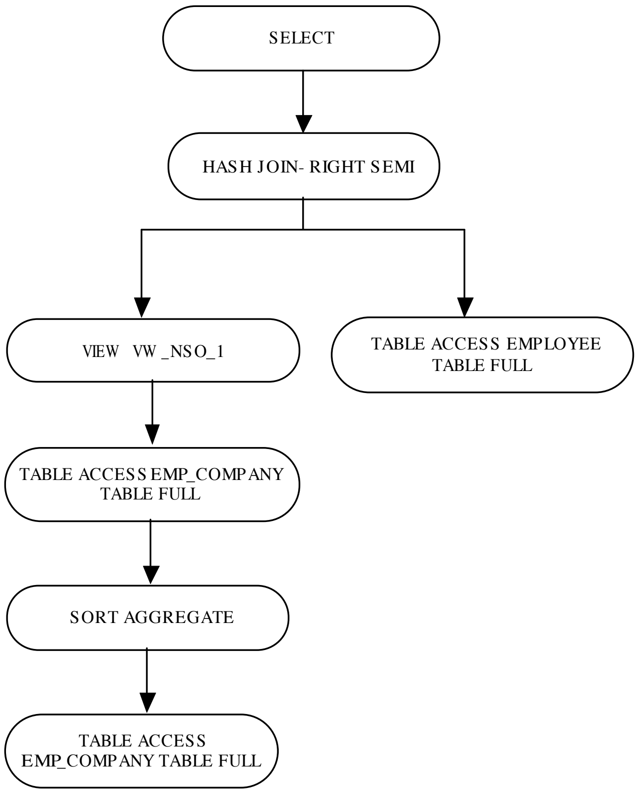
SELECT
HASH JOIN- RIGHT SEMI
VIEW VW_NSO_1
TABLE ACCESS EMPLOYEE TABLE FULL
TABLE ACCESS EMP_COMPANY TABLE FULL
SORT AGGREGATE
TABLE ACCESS EMP_COMPANY TABLE FULL

Figure 1. Query plan for the query

Each node in the query plan tree represents an operation. We extract these query plans from all the queries present in the query load. These plans constitute the tree database (TDB) which is used as an input for the tree mining algorithm.

3.3.1 Query Subtree and Supertree

Consider the two trees $T(R, V, L, E)$ and $T'(R', V', L', E')$ based on tree definition in Section 3.2.5. Assuming T' as a subtree (embedded tree) of T ($T' \subset T$). It implies that $V' \subset V$, $E' \subset E$, $L' \subset L$, $L'(V) = L(V)$. If $(v_1, v_2) \in E'$ and v_1 is the ancestor of v_2 in T, then it is preserved in T' also. If T' is subtree of T then T is called a supertree of T'.

In our paper we are considering only those embedded trees whose set of leaf nodes is a subset of leaf nodes of a tree. Any embedded tree which does not terminate strictly at the leaf level of an enclosing supertree is not considered a valid subtree. Only such subtrees represent a valid query component like subquery or part of query on which materialized view can be created. Figure 2 indicates the subtree.

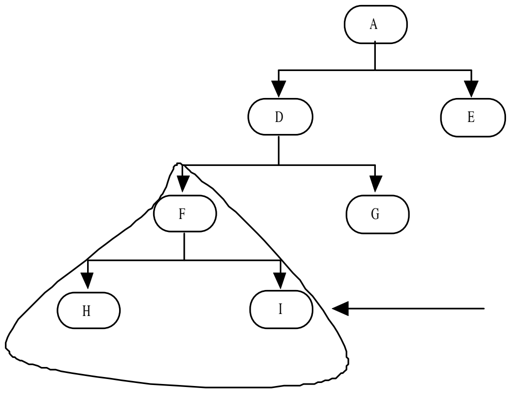

Figure 2. Subtree which represents a subexpression or a subquery

3.3.2 Tree Database (TDB)

It is a set of query execution plan trees collected from traces of database management system. The dataset containing query plans is in the form of binary trees. This data can be easily obtained from the trace utility provided by database management systems. The trace utilities normally dump data in relational tables which includes sql query text, query execution plans. Each query plan in the TDB is associated with an identifier (SQL_ID). It also provides information like number of logical reads, cost of query and number of executions of each query. All such information is available in the dynamic dictionary views which can be easily extracted. The tree database which is extracted from such views is referred to as the query workload.

3.3.3 Support

Given the tree database TDB, assuming a tree $T \in TDB$, and S' is a subtree of T, then support of subtree S' equals to the number of instances of S' in TDB including the instance(s) in T. The task of frequent subtree mining from TDB with given minimum support σ is to find all the candidate subtrees that has support at least equal to σ. The support for subtree given in Figure 3 in the database given in Figure 5 is 2.

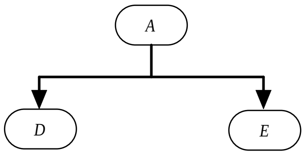

Figure 3. Support of a subtree (i.e. a subquery) is the number of its occurrences in all the query plans

The subtree A-D-E is present in two trees shown in Figures 5 and 6.

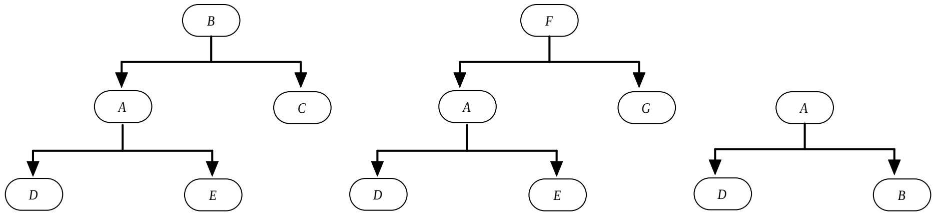

Figure 4. Sample tree dataset (4.1, 4.2, 4.3)

3.3.4 Threshold and Frequent Subtree

It is the minimum support value which qualifies the subtree to be get included into a list of frequent sub-trees. If the support of a sub-tree is greater than or equal to the specified threshold (% value of total trees), then that subtree is called a frequent subtree. The support of subtree S given in Figure 3 is 2. If threshold is 2, then sub-tree S is frequent. If the minimum support value is decreased then more trees will be qualified as frequent trees and more number of materialized views will be created. If more materialized views are present then the high synchronization cost of the materialized views will have adverse impact on performance so the minimum support value should be carefully chosen according to requirement of application. In this paper, for experimentation, the support value is chosen as 50 % so that the optimization is done with less number of materialized views. If the application is having a large number of transactions then this value can be increased to create less number of views, and if the application is having only a large number of retrievals where materialized view synchronization cost is negligible then the support count can be decreased to optimize more queries using materialized views. The value of 50 % allows the user to check performance and adjust it according to the nature of the application.

3.3.5 Maximal Frequent Subtree

The subtree S of any tree T is said to be maximal frequent subtree if S is frequent and there is no supertree S' of S in tree T such that S' is frequent. In other words, there does not exist any frequent tree whose subtree is S. If materialized view is created on maximal frequent subtree then large part of the query is optimized [30].

3.3.6 GAIN Measure (GM)

It is percentage reduction in the number of logical block reads after a query set is optimized using materialized views. For example, a user fires a query "Select sum (salary) from employee group by department_name" then it will require 2 000 logical block reads if the table "employee" occupies 2 000 blocks on the disk. If the materialized view is created on the same query then it will store department wise salary which will take very less space on the disk. If the size of materialized view is 10 blocks then the percentage reduction (GM) is $(1\,990/2\,000) * 100$.

3.4 Tree Mining Algorithm

The proposed algorithm for mining frequent subtrees uses a recursive bottom up approach, i.e., it traverses the search space in a bottom up manner starting from the leaf. The method to mine the components is given below.

```
Input
TDB: Set of query execution plans
δ: Support Threshold
Output
FTDB: The list of frequent components (Subtrees) in trees and list of queries
associated with them.
```

3.4.1 Preprocessing

As, all the queries do not require optimization using materialized view, queries which have very less cost, or which are very infrequent do not need to be optimized using materialized views. To reduce the query load, the following preprocessing is done.

- Query cost is calculated in terms of the number of logical block reads. The query is placed in the experimental load if (number of logical reads) $*$ (frequency of query) is greater than some threshold (theta).
- If the base tables of queries are frequently updated then refresh cost of materialized view is high and such queries can be removed from experimental load.
- Within experimental load the queries are ordered using level of execution plan tree.

```
Variables
TreeNode: It is a data structure pointing to the node of the tree. All nodes in a tree that are the children of TreeNode are accessible from this data structure. In fact, each tree is represented by its root TreeNode.
TreeNode → Left: Left node of TreeNode
TreeNode → Right: Right node of TreeNode
$N$: number of query plans in dataset TDB
$Tr$: Parameter required for Threshold Pruning. $Tr = N * ((1 - \sigma) \div 100)$
FrequentTreeMap<FrequentTree, List_Of_Sql_Ids>: Map of frequent subtrees, present along with their sql ids in which the subtrees are present.
TreeList: List in which all query plans are stored in the form of binary trees. In this list, all elements are TreeNodes [refer]. All TreeNodes are pointing to root of trees.
SQL_Id_List (T): List of all SQL_IDs of which sql queries that includes component subtree $T$.
MaximalFrequentSubtreeList: It is list containing all maximal frequent subtrees encountered.
```

```
Algorithm: FrequentTreeMiner
Preprocessing;
FrequentTreeMap = ();
MaximalFrequentSubtreeList = ();
begin
for $i = 0$; $i < N$; $i$++ do
    MineSubTree(TreeList[$i$]);                                ▷ (1)
end for
```

3.4.2 Analysis

For each query execution plan, all components are searched and if the component is frequent then it is inserted in the frequent tree set as given in step (4) of the algorithm. For each component, searching is done by calling function searchsubtree () in step (6), which tries to find out whether the tree component is a subtree of the existing tree and then the database of all the search trees is scanned as given in

```
Function: MineSubTree          ▷ The function MineSubTree returns a query list which contains the subtree $T$ only if $T$ is frequent subtree, otherwise it returns a null list.
Input
TreeNode: Tree which is to be tested as maximal subtree
Output
FrequentTreeMap: List of frequent subtrees
Status: Boolean variable to indicate whether specified tree is maximal subtree.
```

```
Function MineSubTree (TreeNode) return boolean
if $T$ is null then
    return true
else
    if $T$ is already subtree of any tree in MaximalFrequentSubtreeList then    ▷ (2)
        return true
    else
        if MineSubTree ($T \to$ left) and MineSubTree ($T \to$ right) then    ▷ (3)
            if CheckSubtreeIfFrequent ($T$) then
                FrequentTreeMap. Insert ($T, \text{SQL\_Id\_List}(T)$);    ▷ (4)
                return true
            else
                return false
            end if
        end if
    end if
end if
```

Function: CheckSubtreeIfFrequent (TreeNode) ▷ The function CheckSubtreeIfFrequent (TreeNode T) will check if support of the T is greater than threshold, i.e., if T is frequent.
Input
TreeNode: Tree representing query plan or component of query plan
Output
Status: Boolean variable indicating whether the given tree is frequent.

```
Function CheckSubtreeIfFrequent (TreeNode $T$) return boolean
startIndex = index for tree next to tree that contains component $T$.
count = 1;
for $i$ = startIndex; $i < N$; $i$++ do    ▷ (5)
    if searchSubtree (TreeList[$i$], $T$) == True then    ▷ (6)
        count = count + 1    ▷ (7)
    end if
end for
if count >= ceil ($Tr * N$) then return true; else return false;
end if
```

Function: searchSubtree (TreeNode T, TreeNode subT)
This function checks whether any tree subT is present in tree T (subT represents the TreeNode, i.e., root node of tree to be searched as explained earlier and similarly, T represents TreeNode, i.e., root of tree in which subT is to be searched).

step (5). Step (1) executes N (size of query database) times while step (3) which recursively searches for each component in tree executes $(L - 1)$ times where L is level of the tree. For each component, the database of query plans is searched in step (5) which will also be executed N times and each such search at step (6) takes time proportional to $(L - 1)$. Therefore, the total complexity of the algorithm is $(N * (L - 1))^2$. To get all frequent components, the usual method is to enumerate all components that are in the dataset, and count the support of these item sets, and decide whether they are frequent or not. However, when the number of distinct items is huge, the algorithm that explores the entire search space may be inefficient due to the exponential increase in permutations. To avoid this problem, we employ a few techniques to reduce the search space. Here L, level of the tree, is dependent on the user query and it cannot be controlled, therefore the only way to reduce the cost of the algorithm is to reduce "N" by considering only queries which require performance improvement. Thus, N can be reduced using the following steps.

(1) Threshold pruning: While determining whether a new component is frequent or not, after verifying T_r trees [where $T_r = N * (1 - \text{Threshold}/100)$] without a single instance of a new component, the new component is automatically considered infrequent. A component which does not occur even once in T_r trees (where T_r is $T_r = N * (1 - \text{Threshold}/100)$ and N is the number of trees in the dataset) is considered infrequent. This is because even if it occurs in every tree after T_r trees, it still will not cross the threshold. Thus, new components are not mined after T_r trees. This step can be introduced after step (7) as follows. If $(count < T_r)$ break.

(2) Bottom up pruning: If the component containing two children of a tree node are infrequent then the component containing their parent will also be infrequent. So, while parsing any tree, suppose we find one node infrequent, we do not need to consider its parent, resulting in pruning the search space significantly. This is implemented in step (3).

(3) Maximal frequent lookup pruning: If a component is already frequent then there is no need to check it again as a lookup list of all maximal frequent subtrees is maintained. Every time a component is encountered, it is first searched in the look up list to check if it is a subtree of any other maximal frequent subtree, if it is found in the list, no further checks are carried out to find it is frequent or not, thereby reducing the search space drastically. In absence of the above mentioned look up lists, for every instance of each frequent component, we would have to search in the complete tree database resulting in huge inefficiency for large datasets. This is implemented in step (2).

(4) Filtering based on data source: Step (3) ensures that all the query components are searched and "for loop" specified in step (1) ensures that the whole database of query plans is searched so that the algorithm ensures that if some component is frequent then it is stored in the frequent set. If the component is not frequent then it will not be inserted in the frequent set as given in step (4).

The algorithm ensures that the output frequent set contains all possible frequent components in the database and components that are not frequent will never get a place in the output set.

3.5 Candidate Queries for Materialized View Creation

The aim of the proposed method is to find candidate queries for generating the materialized views, which otherwise cannot be obtained by conventional state of the art methods. The tree mining algorithm discussed in the previous section gives frequently occurring subtrees (components) and list of queries associated with each frequent subtree as an output. These frequent components are then analyzed and used to create materialized views. The following examples shows how the candidate queries are obtained by the algorithm which otherwise cannot be obtained by conventional methods. For example, suppose there are two queries:

Query 1: select e.cname from emp_company e where e.salary > (select avg (salary) from emp_company) group by cname;

Query 2: select e.ename, e.city from employee e where e.ename in (select c.ename from emp_company c where c.cname = 'ACC' and c.salary > (select avg (salary) from emp_company)).

The execution plans of both the queries are shown in Figures 5 and 6, respectively, with the frequent component associated with them.

The materialized view mv2 is created on this frequent component as of the queries on the schema of the data warehouse which contains five dimension tables PRODUCTS:

```
select cname, max(salary), avg(salary), sum(salary), min(salary),
count(salary) from emp_company group by cname.
```

The materialized view mv2 will optimize both queries.

4 EXPERIMENTAL EVALUATION AND RESULTS

The experimentation was performed on a 2.3 GHz Intel core i5 processor with 4 GB main memory, running on Mac OS X. This experimentation was done using standard query workload mentioned in [26] on Oracle 11g Database Management System. The workload consisted of TIMES, CHANNELS, PROMOTIONS, CUSTOMERS having 15, 31, 4, 8 and 15 attributes (columns), respectively. It also contains one FACT table named SALES which contains two measures "QUANTITIES_SOLD" and "AMOUNT_SOLD". The performance on standard workload is compared to the recent established algorithms MVFI [26], ARMMVVM [13] and CBMVS [28] using GAIN measure (GM) [28] as a performance criterion. The optimization tests were carried out using standard query workload of data warehouse in which size is

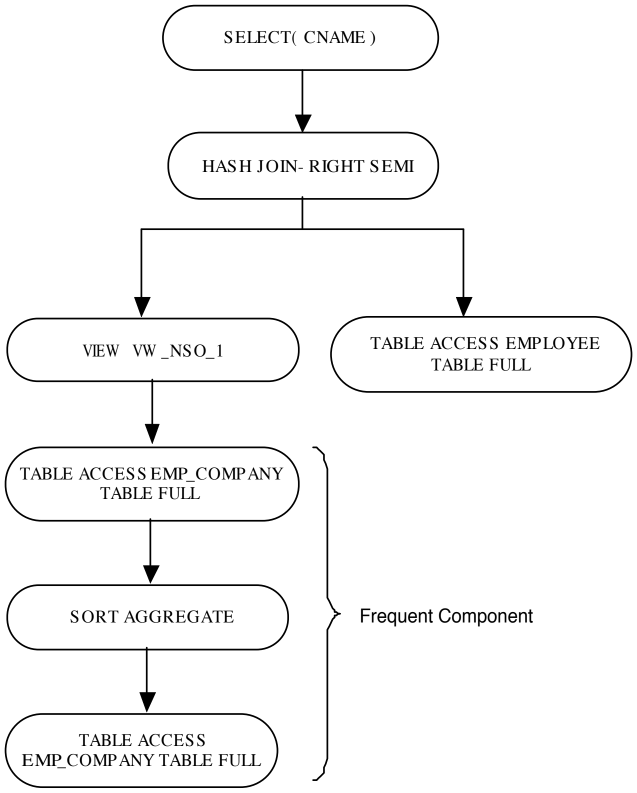
SELECT(CNAME)
HASH JOIN- RIGHT SEMI
VIEW VW_NSO_1
TABLE ACCESS EMPLOYEE TABLE FULL
TABLE ACCESS EMP_COMPANY TABLE FULL
SORT AGGREGATE
TABLE ACCESS EMP_COMPANY TABLE FULL
Frequent Component

Figure 5. Query plan for Query1

Sr. No.	Dataset Size	View Selection Algorithms (Gain Measure)			
		MVFI	ARMMVVM	CBMVS	Proposed
1	0.5 GB	5.5	5	4.4	6.1
2	1.0 GB	11	9	7.7	12.5
3	1.5 GB	17.4	13	10.5	19.8
4	2.0 GB	22.4	18	14.5	25.1

- MVFI – Materialized view selection based on frequent itemset mining algorithm.
- ARMMVVM – An association rule mining for materialized view selection.
- CBMVS or CBDMVS – Clustering based dynamic materialization view selection algorithm.
- GM – Gain Measure.

Table 1. Comparison between recent algorithms with proposed algorithms on the standard query workload

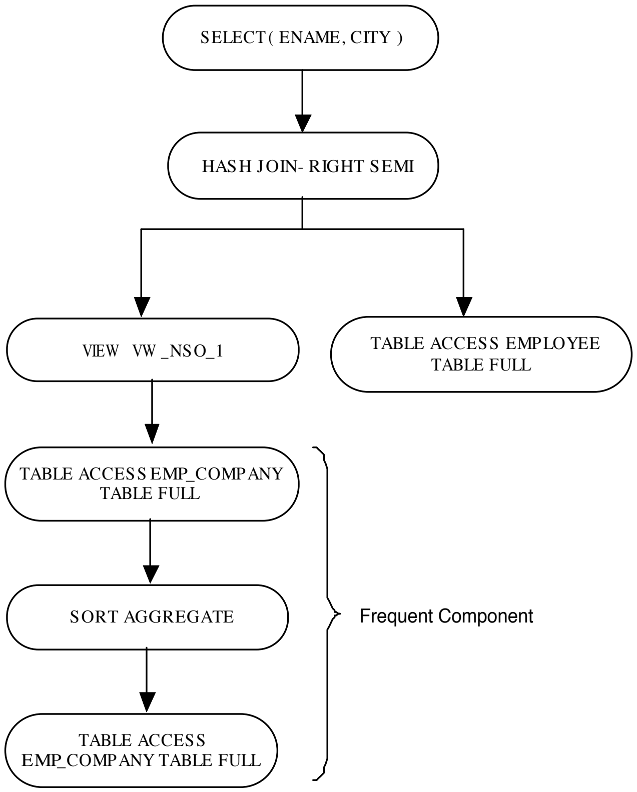
SELECT(ENAME, CITY)
HASH JOIN- RIGHT SEMI
VIEW VW_NSO_1
TABLE ACCESS EMPLOYEE TABLE FULL
TABLE ACCESS EMP_COMPANY TABLE FULL
SORT AGGREGATE
TABLE ACCESS EMP_COMPANY TABLE FULL
Frequent Component

Figure 6. Query plan for Query2

varied from 0.5 GB to 2 GB by controlling the number of rows in table SALES. The result of the experimentation is given in Table 1.

The experimentation results in Table 1 indicate that there is large improvement in GM compared to recent methods for all sizes of query work load by the proposed method. We have also used synthetic and real life datasets for the experimentation, to test the applicability of proposed method on varying types of queries. The real data set is obtained from Management Information System of National Institute of Technology, Nagpur. The majority of queries in almost all real life applications consist of time consuming operations such as joins, aggregations and groupings, hence we have taken query load which is a mixture of queries having such operations. We have composed three query sets QS1, QS2 and QS3 having different composition of join and aggregate queries. The tables referred in the queries use different columns which are aggregated and grouped. We make sure to have more variations in datasets in the form of aggregations and joins. The composition of data sets is described in Table 2. The first row indicates query set QS1 executed on real database of "MIS-

VNIT" which contains 2087 queries having 48 % join queries, 17 % aggregate queries and 29 % queries using both joins and aggregates. The other query sets are shown in successive rows.

Sr. No.	QL	DS	N	QJ	QA	QJA
1	QS1	Real MIS-VNIT	2 087	48	17	29
2	QS2	Synthetic	3 086	26	28	35
3	QS3	Synthetic	2 809	20	39	32

- QL – Query Load
- DS – Data Source
- N – Number of Queries (Total number of complex queries in the query workload)
- QJ – Percentage of queries involving only joins
- QA – Percentage of queries involving only aggregations
- QJA – Percentage of queries involving both joins and aggregations

Table 2. Dataset characteristics

The datasets were cached in the main memory during the algorithms processing stage, to avoid high data access costs. The numbers of frequent trees which are to be mined are controlled by parameter "frequency threshold". If the threshold is higher, then the algorithm produces less frequent trees and lesser number of materialized views are created. Since the number of materialized views cannot be large because of synchronization overheads, we have done the experimentation by setting the threshold to 50 % of the total candidate trees (threshold is taken as 50 % with the assumption that around 50 % of the total workload will be having frequent patterns. If more queries are to be optimized then the threshold can be reduced). The performance is measured using GM. The performance results are shown in Table 3 with different query loads. The proposed tree mining algorithm is implemented in Java.

QL	LR	View Selection Algorithms (Gain Measure)			
		MVFI	ARMMVVM	CBMVS	Proposed
QS1	3 560 642	24.34	21.21	15.38	40.62
QS2	4 701 867	33.68	29.24	27.45	37.80
QS3	3 857 673	31.25	28.41	24.37	40.10

- QL – Query Load
- LR – Logical Reads before creation of materialized views

Table 3. Results showing comparative analysis of best known algorithms with proposed algorithm on real and synthetic datasets described in Table 2

From Table 3, it is interpreted that the GM is considerably increased with the proposed tree mining algorithm when compared to state of the art algorithms in all types of query load because the proposed algorithm is designed to mine frequent queries as well as frequent subquery components. It has also been observed that for the large datasets of query workload of the size 4 million, the improvement is considerable.

4.1 Scalability

The scalability of the tree mining algorithm was analyzed by using query load of three different sized datasets. The query load is obtained by using queries mentioned in Table 2 multiple times. It was found that due to efficient pruning techniques, the processing time increased linearly with size, though the worst case complexity could be $O(N^2)$. The reason behind the experimental linearity is that if the tree is already a part of the frequent tree, then the cost of finding the frequency of the tree or the database scan is minimized.

Another reason for linearity could be that if a subtree is not frequent then its supertree is also not frequent, hence there is no cost of extra database scan. The performance of the algorithm is given in Table 4. In general, execution plans for groupings and aggregations have trees of larger length and hence mining frequent components takes more time. Since the algorithm is executed offline without any hard time constraint, practically the execution time mentioned in the table is well within acceptance level.

Datasets	No. of Queries in the Dataset			
	100 000	200 000	300 000	400 000
QS1	381	708	1 020	1 343
QS2	335	592	829	1 087
QS3	467	931	1 330	1 683

Table 4. Execution time of tree mining algorithm on different datasets with different sizes (number of queries) of query load (time in seconds)

5 CONCLUSION AND FUTURE WORK

It is a challenging task to select a set of queries from a huge query load, for creating materialized views. This is because such a set should not only be small, but should also provide maximum benefit for optimizing most of the queries. Most of the earlier methods rely on approximate text matching algorithms or finding frequent patterns in queries which refers to same set of tables. Such an approach may not work if frequent queries appear as sub queries.

In this paper, an attempt has been made to find frequent queries as well as frequent subqueries. The proposed method uses "query execution plan" for finding frequent query components instead of operating on query text. Such query plan is represented as a binary tree which can then be extracted from dynamic dictionary views which are provided by most of the databases and data warehouse systems, making the proposed method feasible. In the proposed method, finding frequent components in a large set of queries is translated as finding frequent subtrees, and efficient algorithms are proposed to extract subtrees with the correctness proof of the algorithm. The proposed method suggests various pruning techniques to effectively reduce the search space and to combat the huge query load. Certain queries which do not require materialized views are preprocessed and removed from the experimental load. The proposed method is compared with standard workload mentioned in the literature and its performance is compared with the recent methods available in the literature. The experimental evaluation indicates that the proposed method gives better performance than all the recent methods irrespective of query load size. The detailed study is done on real and synthetics data sets to check the performance on various types of workloads. The experimental evaluation indicates that the performance is improved to very large extent by the proposed method in all types of query workloads.

In the future, the selected queries can be analyzed using data synchronization costs of materialized views and total optimization can be done considering reduction of query cost and increase in data synchronization costs. The queries having high data synchronization costs can be modified and optimized using conventional methods and can be pruned in preprocessing steps.

REFERENCES

[1] Afrati, F. N.: Determinacy and Query Rewriting for Conjunctive Queries and Views. Theoretical Computer Science, Vol. 412, 2011, No. 11, pp. 1005–1021, doi: 10.1016/j.tcs.2010.12.031.

[2] Al-Khamaiseh, K.—ALShagarin, S.: A Survey of String Matching Algorithms. International Journal of Engineering Research and Applications, Vol. 4, 2014, No. 7, pp. 144–156.

[3] Aouiche, K.—Jouve, P. E.—Darmont, J.: Clustering-Based Materialized View Selection in Data Warehouses. In: Manolopoulos, Y., Pokorný, J., Sellis, T. K. (Eds.): Advances in Databases and Information Systems (ADBIS 2006). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4152, 2006, pp. 81–95, doi: 10.1007/11827252_9.

[4] Ashadevi, B.: Analysis of View Selection Problem in Data Warehousing Environment. International Journal of Engineering and Technology, Vol. 3, 2012, No. 6, pp. 447–457.

[5] Azgomi, H.—Sohrabi, M. K.: A Game Theory Based Framework for Materialized View Selection in Data Warehouses. Engineering Applications of Artificial Intelligence, Vol. 71, 2018, pp. 125–137, doi: 10.1016/j.engappai.2018.02.018.

[6] Chaudhuri, S.—Krishnamurthy, R.—Potamianos, S.—Shim, K.: Optimizing Queries with Materialized Views. Proceedings of the Eleventh IEEE International Conference on Data Engineering (ICDE '95), Taiwan, 1995, pp. 190–200, doi: 10.1109/ICDE.1995.380392.

[7] Chen, D.—Chirkova, R.—Sadri, F.: Query Optimization Using Restructured Views: Theory and Experiments. Information Systems, Vol. 34, 2009, No. 3, pp. 353–370, doi: 10.1016/j.is.2008.10.002.

[8] Chi, Y.—Xia, Y.—Yang, Y.—Muntz, R. R.: Mining Closed and Maximal Frequent Subtrees from Databases of Labeled Rooted Trees. IEEE Transactions on Knowledge and Data Engineering, Vol. 17, 2005, No. 2, pp. 190–202, doi: 10.1109/TKDE.2005.30.

[9] Daneshpour, N.—Barfourosh, A. A.: Dynamic View Management System for Query Prediction to View Materialization. International Journal of Data Warehousing and Mining, Vol. 7, 2011, No. 2, pp. 67–96, doi: 10.4018/jdwm.2011040104.

[10] Deshpande, P. S.: Data Warehousing Using Oracle. Wiley-DreamTech Press, India, 2005.

[11] Afrati, F.—Chirkova, R.: Selecting and Using Views to Compute Aggregate Queries. Journal of Computer and System Sciences, Vol. 77, 2011, No. 6, pp. 1079–1107, doi: 10.1016/j.jcss.2010.10.003.

[12] Goldstein, J.—Larson, P.-Å.: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, USA. ACM SIGMOD Record, Vol. 30, 2001, No. 2, pp. 331–342, doi: 10.1145/376284.375706.

[13] Gong, A.—Zhao, W.: Clustering-Based Dynamic Materialized View Selection Algorithm. Proceedings of the 2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2008), Vol. 5, 2008, pp. 391–395, doi: 10.1109/FSKD.2008.96.

[14] Gosain, A.—Sachdeva, K.: A Systematic Review on Materialized View Selection. In: Satapathy, S., Bhateja, V., Udgata, S., Pattnaik, P. (Eds.): Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications. Springer, Singapore, Advances in Intelligent Systems and Computing, Vol. 515, 2017, pp. 663–671, doi: 10.1007/978-981-10-3153-3_66.

[15] Gupta, A.—Mumick, I. S.: Maintenance of Materialized Views: Problems, Techniques, and Applications. IEEE Data Engineering Bulletin, Vol. 18, 1995, No. 2, pp. 3–18.

[16] Gupta, H.—Mumick, I. S.: Selection of Views to Materialize Under a Maintenance Cost Constraint. In: Beeri, C., Buneman, P. (Eds.): Database Theory – ICDT '99. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1540, 1999, pp. 453–470, doi: 10.1007/3-540-49257-7_28.

[17] Gupta, H.—Mumick, I. S.: Selection of Views to Materialize in a Data Warehouse. IEEE Transactions on Knowledge and Data Engineering, Vol. 17, 2005, No. 1, pp. 24–43, doi: 10.1109/TKDE.2005.16.

[18] Hung, M.-C.—Huang, M.-L.—Yang, D.-L.—Hsueh, N.-L.: Efficient Approaches for Materialized Views Selection in a Data Warehouse. Information Sciences, Vol. 177, 2007, No. 6, pp. 1333–1348, doi: 10.1016/j.ins.2006.09.007.

[19] Hylock, R.—Currim, F.: A Maintenance Centric Approach to the View Selection Problem. Information Systems, Vol. 38, 2013, No. 7, pp. 971–987, doi: 10.1016/j.is.2013.03.005.

[20] Karanjekar, J. B.—Chandak, M. B.: Uniform Query Framework for Relational and NoSQL Databases. Computer Modeling in Engineering and Sciences, Vol. 113, 2017, No. 2, pp. 177-187, doi: 10.3970/cmes.2017.113.177.

[21] Kumar, A.—Vijay Kumar, T. V.: Improved Quality View Selection for Analytical Query Performance Enhancement Using Particle Swarm Optimization. International Journal of Reliability, Quality and Safety Engineering, Vol. 24, 2017, No. 6, Art. No. 1740001, doi: 10.1142/S0218539317400010.

[22] Lian, Y. S.—Liou, M. S.: Mining of Data From Evolutionary Algorithms for Improving Design Optimization. CMES: Computer Modeling in Engineering and Sciences, Vol. 8, 2005, No. 1, pp. 61–72, doi: 10.3970/cmes.2005.008.061.

[23] Lovis, C.—Baud, R. H.: Fast Exact String Pattern-Matching Algorithms Adapted to the Characteristics of the Medical Language. Journal of the American Medical Informatics Association, Vol. 7, 2000, No. 4, pp. 378–391.

[24] Mohania, M.—Kambayashi, Y.: Making Aggregate Views Self-Maintainable. Data and Knowledge Engineering, Vol. 32, 2000, No. 1, pp. 87–109, doi: 10.1016/S0169-023X(99)00016-6.

[25] Ross, K. A.—Srivastava, D.—Sudarshan, S.: Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '96), Montreal, Quebec, Canada. ACM SIGMOD Record, Vol. 25, 1996, No. 2, pp. 447–458, doi: 10.1145/235968.233361.

[26] Vishwanath, R. P.—Rajyalakshmi, R.—Reddy, S.: An Association Rule Mining for Materialized View Selection and View Maintenance. International Journal of Computer Applications, Vol. 109, 2015, No. 5, pp. 15–20.

[27] Sohrabi, M. K.—Azgomi, H.: TSGV: A Table-Like Structure-Based Greedy Method for Materialized View Selection in Data Warehouses. Turkish Journal of Electrical Engineering and Computer Sciences, Vol. 25, 2017, No. 4, pp. 3175–3187, doi: 10.3906/elk-1608-112.

[28] Sohrabi, M. K.—Ghods, V.: Materialized View Selection for a Data Warehouse Using Frequent Itemset Mining. Journal of Computers, Vol. 11, 2016, No. 2, pp. 140–148, doi: 10.17706/jcp.11.2.140-148.

[29] Yang, J.—Karlapalem, K.—Li, Q.: A Framework for Designing Materialized Views in Data Warehousing Environment. Proceedings of the 17th International Conference on Distributed Computing Systems, IEEE, Baltimore, MD, USA, 1997, pp. 458–465, doi: 10.1109/ICDCS.1997.603380.

[30] Yan, X.—Han, J.: gSpan: Graph-Based Substructure Pattern Mining. Proceedings of the 2002 IEEE International Conference on Data Mining, Maebashi City, Japan, IEEE, 2002, pp. 721–724, doi: 10.1109/ICDM.2002.1184038.

[31] Yousri, N. A. R.—Ahmed, K. M.—El-Makky, N. M.: Algorithms for Selecting Materialized Views in a Data Warehouse. Proceedings of the 3rd ACS/IEEE International Conference on Computer Systems and Applications, IEEE, Cairo, Egypt, 2005, doi: 10.1109/AICCSA.2005.1387024.

Atul Thakare received his post-graduate degree (M.Eng. (CSE)) from SGB Amravati University, Maharashtra, India, and his undergraduate degree (Bc.Eng. (CT)) from RTM Nagpur University, Maharashtra, India. Currently, he is pursuing his Ph.D. from VNIT, Nagpur, Maharashtra, India. He has a total of 16 years experience, six years in the IT industry and ten years in academic profession.

Srijay Deshpande has received his B-Tech from Visvesvaraya National Institute of Technology, Nagpur, M-Tech from IIT-Bombay, and currently he is working as Data Scientist at Microsoft Private Ltd., India.

Amit Kshirsagar has received his B-Tech from Visvesvaraya National Institute of Technology, Nagpur, and Masters from the University of Florida and currently he is working as Senior Software Developer at Visa Inc., (USA) in Cyber Security.

Parag Deshpande has received his Ph.D. from Nagpur University, Nagpur, India and his M-Tech from IIT Powai, Mumbai, India. He is currently working as Professor in the Department of Computer Science and Engineering, VNIT, Nagpur, Maharashtra, India. He has 31 years of academic experience. He is the author of several books including C and Data Structure, Data Warehousing Using Oracle, SQL and PL/SQL for Oracle 11g. He is a member of ISTE and SAE-India.