Computer Science • 19(4) 2018 https://doi.org/10.7494/csci.2018.19.4.3006

Atul Thakare
Parag Deshpande

AN EFFICIENT APPROACH FOR VIEW SELECTION FOR DATA WAREHOUSE USING TREE MINING AND EVOLUTIONARY COMPUTATION

Abstract *The selection of a proper set of views to materialize plays an important role in database performance. There are many methods of view selection that use different techniques and frameworks to select an efficient set of views for materialization. In this paper, we present a new efficient scalable method for view selection under the given storage constraints using a tree mining approach and evolutionary optimization. The tree mining algorithm is designed to determine the exact frequency of (sub)queries in the historical SQL dataset. The Query Cost model achieves the objective of maximizing the performance benefits from the final view set that is derived from the frequent view set given by the tree mining algorithm. The performance benefit of a query is defined as a function of query frequency, query creation cost, and query maintenance cost. The experimental results show that the proposed method is successful in recommending a solution that is fairly close to an optimal solution.*

Keywords database management systems, data warehousing and data mining, query optimization, graph mining, algorithms for parallel computing, evolutionary computations, genetic algorithms

Citation Computer Science 19(4) 2018: 431–455

1. Introduction

Among all of the query optimization techniques, indexing and view materialization have been proven to be the most effective ones. A materialized view improves data access time by pre-computing the intermediate results. The user queries can be processed efficiently by using the data stored within the materialized views. Hence, the materialized view can speed up the analytical query processing in a data warehouse. However, the creation of materialized views demands additional storage space and incurs view maintenance overheads; because of this, it is not possible to materialize all possible views. Hence, it is an important issue in data warehousing systems to select an appropriate set of materialized views that minimizes the total query response time and cost of maintaining the selected views under the given storage constraints. To achieve this goal, views that are closely related to most of the workload queries need to be materialized [9]. To address this problem, several cost models (solutions) are proposed in the existing literature, and most of the models use some or all of the cost metrics out of query execution frequencies, base relationship update frequencies, query access costs, view maintenance costs, and the systems storage space constraints. The solutions mainly consist of two parts, out of which the first part finds a set of candidate views and the second defines the final set of views to materialize (which is essentially a subset of the candidate views). This selection of a final view set is done under specified resource constraints (storage cost and view maintenance cost constraints) by using some approximate heuristic algorithms(s).The known algorithms related to the selection of a final view set can be classified into four categories: deterministic algorithms, randomized algorithms, hybrid algorithms, and constraint programming. For the first part of constructing a candidate view set, a Directed Acyclic Graph (DAG) of the queries is usually used. This DAG represents the dependence relationship of the queries and is used to detect common sub-expressions between different queries. The most commonly used DAGs for materialized view selection are the AND/OR view graph, Multiple View Processing Plan (MVPP), and data cube lattice [16]. Some of the view selection methods use identification techniques such as a syntactical analysis of the workload and query rewriting instead of DAGs. We have designed a novel tree mining algorithm to detect common sub-queries between different workload queries and to determine the exact execution frequency of these sub-queries in the past workload.

2. Our contribution

To predict future queries, we determine frequent subqueries from the past workload, as these query components will have a high probability of being repeated in the near future as an independent query and as a part of many complex or nested queries. Considering this syntactical similarity (link) between past workload frequent query components and the queries arriving in the near future, we suggests that extracting such frequent components from the past workload and converting the most profitable

subset of it to the materialized views will ensure that the future workload will be better optimized due to the use of views while processing future queries. We define the similarity and dissimilarity measures among the queries, which enables to form groups of similar ,i.e., related queries (defined in section 4.1). Each group will have a set of queries having one or more identical component(s) common amongst them. Different queries from all the groups (also called as clusters or partitions) are used to build a set of candidate views. As we cluster the related queries together, most of the identical or similar query components originating from different complex queries (which may not be frequent) will be added to the same cluster. To determine the list of candidate queries with their final frequencies, we designed two phases of frequency roll-ups.

The phase one frequency roll-ups which is also called as inter-cluster/ phase-1 counting is performed within each cluster. This phase determines the aggregate frequency of each distinct query component within each cluster. For example, If there are ten queries $\{Q_1, Q_2, ...Q_n\}$ having a common subquery S, then the number of executions (frequency) of S will be, $E = \sum_{k=1}^{n} E_k$ where E_k is the number of executions (frequency) of Query Q_k. As ten different instances of S will join the same cluster say P1 through queries $\{Q_1, Q_2, ...Q_n\}$, phase-1 counting will assign aggregate frequency E to first instance of S and will remove all the other identical 9 instances. This operation is performed on each distinct query component within each cluster in phase-1 counting. At the end of phase-1 counting, every query component will be unique within its cluster having aggregate frequency assigned to it. This phase determines the list of candidate queries along with their aggregate, i.e., intermediate frequency within a cluster. In other words, frequency rollups during phase-1 counting remove all of the duplicates of each candidate component within the cluster, leaving distinct candidate components in it along with the updated number of executions of each one. This phase also optimizes the process of finding the total number of executions (final frequency) of each candidate query by reducing the search space.

To determine the final frequency of each candidate query, we propose the use of phase two frequency roll-ups which is also called as intra-cluster/ phase-2 counting, which is performed while merging any two partitions (clusters). In this phase, we recursively merge 2 partitions at a time till all the partitions are merged to form a single partition. This phase takes care of exceptional cases where candidate component of one cluster is also related to candidate component from other cluster(s). Frequency rollups during phase-2 counting occurs for only those queries from the first partition, which is having an exact matching (isomorphic) candidate query in the other partition. In such cases, frequency rollups will also be performed for all of the candidate queries in the subcomponent hierarchy of the matching queries from both the partitions (explained in section 6.4.3).

Phase one frequency roll-ups followed by phase two frequency roll-ups gives us all the candidate components with their final frequencies. During these two phases, in addition to frequency roll-ups the algorithm also captures the relationship links

between different candidate components when one candidate query is a subcomponent of the other candidate query(s). Hence, this process can be seen as iteratively building lattices of related queries. During this process, the frequency of each candidate query is rolled-up by adding to its frequency the frequencies of all of its duplicates and the frequencies of all of its super-queries. This process also involves removing all of the duplicates of each candidate query. This keeps on reducing the number of queries under consideration (hence, reducing the time complexity of the process).

As the main contributions of this paper, a new tree mining method is developed for finding a set of top frequent candidate queries, and a novel query cost model based on evolutionary computation is introduced, which takes a storage cost, creation cost, and maintenance cost of queries under consideration for the recommending fairly optimal final view set.

3. Related work

The problem of finding views to materialize to optimize query performances has traditionally been studied under the name of view selection. This selection is non-trivial in nature and is an NP-complete problem. Harinarayan and Ullman [9] were the first to recognize the problem of materialized view selection for supporting a multidimensional analysis in OLAP. The authors proposed a cost model defined on a lattice of views and provided a polynomial-time greedy algorithm for view selection that minimizes the query processing cost (subject to the space constraint in the special case of data cubes).

Gupta [6] further improved the work by providing a solution for materializing view indexes. Ezeife [3] also considered the same problem but proposed a uniform approach using a more detailed cost model. Ross and Sudarshan [16] presented heuristics to determine the additional set of views to materialize under a given storage constraint to reduce the overall maintenance cost and query response time of all of the views. Yang [19] proposed a heuristic algorithm that utilizes a Multiple View Processing Plan (MVPP) to obtain an optimal materialized view selection such that the best combination of good performance and low maintenance cost can be achieved.

The work in [18] considered the queries that include select, project, join, and aggregation operations. The paper proposed a greedy algorithm to select a set of materialized views so that the combined query processing and view maintenance cost is minimized. A genetic algorithm has been used in [10, 20] in conjunction with the Multiple View Processing Plan (MVPP) framework to solve the view selection problem. The views have been selected based on a reduction in the combined query processing and view maintenance cost. Himanshu Gupta and Inderpal Singh Mumick [8] developed algorithms to select a set of views to materialize in order to minimize the total query response time under the view maintenance time constraint. Lin and Kuo [14] proposed a greedy genetic algorithm that selects a set of materialized cubes from data cubes under the storage space constraints in order to reduce the amount of query cost as well as the cube maintenance cost.

Gupta [7] presents a greedy view selection algorithm in AND/OR view graphs, which describes all possible ways to generate warehouse views and selects the best query path that can be maximally utilized to optimize the response cost on the workload queries under the maintenance cost constraints. In [2], the authors proposed a framework for materialized view selection that exploits a data mining technique (clustering) in order to determine clusters of similar queries. The paper also proposed a view merging algorithm that builds a set of candidate views by iteratively building a lattice of the views. To determine the final view set, a greedy process is used that considers the cost of accessing data from the views and cost of storing views as important criteria of the selection process.

The authors of [11] proposed a cost model having well-defined gain metrics and loss metrics to decide the membership of a view in the view set. For candidate generation, the data cube is represented as a lattice, and the lattices are expressed in the form of a vector. The vector is used to search for a dependency of views. In [4], An Gong proposed a clustering based dynamic materialized view selection algorithm (CBDMVS). It finds a cluster of SQL queries using a similarity threshold τ; if a new querys similarity is below for all of the existing clusters, then a new cluster is formed. The similarity between queries is measured based on certain parameters like the 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 the lowest gains where the system lacks storage space for the new query. In essence, it not only improves the overall query response time but also reduces the computational cost that is spent while updating the materialized view. Afrati and Chirkova [1] explained 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. Mohammad and Vahid [15] discussed the usage of Directed Acyclic Graphs and a data cube lattice in the candidate generation step and the various heuristic algorithms in the second step of the view selection. The paper also proposed a novel algorithm based on the frequent itemset mining technique that aims at minimizing the view creation and maintenance costs. Hylock and Currim [12] presented a View Selection Problem heuristic model that combines the previous methods and minimizes and bounds the view maintenance time. In [17], Rajyalakshmi proposed an 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 a frequent mining algorithm along with adjustments to the view set. In [5], the authors considered the problem of view selection in Big Data warehousing systems and defined it as a multi-objective optimization problem for minimizing the total query processing MapReduce cost and MapReduce cost for maintaining the materialized views. In [13], the author proposed a swarm optimization-based view selection algorithm to selects the top K views from a multidimensional lattice in order to improve the performance of the analytical queries.

4. Proposed work

4.1. Terminologies

View: A view is a derived relationship defined by a query in terms of base relationships 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 query workload is a given set of queries $\{Q = Q_1, Q_2, ...Q_n\}$. Based on the query workload set, materialized views can be defined. Each query in the query workload can be described using its frequency, storage cost, and update cost. Based on some combination of these parameters, the set of views to materialize can be defined.
View Selection: Given a database schema and query workload, the objective is to select an appropriate set of materialized views to improve the performance of a database in processing the workload; i.e., in executing queries in the workload. The ideal view set can be comprised of queries that are useful in optimizing the performance of a large number of queries in the workload.
View Maintenance: Whenever a base relationship is changed, the materialized views built on it have to be updated in order to compute up-to-date query results. The process of updating a materialized view is known as view maintenance.
Related Candidate Queries: Candidate Queries are said to be related if they have a common super query; i.e., they are subcomponents of the same query OR they have a common subquery. For example:

- If query Q4 has Q1 and Q2 as its subqueries, then Q1 and Q2 are related queries.
- If queries Q1 and Q2 have common subquery Q3, then Q1 and Q2 are related queries.
- If query Q4 has Q1 as its subquery, then Q4 is related to all of the subqueries as well as the super queries of Q1; similarly, Q1 is related to all of the subqueries and super queries of Q4.

4.2. Problem formulation

The problem of view selection can be formulated as follows. Given database schema $R = \{R_1, R_2, ...R_r\}$ and query workload $Q = \{Q_1, Q_2, ...Q_q\}$ defined over R, the problem is to select an appropriate set of materialized views $M = \{V_1, V_2, ...V_m\}$ such that the query workload is answered with the lowest cost under a limited number of resources; e.g., storage space and/or view maintenance cost.

4.3. Architecture of proposed work

We have done this experiment on an Oracle database system. We have gathered all of the information related to the historical SQL queries, their executions plans, and statistics from three data dictionary views; $v\$sqlarea$, $v\$sql$, and

v$sql_plan_statistics_all [Steps 1 through 3 of Figure 1]. The tree mining algorithm analyzes the query execution plans to find the set of frequent subqueries (F_{sq}). The Query Cost Model takes (F_{sq}) as an input and gives the final view set (F_v) by using the proposed genetic algorithm over the parameter storage cost, processing cost, and update cost of the queries in F_{sq} (as represented in Figure 1).

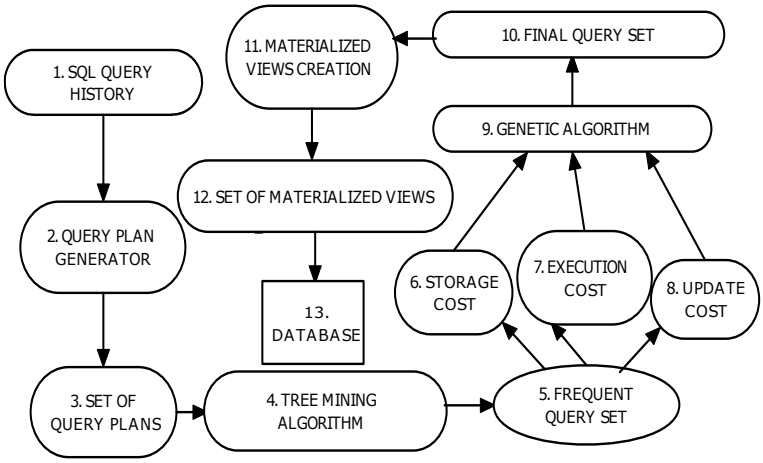

Figure 1. Architecture of Proposed Work

5. Algorithms

A Tree Mining Algorithm is used for finding the top K frequent candidate queries/subqueries. The detailed explanation of this algorithm (which is divided into several small algorithms) is given in Section 6.

Algorithm 1 finds out all of the active components (frequency above threshold) from the set of query plans. Algorithm 2 takes the active component list (output of Algorithm 1) as the input and clusters the active components into different groups based on the relationships between them. The output of Algorithm 2 is the same list in which each component is assigned a partition number or group number based on which the active components can be separated into groups.

Algorithm 3 takes the output of Algorithm 2 as an input and actually partitions the active component list based on the partition key column. Before this, it determines the frequency (i.e., the number of executions of each distinct component belonging to each partition). The output of Algorithm 3 is the Candidate List, which contains different candidate components along with their partition keys and the number of executions within their partitions. The output of Algorithm 3 is the input to Algorithm 4, which merges the disjoint pair of partitions in parallel and repeats this process until the table is unpartitioned. Algorithm 4 gives the final list of frequent components (Candidates) along with their final frequencies.

5.1. Algorithm 1. Finding all occurrences of all subqueries having frequent database objects and frequent database operations from SQL History H

Assumptions and Terminologies:
H: Query Plans for all SQL Queries from SQL History.
Edges of Query Plans having no labels are assigned label L.
F: Threshold Frequency.
Active Vertex: Vertex Label whose frequency is above F.
Active Edge: Edge Label whose frequency is above F.
Active component of a query plan: connected component having all vertices active, connected by active edges. Each leaf node of an active component should be a subset of the leaf nodes of the query plan (from which it is derived).
COMPL: Active Component List
Each Component's Record in COMPL has five fields: COMPID, Component, matchcode, partitionkey, and Freq (Initially set to number of executions of Query plan from which a component is derived). This information is available in Data Dictionary Views.

Algorithm 1. Finding SubQueries from SQL History

```
1: Input: Query plans Set H , Threshold F
2:
3: for each query plan tree Q_j in H do                ▷ Finding active components in H
4:     if Q_j contains at least one table join and one aggregate function then
5:         for each Active Component AC in Q_j do      ▷ AC represents a subquery in Q_j
6:             Compute Matchcode M
7:             Generate new COMPID
8:             Add record {COMPID,AC,M,NULL,NULL} to COMPL.
9:         end for
10:    end if
11: end for
12: Output: Active Component List COMPL
```

5.2. Algorithm 2. Assigning partitioning key for each active component record in COMPL for clustering-related subqueries in H

Assumptions and Terminologies:
Size of component: number of edges in a component.
Closed component: component having no super-component in COMPL.
C_k : K^{th} component in COMPL.
C_m : M^{th} component in COMPL.
$C_k \subseteq C_m$: Component C_k is a subcomponent of Component C_m. In other words, Query C_k is a Subquery of Query C_m. Subquery means C_k may also be equal to C_m.
PKey: Partition Key

Algorithm 2. Assigning partitioning key for each component in COMPL

Input: COMPL: Component List. Size: number of components in COMPL. (Output of Algorithm 1)
Sort COMPL based on sizes of components in descending order. ▷ Hence, all of the closed components will be at the beginning of the list.
CurrentLast ← 1
PKey ← 1
for k=1; k ≤ Size; k++ **do** ▷ Assigning partition key to each component
  for m=1; m ≤ CurrentLast; m++ **do**
    if $C_k \subseteq C_m$ **then**
      C_k.partitionkey ← C_m.partitionkey
      CurrentLast ← CurrentLast + 1
      Continue outer loop for next value of k
    end if
  end for
  C_k . partitionkey ← PKey++
  CurrentLast ← CurrentLast + 1
end for
Output: COMPL list with partition key assigned for each component within it (4^{th} field of each Component record), number of partitions (Pkey − 1)

5.3. Algorithm 3. Partitioning component list based on partition key and determining frequency of each distinct component (called a candidate) within each partition [Phase-1 frequency counting]

Assumptions and Terminologies:
CANDL: Candidate List. This list has PARTID, CANDID, COMPID, and FREQ fields, where COMPID of CANDL references COMPID of COMPL.
{Set of distinct components within each partition forms a list of candidates within that partition}
PKey –1: number of partitions from Algorithm 2. ▷ Last partition number will be PKey – 1
PSizen: Number of components in n^{th} partition.
TempL: List of deleted components.
SubL: SubList of components.
RLIST: Relationship List storing relationships between different non-identical components.
Relationship (a, b): Function that stores relationship between components a and b in RLIST.
cn-id: candidate id initialized to 1.
newrec: returns record corresponding to new candidate to be inserted to CANDL.
C_{r_n}: r^{th} component in n^{th} partition in COMPL.
$COMPL_n$: list of components from n^{th} partition in COMPL.

Algorithm 3. Defining a Candidate List

Input: COMPL: Component list; Size: size of COMPL List; number of partitions: PKey - 1 output of Algorithm 2
Partition the COMPL on partitionkey field. Hence, each partition will contain all occurrences (originated from historical queries in H) of related subqueries.
for n=1; n $\leq$ PKey -1; n++ **do**
    r = 1
    Sort the n^{th} partition based on Component sizes.
    TempL $\leftarrow$ {} ▷ Empty List
    while SubL .end () $\neq$ true **do**
        SubL $\leftarrow$ $COMPL_n$ - {C_{r_n} $\cup$ TempL};
        for each x in SubL **do**
            if C_{r_n} $\subset$ x **then**
                C_{r_n}.freq = C_{r_n}.freq + x.freq
                RLIST $\leftarrow$ Relationship (C_{r_n}, x)
            end if
            if C_{r_n} == x **then**
                C_{r_n}.freq = C_{r_n}.freq + x.freq;
                TempL $\leftarrow$ {TempL $\cup$ x}
            end if
        end for
        CANDL $\leftarrow$ CANDL $\cup$ newrec (n, cnid, C_{r_n}, C_{r_n}.freq)
        cnid $\leftarrow$ cnid + 1
        r $\leftarrow$ SubL.next () ▷ Next undeleted component from SubL
    end while
end for
Output: CANDL List
Partition the CANDL on PARTID.

5.4. Algorithm 4. Finding final candidate list by merging all partitions of candidate list [Phase-2 frequency counting]

Assumptions and Terminologies:
FCANDL: Final Candidate List having same structure as CANDL.
X, Y: Partition Identifiers.
TempL: List of deleted components.
n: integer.
L_x: List of candidates of Partition X.
L_y: List of candidates of Partition Y.
A_i: List of direct/indirect children of Candidate i within its partition.

Algorithm 4. Final Unpartitioned Candidate List having Candidates with Final Frequency

Input: CANDL: partitioned Candidate List (output of Algorithm 3) ▷ Merging partitions of CANDL recursively until list is unpartitioned

2: **for** n=2; n ≤ PKey -1; n++ **do**
3: X ← 1
4: Y ← n
5: TempL ← {} ▷ Empty List
6: **for** Each $z_1 \in L_x \bowtie z_2 \in L_y$ **do** ▷ ⋈: left right outer join
7: **if** $\{z_1,z_2\} \notin$ TempL $\wedge$ $z_1 \equiv z_2$ **then**
8: z_1.freq = z_1.freq + z_2.freq
9: **for** each p $\in A_{z_1}$ **do**
10: p.freq = p.freq + z_2.freq
11: **end for**
12: **for** each q $\in A_{z_2}$ **do**
13: q.freq = q.freq + z_1.freq
14: **end for**
15: Create relationship (Link) of child subtree of z_2 to z_1.
16: TempL ← {TempL ∪ z_2} ▷ Deletes z_2
17: **else if** $\{z_1,z_2\} \notin$ TempL **then**
18: Add z_2 to Partition X.
19: **end if**
20: **end for**
21: Update partition table by adding relationships of all candidates, which migrates to Partition X from Partition Y.
22: **end for**
23: FCANDL ← CANDL
24: **Output:** Final Candidate List FCANDL containing unpartitioned CANDL List with final frequency of each candidate component

5.5. Finding relationship between query components (used by Algorithms 2 and 4)

This procedure compares two query components by comparing string representations of them. The string representation of a query component (matchcode) is generated by traversing a tree in an in-order fashion and adding all of the node labels (delimited by =) in a matchcode.

The matchcode of the query plan in Figure 2 will be as follows:
$TABLE\ ACCESS\ EMP\_COMPANY\ TABLE\ FULL$
$= SORT\ AGGREGATE = TABLE\ ACCESS\ EMP\_COMPANY\ TABLE\ FULL$
$= VIEW\ VW\_NSO\_1 = HASHJOIN - RIGHTSEMI$
$= TABLE\ ACCESS\ EMPLOYEE\ TABLE\ FULL$
$= SELECT$

Consider two query components A and B. Let the matchcode of Query A be String S1 and the matchcode of Query B be String S2. If S1 is a substring of S2, then Query A is a subquery of Query B and vice-versa. If S1 and S2 are identical strings, then Query A and Query B represent the same query. If none of the above conditions are true, then Queries A and B are not related.

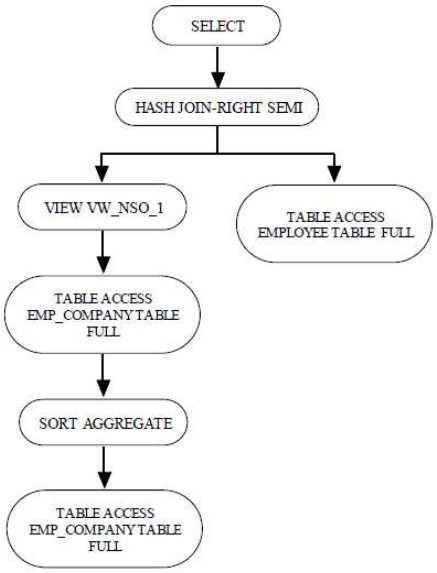

Figure 2. Query Plan

6. Working of graph mining algorithm

6.1. Terminologies

Closed candidate component (sub-tree): is a frequent substructure that does not have any frequent super structure.
Related candidates: Frequent components that are related to one another with a subquery-super query property.

First, our algorithm finds out all of the frequent base components (the vertices and edges whose frequency in Query History H is above threshold τ). Then, for candidate generation, it goes through each query plan tree in H and finds out the valid connected components obtained by connecting all of the frequent base components in a plan tree. For the validity of a connected component, it is checked that all of the branches of a connected component are terminating strictly at a leaf level of the query plan tree from which it is derived. All of the valid components derived from each query plan tree are stored in the tree database with a unique component ID. Each component basically represents a frequent SQL subquery that is a full/sub part of the historical SQL query. If we take any two valid components A and B from tree database, then the following possibilities exist:

- $A \equiv B$ (Components A and B represent the same query).
- $A \neq B \wedge A$ *related* B (Candidates A and B represent different queries but are related).
- $A \neq B \wedge A$ *notrelated* B (Candidates A and B represent different queries and are not related).

6.2. Clustering-related frequent queries (Algorithm 2)

All of the frequent candidate queries that are related will be stored in a single partition. Hence, we will have clusters of frequent candidate queries that have high commonality between queries within the cluster and much less commonality among queries across the clusters. Example: Let D1, D2, D3, and D4 be unrelated closed components in our tree database. Hence, our tree database will be partitioned into four partitions (say, P1, P2, P3, and P4). Partition P1 contains all of the occurrences of D1 and subcomponents of D1. Similarly, Partition P2 contains all of the occurrences of D2 and subcomponents of D2, and so on.

6.3. Frequency counting of candidate queries (Algorithm 3, Algorithm 4)

The frequency counting of candidate queries involves two steps:

- Determining frequency of distinct candidates within the partition.
- Merging the partition to get the final frequency (i.e., the number of executions) of the candidate components.

The above process is optimized by the application of the following pruning techniques.

6.4. Pruning techniques

6.4.1. Pruning by partitioning (Algorithm 2)

The advantage of partitioning the candidate list is that it reduces the search space while performing inter-cluster frequency rollup's for each distinct candidate component in Phase-1 counting. The search will be local to the partition in which candidate component in present.

6.4.2. Pruning by deletion (Algorithm 3)

In Phase-1 counting when we find two candidates (A and B) to be identical (isomorphic; i.e., A = B) within a particular partition, then make A.frequency = A.frequency + B.frequency and delete Candidate B from that partition.

6.4.3. Pruning by partition merging (Algorithm 4)

While merging partitions (say, P1 and P2) in Phase-2 counting, if we find
{Candidate A from Partition P1} = {Candidate B from Partition P2},
then do the following things:

- A.frequency = A.frequency + B.frequency.
- For each frequent component C_i in the child hierarchy of A in Partition P1, C_i.frequency = C_i.frequency + B.frequency.
- For each frequent component C_j in the child hierarchy of B in Partition P2, C_j.frequency = C_j.frequency + A.frequency.
- Link (update relationship table) child subtree of Candidate B to Candidate A of Partition P1.

- Delete Candidate B (and all of its relationships from the relationship table) of Partition P2.
- After handling all such cases of isomorphic candidates across P1 and P2, add all remaining candidates from P2 to P1 with no change in their frequency. Update the relationship table by adding new entries for partition P1. {now, P1 can have multiple disconnected relationship trees}.

This process reduces the number of tree comparison tests substantially which are the highly computation-intensive operations. Clustering related query components in a single partition by finding out the correlations among the different query components reduces the search space and speeds up the process of finding a final frequency count for each candidate component. Figures 3–5 give graphical representations of a partition merging between Partitions P1 and P2 in which Candidate C3 in Partition P1 is isomorphic to Candidate C7 of Partition P2.

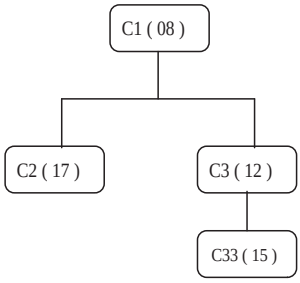

Figure 3. Relationship Tree of Partition P1

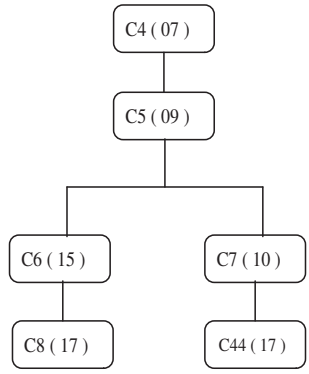

Figure 4. Relationship Tree of Partition P2

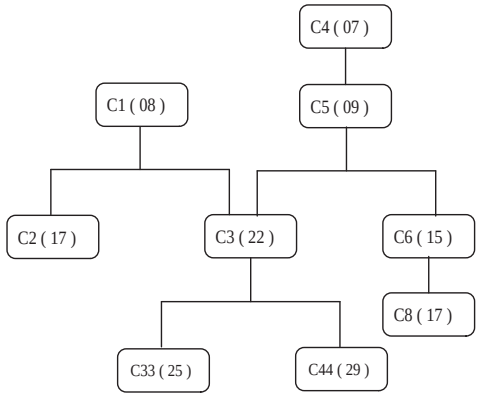

Figure 5. Relationship Tree of Partition P1 after merging P2 to P1

6.5. Advantages of proposed graph mining approach

The advantage of implementing graph mining in a relational database system is that we can use database optimization techniques like indexing, partitioning, and pipelining (parallel execution) to enhance the performance of our algorithm. On multiprocessor or multi-core systems, database system can carry out the task of finding the intermediate frequencies of the candidate components in different partitions independently & concurrently in different threads of the execution (In Phase-1 counting).

In Phase-2 counting, the merging of disjoint pairs of partitions can also be carried out concurrently on different processors. This is conceptually similar to the Map--Reduce Technique in the Big Data Scenario. This will scale the performance gain involved in finding frequent components linearly along with the data size. We can increase the speed up by investing additional computing resources.

Another benefit is that creating an index on Candidate ID (graph indices) will read the plan of a candidate component quickly into the memory by directly accessing blocks containing the plan of the candidate component. This is similar to maintaining a pointer along with a candidate ID that points to the disk blocks containing a candidate's query plan. Unlike many graph mining algorithms, this algorithm does not require all of the graphs to be in the main memory while counting the total number of instances of a particular subgraph.

7. Finding final view set using evolutionary computation

7.1. Genetic algorithms

Genetic Algorithms (GAs) are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. As such, they represent an intelligent exploitation of a random search used to solve optimization problems.

Algorithm 5. Generic Genetic Algorithm

```
1: randomly initialize population (p)
2: determine fitness of population (p)
3: while best individual is not good enough or number of evolutions does not reach its
   limiting value do
4:     select parents from population (p)
5:     perform crossover on parents creating population (p+1)
6:     perform mutation of population (p+1)
7:     determine fitness of population (p+1)
8: end while
```

After an initial population is randomly generated, the algorithm evolves through three operators:

- Selection, which equates to survival of the fittest.
- Crossover, which represents reproduction by crossover between solutions.
- Mutation, which introduces random modifications.

7.2. Query cost model

The problem of view selection in data warehouses can be reduced to a non-linear constrained optimization problem where we have to predefine some storage constraints (storage threshold) and create an initial population of n solutions within those constraints. A tree mining algorithm will give a set of frequent queries S. Each solution of the initial population is defined by randomly selecting Subset S' from Set S such that the summation of the storage cost of all queries in S' is less than or equal to the storage threshold. In the tree mining algorithm, we are also finding groups of related frequent queries (say, total G groups/clusters. where the frequent queries in each group are related by the subquery-super query property). Each solution has a set of frequent queries chosen by following the above constraints. Each query in the solution has the following attributes:

<QueryID, StorageCost, CreationCost,UpdateCost, FitnessValue >

Our optimization process has two objectives:

- Maximize total creation cost.
- Minimize total maintenance (update) cost.

Hence, this is a multiple-objective problem that can be solved by a multiple--objective optimization method inspired by evolutionary computations. Here, the total creation cost (T_c) is a summation of the creation cost of all of the queries within the solution, and the total update cost (T_u) is a summation of the update cost of all of the queries within the solution.

Hence, we can say that $T_c(S)$ is nothing but the creation cost of Solution S (which is desired to be as high as possible) and $T_u(S)$ is nothing but the update cost of Solution S (which is desired to be as low as possible).

Hence, our two objective functions (i.e., the creation and update costs of Solution S) are as follows:

- $T_c(S) = \sum_{i=1}^{x}$ CreationCost(i) such that $T_c(S)$ should be maximum.
- $T_u(S) = \sum_{i=1}^{x}$ UpdateCost(i) such that $T_u(S)$ should be minimum,

where S is a solution having x frequent candidate queries.

The creation and update cost of any query Q_i within Solution S can be defined as follows:

- CreationCost (Q_i) =[ProcessingCost (Q_i) $*$ ExecutionFrequency (Q_i)],
- UpdateCost (Q_i) =[UpdateCost (Q_i) $*$ UpdateFrequency (Q_i)],

where ProcessingCost (Q_i) is the average cost of executing query Q_i.

ExecutionFrequency (Q_i) is the number of executions of query Q_i in a week.

UpdateCost (Q_i) is the average cost of refreshing the materialized view of query Q_i. This depends on the update cost of the base database objects to which query Q_i refers.

UpdateFrequency (Q_i) is the number of times refreshing the materialized view of query Q_i is required in a week. This depends on the update frequency of the underlying database objects to which query Q_i refers. We can get this information

by querying data dictionary views and audit tables of the database. Ideally, Update-Frequency (Q_i) is the number of times the result stored in the materialized view of query Q_i becomes invalid within a week. Hence, fitness value of query (Q_i),

$$\text{FitnessValue } (Q_i) = \text{CreationCost } (Q_i) - \text{UpdateCost } (Q_i).$$

Hence, our two objective functions (i.e., creation and update cost of Solution S) becomes

- $T_c(S) = \sum_{i=1}^{x} [\text{ ProcessingCost } (Q_i) * \text{ExecutionFrequency } (Q_i)]$ such that $T_c(S)$ should be maximum.
- $T_u(S) = \sum_{i=1}^{x} [\text{ UpdateCost } (Q_i) * \text{UpdateFrequency } (Q_i)]$ such that $T_u(S)$ should be minimum.

We can combine the above two objective functions in a single objective function as follows:

$$\text{FitnessValue } (S) = T_c(S) - T_u(S) \text{ should be maximum.}$$

7.3. Implementation of query cost model using genetic algorithm

After merging all of the partitions into a single one, we will find a number of lattices (say, $L_1, L_2, L_3, \dots$) where each lattice describes the set of queries related to each other. This description also contains the final frequency of each candidate query in each lattice.

7.3.1. Database of candidate hierarchy

Let us consider Figure 6, which represents a lattice of the candidate queries. We can represent the different hierarchies of the candidate queries embedded inside the lattice as follows. Hence, our Query Cost Model finds all such hierarchies of the related queries from all of the lattices.

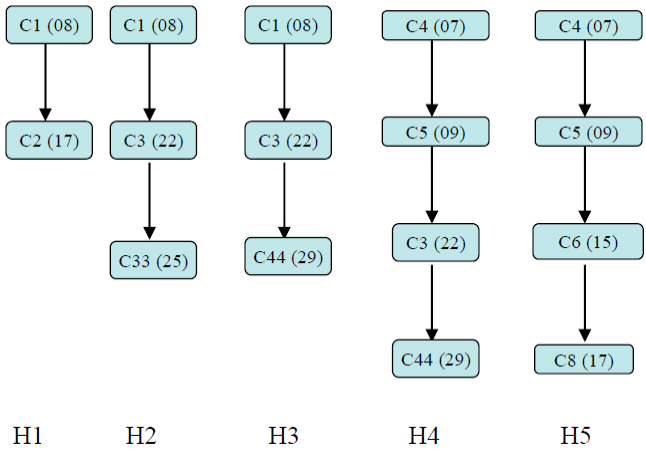

Figure 6. Different hierarchies of candidate queries in lattice of Figure 5

7.3.2. Creating initial population of solutions

We create an initial population of 100 solutions. We create each solution by randomly selecting different hierarchies and selecting exactly one query from each hierarchy. If an identical/same query is already admitted to a solution (picked up from some other hierarchy), then that hierarchy is completely ignored for completion of that solution.

Each solution observes the following constraint (Rule (1)):

- The total storage cost of a solution should be between 90–100 percent of the available storage (the available storage is dependent on the free disk space available for the database system used for performing the experimentation).
- All of the queries in a solution should be distinct. The fitness value of a solution is the summation of the fitness values of the queries in it.

The mutation and crossover operators also observe the above constraints while creating new solutions.

7.3.3. Mutation operator

The mutation operator works on the following principle.

Each candidate query within a solution is picked from some hierarchy. In each hierarchy, the top-level query will be the most complex one; hence, that query will have the highest creation cost and update cost among all of the queries in it. However, as the fitness value of a query is a function of the difference between the creation and update costs of a query, hence any query can be the fittest query among all queries in a hierarchy.

Considering this, the mutation operator will perform the following steps:

1. Select any one query from a solution.
2. Pretend to replace the selected query with another query from the hierarchy of the selected query.
3. If the mutated solution is valid according to Rule (1), then compute the fitness value of the mutated solution and add it to the current population. Exit.
4. Otherwise, repeat Steps (1) through (3).

7.3.4. Crossover operator

The crossover operator will perform the following steps:

1. Select two solutions randomly from the population.
2. Select some queries randomly from both solutions and combine them to form a new solution.
3. If the new solution is valid according to Rule (1), then compute the fitness value of the new solution and add it to the current population. Exit.
4. Otherwise, repeat Steps (1) through (3).

7.3.5. Selection operator

The current population evolves to a new population by means of a selection operator. The selection operator selects the top 25% solutions from the current population based on fitness values and copies it to a new population. The remaining 75% of the population is generated by using mutation and crossover operators.

In brief, the purpose of the crossover operator is to find the correct set of candidate hierarchies for the optimal solution, and the purpose of the mutation operator is to find the fittest candidate query from each candidate hierarchy in that solution. Hence, each member of the fittest solution (an optimal solution that is the fittest one in the last population of the evolution process) is ideally the top-most fittest query in its own candidates hierarchy and one of the top N candidate queries (in terms of fitness) among all of the hierarchies in the population of solutions.

8. Experimental evaluation and results

The experimentation was performed on an Oracle Database 11g installed on a system with a 2.3 GHz Intel Core i5 processor and 4GB of main memory on the Mac OS X platform. In the first phase, we performed the experiment four times on the four different datasets used in [17]. The final result is the exact gain of the proposed method we recorded on each dataset by using the Gain Measure described in [15]. A comparison of the gains of the proposed method with the known methods from [4, 15], and [17] is shown in Figure 7.

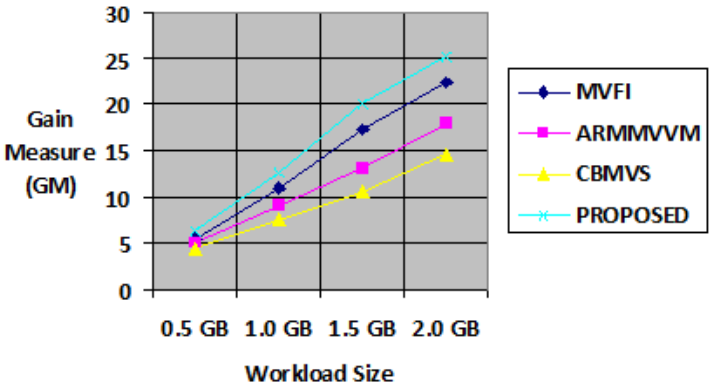

Figure 7. Comparison of known methods with proposed methods

MVFI – materialized view selection based on frequent itemset mining algorithm.
ARMMVVM – association rule mining for materialized view selection.
CBMVS or CBDMVS – clustering-based dynamic materialization view selection algorithm.

The parallel graph mining algorithm was executed on an Intel Xeon E5-2687W V4 3.0 GHz 12-core LGA 2011 processor by importing its input (set of query plans)

from the Oracle database. The implementation of the Query Cost Model (executed on the Xeon 12-core LGA processor) makes 50 evolutions from the initial population and selects the solution (recommended view set) that has the highest fitness value from the last (i.e., 50^{th}) evolution. The recommended view set for each workload is exported to the Oracle database, which records the database performance of the proposed method for the given workload. This experimentation was done using the standard query workload mentioned in [17] as well as real and synthetic datasets. The performance on the standard workload is compared to the recently established algorithms mentioned in [4, 15], and [17] using GAIN measure (GM) [15] as a performance criterion. The optimization tests were carried out using the standard query workload on four different data warehouses of various sizes (from 0.5 GB to 2 GB). The results of the experimentation are given in Figure 7.

The experimentation results in Figure 7 indicate that there is a large improvement in proposed method's GM as compared to that of recent methods for all four sizes of query workloads.

We have also used synthetic and real-life datasets for the experimentation to test the applicability of the proposed method on various types of queries. The real data set is obtained from the Management Information System of the National Institute of Technology in Nagpur. The workload queries in real-life applications consist of time-consuming operations such as joins, aggregations, and groupings. We made sure to have more variations in the datasets in the form of aggregations and joins. The composition of the data sets is described below (Table 1).

Table 1
Dataset characteristics

SN	QL	DS	N	QJ	QA	QJA
1	QS1	Real-MIS	2087	48	17	29
2	QS2	Synthetic	3086	26	28	35
3	QS3	Synthetic	2809	20	39	32

SN – Serial number
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

The datasets were cached in the main memory during the algorithm-processing stage to avoid high data access costs. The experimentation was performed by setting the frequency threshold to 50% of the total candidate trees (the threshold is taken as 50% with the assumption that around 50% of the total workload will have frequent

patterns. If more queries are to be optimized, then threshold can be reduced). The performance is measured using GM. The performances with different query loads are shown in Figure 8 and Table 2.

Table 2
Comparison of proposed method with existing methods

QL	LR	View Selection Algorithms			
		MVFI	ARMMVVM	CBMVS	PROPOSED
QS1	3,560,642	24.34	21.21	15.38	37.85
QS2	4,701,867	33.68	29.24	27.45	42.67
QS3	3,857,673	31.25	28.41	24.37	39.81

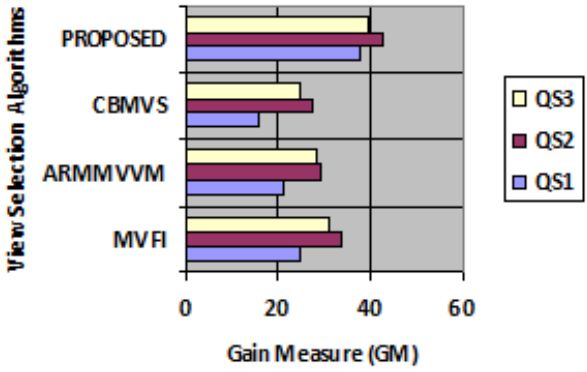

Figure 8. Performance Statistics on QS1, QS2, & QS3

From Figure 8, it is interpreted that the GM is considerably increased using the proposed tree mining algorithm and query cost model as compared to state-of-the-art algorithms on different query loads.

8.1. Time efficiency with parallel execution

The parallel graph mining algorithm was tested on an Intel Xeon E5-2687W V4 3.0 GHz 12-core LGA 2011 processor. We have three datasets (QS1, QS2, and QS3) with different complexities (varying percentages of complex queries). The maximum speeds of up of 7.93, 5.93, and 4.30 were recorded for the QS1, QS2, and QS3 datasets on 12 cores. The details are described in Table 3. The speed-up and execution times on different numbers of cores within a range of 1 to 12 are shown in Figures 9 and 10. The results show that the performance improvement of our algorithm is scalable with increasing numbers of cores on all the three datasets.

Table 3
Performance with and without partitioning

SN	Query load	No. of partitions	Execution time (without parallel execution) [s]	Execution time (with parallel execution) [s]
1	QS1	154	391.20	49.49
2	QS2	127	567.23	96.56
3	QS3	111	496.21	112.57

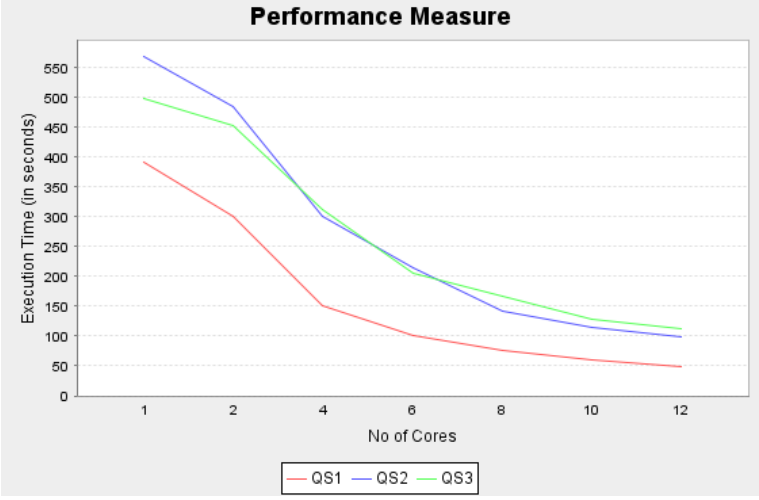

Figure 9. Execution times on three datasets for different numbers of cores

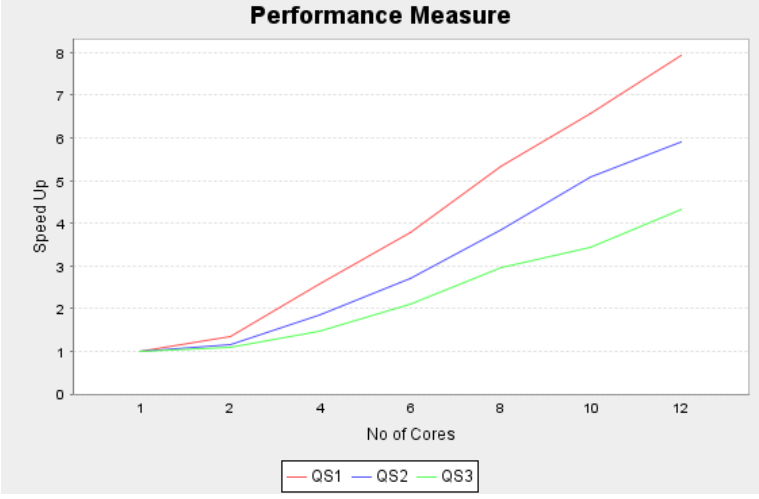

Figure 10. Speed-up on three datasets for different numbers of cores

9. Conclusion

In this paper, we considered the problem of selecting views to optimize the aggregate queries so that the sum of the query evaluation and view maintenance costs for workload queries can be minimized. The algorithm proposed in this paper analyzes the query plans of all of the historical queries and aims at finding frequent queries as well as frequent query components. The creation of materialized views on frequent components will result in the optimization of all of the queries having these components. The paper also proposes various pruning techniques to effectively reduce the search space and apply clustering and parallel execution to improve the performance of a view selection algorithm to combat the huge query load. The experimental results obtained from an implementation on an Oracle SQL Server showed that the algorithm is fast and scales to very large numbers of queries.

The proposed method is compared with standard workloads mentioned in the literature, and its performance is compared with recent methods available in the literature. The experimental evaluation indicates that the proposed method gives a better performance than all of the recent methods irrespective of query load size. The experimentation also proves that the algorithm is highly time-efficient on a multicore architecture system, as the computations are decomposed into many independent parts that can be executed on different cores. A highest speed of up of 7.9 is recorded on an Intel Xeon 3.0 GHz LGA processor with 12 cores. The detailed study is done on real and synthetic data sets to check the performance on various types of workloads.

The experimental evaluation indicates that the performance is improved to a large extent by the proposed method on all of the examined datasets. The experimental results also show that the proposed Query Cost Model is successful in finding the fairly optimal solution on the varieties of workloads used in the experimentation.

References

[1] Afrati F., Chirkova R.: Selecting and using views to compute aggregate queries, *Journal of Computer and System Sciences* vol. 77(6), pp. 1079–1107, 2011. https://doi.org/10.1016/j.jcss.2010.10.003.

[2] Aouiche K., Jouve P., Darmont J.: Clustering-Based Materialized View Selection in Data Warehouses. In: Manolopoulos Y., Pokorn J., Sellis T.K. (eds.), *Advances in Databases and Information Systems. 10th East European Conference, ADBIS 2006, Thessaloniki, Greece, September 3–7, 2006. Proceedings*, Lecture Notes in Computer Science, vol. 4152, Springer, Berlin–Heidelberg, pp. 81–95, 2006. https://doi.org/10.1007/11827252_9.

[3] Ezeife C.I.: A uniform approach for selecting views and indexes in a data warehouse. In: *Proceedings of the 1997 International Database Engineering and Applications Symposium (Cat. No.97TB100166)*, Montreal, Quebec, Canada, pp. 151–160, 1997. https://doi.org/10.1109/IDEAS.1997.625671.

[4] Gong A., Zhao W.: Clustering-Based Dynamic Materialized View Selection Algorithm. In: *2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery, FSKD'08*, Shandong, China, pp. 1333–1348, 2008. https://doi.org/10.1109/FSKD.2008.96.

[5] Goswami R., Bhattacharyya D.K., Dutta M.: Materialized view selection using evolutionary algorithm for speeding up big data query processing, *Journal of Intelligent Information Systems*, vol. 49(3), pp. 407–433, 2017. https://doi.org/10.1007/s10844-017-0455-6.

[6] Gupta H., Selection of views to materialize in a data warehouse. In: Afrati F., Kolaitis P. (eds.), *Database Theory – ICDT'97. 6th International Conference Delphi, Greece, January 8–10, 1997 Proceedings*, Lecture Notes in Computer Science, vol. 1186, Springer, Berlin–Heidelberg, pp. 98–112, 1997. https://doi.org/10.1007/3-540-62222-5_39.

[7] Gupta H., Mumick I.S.: Selection of views to materialize in a data warehouse, *IEEE Transactions on Knowledge and Data Engineering*, (17)1, pp. 24–43, 2005. https://doi.org/10.1109/TKDE.2005.16.

[8] Gupta H., Mumick I.S.: Selection of Views to Materialize Under a Maintenance Cost Constraint. In: Beeri C., Buneman P. (eds.), *Database Theory – ICDT99. 7th International Conference Jerusalem, Israel, January 10–12, 1999 Proceedings*, Lecture Notes in Computer Science, vol. 1540, Springer, Berlin–Heidelberg, pp. 453–470, 1999. https://doi.org/10.1007/3-540-49257-7_28.

[9] Harinarayan V., Rajaraman A., Ullman J.D.: Implementing data cubes efficiently. In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pp. 205–216, 1996. https://doi.org/10.1145/235968.233333.

[10] Horng, J-T., Chang Y-J., Liu B-J.: Applying evolutionary algorithms to materialized view selection in a data warehouse, *Soft Computing*, vol. 7(8), pp. 574–581, 2003. https://doi.org/10.1007/s00500-002-0243-1.

[11] 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(6), pp. 1333–1348, 2007. https://doi.org/10.1016/j.ins.2006.09.007.

[12] Hylock R., Currim F.: A maintenance centric approach to the view selection problem, *Information Systems*, vol. 38(7), pp. 971–987, 2013. https://doi.org/10.1016/j.is.2013.03.005.

[13] 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(6), p. 1740001, 2017. https://doi.org/10.1142/S0218539317400010.

[14] Lin W.Y., Kuo I.C.: A Genetic Selection Algorithm for OLAP Data Cubes, *Knowledge and Information Systems*, vol. 6(1), pp. 83–102, 2004. https://doi.org/10.1007/s10115-003-0093-x.

[15] Mohammad K.S., Vahid G.: Materialized View Selection for a Data Warehouse Using Frequent Itemset Mining, *Journal of Computers*, vol. 11(2), pp. 140–148, 2016. https://doi.org/10.17706/jcp.11.2.140-148.

[16] Ross K.A., Srivastava D., Sudarshan S.: Materialized view maintenance and integrity constraint checking: trading space for time. In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pp. 447–458, 1996. https://doi.org/10.1145/235968.233361.

[17] Vishwanath P.R., Rajyalakshmi, Reddy S.: An Association Rule Mining for Materialized View Selection and View Maintenance, *International Journal of Computer Applications*, vol. 109(5), pp. 15–20, 2015. https://doi.org/10.5120/19184-0670.

[18] Yang J., Karlapalem K., Li Q.: Algorithms for Materialized View Design in Data Warehousing Environment. In: *VLDB'97 Proceedings of the 23rd International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, San Francisco, pp. 136–145, 1997.

[19] Yang J., Karlapalem K., Li Q.: A framework for designing materialized views in data warehousing environment. In: *Proceedings of 17th IEEE International Conference on Distributed Computing Systems*, Maryland, USA, 1997. https://doi.org/10.1109/ICDCS.1997.603380.

[20] Zhang C., Yang J.: Genetic Algorithm for Materialized View Selection in Data Warehouse Environments. In: Mohania M., Tjoa A.M. (eds.), *DataWarehousing and Knowledge Discovery. First International Conference, DaWaK'99 Florence, Italy, August 30 – September 1, 1999 Proceedings*, Lecture Notes in Computer Science, vol. 1676. Springer, Berlin–Heidelberg, pp. 116–125, 1999. https://doi.org/10.1007/3-540-48298-9_12.

Affiliations

Atul Thakare
Visvesvaraya National Institute of Technology, Computer Science & Engineering Department, South Ambazari Road, Nagpur (Maharashtra) India 440010, aothakare@gmail.com, ORCID ID: https://orcid.org/0000-0003-3897-5973

Parag Deshpande
Visvesvaraya National Institute of Technology, Computer Science & Engineering Department, South Ambazari Road, Nagpur (Maharashtra) India 440010, psdeshpande@cse.vnit.ac.in, ORCID ID: https://orcid.org/0000-0003-4051-4666

Received: 06.08.2018
Revised: 30.08.2018
Accepted: 12.09.2018