ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20230501083155

Original scientific paper
Received: 2023-05-01, Accepted: 2023-05-27

QOTUM: The Query Optimizer for Distributed Database in Cloud Environment

Archana Bachhav*, Vilas Kharat, Madhukar Shelar

Abstract: In distributed and cloud database system, large amount of resources are used by complex queries that increase in payment overheads of cloud users. These resource needs can be minimised by evaluating common join operations and caching their results so that they could be applied to the execution of additional queries. In this research, the Query Optimization Technique Using Materialization (QOTUM) system is designed and developed that uses the mechanism of materialized views during query execution at distributed database in cloud environment. Extensive experiments are conducted with the help of standard benchmarks datasets and query workloads in cloud environment. The performance of QOTUM system is studied and evaluated against conventional SQL system based on various performance parameters.

Keywords: cloud computing; distributed databases; materialized views; query optimizer; query optimization technique

1 INTRODUCTION

This paper continues the earlier research we published in Bachhav et al. [1]. To perform complex queries on large databases more effectively, cloud customers can hire a lot of resources for a short period of time with the help of virtual machines [2]. One of the most difficult research problems in the Big Data age is how to make the requirements for relational query processing efficiency across clouds [3].

Through the use of stronger query optimisation strategies, the cost of hiring resources for cloud users can be decreased [4]. The devise of *Query Optimization Technique Using Materialization* (*QOTUM*) is the main contribution of this research in the field of database query optimization. It materializes views during query execution at distributed database in cloud environment. The *QOTUM* system will provide the following contributions in the field of database management and query optimization in distributed system on cloud:

- It can be used as a pre-processor for effective query optimization on the top of any databases system.
- The design of query optimizer that uses materialized views to execute future queries.
- An optimization strategy has been introduced that reduces total execution cost of queries in distributed databases which will be more beneficial in cloud environment.
- Cloud resource utilization has been improved by reducing query evaluation time and overall execution cost.

QOTUM system has been developed on Java platform with the help of ZQL Parser [5]. The performance of the system is tested on cloud platform of Amazon Relational Database Service (RDS) [6] using standard benchmark dataset and the workload of TPC-H [7]. The MySQL database engine is initially setup on Amazon RDS instances and TPC-H database is distributed among all these RDS instances. Numerous tests have been carried out using all different types of TPC-H query workloads.

Remainder of this research paper is organized as follows. The working of *QOTUM* system is elaborated in section 2. Section 3 presents an experimental setup for testing performance of *QOTUM*. The result analysis is explained in section 4. Observations and Plugin to deal with *QOTUM* system are presented in section 5 and section 6 concludes the work with future directions.

2 QOTUM (QUERY OPTIMIZATION TECHNIQUE USING MATERIALIZATION)

After extensive literature survey published in our paper [1, 8] on numerous methods of database query optimization, it is determined that there can be a query optimization technique which will work as a pre-processor on the top of conventional database management system. Query optimization technique named *QOTUM* uses the mechanism of materialized views during query execution at distributed database in cloud environment. The main aim of this study to innovate the query optimizer in distributed and cloud environment that reduces overall query execution cost. *QOTUM* materializes results generated during query evaluation that can be reused for evaluation of additional queries. The Fig. 1 shows architecture of *QOTUM*, the query optimizer technique using materialized views presented in [1, 9].

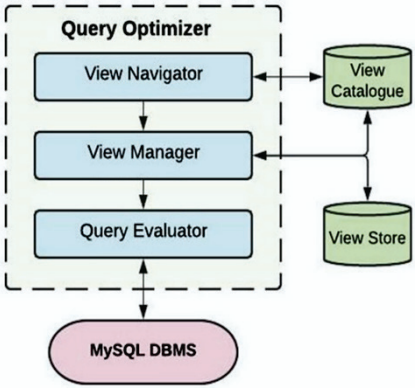

Figure 1 An Architecture of *QOTUM* [1, 9]

During evaluation of a future query, if materialized view matches with any of its subqueries then instead of evaluating that matched subquery again, materialized view can be directly used for further processing [10-13]. Aggregation

queries across big relations can be processed more quickly by using materialised views [14, 15].

The relations requested by a query can be stored in several places in cloud and distributed systems. The cost of moving data between various cloud sites should now be taken into account in addition to the cost of local computing [16]. It can be difficult to determine the best join order for queries with many relations in distributed systems since the search space expands exponentially as the number of relations increases [17]. *QOTUM* trying to reduce the execution time, I/O operations and communication cost.

2.1 How *QOTUM* works?

QOTUM system performs the query optimization process with the help of three modules – View Manager, View Navigator and Query Evaluator by maintaining View Store and View Catalogue as elaborated in Algorithm 1.

QOTUM performs the following steps.

1. Submitted query is divided into subqueries (Makes use of Zql Parser [5]).
2. Finds the relation names and predicates from subqueries
3. View Navigator navigates the View Catalogue and using the view matching algorithm provided by Bachhav et al. [1], attempts are made to locate the view store entry to find longest match view name.
4. View Manager finds a matching view name in the view catalogue and maps it to the view store.
5. If match is found, then returns the matched position of View Store. Additionally, partly interim findings are used to evaluate upcoming queries. The commutative rule for natural join is followed during match.
6. The query execution is performed after replacing the matched views in a query.
7. After evaluation of query, the generated views are maintained in View Store by View Manager.
8. View Manager also updates the View Catalogue after storing views in view store.
9. View Manager updates the read/write timestamps and reference count for the view which is referred.
10. When View Store becomes full, Views those are not referred from longest period of time are deleted by View Manager and also updates the corresponding entries in View Catalogue by invoking logistic regression algorithm as per step 11.
11. The value of Read/Write timestamp for a view in View Store is used to take decision for deletion of view from View Store.
    Logistic Regression algorithm of Machine Learning [18] is applied to classify View Store entries into two classes.
    (I) Entries to be deleted
    (II) Entries not to be deleted.
    Logistic regression algorithm uses Eq.(1) to find the value between 0 and 1 that predicts in which class a View Store entry belongs [19].
    $$y = \frac{1}{1 + e^{-x}} \quad (1)$$
    Where x is read/write timestamp of view, and y is the predicted value generated.
    A threshold value is set to classify the predicted value into appropriate class.
    For example – if threshold value is 0.5 and predicted value y is 0.3 then our predicted value <0.5, will classify it as the class "Entry to be deleted", otherwise classify it as the class "Entry not be deleted".
12. Materialized view get removed from View Store, if there are updates in base relations involved in it to avoid inconsistency in resultant set (Details presented in section 5.2)

Algorithm 1 Algorithm for *QOTUM*

	Algorithm *QOTUM*
	Input: - Query to be evaluated Output: - Updated View Store - Updated View Catalogue
1	Begin
2	Split a query into subqueries using Zql Parser
3	Get view pattern used in subqueries
4	Location ← *Call Algorithm ViewMatching(catalogue, view pattern)*[1]
5	If location is in View Catalogue
6	Get corresponding Materialized Views from View Store
7	Evaluate query using Materialized Views
8	Else
9	Evaluate query using conventional method
10	End If
11	If View Store is Full
12	Find appropriate views to remove using Logistic Regression algorithm
13	Remove such views from View Store
14	Update corresponding entries in View Catalogue
15	End If
16	Materialize the newly generated view in View Store
17	Add entry into View Catalogue about newly generated view
18	Update View Reference Count and Read/Write Timestamps in View Store
19	End

3 EXPERIMENTAL SETUP

The *QOTUM* System is developed on Java platform using Zql Parser Java library. The performance of this

technique has been already tested at small scale level where horizontal fragmentation is applied on benchmark relations and fragments are randomly distributed on two nodes [1]. However, this research paper presents the performance of *QOTUM* at large scale level on cloud platform of Amazon Relational Database Service (RDS) using standard benchmark dataset and the query workload of TPC Benchmark™H (TPC-H). TPC-H is made up of a collection of business databases and queries that are intended to test the functionality of any sophisticated business analysis applications [7]. The process of a wholesale supplier has been realistically shown by the database and queries. Given SQL statements in TPC-H workload are parsed by Zql parser and represent it in Java data structures [5].

3.1 Database and Workload for Experiments

The TPC-H database consists of 8 individual base tables with different relationship between them. Tab. 1 illustrates the TPC-H Schema with the relationships between different tables. The number of rows (cardinality) of each table is represented using the number or formula where SF represents the Scale Factor which can be chosen by researcher to obtain the required database size. The total storage space required with $SF = 1$ is 2.21 GB.

The testing has been performed after fragmentation and distributing these benchmark relations as per the distribution scheme specified in section 3.2 over different instances of Amazon RDS after applying the horizontal fragmentation. For the TPC-H benchmark, a set of 22 original queries were used to provide a realistic context that reflected the activity of a wholesale supplier.

Table 1 TPC-H schema relations [7]

Sr. No.	Table name	Cardinality
1	Region	5
2	Nation	25
3	Part	$SF \times 2{,}00{,}000$
4	Supplier	$SF \times 10{,}000$
5	Partsupp	$SF \times 8{,}00{,}000$
6	Customer	$SF \times 1{,}50{,}000$
7	Order	$SF \times 15{,}00{,}000$
8	Lineitem	$SF \times 60{,}00{,}000$

Table 2 Query workload variations [1]

Workload	Description
W1	Set of queries without common join operations
W2	Set of queries with about 25% common join operations
W3	Set of queries with about 50% common join operations
W4	Set of queries with about 75% common join operations
W5	Set of queries with 100% common join operations

Five different workload variations (shown in Tab. 2) each containing 100 queries were created using 22 TPC-H base queries to evaluate the performance of the *QOTUM* system. With the aid of the right range of probable values according to the benchmark, the selection predicates were constructed for each query in accordance with the defined workload. The effectiveness of the *QOTUM* has been assessed using a variety of performance metrics, including the number of iterations, query execution time, memory usage, and overall execution cost.

3.2 Database Setup on AWS-RDS

The Relational Database Service from Amazon Web Services (AWS-RDS) is reasonably priced, easily scaled, and configurable [20]. RDS MYSQL is a cloud-based database that is fully managed and provides high availability [21, 22]. Hence AWS-RDS with MySQL database engine has been used on cloud to deploy the TPC-H database and to test the workload of queries. Total 10 instances have been setup on AWS-RDS each of *db.t2.micro* class with the configuration of 1 vCPU, 1 GB RAM and 20 GB SSD storage. The TPC-H benchmark database of 2.21 GB is used for testing effectiveness of *QOTUM* on Amazon RDS. The TPC-H benchmark database is deployed on 10 RDS instances named as Instance-1 to Instance-10 as per the distribution mentioned in Tab. 3, for conducting various experiments.

RDS Instance-1 stores the whole TPC-H database of 2.21 GB and other instances maintain the horizontal fragments of the database. Initially for first slot of experiments, database is horizontally partitioned into 2 fragments and deployed among Instance-2 and Instance-3. To conduct second slot of experiments, database is partitioned into 3 fragments and deployed among Instance-4 to 6. Final slot of experiments is conducted on Instances-7 to 10, those holds 4 horizontal partitions of database.

Table 3 TPC-H database distribution on Amazon RDS instances

AWS-RDS Instances	Database Distribution
Instance-1	Maintains the whole TPC-H database
Instance-2	Maintains the fragments generated from 2 horizontal partitioning of TPC-H database
Instance-3	
Instance-4	Maintains fragments generated from 3 horizontal partitioning of TPC-H database
Instance-5	
Instance-6	
Instance-7	Maintains fragments generated from 4 horizontal partitioning of TPC-H database
Instance-8	
Instance-9	
Instance-10	

4 RESULTS AND DISCUSSION

All combinations of the stated query workloads have been used to conduct extensive experiments. Based on a number of performance parameters, the performance of *QOTUM* is examined and compared with that of a traditional SQL system.

4.1 Experiments Conducted

The MySQL database engine was initially setup on AWS-RDS instances and TPC-H database was distributed among all those RDS instances. The *QOTUM* system which is developed in Java has been tested by conducting various experiments on all deployed RDS instances by providing variety of query workloads. Tab. 4 describes the list of performance parameters those are considered for the comparative study.

All different types of query workloads are used to run experiments using fragments 2, 3 and 4. Primarily, the experiment was conducted on the RDS Instance-1 that holds the whole TPC-H database without partitioning. The

distributed database environment is provided to test the performance of the system. The system is then tested on two partitions of TPC-H database viz. Instance-2 and Instance-3, followed by on three partitions Instance-4 to Instance-6 and finally on four partitions Instance-7 to Instance-10. All test experiments are conducted by providing query workloads W1 to W5.

Table 4 List of Performance Parameter

Performance Parameter	Description
Average number of iterations	Average number of iterations required to complete the execution of all queries in workload
Average execution time	Average execution time required for execution of all queries in workload
Total memory consumption	Total memory consumed by all queries in query workload
Total cost of execution	The total cost required to execute all queries in workload comprises of time for execution, memory consumption, cost required for IO operations and view matching time in view store.

The system *QOTUM* acts as a pre-processor because the results generated from *QOTUM* are in the form of optimized queries, which can be provided as inputs to the conventional RDBMS like MySQL for further evaluation.

4.2 Results

Results are evaluated with traditional SQL system on various parameters such as number of iterations, execution time, memory consumption and total execution cost. The analysis of summarized results is presented using charts in Figs. 2 to 4. The comparative study on average execution time in conventional as well as *QOTUM* system is presented with the help of chart in Fig. 5.

5 OBSERVATIONS AND PLUG-IN

5.1 Observations

- We observed the substantial reduction of execution time in the *QOTUM* system as compare to conventional system for query workloads W2 to W5. Repetition of join operations in query workloads W2 to W5 takes the benefit of materialization that results in substantial reduction of execution time.
- However, in workload W1 no repetition of join operations, so it does not take a benefit of materialization. The *QOTUM* system adds an extra overhead of View Store searching time in query execution time. It leads to degrade the performance of the *QOTUM*.
- In the *QOTUM* system, average memory requirement for query processing decreases with increase in repetition of join operations using workloads W2 to W4.
- In *QOTUM* system, for workload W1 View Store becomes full more frequently than in workload W2 to W4 because in W1 there are no repeated joins, every query after execution adds new view in View Store.

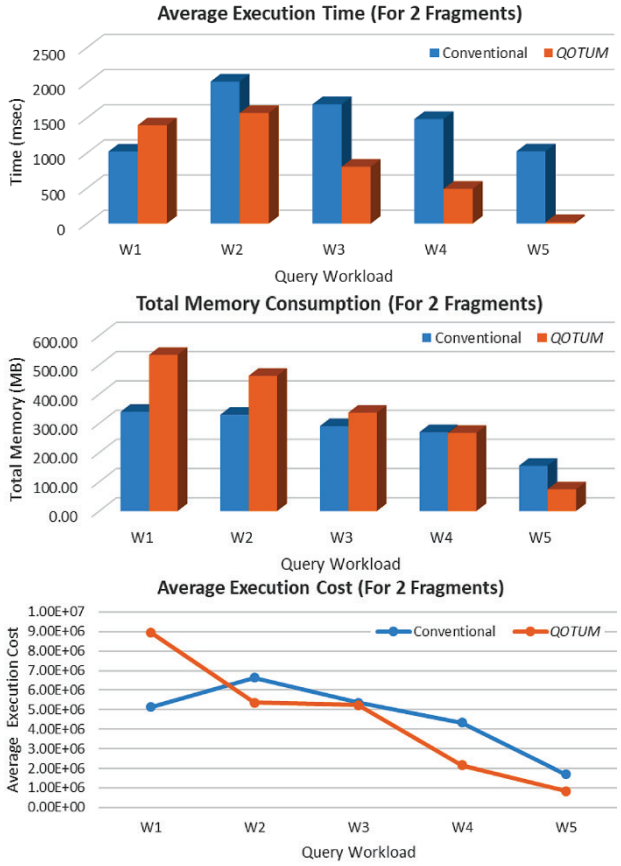

Figure 2 Comparative analysis on performance parameters for 2 fragments

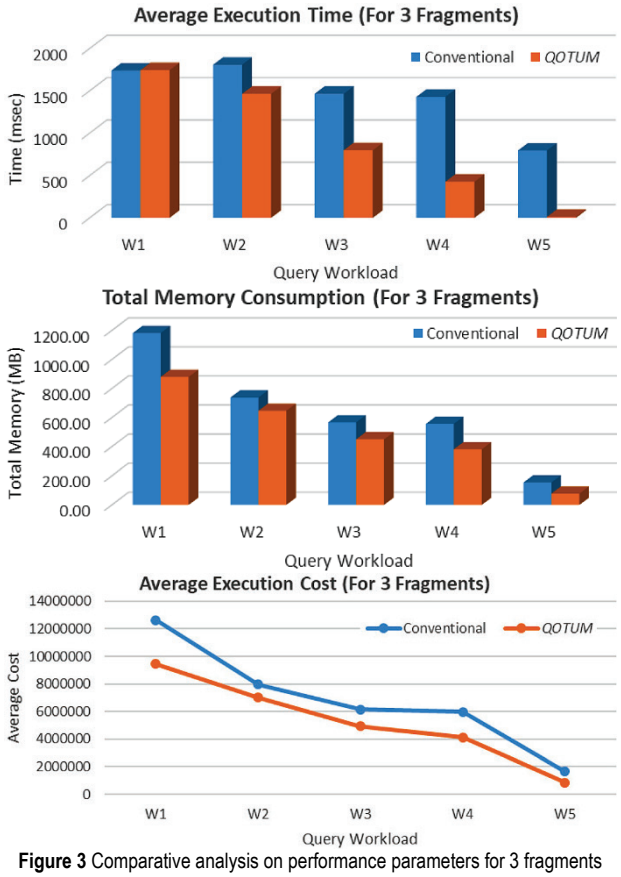

Figure 3 Comparative analysis on performance parameters for 3 fragments

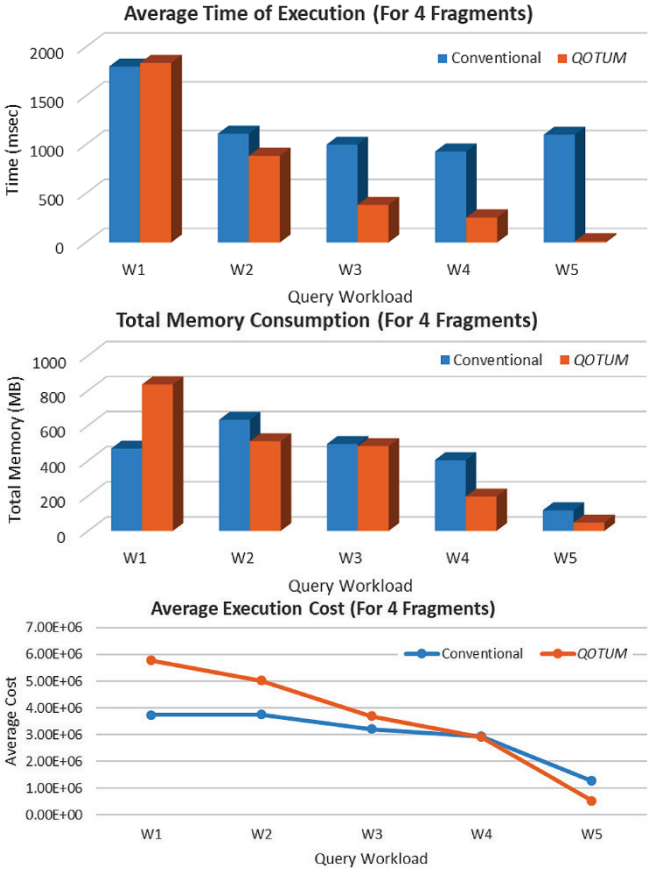

Figure 4 Comparative analysis on performance parameters for 4 fragments

- Due to materialization in *QOTUM* system, number of iterations gradually decreases with increase of repeated join operations in workloads W2 to W5. But it remains almost same for conventional and *QOTUM* using workload W1.
- The *QOTUM* system is more efficient with increase in number of fragments and repeated join operations.

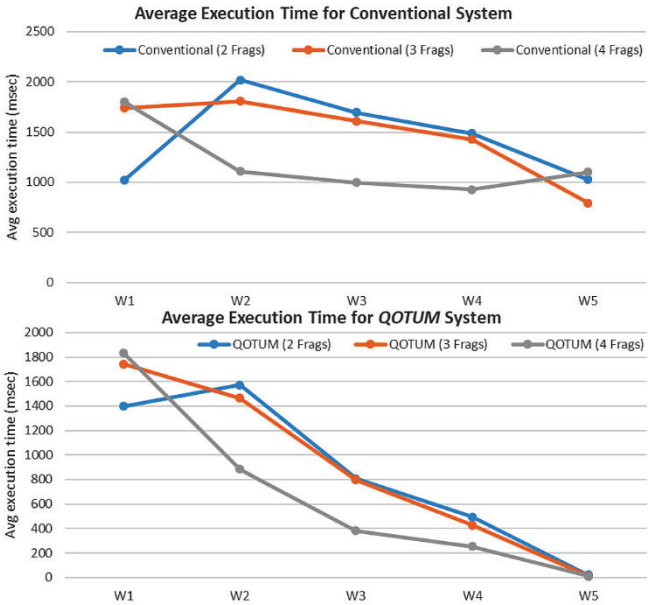

Figure 5 Comparative study on execution time with 2,3 and 4 fragments

5.2 Plug-In to deal with *QOTUM* System

Materialized views those are stored in View Store comprise of various base relations. However, there may be updates time to time in base relations that are involved in materialized views which will make those materialized views stale. Hence, to handle the problem of stale materialization, trigger and store procedure should be plugged in to the database in order to remove stale views. *QOTUM* System handles the stale materialization with the help of trigger, stored procedure and java class as mentioned in Tab. 5.

Table 5 Handling Stale Materialization

Plugin	Plugin Content

Java Class	import java.io.*;
	public class Rmview
	{ public static void remove(tablename)
	{ //Remove views from viewstore
	//matches with tablename
	}
	}

Stored Procedure	CREATE OR REPLACE PROCEDURE Remove_View(tablename) AS LANGUAGE JAVA NAME 'Rmview.remove(tablename)';

Trigger	CREATE OR REPLACE TRIGGER tablename_upd
	AFTER INSERT, DELETE, UPDATE ON tablename
	begin
	Call Remove_View(tablename);
	end;

6 CONCLUSION AND FUTURE DIRECTION OF RESEARCH

In this research work, the query optimization technique, *QOTUM* has been presented. The technique seeks to reuse intermediate results from previously run searches to execute new queries. The intermediate or materialized views can be used to execute queries that have repeating joins or similar sub expressions, which reduce the average execution time per query and improve system performance. As a result of the materialisation of intermediate views, the *QOTUM* system has been put through testing on large scale cloud infrastructure, and it has been found to significantly lower query execution time and memory requirement when compared to the conventional query processing system. In this system, the overall cost of query execution is inversely proportional to the number of repeated joins in query

workloads. That means increase in number of repeated joins leads to decrease in execution cost.

In cloud environments, Map-Reduce platforms such as Hadoop works on NoSQL or non-relational database management systems that are now become standard for large-scale data processing, but they increase overall processing cost for join-intensive workloads [23]. However, the *QOTUM* can only works on top of relational database management system. Hence as a future direction of the research, this system can also be extended to work on the top of NoSQL or non-relational database management system that will reduce the overall processing cost of join intensive workloads using Map-Reduce.

7 REFERENCES

[1] Bachhav, A., Kharat, V. & Shelar, M. (2021). An Efficient Query Optimizer with Materialized Intermediate Views in Distributed and Cloud Environment. *Tehnički Glasnik, 15*(1), 105-111. https://doi.org/10.31803/tg-20210205094356

[2] Dokeroglu, T., Sert, S. A. & Cinar, M. S. (2014). Evolutionary Multiobjective Query Workload Optimization of Cloud Data Warehouses. *The Scientific World Journal, 2014*, Article ID 435254. https://doi.org/10.1155/2014/435254

[3] Cuzzocrea, A., Karras, P. & Vlachou, A. (2022). Effective and efficient skyline query processing over attribute-order-preserving-free encrypted data in cloud-enabled databases. *Future Generation Computer Systems, 126*, 237-251. https://doi.org/10.1016/j.future.2021.08.008

[4] Doshi, P. & Raisinghani, V. (2011). Review of Dynamic Query Optimization Strategies in Distributed Database. ICECT 2011 - *The 3rd International Conference on Electronics Computer Technology, 6*, 145-149. https://doi.org/10.1109/ICECTECH.2011.5942069

[5] Zql Parser. http://zql.sourceforge.net/.

[6] Amazon Relational Database. https://aws.amazon.com/rds/.

[7] Council, Transaction Processing Performance. TPC-H - Ad-Hoc, Decision Support Benchmark. http://www.tpc.org/tpch/.

[8] Bachhav, A., Kharat, V. S. & Shelar, M. N. (2017). Query Optimization for Databases in Cloud Environment: A Survey. *International Journal of Database Theory and Application, 10*(6), 1-12. https://doi.org/10.14257/ijdta.2017.10.6.01

[9] Bachhav, A., Kharat, V. S. & Shelar, M. N. (2018). Novel Architecture of an Intelligent Query Optimizer for Distributed Database in Cloud Environment. *Journal of Advanced Database Management & Systems, 5*(2), 28-32.

[10] Perez, L. L. & Jermaine, C. M. (2014). History-Aware Query Optimization with Materialized Intermediate Views. Proceedings *30th IEEE International Conference on Data Engineering*, Chicago, IL, USA, 520-531. https://doi.org/10.1109/ICDE.2014.6816678

[11] Tan, K.-L., Goh, S.-T. & Ooi, B.-C. (2001). Cache-on-Demand: Recycling with Certainty. *Proceedings 17th International Conference on Data Engineering*, Heidelberg, Germany, 633-640. https://doi.org/10.1109/icde.2001.914878

[12] Kossmann, D., Franklin, M. J. & Drasch, G. (2000). Cache Investment: Integrating Query Optimization and Distributed Data Placement. *ACM Transactions on Database Systems, 25*(4), 517-558. https://doi.org/10.1145/377674.377677

[13] Phan, T. & Li, W.-S. (2008). Dynamic Materialization of Query Views for Data Warehouse Workloads. *Proceedings 24th IEEE International Conference on Data Engineering*, Cancun, Mexico, 436-445. https://doi.org/10.1109/ICDE.2008.4497452

[14] Goldstein, J. & Per-Åke, L. (2001). Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. *ACM SIGMOD Record, 30*(2), 331-342. https://doi.org/10.1145/376284.375706

[15] Ivanova, M. G., Kersten, M. L., Nes, N. J. & Gonçalves, R. A. P. (2010). An Architecture for Recycling Intermediates in a Column-Store. *ACM Transactions on Database Systems, 35*(4). https://doi.org/10.1145/1862919.1862921

[16] Azhir, E., Navimipour, N. J., Hosseinzadeh, M., Sharifi, A., Unal, M. & Darwesh, A. (2022). Join queries optimization in the distributed databases using a hybrid multi-objective algorithm. *Cluster Computing*, 1-16. https://doi.org/10.1007/s10586-021-03451-9

[17] Mancini, R., Karthik, S., Chandra, B., Mageirakos, V. & Ailamaki, A. (2022, June). Efficient massively parallel join optimization for large queries. In Proceedings of the 2022 International Conference on Management of Data (pp. 122-135). https://doi.org/10.1145/3514221.3517871

[18] de Menezes, F. S., Liska, G. R., Cirillo, M. A. & Vivanco, M. J. F. (2017). Data Classification with Binary Response through the Boosting Algorithm and Logistic Regression. *Expert Systems with Applications, 69*, 62-73. https://doi.org/10.1016/j.eswa.2016.08.014

[19] Rao, S. J. (2003). Regression Modeling Strategies: With Applications to Linear Models, Logistic Regression, and Survival Analysis. *Journal of the American Statistical Association, 98*(461), 257-258. https://doi.org/10.1198/jasa.2003.s263

[20] Salecha, R. (2023). *Introduction to AWS. In: Practical GitOps*. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-8673-9_2

[21] Juncosa Palahí, M. (2022). Platform for deploying a highly available, secure and scalable web hosting architecture to the AWS cloud with Terraform (*Bachelor's thesis*, Universitat Politècnica de Catalunya).

[22] Soni, R. K., Soni, N., Soni, R. K. & Soni, N. (2021). Deploy a Spring Boot Application Talking to MySQL in AWS. *Spring Boot with React and AWS: Learn to Deploy a Full Stack Spring Boot React Application to AWS*, 103-141. https://doi.org/10.1007/978-1-4842-7392-0_4

[23] Anyanwu, K., Kim, H. & Ravindra, P. (2013). Algebraic optimization for processing graph pattern queries in the cloud. *IEEE Internet Comput., 17*(2), 52-61. https://doi.org/10.1109/MIC.2012.22

Authors' contacts:

Archana Bachhav
(Corresponding author)
MVP Samaj's KSKW Arts,
Science and Commerce College,
Ambad-Uttam Nagar Road, CIDCO, Nashik, Maharashtra 422008, India
77.archana@gmail.com

Vilas Kharat
School of Mathematical and Computational Sciences,
S. P. Pune University, Pune, India
laddoo1@yahoo.com

Madhukar Shelar
MVP Samaj's Commerce,
Management & Computer Science (CMCS) College,
Nashik, India
mnshelar70@gmail.com