Journal of Science and Technology
ISSN: 2362-1443 (Print) 3059-9490 (Online) 2026, Vol. 5, No. 1

Distributed Query Optimization by Index Tuning

Rabindra Tandukar
Nepal College of Information Technology,
Pokhara University, Nepal
ravindratandukar@gmail.com

Anil Verma
Pulchowk Campus, Institute of Engineering
Tribhuvan University, Nepal
anil@ioe.edu.np

Abstract - Query language access a data from databases. When data grows exponentially, the optimization techniques must to be adopt for better results. Query performance tuning and optimization can be achieved by query reformation and/or index selection. A database index is a way of sorting a number of records on multiple fields. Creating an index on a field in a table creates another data structure which holds the field value, and a pointer to the record it relates to. The creation of index in database adds more complexity in data storage, memory consumptions as well as computational complexity. It causes the significant impact on data manipulation operation. On distributed database the query execution optimization is more complex due to various sites are responsible to execute a query and also data is not stored in single site but is distributed to the network. Hence it is needed to define the indexes in more efficient way to reduce the complexities. The proposed method is to reduce the overhead of the data manipulation operation due to index creation; the exploration of multiple column index to identify the minimum number of index creation for optimum level of the database storage consumption and efficient query execution for most frequently used query.

Keywords: Database Index, index selection, Index Performance, Distributed Database, Optimization, multiple column index

I. INTRODUCTION

Designing efficient indexes is vital in achieving the good database and application performance. Poorly designed indexes and lack of indexes are primary sources of database application performance bottlenecks. Indexes can be added, modified, and dropped without affecting the database schema or application design. It does not always equate index usage with good performance, and good performance with efficient index use. The overall index design strategy should provide the variety of indexes for the query optimizer to choose from and trust it to make the right decision. Indexes are very important in query data from database, especially in large scale of data [6]. Rational use of index technology is essential in improving database's query performance. SQL Server database use B+ tree structure to store indexes. Clustered index can change data's physical position. Both of Clustered index and non-clustered index depend on B+ index tree to query data. Non-clustered index should depend on either data row pointer or clustered index key to find the retrieve data. Two algorithms are used in SQL Server database to retrieve data. Usually, table scan should be avoided except that large-scale of data involved in the query or most columns of the table are covered in the query. Table seek has a better efficiency for most cases. Some principle of how to use index properly are presented in this thesis. The functions, calculations and some query conditions should to be avoided or replaced, or to be used as little as possible in query statement to make indexes effective.

The query optimizer in the database reality chooses the most effective index in the vast majority of cases. The overall index design strategy should provide a variety of indexes for the query optimizer to choose from and trust it to make the right decision. This reduces analyses time and produces good performance over variety of situations. The indexes on individual column can work efficiently when each column has a high selectivity [2]. The resulted access plan using hash join between two index range scans usually works quite well. Columns with low selectivity make no sense to be indexed on their own, the query optimizer will simply ignore them. However, low selectivity columns many times make good composite keys when their paired with high selectivity column. Creating an index on each column is fast in terms of lookup of either columns separately or both of them together, but it will use up more memory to store the multiple B-tree. Also, inserts and updates becomes even shower with multiple indexes. Creating a multiple column indexes gives similar speed benefits with lower memory costs and insert update overhead as long as the queries involve lookup of columns in the database. A multiple column index can be sorted considered a sorted array, the rows of which contain values that are created by concatenating the values of the indexed columns. Even though the database creates the index for the primary key automatically, there is still room for manual refinements if the key consists of multiple columns. In that case the database creates an index on all primary key columns a so-called concatenated index (also known as multicolumn, composite or combined index). The column order of a concatenated index has great impact on its usability so it must be chosen carefully. SQL Server allows to build indexes on specific columns within a table. Index can have a single column or index multiple columns (called a compound index). It is better to use to use multiple column indexes if query on all columns frequently and or there is a hierarchy in terms of which columns queried.

A distributed database system is a collection of partially independent database systems that (ideally) share a common schema, and coordinate processing of transactions that access

DOI : https://doi.org/10.3126/jost.v5i1.XXXXX

Article History: Received: 11 Nov. 2025, Revised: 12 Jan. 2026, Accepted: 19 Feb. 2026.

nonlocal data [7]. A distributed database (DDB) is a collection of data that logically belongs to the same system but is spread over the sites of a computer network [8]. It is not necessary that database system have to be geographically distributed. The sites of the distributed database can have the same network address and may be in the same room but the communication between them is done over a network instead of shared memory. As communication technology, hardware, software protocols advance rapidly and prices of network equipment's falls every day, developing distributed database systems become more and more feasible. Design of efficient distributed database is one of the major research problems in database & information technology areas.

II. RELATED WORK

Query performance tuning and optimization can be achieved by query reformation and index selection [1]. Searching tuples from millions of results is overhead and it degrades overall system performance. Index Selection Problem (ISP) is optimization problem and it can be solved by different approaches like greedy approach, dynamic programming, linear programming, branch and bound, genetic algorithm, etc. There are many algorithms are exits like Deterministic Algorithms, Randomized Algorithms, Genetic Algorithms can be used to optimize the query. Best index selection can be done by analyzing query execution plan for input query. Clustered Index, Multilevel Index, Hash, Tree, Bitmap indexes are discussed. Researchers developed different searching and selecting strategies which can help to maximized objective function. Linear programming assures optimal solution by following branch and bound. Best index selection will give fast results and also helps for future coming query. Future scope of this research is to find cost execution plan of particular query based on the number of tuples satisfy given predicate and to solve predicates based on minimum tuples return by it.

Good index structures are useless if we do not employ an intelligent query optimizer to select a suitable indexing technique to process queries [2]. Data warehouse system is becoming more and more important for decision-makers. When right index structures are built on columns, the performance of queries, especially ad hoc queries will be greatly enhanced. Indexing techniques have already been in existence for decades for the OLTP relational database system but they cannot handle large volume of data and complex and iterative queries that are common in OLAP applications. A proper indexing technique is crucial to avoid I/O intensive table scans against large data warehouse tables. B-Tree Indexes should only be used for high cardinality data and predicted queries. Bitmap Indexes play a key role in answering data warehouse's queries because they have an ability to perform operations on index level before retrieving base data. Most commercial data warehouse products except Teradata database implement Bitmap Indexes. Data mining techniques could be used to develop an intelligent optimizer whereas paralleling is another issue that we should consider.

An algebraic transformation does not necessarily reduce cost and therefore they must be applied in a cost-based manner to ensure a positive benefit [3]. The research mainly focusses on the actual objective of the Optimizer that is to find a Strategy close to the optimal. It discussed on the Search Space, Search Strategy and Processing Tree which describes the Query Optimization problem and the associated cost model. Syntactical Optimization, Cost based Optimization and Semantic Optimization techniques have been discussed. The model of Cost based Optimization is represented in the paper the diagram of the model given below. The paper tried to quantify the factors that enhance the performance of the query. It also proposed a simple operational model for Query Optimization that incorporates the modularity and flexibility necessary to implement an extensible Query Optimizer as required for the new generation of database management systems. The results of this paper will help to extend the existing Optimization techniques to the demanding requirements of new application areas. Usage of constraints and methods like introducing an additional join to the original query requires further study from an implementation point of view.

Database cracking technique is adaptive merging, an adaptive increment, and efficient technique for index creation to focus on key ranges used in actual queries and sort efficiency is comparable to that of traditional B-tree creation [4]. Nonetheless, the new technique promises better query performance than database cracking, both in memory and on block access storage. Database cracking, which combines features of both automatic index selection and partial indexes. Database cracking and adaptive merging offer a promising alternative to traditional index tuning. Database cracking was designed for in-memory arrays, adaptive merging enables automatic creation and incremental improvements of indexes in large data warehouses on "external" block-access storage. First an artificial leading key field permits creation and removal of partitions by insertion and deletion of records with specific partition identifiers. Second, index creation is divided into run generation and merging. Both can be side effects of query execution or other scans over the data. Third, query execution may optimize such an index by merging the key ranges required to answer actual queries, with no effort spent on any unused key ranges. Fourth, those non-optimized key ranges are automatically in a format that can readily be searched and optimized later if the query pattern changes. The described techniques have design goals very similar to database cracking, namely automatic and adaptive index selection as well as incremental optimization of indexes focused on key ranges of interest in actual queries.

A new data warehouse query acceleration feature based on a new index type called a column store index, the new index type combined with new query operators processing batches of rows greatly improves data warehouse query performance [5]. The paper gives an overview of the design and implementation of column store indexes including enhancements to query processing and query optimization to take full advantage of the new indexes. The resulting performance improvements are illustrated by a number of example queries. Use of partitioning to load a staging table, index it with a column store index. Using a column store index as the primary organization is not supported in the initial release; they can only be used for secondary indexes. The research has plan to lift this restriction in a future release.

III. RESEARCH FRAMEWORK

The overall research framework is shown in Fig. 1 at the end of this paragraph , which clearly describes the designing database, applying different index techniques and analysis the result. On the basis of the objectives and literature review, the indexing model is designed and developed. This research is based on multiple columns key index. MSSQL server in targeted hardware and other resource infrastructure is used. Designed database has under consideration of research purpose. The same piece of data has not stored in more than one place. This is because duplicate data not only waste storage spaces but also easily lead to inconsistencies. The limited tables are created and have full relationship between them with the norms. Some relation entity in the database with predefined constrained is created. Primary key and foreign key is maintained in each table definitions. The test performance is observed under millions of numbers of records. Data generation is done particular on manual basis and iteration process. The experiment was also taken on the standard database of the source from Microsoft site. It supports the use of the multi column indexing for more accurate way. The impact of multiple columns indexing in the query optimization process are significantly studied and gathered the analysis.

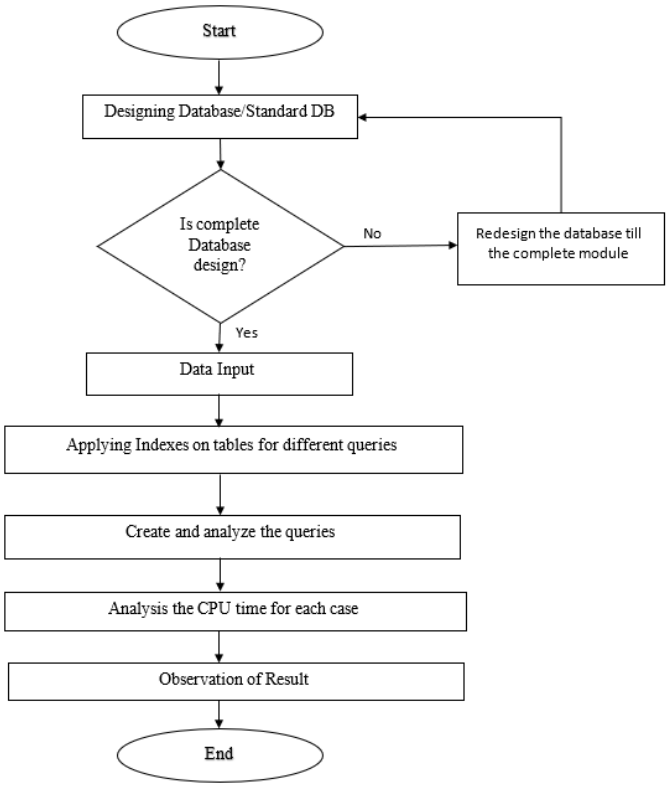

Fig. 1. Research Framework

IV. DATABASE INDEX STRUCTURE MODEL

Fig. 2 Index Structure Overview shows the overview of the designed index model. The figure explains how the research uses the indexing or how the index flow is maintained for the designed database or on standard database. The flow diagram gives the general idea on the index uses on the relational database query.

- Create the clustered index if the primary key column has no index.
- Create the multiple column index as per requirement.
- Study the query execution time on query if the query contains the where clause.
- Apply the multiple column index on where clause columns for the best performance.
- Create index on filtered columns if the query returns a lot of data.
- Apply the index on aggregate columns.
- Apply the index on ordered columns.

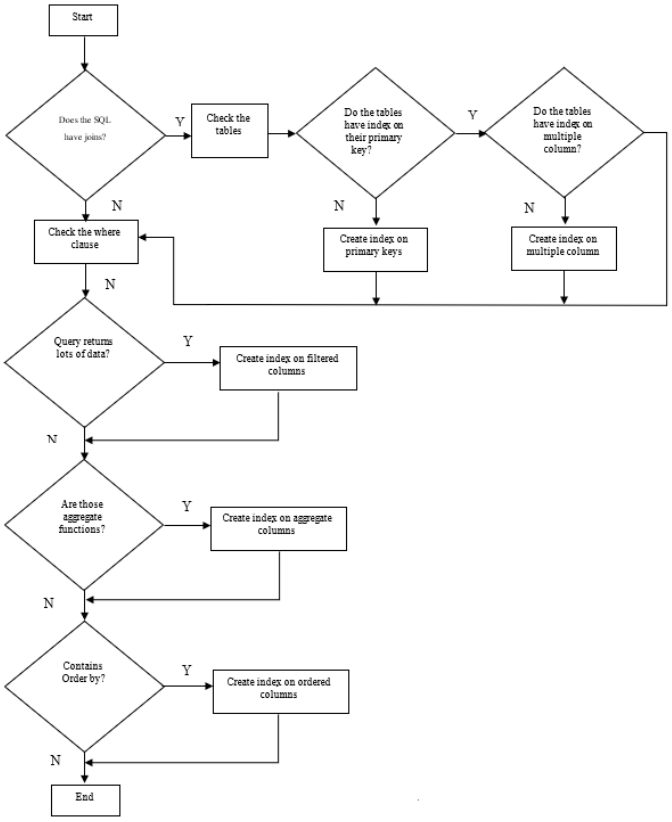

Fig. 2. Index Structure Overview

V. EXPERIMENT AND ANALYSIS

The research to the query optimization using multiple column index in SQL Server. Some queries on the database tables have been created. Query execution time with and without using the multiple column index are observed from experiment. The results are captured on each case. Here are the queries used in each observations: -

1. Index tuning and used for observation 1

```
CREATE NONCLUSTERED INDEX [Employee_index3] ON
[dbo].[Employee](MaritalStatus, FirstName, LastName, OnJob)
```

2. Index tuning and used for observation 2

```
CREATE NONCLUSTERED INDEX [EmployeeName2_index]
ON [dbo].[Employee](LastName, FirstName)
```

3. Index tuning and used for observation 3

```
CREATE NONCLUSTERED INDEX [Employee_index3] ON
[dbo].[EmployeeFrag1] (FirstName,LastName,Address)
```

4. Index tuning and used for observation 4

```
CREATE NONCLUSTERED INDEX [Employee_index3] ON
[dbo].[EmployeeFrag1] (MaritalStatus, OnJOb)
```

5. Index tuning and used for observation 5

```
CREATE NONCLUSTERED INDEX [Employee_index3] ON
[dbo].[EmployeeFrag1] (FirstName, LastName)
```

Similarly, algebraic expression for various observations are given below: -

1. Algebraic Expression of Observation 1

π E.EmployeeID, E.FirstName, E.LastName, E.OnJOb, SUM(TS.GrossSalary), SUM(TS.Tax)
γ E.EmployeeID, E.FirstName, E.LastName, E.OnJOb, SUM(TS.GrossSalary), SUM(TS.Tax)
σ E.FirstName='Chandan' **∧** E.LastName='Bhandari' **∧** E.OnJob=0
(**ρ**TS(TblSalary) **⋈** **ρ**E(EmployeeFrag1))
TS.EmployeeGUID=E. EmployeeGUID

2. Algebraic Expression of Observation 2

π E.Employeeid, E.FirstName, E.LastName, A.StartDate, SUM(TS.NetSalary), SUM(TS.Tax), MAX(TS.Salary)
γ E.Employeeid, E.FirstName, E.LastName, A.StartDate, SUM(TS.NetSalary), SUM(TS.Tax), MAX(TS.Salary)
σ FirstName LIKE 'C%' ∧ LastName='Bhandari'
(ρE(EmployeeFrag1) ⋈ ρTS(TblSalary) ⋈ ρA(Assignment) ⋈ ρO(Order)
E.EmployeeGUID = TS.EmployeeGUID TS.AssignmentGUID = A.AssignmentGUID A.StaffingOrderGUID = O.StaffingOrderGUID
⋈ ρO(Order) ⋈ ρCR(CustomerSalaryRange) ⋈ ρC(Customer))
O.CustomerGUID = CR.CustomerGUID
CR.CustomerGUID = C.CustomerGUID
O.ConfigSkillCodeID = CR.ConfigSkillCodeID
O.DesignationID = CR.DesignationID

3. Algebraic Expression of Observation 3

π E.Employeeid, E.FirstName, E.LastName, A.StartDate, C.CustomerID, C.CustomerName
γ E.Employeeid, E.FirstName, E.LastName, A.StartDate, C.CustomerID, C.CustomerName
σ E.FirstName = 'Chandan' ∧ E.LastName='Bhandari' ∧ E.Address='Kathmandu
(ρE(EmployeeFrag1) ⋈ ρTS(TblSalary) ⋈ ρA(Assignment) ⋈ ρO(Order)
E.EmployeeGUID = TS.EmployeeGUID TS.AssignmentGUID = A.AssignmentGUID A.StaffingOrderGUID = O.StaffingOrderGUID
⋈ ρO(Order) ⋈ ρCR(CustomerSalaryRange) ⋈ ρC(Customer))
O.CustomerGUID = CR.CustomerGUID CR.CustomerGUID = C.CustomerGUID
O.ConfigSkillCodeID = CR. ConfigSkillCodeID
O.DesignationID = CR.DesignationID

4. Algebraic expression of Observation 4 and 5

π E.Employeeid, E.FirstName, E.LastName, A.StartDate, C.CustomerID, C.CustomerName
γ E.Employeeid, E.FirstName, E.LastName, A.StartDate, C.CustomerID, C.CustomerName
σ E.MaritalStatus = 1 ∧ E.OnJob=0 ∧ E.FirstName='Chandan'
(ρE(EmployeeFrag1) ⋈ ρTS(TblSalary) ⋈ ρA(Assignment) ⋈ ρO(Order)
E.EmployeeGUID = TS.EmployeeGUID TS. AssignmentGUID = A.AssignmentGUID A.StaffingOrderGUID = O.StaffingOrderGUID
⋈ ρO(Order) ⋈ ρCR(CustomerSalaryRange) ⋈ ρC(Customer))
O.CustomerGUID = CR.CustomerGUID CR.CustomerGUID = C.CustomerGUID
O.ConfigSkillCodeID=CR. ConfigSkillCodeID
O.DesignationID=CR.DesignationID

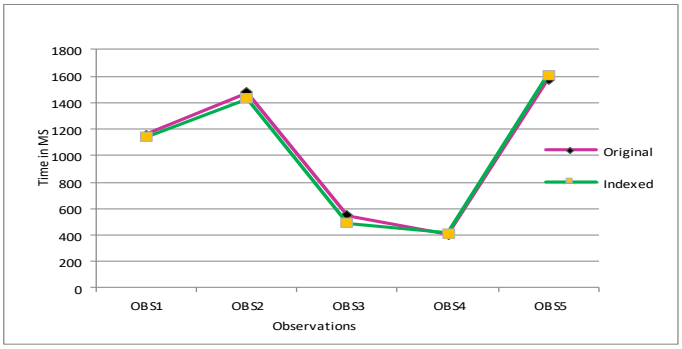

Fig. 3. Experiment result from Observation 1 to Observation5

VI. CONCLUSION

In this research, the multiple columns indexing to the relational database query has been dealt. The possible queries format has been taken and applied the various indexes to them for the experiment. The best suitable index has been in terms of time complexity has been observed. In most of the cases the single column clustered indexes have not fully used and found they are used only for the index scan and to another hand the multiple column index has been used by query optimizer and get seek and so it gives the improvement in performance. The result of the time complexity on query execution between single column clustered index and multiple columns index has been represented in graphs. On Observation1 the multiple column index on columns with equal operator shows improvement in query execution time. On Observation2 Indexed column is not getting fully seek when it has the like operator. On Observation3 It is found the columns in where clause is kept in same order in left base in index creation then the result get improved. On Observation4 Using bit columns in index gives worse performance. On Observation5 Even if the where clause have bit columns they Should not be included in index. Observation6 It is needed to consider the index column sort order.

VII. LIMITATION AND FUTURE WORK

An update to any attribute of a multiple column index causes the index to be modified. One of the limitations in this research is the columns chosen should not be those that are updated frequently. Indexes that are nearly as wide as the table. Also, the multiple column indexes where only a minor key is used in the where clause is another limitation.

The database query optimization is the massive field and it could be said the optimization on query can be done on various aspects. The optimization on query by index tuning could be considered on page split and fragmentation. The fragmentation is simple just as linked list where each page gets linked based on key column. The improved result could be achieved if the fragmentation is avoided to the page split. The page split should be considered to the linked list on ordered basis.

ACKNOWLEDGMENT

This research is supported by the Nepal College of Information Technology (NCIT), Department of Computer Science and Engineering, Pokhara University, Nepal.

REFERENCES

[1] L. Bajaj, “A Survey on Query Performance Optimization by Index Recommendation", International Journal of Computer Applications (0975 – 8887), Volume 113 – No. 19, March 2015

[2] Sirirut Vanichayobon, Le Gruenwald, “Indexing Techniques for Data Warehouses’ Queries”, Addison-Wisley Publishing Company, March 2008

[3] Tejy Johnson, Dr.S.K.Srivatsa, “A Study on Optimization Techniques and Query Execution Operators That Enhances Query Performance”, International Journal of Advanced Research in Computer Science, Volume 3, No. 3, May-June 2012

[4] Goetz Graefe, Harumi Kuno, “Self-selecting, self-tuning, incrementally optimized indexes”, Proceedings of the 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 2010

[5] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, Qingqing Zhou, “SQL Server Column Store Indexes”, Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, Pages 1177-1184, Athens, Greece, June 2011

[6] Chunxia Qi, “ON index-based query in SQL Server database”, 2016 35th Chinese Control Conference (CCC), August 2016

[7] Abraham Silberschatz, Henry F. Korth and S. Sudarshan, “DATABASE SYSTEM CONCEPTS”,6th Ed., McGraw-Hill (2011)

[8] Ms. P. R. Bhuyar, Dr.A.D.Gawande, Prof. A.B.Deshmukh , “Horizontal Fragmentation Technique in Distributed Database”, May 2012

[9] Don Jones, “The Definitive Guide To tm SQL Server Performance Optimization”, July 2002

[10] Anas Amro, Mohammed Abutaha, “Case Study: Comparison between Traditional Indexing and Columnstore Indexes in SQL Server 2012”, January 2013

[11] Michael Cain, Kent Milligan, “IBM DB2 for i indexing methods and strategies”, July 2011

[12] Markus Winand, “SQL Performance Explained”, August 2014

[13] https://en.wikipedia.org/wiki/Microsoft_SQL_Server

[14] https://en.wikipedia.org/wiki/Database_index

[15] https://en.wikipedia.org/wiki/Query_plan

[16] https://docs.microsoft.com/en-us/sql/tools/sql-server-profiler/sql-server-profiler?view=sql-server-2017

[17] https://www.essentialsql.com/what-is-a-data-dictionary/

[18] https://www.red-gate.com/simple-talk/sql/sql-training/the-sql-server-query-optimizer/

[19] https://docs.oracle.com/

[20] https://web.cs.ucdavis.edu/~green/courses/ecs165a-w11/8-query.pdf

[21] https://en.wikipedia.org/wiki/Distributed_database

[22] http://www.exploredatabase.com/2017/05/list-characteristics-of-distributed-dbms.html