IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

Impact of Clustered, Non-Clustered, and Composite Indexes on Query Performance in Microsoft SQL Server

Vishal Chhaganlal Kumawat
MCA Student, Vidyavardhini's College of Engineering and Technology,
University of Mumbai CDOE, Mumbai, India

Abstract: *Efficient query execution is essential for maintaining performance in database-driven applications as data volume and workload complexity increase. Indexing is a fundamental optimization technique used to reduce data-access cost and improve retrieval efficiency in relational database systems. This paper presents a comparative experimental analysis of clustered, non-clustered, and composite indexing strategies in Microsoft SQL Server and evaluates their effect on point lookup, range retrieval, sorting, and aggregation queries. Performance was assessed using execution time, CPU time, logical reads, and execution-plan behavior. The results show that appropriate indexing substantially improves query efficiency, with execution time reductions of up to approximately 88% and logical-read reductions exceeding 90% for selected workloads. The findings indicate that non-clustered indexes are most effective for selective lookup queries, clustered indexes provide strong benefits for range-based and ordered retrieval, and composite indexes improve performance for multi-column access patterns.*

Keywords: Database Indexing, Query Performance, Microsoft SQL Server, Clustered Index, Non-Clustered Index, Composite Index, Query Optimization, Execution Plan Analysis

I. INTRODUCTION

The growth of transactional and analytical data has made query efficiency a critical requirement in modern database applications. In Microsoft SQL Server, index design strongly influences data-access cost, retrieval efficiency, and execution-plan selection by the cost-based optimizer [1]–[3]. However, indexing is not universally beneficial, since additional indexes also increase storage consumption and maintenance overhead for insert, update, and delete operations [1], [4].

Prior research has shown that index selection is a workload-dependent optimization problem. The effectiveness of an index depends on query structure, predicate selectivity, key-column order, and optimizer behavior [4]–[8]. Although existing studies provide strong theoretical and system-level insight, there remains a practical need for controlled empirical analysis of how standard rowstore index types affect common operational queries within a specific DBMS environment.

Motivated by this need, the present study investigates the impact of clustered, non-clustered, and composite indexes on query performance in Microsoft SQL Server. The paper presents a controlled SQL Server-specific experimental comparison of these indexing strategies under representative workload patterns, including point lookup, range retrieval, sorting, and aggregation. Performance is evaluated using execution time, CPU time, logical reads, and execution-plan behavior in order to provide practical guidance for workload-aware index design in SQL Server-based applications.

II. LITERATURE REVIEW

Database indexing is a central component of physical database design because access-path selection directly affects query latency, CPU usage, and I/O behavior. In Microsoft SQL Server, rowstore indexes are implemented as B+ trees,

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 5, May 2026 Impact Factor: 8.2

and the Query Optimizer evaluates alternative access paths using a cost-based model informed by schema metadata, index definitions, and statistics [1]–[3]. Current SQL Server guidance also emphasizes that both missing indexes and over-indexing can degrade performance, indicating that effective index design is a trade-off between read acceleration, storage overhead, and update cost [1], [3].

Prior research has shown that index selection is a workload-dependent optimization problem rather than a simple per-query decision. Studies on automated tuning for Microsoft SQL Server demonstrated that the value of an index depends on workload context, optimizer behavior, and interaction among candidate indexes [6], [7]. Later experimental work further confirmed that index effectiveness varies with workload characteristics, multi-column support, and the difficulty of estimating benefit without full deployment [4].

More recent studies have shown that estimated optimizer cost alone may not always reflect real execution behavior and that index recommendations must be validated under actual workloads [5], [8], [9]. These findings are especially relevant for composite indexes, whose usefulness depends strongly on key-column order and predicate alignment [1], [4].

Despite these advances, there remains a practical need for controlled empirical studies that examine how standard rowstore index types affect common operational queries within a specific DBMS environment. The present study addresses this need by providing a focused experimental comparison of clustered, non-clustered, and composite indexes in Microsoft SQL Server using execution time, CPU time, logical reads, and execution-plan behavior as evaluation metrics.

III. TYPES OF INDEXING STRATEGIES

In Microsoft SQL Server, rowstore indexes are implemented as B+ tree structures that reduce the cost of locating qualifying rows. However, index effectiveness depends on workload characteristics, including predicate selectivity, data distribution, sort requirements, and optimizer cost estimation [1], [2].

A. Clustered Indexes

A clustered index stores table rows in key order and is therefore well suited to range predicates and ordered retrieval. Because the leaf level contains the actual data pages, clustered indexes are especially effective for queries involving sequential access or ORDER BY operations on the indexed key [3]. Their main limitation is added maintenance overhead for inserts and updates on key values, particularly in write-intensive workloads[1], [3].

B. Non-Clustered Indexes

A non-clustered index stores key values separately from the underlying data rows and uses row locators to reference the base table. These indexes are especially effective for highly selective predicates such as point lookups, where the optimizer can use index seek operations to locate a small number of qualifying rows efficiently [1], [3]. Their benefit may be reduced when queries require many additional columns not covered by the index.

C. Composite Indexes

A composite index contains more than one key column within a single index definition. Its effectiveness depends strongly on key-column order, since SQL Server typically benefits most when predicates, grouping, or sorting begin with the leading key sequence [1]. As a result, composite indexes should be designed according to actual workload patterns rather than by simply combining commonly used columns.

D. Selectivity, Cost, and Maintenance Trade-offs

The effectiveness of any indexing strategy is shaped by predicate selectivity, update frequency, and overall workload composition. High-selectivity columns generally offer stronger filtering benefits, whereas low-selectivity columns may lead the optimizer to prefer scanning when scan cost is estimated to be lower than indexed access [1], [2]. At the same time, every additional index increases storage consumption and write overhead because SQL Server must maintain indexes whenever table data is modified [3]. For this reason, the present study focuses on clustered, non-clustered, and composite indexes as the most practically relevant rowstore index designs for point lookup, range retrieval, sorting, and aggregation workloads in Microsoft SQL Server.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

IV. EXPERIMENTAL SETUP

To evaluate the effect of indexing strategies on query performance, controlled experiments were conducted in a Microsoft SQL Server environment. The objective of the experimental design was to compare baseline query behavior with performance under different index configurations while keeping the workload, schema, and execution environment constant. The study focused on four representative query patterns: point lookup, range retrieval, ordered retrieval, and aggregation.

A. System Environment

All experiments were performed using Microsoft SQL Server 2019 Developer Edition on Windows 11 Pro 64-bit, with query execution and analysis conducted through SQL Server Management Studio 19. The experiments were executed on a single system configured with an Intel Core i5 processor, 8 GB RAM, and 512 GB SSD. The database compatibility level was fixed at 150 throughout the experiment. No schema modifications other than the index configurations under test were introduced during performance evaluation. This ensured that the observed performance differences were attributable to the indexing strategy rather than to changes in hardware, software configuration, or execution environment.

B. Dataset and Schema

The experimental workload was executed on an Orders table representing a transactional sales dataset. The table contained 122456 rows and included the attributes required for the selected queries, namely OrderId, CustomerId, OrderDate, and TotalAmount. The dataset used in this study was synthetic. The synthetic dataset was generated to simulate realistic transactional behavior with repeated customer activity, date-based order distribution, and varying order amounts. In particular, the dataset was designed to reflect repeated customer purchases over time, with multiple orders per customer and variation in transaction values across the evaluation period.Customer identifiers were distributed across repeated purchase records, order dates were spread over the evaluation period, and transaction amounts varied to reflect heterogeneous sales activity.

To avoid ambiguity in the clustered-index experiment, the primary key on OrderId was defined as NONCLUSTERED. This design choice was necessary because the clustered index under evaluation was created on OrderDate, and SQL Server permits only one clustered index per table. In the baseline state, no additional secondary indexes were created beyond the primary key definition. This configuration served as the reference state for comparing the impact of the tested indexing strategies.

C. Query Workload

The workload was designed to represent common query patterns in relational applications:

1) Point lookup query

```
SELECT * FROM Orders
WHERE CustomerId = 500;
```

2) Range retrieval query

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN '2024-01-01' AND '2024-12-31';
```

3) Ordered retrieval query

```
SELECT * FROM Orders
ORDER BY OrderDate DESC;
```

4) Aggregation query

```
SELECT CustomerId, SUM(TotalAmount)
FROM Orders
GROUP BY CustomerId;
```

These queries were selected because they exercise distinct access behaviors, including selective filtering, sequential range access, sorting, and grouping.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 5, May 2026 Impact Factor: 8.2

D. Index Configurations and Isolation Procedure

Four experimental states were evaluated:

Baseline state: no additional secondary indexes beyond the declared primary key constraint;

Clustered-index state: clustered index on OrderDate;

Non-clustered-index state: non-clustered index on CustomerId;

Composite-index state: non-clustered index on (CustomerId, OrderDate).

Each experimental state was evaluated independently. Before moving from one indexing phase to the next, indexes from the previous phase were dropped so that performance changes could be attributed to the active index configuration rather than to cumulative indexing effects. This isolation procedure is necessary for a valid comparison of indexing strategies.

E. Measurement Procedure and Evaluation Metrics

For each experimental state, the same query text was executed under the same session settings. The SQL Server options SET STATISTICS IO ON and SET STATISTICS TIME ON were enabled to collect I/O and timing information, and actual execution plans were captured for operator-level analysis. Prior to measurement, statistics were explicitly refreshed using UPDATE STATISTICS. Each query was executed 5 times, and the reported values correspond to the median obtained across repeated runs after one warm-up run. This procedure was adopted to reduce the effect of transient fluctuations and improve the reliability of the reported results.Performance was evaluated using elapsed execution time, CPU time, logical reads, and execution-plan behavior, where elapsed time measured response latency, CPU time reflected processing cost, logical reads represented data-access effort, and execution plans were used to interpret optimizer behavior.

V. IMPLEMENTATION

This section describes the step-by-step implementation of indexing strategies and their impact on query execution behavior in Microsoft SQL Server. The implementation was conducted in a controlled manner to ensure that performance differences could be directly attributed to the applied indexing configurations.

A. Baseline Execution (Primary Key Only)

The implementation began by executing all selected queries on the Orders table in a baseline state in which no additional secondary indexes were defined beyond the primary key constraint. Under this configuration, the optimizer relied primarily on scan-oriented access paths for the evaluated workloads, resulting in relatively high logical reads and longer execution times. This baseline state served as the reference point for evaluating subsequent indexing strategies.

B. Clustered Index Implementation

A clustered index was created on the OrderDate column to evaluate its effect on range-based and sorting queries:

```
CREATE CLUSTERED INDEX idx_orderdate
ON Orders(OrderDate);
```

Since SQL Server permits only one clustered index per table, the primary key on OrderId was defined as NONCLUSTERED. This configuration improved range retrieval and ORDER BY performance by aligning physical row order with the access pattern.

C. Non-Clustered Index Implementation

A non-clustered index was created on the CustomerId column to optimize point lookup queries:

```
CREATE NONCLUSTERED INDEX idx_customerid
ON Orders(CustomerId);
```

This enabled the optimizer to use index seek operations for selective filtering and substantially reduced execution time and logical reads for lookup queries.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

D. Composite Index Implementation

To support multi-column access patterns, a composite non-clustered index was created on (CustomerId, OrderDate):

```
CREATE NONCLUSTERED INDEX idx_customer_orderdate
ON Orders(CustomerId, OrderDate);
```

This index was designed to support multi-column access patterns. Its effectiveness depended on how well the leading key order matched the evaluated query requirements. In the experiment, it improved execution behavior for the tested multi-column and aggregation-oriented workload.

E. Execution Plan Evaluation

Execution-plan analysis was used to examine how the SQL Server Query Optimizer responded to each indexing strategy. In the baseline state, scan-oriented operators were more common, resulting in higher logical reads and longer execution times. After indexing was introduced, the optimizer more frequently selected efficient access methods such as Index Seek, particularly when the index structure aligned with the query pattern. However, index availability alone did not guarantee index usage, since SQL Server continued to prefer scan-based plans when indexed access was estimated to be less efficient.

F. Implementation Phases Summary

The implementation was carried out in five stages: baseline execution with only the primary key defined, evaluation of a clustered index on OrderDate, evaluation of a non-clustered index on CustomerId, evaluation of a composite non-clustered index on (CustomerId, OrderDate), and execution-plan analysis across all indexing conditions. To improve reproducibility, all query tests were executed under the same session settings and previously created indexes were dropped before each new evaluation phase.

VI. RESULTS AND ANALYSIS

The experimental results show that indexing had a substantial effect on query performance across all tested workload patterns. In the baseline condition, SQL Server relied primarily on scan-based access paths, resulting in higher logical reads, greater CPU cost, and longer elapsed execution time. After the application of clustered, non-clustered, and composite indexes, the optimizer selected more efficient access strategies for the corresponding query types.

A. Performance Comparison by Query Type

Table I summarizes the measured execution time, logical reads, and CPU time under the evaluated index configurations.

Table I - Query Performance Under Different Index Configurations

Query Type	Index Type	Execution Time (ms)	Logical Reads	CPU Time (ms)
SELECT	Baseline (Primary Key Only)	1180	48920	950
SELECT	Non-Clustered	145	1240	120
RANGE	Baseline (Primary Key Only)	1520	52300	1100
RANGE	Clustered	210	2100	180
ORDER BY	Baseline (Primary Key Only)	1390	50100	1020
ORDER BY	Clustered	260	2500	210
GROUP BY	Baseline (Primary Key Only)	1310	49800	980
GROUP BY	Composite	310	3200	260

For the point lookup query, the non-clustered index on CustomerId reduced execution time from 1180 ms to 145 ms and logical reads from 48,920 to 1,240. For the range query, the clustered index on OrderDate reduced execution time from 1520 ms to 210 ms and logical reads from 52,300 to 2,100. A similar improvement was observed for the ORDER BY query, where execution time decreased from 1390 ms to 260 ms and logical reads fell from 50,100 to 2,500 after

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

clustering on OrderDate. For the aggregation query, the composite index produced the best measured result, reducing execution time from 1310 ms to 310 ms and logical reads from 49,800 to 3,200.

B. Comparative Query-Level View

Table II presents a consolidated comparison of the best observed performance across query types.

Table II - Comparative Execution Time Across Query Types

Query Type	Baseline Execution Time (ms)	Indexed Execution Time (ms)	Baseline Logical Reads	Indexed Logical Reads	Best Performing Index
Point Lookup	1180	145	48920	1240	Non-Clustered
Range Retrieval	1520	210	52300	2100	Clustered
Ordered Retrieval	1390	260	50100	2500	Clustered
Aggregation	1310	310	49800	3200	Composite

Table II shows that the relative benefit of indexing depended strongly on query structure. The non-clustered index was most effective for selective point lookup, the clustered index was most effective for range retrieval and ordered access, and the composite index produced the strongest measured result for the evaluated aggregation-oriented workload.

C. Execution Plan Analysis

Execution-plan inspection provided additional evidence for the observed performance differences. In the baseline condition, SQL Server predominantly used scan-oriented operators, which required access to a larger number of pages and resulted in higher I/O cost. After indexing was introduced, the optimizer more frequently selected efficient access operators such as Index Seek, which explains the reduction in execution time and logical reads. However, index availability alone did not guarantee index usage, since SQL Server continued to prefer scan-based plans when indexed access was estimated to be less efficient.

D. Key Findings

The results lead to four main findings. First, appropriate indexing substantially reduced execution time, CPU time, and logical reads across all tested workloads. Second, non-clustered indexing was most effective for selective point lookup conditions. Third, clustered indexing produced the strongest improvement for range retrieval and ordered-access workloads. Fourth, the composite index provided the best measured result for the evaluated multi-column and aggregation-oriented workload, although this benefit remained dependent on query structure and key-column alignment.

VII. DISCUSSION

The findings of this study show that indexing directly influences query execution behavior in Microsoft SQL Server. Across all tested workload types, indexed configurations outperformed the baseline state in execution time, CPU time, and logical reads. However, the results also confirm that index effectiveness is workload-dependent rather than universal.

Different query types benefited from different index structures. The non-clustered index on CustomerId was most effective for point lookup operations, while the clustered index on OrderDate provided the strongest improvement for range retrieval and sorting workloads. The composite index produced the best measured result for the evaluated aggregation-oriented workload, indicating that multi-column index design can improve performance when key-column order aligns with query requirements.

At the same time, indexing involves trade-offs. Additional indexes can improve read performance but also increase storage usage and maintenance overhead for insert, update, and delete operations. This means that effective index design should be based on actual workload patterns rather than indiscriminate index creation.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 5, May 2026 Impact Factor: 8.2

The study also highlights the value of execution-plan analysis. The transition from scan-oriented plans in the baseline state to more efficient seek-oriented plans in indexed states shows that performance improvement is reflected not only in runtime metrics but also in optimizer behavior.

Despite these useful findings, the study has several limitations. The analysis was confined to Microsoft SQL Server, the workload consisted of a limited set of representative queries, and write-oriented index-maintenance overhead was not measured systematically. In addition, because the dataset used in the experiment was synthetic, the observed performance behavior may not fully capture all characteristics of real-world enterprise workloads.

Overall, the results support a workload-driven approach to index design in SQL Server-based applications and highlight the need to balance retrieval performance, storage overhead, maintenance cost, and long-term scalability.

VIII. CONCLUSION

This study examined the impact of clustered, non-clustered, and composite indexing strategies on query performance in Microsoft SQL Server through a controlled experimental comparison of point lookup, range retrieval, sorting, and aggregation workloads. Performance was evaluated using execution time, CPU time, logical reads, and execution-plan behavior.

The findings show that appropriate indexing can substantially improve query efficiency when the index structure is aligned with workload requirements. Non-clustered indexing was most effective for selective lookup queries, clustered indexing provided strong benefits for range-based and ordered retrieval, and composite indexing improved performance for the evaluated multi-column access pattern. At the same time, the results demonstrate that index effectiveness is conditional and depends on query structure, predicate selectivity, and optimizer cost estimation.

Overall, the study reinforces the importance of workload-aware index design in SQL Server-based applications and provides practical guidance for improving query performance while balancing storage and maintenance overhead. Future work may extend this analysis to more complex workloads, write-oriented performance evaluation, advanced indexing techniques, and comparative studies across multiple DBMS platforms.

REFERENCES

[1] Microsoft, “Index Architecture and Design Guide - SQL Server,” Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver17. [Accessed: Apr. 4, 2026].

[2] Microsoft, “Query Processing Architecture Guide - SQL Server,” Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver17. [Accessed: Apr. 4, 2026].

[3] Microsoft, “Clustered and Nonclustered Indexes - SQL Server,” Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver17. [Accessed: Apr. 4, 2026].

[4] J. Kossmann, S. Halfpap, M. Jankrift, and R. Schlosser, “Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms,” Proc. VLDB Endowment, vol. 13, no. 11, pp. 2382-2395, 2020.

[5] S. Chakkappen, S. Kunjibettu, D. McGreer, M. J. Kishi, H. Su, M. Ziauddin, M. Zait, Z. Li, and Y. Zhang, “Automatic Indexing in Oracle,” Proc. VLDB Endowment, vol. 18, no. 12, pp. 4924-4937, 2025.

[6] S. Chaudhuri and V. R. Narasayya, “An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server,” in Proc. 23rd Int. Conf. Very Large Data Bases (VLDB), 1997, pp. 146-155.

[7] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, “Database Tuning Advisor for Microsoft SQL Server 2005,” in Proc. 30th Int. Conf. Very Large Data Bases (VLDB), Toronto, ON, Canada, 2004.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

[8] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. Narasayya, "AI Meets AI: Leveraging Query Executions to Improve Index Recommendations," in Proc. 2019 Int. Conf. Manage. Data (SIGMOD), Amsterdam, The Netherlands, 2019, pp. 1241-1258.
[9] Microsoft, "Automatic tuning - SQL Server," Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning?view=sql-server-ver17. [Accessed: Apr. 4, 2026].