Jurnal Pendidikan dan Teknologi Indonesia (JPTI)
Vol. 5, No. 7, Juli 2025, Hal. 1845-1856
DOI: https://doi.org/10.52436/1.jpti.862
p-ISSN: 2775-4227
e-ISSN: 2775-4219

Optimasi Kueri pada Database Oracle Melalui Indeks dan Partisi Tabel untuk Data Besar

Bambang Sugiarto[1], Argan Imam Bagusputra[*2], Samidi[3]

[1,2,3]Master of Computer, Faculty of Information Technology, Universitas Budi Luhur, Indonesia
Email: [1]2311601682@student.budiluhur.ac.id, [2]2311601732@student.budiluhur.ac.id,
[3]samidi@budiluhur.ac.id

Abstrak

Mengoptimalkan kinerja database sangat krusial di era dominasi data. Pertumbuhan data eksponensial, khususnya pada data Kredit Usaha Rakyat (KUR) sejumlah 227.587.131 baris di database Oracle yang digunakan dalam penelitian ini, menjadi tantangan utama. Eksekusi kueri SQL yang lambat menghambat efisiensi operasional. Penelitian ini menerapkan strategi optimasi kinerja database Oracle melalui teknik pengindeksan (indeks tunggal dan komposit) dan partisi tabel berdasarkan rentang (kolom tahun). Kedua teknik ini bertujuan mempercepat pengambilan data dan meningkatkan efisiensi akses pada tabel besar. Tujuan penelitian adalah mengoptimalkan eksekusi kueri SQL pada data KUR yang besar tersebut. Evaluasi dilakukan dengan membandingkan waktu respons kueri pada empat skenario tabel (tanpa indeks, indeks tunggal, indeks komposit, serta indeks komposit dengan partisi). Hasil evaluasi menunjukkan bahwa penerapan indeks (tunggal dan komposit) serta partisi tabel secara umum meningkatkan kinerja kueri seleksi dan join secara signifikan dibandingkan tanpa optimasi, dengan waktu tercepat dicapai pada tabel berpartisi dengan penyebutan partisi eksplisit (0,082 detik untuk seleksi sederhana). Namun, untuk kueri agregasi, tabel tanpa indeks memberikan waktu respons lebih cepat (58.100 detik) dibandingkan tabel dengan indeks tunggal (71.700 detik) ataupun kombinasi indeks komposit dan partisi.

Kata kunci: *Database, Indexing, Optimization, Oracle Database, Performance, Query.*

Query Optimization on Oracle Database Via Index and Partition Tables for Large Data

Abstract

Optimizing database performance is crucial in the data-dominated era. Exponential data growth, particularly with the 227,587,131 rows of Kredit Usaha Rakyat (KUR) data in the Oracle database used in this study, presents a primary challenge. Slow SQL query execution hinders operational efficiency. This research applies Oracle database performance optimization strategies through indexing techniques (single and composite indexes) and table partitioning based on range (year column). Both techniques aim to accelerate data retrieval and improve access efficiency for large tables. The research objective is to optimize SQL query execution on the large KUR data. Evaluation was conducted by comparing query response times across four table scenarios (without index, with a single index, with a composite index, and with a composite index and partitioning). Evaluation results show that implementing indexes (single and composite) and table partitioning generally significantly improved selection and join query performance compared to no optimization, with the fastest time achieved on a partitioned table with explicit partition specification (0.082 seconds for simple selection). However, for aggregation queries, the table without an index provided faster response times (58,100 seconds) compared to the table with a single index (71,700 seconds) or a combination of composite index and partitioning.

Keywords: *Database, Indexing, Optimization, Oracle Database, Performance, Query.*

1. INTRODUCTION

The escalating volume of data in modern information systems presents a significant performance challenge for organizations [1]. Relational Database Management Systems (RDBMS) like Oracle are widely employed to manage this data, using Structured Query Language (SQL) for data access and manipulation [2]. However, as datasets grow, particularly to the scale of Kredit Usaha Rakyat (KUR) data involving hundreds of millions of

p-ISSN: 2775-4227
e-ISSN: 2775-4219

rows as detailed in this study (analyzing 227,587,131 rows from a KUR table), the efficiency of SQL query execution can drastically decrease. This slowdown in data retrieval directly impacts operational tasks, hinders timely data processing, and can disrupt organizational productivity [3]. Therefore, optimizing query performance in such large-scale Oracle database environments is of critical importance [4].

To address these performance bottlenecks, particularly for data retrieval, database optimization techniques such as indexing and table partitioning are commonly implemented [5], [6], [7]. Indexing, by creating auxiliary data structures based on table columns (often B-tree structures in Oracle), aims to accelerate data access and reduce the need for full table scans during query execution [4], [8], [9]. The design of these indexes, whether single-column or composite as tested in this research, plays a crucial role in their effectiveness [10], [11]. While indexing significantly benefits query performance, it is a factor in overall database design to consider its maintenance overhead, particularly for data modification operations [12], [13], though this study focuses on retrieval performance.

Table partitioning is another established technique for managing very large tables by dividing them into smaller, more manageable segments based on defined keys, such as range partitioning by year as explored in this study. This approach can enhance query performance by allowing the database optimizer to scan only relevant partitions when retrieving data, which is especially beneficial for queries on large tables [7], [14], [15], [16]. Studies indicate that partitioning can improve query response times significantly [14], [17], and its combination with appropriate indexing strategies can further optimize data access in large database systems [7].

The challenge of optimizing query performance in Oracle databases handling extensive data like the KUR dataset, which involves over 227 million rows in the sample table used in this research, requires a systematic approach. Therefore, the primary objective of this research is to experimentally evaluate and compare the impact of specific B-tree indexing strategies (single and composite) and range-based table partitioning on SQL query execution performance in an Oracle 19c database. This study aims to quantify the performance changes across different optimization scenarios, providing empirical insights for enhancing query efficiency in similar large-scale data environments.

2. RESEARCH METHOD

This research adopts an experimental methodology to evaluate and compare the performance of SQL query execution on an Oracle database, specifically focusing on the application of indexing and table partitioning as optimization techniques. An experimental approach is suitable as it facilitates a detailed quantitative analysis of the impact these methods have on query response times, particularly crucial when dealing with large-scale datasets such as the Kredit Usaha Rakyat (KUR) data utilized in this study. The overall research workflow is depicted in Figure 1.

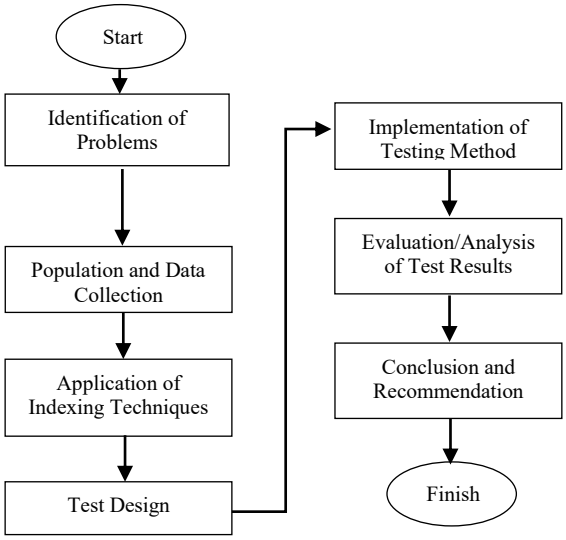

Figure 1. Framework

Figure 1 illustrates the structured framework guiding this research. It commences with problem identification related to slow query performance on large datasets. This is followed by population and data collection, focusing on the KUR dataset. Subsequently, a detailed test design is formulated, outlining the different database scenarios (with and without indexing/partitioning). The core of the experimental phase

p-ISSN: 2775-4227
e-ISSN: 2775-4219

involves the application of indexing techniques and table partitioning, followed by the implementation of testing methods to measure query execution times. Finally, the test results are evaluated and analyzed, leading to conclusions and recommendations regarding the effectiveness of the applied optimization strategies.

2.1. Identification of Problems

This research focuses on optimizing SQL query execution performance within an Oracle database environment tasked with managing large-scale data, specifically the Kredit Usaha Rakyat (KUR) dataset. The primary problem identified is the suboptimal or slow execution of SQL queries, which significantly impedes operational efficiency and the timeliness of data processing. Such performance degradation can negatively impact critical business decision-making and overall organizational productivity. To systematically address this, the research identifies query execution time as the key performance indicator. Consequently, this study aims to investigate and quantify the optimization achievable by implementing selected indexing and table partitioning techniques to accelerate queries that currently exhibit slow or inefficient performance.

2.2. Population and Data Collection

This research utilizes Kredit Usaha Rakyat (KUR) data stored in an Oracle database. This dataset is characterized by its complexity and vast volume of information, comprising 227,587,131 rows in the sample used, which inherently poses significant challenges in achieving efficient SQL query execution [1]. The data for this study was sourced from the Ministry of Finance's Kredit Usaha Rakyat (KUR) database. Specifically, one of the primary tables, the KUR detailed billing data table (TB_TAG_DTL), was copied from the production environment to a dedicated research database environment. The structure of this TB_TAG_DTL table, which includes key fields such as ID, BANK_CODE, YEAR, MONTH, NIK_RECIPIENT, and various financial indicators like OUTSTANDING and SUBSIDY, is detailed in Table 1.

Table 1. TB_TAG_DTL Table Structure

No.	Field Name	Type
1	ID	NUMBERS
2	BANK_CODE	VARCHAR2(4)
3	YEAR	VARCHAR2(4)
4	MONTH	VARCHAR2(2)
5	SCHEME	VARCHAR2(1)
6	SEK_NGR_OBJECTIVE	VARCHAR2(3)
7	NIK_RECIPIENT	VARCHAR2(16)
8	REC_NEW	VARCHAR2(40)
9	OUTSTANDING	NUMBERS
10	LONG_DAYS	NUMBERS
11	SUBSIDY	NUMBERS
12	NO_AKAD	VARCHAR2(45)
13	FILENAME	VARCHAR2(255)

The data sample selected for this research comprises a subset of the available KUR data, specifically covering transactions from the years 2018 to 2023. As mentioned, this sample consists of 227,587,131 rows extracted from the TB_TAG_DTL table. Table 2 provides a breakdown of the number of rows per year within this sample, illustrating the data distribution over the covered period. This extensive dataset was subsequently used to measure and compare query execution performance before and after the application of the proposed indexing and table partitioning techniques.

Table 2. Number of rows per year

No.	Year	Row
1	2018	81,353,636
2		19,894,688
3		18,994,388
4		32,385,801
5		38,964,346
6		35,994,272

p-ISSN: 2775-4227
e-ISSN: 2775-4219

Testing was conducted in a server environment specifically designed to handle large-scale workloads. The server specifications used are a virtual machine (VM) server with the Ubuntu 20.04.3 LTS operating system, an AMD EPYC 7543 32-Core Processor, 126 GB RAM, 1 TB HDD storage, and an Oracle 19c database. The instrument used for client-side measurements is a PC with the Windows 10 Pro Operating System, connected to the server via a local area network (LAN), and the tool for running SQL queries is DBeaver version 24.

2.3. Application of Indexing Techniques

Indexing is a primary technique employed to enhance data search performance within a database system. This method improves the efficiency of the data search process primarily by obviating the need for a full table scan, which can be resource-intensive, especially for large tables [4], [8]. Table indexes function analogously to a book's index, creating a link between logical data representations and their physical storage locations, thereby accelerating data lookup [2]. In Oracle databases, these indexes commonly utilize a B-tree structure, which stores "Keys" derived from one or more table or view columns to expedite data queries [9].

Indexes enable data searches to bypass row-by-row scanning, instead directing the search to a key within the index structure which then points to the data's physical location [12]. This mechanism, consisting of indexed column values and pointers (like ROWID in Oracle) to the actual data blocks, is crucial for avoiding time-consuming full table scans, particularly in tables with a substantial number of rows [7]. Figure 2 visually contrasts the data search process in a table without an index, which necessitates a full table scan, against the more direct and efficient path facilitated by searching via a table index.

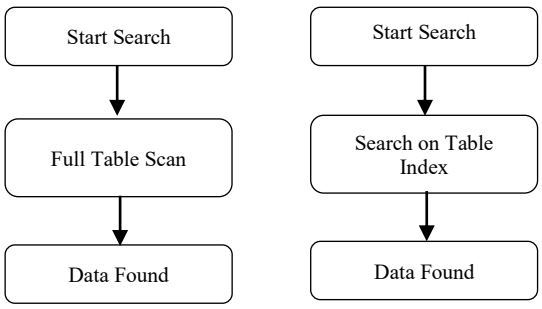

Figure 2. Search Flow Without Index and With Index

Beyond faster data location, indexing aims to reduce computational complexity and I/O load during data retrieval [4]. However, the mere presence of an index does not guarantee improved performance, ineffective query design can prevent the database engine from utilizing indexes efficiently, potentially worsening query performance. Therefore, designing effective indexes involves strategic creation on columns frequently used in query predicates, such as those involved in WHERE clauses, join conditions, or columns often used for sorting (ORDER BY) or grouping (GROUP BY) data [4].

2.4. Application of Table Partition Techniques

Table partitioning is an essential technique in physical database design, capable of significantly improving database performance, simplifying management, and enhancing data availability, especially for large tables [6], [7]. By employing table partitioning, large tables, and their associated indexes, can be divided into smaller, more manageable segments called partitions. This segmentation allows database objects to be managed and accessed with finer granularity [7]. Oracle Database, as used in this research, provides a variety of partitioning strategies, such as range, list, and hash partitioning, to cater to diverse business requirements [6]. A key advantage is that partitioning can often be implemented transparently to applications, minimizing the need for extensive application code changes.

Logically, a partitioned table is still treated as a single table by applications, however, physically, its data is stored across multiple distinct segments. This physical separation can contribute to increased I/O efficiency, for instance, by distributing partitions across different storage devices [1]. Partitioning is particularly beneficial for improving the performance of queries, including those involving multi-table joins, as operations can often be localized to relevant partitions or performed in parallel across partitions [6]. Figure 3 provides a conceptual illustration of how a main table can be divided into several smaller, independent partitions.

p-ISSN: 2775-4227
e-ISSN: 2775-4219

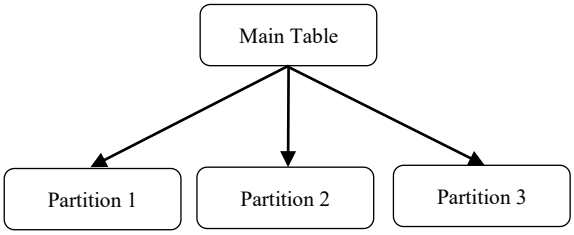

Figure 3. Table Partition Diagram

Each segment of a partitioned table is termed a 'partition,' possessing its own name and potentially unique storage characteristics. The distribution of data into these partitions is determined by a 'partition key,' which consists of one or more columns whose values dictate the target partition for each row [7]. Among the primary benefits of partitioning is improved data availability; if an issue affects one partition (e.g., data corruption or maintenance), other partitions can remain accessible, thus enhancing the overall resilience of the system [6]. This study specifically implements range partitioning based on the YEAR column, a common approach for time-series data like the KUR dataset.

2.5. Test Design

To evaluate the performance of SQL query execution on an Oracle database, this research established four different table scenarios. Each scenario utilized a table with the same base structure as the TB_TAG_DTL table, but with varying indexing and partitioning techniques applied. The four scenarios are as follows:

1. TB_1: Table without index
   This table served as the baseline, representing a non-optimized scenario, to measure query performance without any indexing.

2. TB_2: Table with a single index
   This scenario involved creating three separate single-column B-tree indexes on the BANK_CODE, YEAR, and MONTH columns. This setup was designed to test the impact of basic, individual indexes on query performance.

3. TB_3: Table with composite index
   For this scenario, a single composite B-tree index was created, combining the BANK_CODE, YEAR, and MONTH columns. This allowed for testing the performance implications of a multi-column index.

4. TB_4: Table with a composite index and table partition
   This scenario built upon TB_3 by retaining the same composite index and adding range partitioning based on the YEAR column (with partitions for each year from 2018 to 2023). This was designed to test the combined effect of composite indexing and table partitioning.

The complete Data Definition Language (DDL) statements for creating these four tables and their indexes are provided in Appendix A.

Each of these table configurations was then subjected to a series of identical SQL SELECT queries to measure execution time. The results were subsequently compared to determine the relative effectiveness of each optimization technique. Table 3 provides a summary of these test scenarios, outlining the specific conditions applied to each table.

Table 3. Test Scenario Description Table

Table	Condition	Partition
TB_1	No index	-
TB_2	*Single index*	-
TB_3	*Composite index*	-
TB_4	*Composite index* + Partition	YEAR

p-ISSN: 2775-4227
e-ISSN: 2775-4219

Additionally, a reference table named R_BANK was utilized for join query tests. This table contained bank codes and their corresponding names. Its structure is detailed in Table 4.

Table 4. R_BANK Table Structure

No.	Field Name	Type
1	BANK_CODE	VARCHAR2(4)
2		VARCHAR2(100)

To measure query execution performance on the Kredit Usaha Rakyat (KUR) data across these scenarios, several types of SELECT queries were executed, designed to reflect frequently performed data retrieval operations:

1. Simple Selection Query

This type of query retrieves a specific subset of data based on exact match conditions on indexed columns. An example is:

```
SELECT * FROM TB_x WHERE bank_code = ' 200 ' AND year = '2023' AND month = '05';
```

This query, in the context of this study, typically retrieved around 10,502 rows from the total 227,587,131 rows. It was used to measure query execution time under highly specific data search conditions.

2. Selection Query with Date Range

This query retrieves data based on a range condition, often applied to date-related columns. An example is:

```
SELECT * FROM TB_x WHERE bank_code = ' 200 ' AND year = '2022' AND month BETWEEN '04' AND '06';
```

This query typically yielded around 14,002 rows. It aimed to measure execution time under broader, yet still targeted, data search conditions.

3. Aggregation Queries

This query type involves calculations across a group of rows, such as counting records per year. An example is:

```
SELECT year, COUNT(*) FROM TB_x GROUP BY year;
```

The results of this query, illustrating the number of records for each year in the dataset, are exemplified in Table 5. This test was designed to measure query execution time under data aggregation conditions.

Table 5. Aggregation Query Results

Year	Count(*)
2018	81,353,636
	19,894,688
	18 ,994,388
	32,385,801
	38,964,346
	35,994,272

4. Join Query

This query combines data from the main data table (TB_x) with the reference table (R_BANK) based on a common key. An example is:

```
SELECT a.*, b.bank_name FROM TB_x a JOIN R_BANK b ON a.kode_bank = b.kode_bank
WHERE a.kode_bank = '200' AND year = '2022' AND month = '08';
```

This query typically produced around 5,725 rows. It aimed to measure query execution time under conditions involving a simple data join.

p-ISSN: 2775-4227
e-ISSN: 2775-4219

Furthermore, for scenarios involving partitioned tables (specifically TB_4), queries explicitly referencing partition names were also executed to evaluate the direct impact of partition pruning. An example of such a query is:

```
SELECT * FROM TB_x PARTITION (P_2023)
WHERE bank_code = '200' AND year = '2023' AND month = '05';
```

The testing procedure involved several key steps. First, data preparation was conducted, where the 227,587,131 rows of KUR data were loaded into each of the four prepared tables (TB_1, TB_2, TB_3, TB_4). Reference data for the R_BANK table was also loaded. Next, query execution tests were performed on each table. Each designed query was run iteratively 10 times to ensure the accuracy of the results and to minimize variability due to transient system factors. The average execution time was then calculated from these repeated executions. To mitigate the impact of client-side caching and ensure consistent network speed evaluation, each query was executed alternately on the different source tables across the 10 attempts. This approach helps ensure that the measured query execution times are not unduly influenced by caching optimizations or network speed fluctuations, leading to more accurate and representative results

Finally, the collected measurement results were analyzed by comparing the average query execution times among the four table scenarios. This comparative analysis aimed to understand the specific impact of the different indexing and table partitioning techniques on query performance and to determine which configurations provided the most significant improvements for the types of queries tested. These systematic steps, particularly the use of a very large real-world dataset, aim to provide robust insights into the performance limits and benefits of these optimization techniques in practical scenarios.

3. RESULTS AND DISCUSSION

This chapter presents the results of the performance tests conducted on the four defined table scenarios. It is followed by a detailed discussion and analysis of these results, correlating them with existing literature to provide a comprehensive understanding of the impact of indexing and partitioning techniques on query execution performance in an Oracle database handling large-scale data.

3.1. Results

Testing was carried out by running the designed queries on four table scenarios: TB_1 (without index), TB_2 (with single-column indexes), TB_3 (with a composite index), and TB_4 (with a composite index and partitions). Each query was executed 10 times, and the average execution time (in seconds) was used for the primary analysis. The detailed results for each query type across the 10 iterations and their averages are presented in Tables 6 through 10.

1. Simple Selection Query Results

Table 6. Simple Selection Query Test Results (10 Times)

Query	TB_1 (seconds)	TB_2 (seconds)	TB_3 (seconds)	TB_3 (seconds)
1	7,000	0.211	0.108	0.148
2	6,000	0.144	0.108	0.076
3	6,000	0.155	0.148	0.101
4	6,000	0.155	0.107	0.115
5	8,000	0.140	0.081	0.087
6	10,000	0.215	0.104	0.084
7	6,000	0.151	0.131	0.090
8	6,000	0.136	0.084	0.095
9	6,000	0.116	0.068	0.086
10	6,000	0.118	0.103	0.119
AVG	**6,700**	**0.154**	**0.104**	**0.100**

2. Selection Query Results with Date Range

Table 7. Query Test Results with Date Range (10 Times)

Query	TB_1 (seconds)	TB_2 (seconds)	TB_3 (seconds)	TB_3 (seconds)

p-ISSN: 2775-4227
e-ISSN: 2775-4219

1	8,000	0.156	0.101	0.105
2	5,000	0.155	0.134	0.094
3	6,000	0.125	0.103	0.115
4	6,000	0.123	0.085	0.089
5	6,000	0.137	0.116	0.104
6	5,000	0.136	0.098	0.091
7	6,000	0.139	0.088	0.095
8	5,000	0.170	0.098	0.091
9	6,000	0.119	0.089	0.106
10	6,000	0.118	0.089	0.079
AVG	**5,900**	**0.138**	**0.100**	**0.097**

3. Aggregation Query Results

Table 8. Aggregation Query Test Results (10 Times)

Query	TB_1 (seconds)	TB_2 (seconds)	TB_3 (seconds)	TB_3 (seconds)
1	21,000	116,000	85,000	52,000
2	30,000	53,000	79,000	57,000
3	19,000	54,000	32,000	67,000
4	70,000	51,000	38,000	40,000
5	91,000	116,000	28,000	89,000
6	69,000	41,000	84,000	59,000
7	97,000	97,000	79,000	90,000
8	86,000	63,000	42,000	59,000
9	31,000	59,000	48,000	46,000
10	67,000	67,000	77,000	73,000
AVG	**58,100**	**71,700**	**59,200**	**63,200**

4. Join Query Results

Table 9. Join Query Test Results (10 Times)

Query	TB_1 (seconds)	TB_2 (seconds)	TB_3 (seconds)	TB_3 (seconds)
1	8,000	0.156	0.101	0.105
2	5,000	0.155	0.134	0.094
3	6,000	0.125	0.103	0.115
4	6,000	0.123	0.085	0.089
5	6,000	0.137	0.116	0.104
6	5,000	0.136	0.098	0.091
7	6,000	0.139	0.088	0.095
8	5,000	0.170	0.098	0.091
9	6,000	0.119	0.089	0.106
10	6,000	0.118	0.089	0.079
AVG	**5,900**	**0.138**	**0.100**	**0.097**

5. Simple Selection Query Results with Explicit Partitioning

Table 10. Simple Selection Query Test Results with Partitions (10 Times)

Query	TB_1 (seconds)	TB_2 (seconds)	TB_3 (seconds)	TB_3 (seconds)
1	N/A	N/A	N/A	0.115
2	N/A	N/A	N/A	0.093
3	N/A	N/A	N/A	0.071
4	N/A	N/A	N/A	0.078
5	N/A	N/A	N/A	0.077
6	N/A	N/A	N/A	0.070
7	N/A	N/A	N/A	0.083
8	N/A	N/A	N/A	0.075
9	N/A	N/A	N/A	0.080

p-ISSN: 2775-4227
e-ISSN: 2775-4219

10	N/A	N/A	N/A	0.073
AVG	**N/A**	**N/A**	**N/A**	**0.082**

A summary of the average execution time values for all types of queries run on the four different table scenarios (TB_1 to TB_4) is presented in Table 11. This table provides a comprehensive overview of query execution times for each indexing and table partitioning technique tested.

Table 11. Average Query Execution Time for Different Scenarios Table

Query	TB_1 (seconds)	TB_2 (seconds)	TB_3 (seconds)	TB_4 (seconds)
Simple Selection	6,700	0.154	0.104	0.100
Selection by Date Range	5,900	0.138	0.100	0.097
Aggregation	58,100	71,700	59,200	63,200
Join	5,900	0.138	0.100	0.097
Simple Selection with Partitions	N/A	N/A	N/A	0.082

3.2. Discussion

After performing query testing on the four different table scenarios (TB_1 to TB_4), the results were analyzed to determine the effectiveness of table indexing and partitioning techniques on query execution performance. The findings for each query type are discussed below, integrating insights from existing literature.

1. Simple Selection Query and Selection Query with Date Range

For both Simple Selection queries and Selection Queries with Date Range, the application of indexes (TB_2, TB_3, and TB_4) resulted in dramatically faster execution times compared to the baseline TB_1 (no index), which showed average times of 6,700 and 5,900 seconds respectively. TB_2 (single indexes) reduced these times to 0.154 and 0.138 seconds. TB_3 (composite index) further improved performance to 0.104 and 0.100 seconds. TB_4 (composite index + partition) showed similar or slightly better times (0.100 and 0.097 seconds).

This significant improvement aligns with established database optimization principles, where indexes provide a more direct path to data, avoiding costly full table scans, especially on large tables [4], [8]. The use of a composite index (TB_3) generally outperformed single-column indexes (TB_2) when multiple columns were involved in the WHERE clause, a common strategy for enhancing query performance [4]. The slight additional improvement or comparable performance in TB_4 for these selective queries suggests that while partitioning offers benefits, its impact might be less pronounced than indexing when queries are already highly selective due to effective indexing on the query predicates. However, the consistent sub-second response times for indexed tables versus thousands of seconds for the non-indexed table clearly demonstrate the fundamental importance of indexing for selective queries.

2. Aggregation Queries

The results for aggregation queries presented an interesting anomaly. TB_1 (no index) had an average execution time of 58,100 seconds. Surprisingly, TB_2 (single indexes) performed worse, with an average of 71,700 seconds. TB_3 (composite index) was slightly better than TB_2 at 59,200 seconds (comparable to TB_1), and TB_4 (composite index + partition) recorded 63,200 seconds.

This finding, where a non-indexed table outperforms an indexed one for certain aggregation queries, particularly COUNT(*) grouped by a column like YEAR which might access a large portion of the table, is not entirely uncommon. The EXPLAIN PLAN outputs for TB_1 and TB_2 (presented as Table 12 and Table 13) indicated that both scenarios ultimately resorted to a TABLE ACCESS FULL and a HASH GROUP BY operation.

EXPLAIN PLAN FOR SELECT year, COUNT(*) FROM TB_1 GROUP BY year;

Table 12. Output Explain Plan TB_1

Id	Operations	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		6	30	834K (2)	00:00:33
1	HASH GROUP BY		6	30	834K (2)	00:00:33
2	TABLE ACCESS FULL	TB_1	227M	1085M	826K (1)	00:00:33

p-ISSN: 2775-4227
e-ISSN: 2775-4219

EXPLAIN PLAN FOR SELECT year, COUNT(*) FROM TB_2 GROUP BY year;

Table 13. Output Explain Plan TB_2

Id	Operations	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		6	30	834K (2)	00:00:33
1	HASH GROUP BY		6	30	834K (2)	00:00:33
2	TABLE ACCESS FULL	TB_2	227M	1085M	826K (1)	00:00:33

The identical execution plans suggest that the Oracle optimizer, in these specific aggregation scenarios, chose not to use the available single indexes on YEAR in TB_2 for the primary data access path, likely deeming a full scan more efficient for gathering all necessary data for the GROUP BY operation. The slight performance degradation in TB_2 compared to TB_1 could be attributed to the overhead associated with maintaining indexes, even if they are not directly used for the primary scan in the chosen plan [12], [13]. When an aggregation query needs to access a large percentage of rows, a full table scan can sometimes be faster than an index scan followed by numerous table accesses by ROWID, especially if the data is not highly clustered or if the index is not covering [4].

The fact that partitioning (TB_4) did not consistently improve performance over the non-indexed or composite-indexed (TB_3) table for this specific GROUP BY YEAR query also warrants discussion. While partitioning by YEAR might seem intuitive for a query grouping by YEAR, if the aggregation still requires scanning all data within each partition (or many partitions) to get the counts, the benefit of partition pruning might be limited for this specific query type if all years are being aggregated. Some studies note that the effectiveness of partitioning is highly dependent on the query's ability to prune partitions effectively [6], [7]. If the GROUP BY spans all or most partitions, the overhead of managing multiple segments might not be offset by pruning benefits.

3. Join Query

For the simple join query, TB_1 (no index) was again the slowest at 5,900 seconds. TB_2 (single indexes) improved this to 0.138 seconds, TB_3 (composite index) to 0.100 seconds, and TB_4 (composite index + partition) achieved the best time at 0.097 seconds.

These results affirm that both indexing on join keys and table partitioning can significantly benefit join operations [6]. Indexes allow for faster lookups of matching rows between tables, and partitioning can enable partition-wise joins, where smaller joins are performed between corresponding partitions, potentially reducing the overall complexity and I/O [1]. The superior performance of TB_4 suggests that the combination of a composite index (covering join and filter columns) and partitioning provided the most optimal execution path for this join query.

4. Simple Selection Query with Explicit Partitioning

Additional testing on TB_4 using a simple selection query that explicitly specified a partition (e.g., PARTITION (P_2023)) yielded an average execution time of 0.082 seconds. This was faster than the same query on TB_4 without explicit partition naming (0.100 seconds).

This highlights the effectiveness of partition pruning, a key benefit of table partitioning [14], [17]. When the optimizer can definitively identify and access only the relevant partition(s), the amount of data scanned is drastically reduced, leading to faster query responses. This is consistent with literature emphasizing that partitioning's main performance advantage comes from limiting data access to only necessary segments [7].

5. General Discussion on Findings

The overall results indicate that for queries involving specific row lookups or small range scans (Simple Selection, Selection by Date Range, Join), indexing (both single and composite) provides massive performance improvements over no indexing. Composite indexes generally offer an edge when multiple indexed columns are used in predicates.

The relatively small difference in execution times for these selective queries between the optimized scenarios (TB_2, TB_3, TB_4 – all sub-second) compared to the non-optimized TB_1 (thousands of seconds) might also be influenced by the high specifications of the test server (32-core CPU, 126 GB RAM). On such powerful hardware, once data is efficiently located via an index, subsequent processing can be very fast, potentially masking finer-grained differences between various optimized approaches for less complex queries. However, the fundamental benefit of indexing remains clear.

The aggregation query results underscore that optimization techniques are not universally beneficial for all query types. The decision by the Oracle optimizer to use a full table scan even when indexes were available

p-ISSN: 2775-4227
e-ISSN: 2775-4219

suggests a cost-based analysis where the overhead of using the index for that specific aggregation pattern was deemed higher than a direct scan. Factors like data distribution, index clustering factor (how well the physical data order on disk matches the index order), and the percentage of table blocks that would need to be visited via an index scan play a role in such optimizer decisions.

The application of indexing and table partitioning techniques in Oracle databases, as demonstrated, generally enhances query execution performance, particularly for selective data retrieval and join operations on large datasets. However, the effectiveness is contingent on the query type, data characteristics, and how well the chosen optimization strategy aligns with data access patterns [5]. The anomaly with aggregation queries suggests that for certain operations that require accessing a large portion of the data, the optimizer might correctly favor full scans, and the presence of indexes (especially if not covering) might not offer benefits or could even introduce slight overhead.

4. CONCLUSION

This research demonstrated that applying indexing (single and composite B-tree) and range-based table partitioning techniques can significantly enhance SQL query execution performance on an Oracle 19c database handling large-scale Kredit Usaha Rakyat (KUR) data, although the extent of improvement varies by query type. For selective queries, including simple selections, range selections, and joins, indexing provided substantial performance gains over non-indexed tables, with composite indexes generally outperforming single-column indexes. The addition of table partitioning, particularly when explicitly referenced, offered further, albeit sometimes marginal, improvements for these query types. Conversely, for the specific aggregation query tested (GROUP BY YEAR), a non-indexed table surprisingly yielded faster execution times than scenarios with single indexes or a combination of composite indexes and partitioning; the database optimizer consistently chose full table scans for these aggregation operations across all scenarios.

The findings imply that while indexing is crucial for optimizing most data retrieval operations on large datasets, its universal application without considering query-specific access patterns, especially for broad aggregations, may not always yield the best performance. The choice of optimization strategy—whether indexing, partitioning, or a combination—must be carefully tailored to the predominant query types and data characteristics. For future research, it is recommended to explore these techniques with more diverse and complex query workloads, include Data Manipulation Language (DML) operations, and investigate other optimization methods such as advanced query tuning and materialized views. Practically, organizations should regularly profile their database performance and selectively apply optimization techniques, supported by adequate server infrastructure, to ensure efficient data processing and support timely decision-making.

REFERENCE

[1] V. B. Ramu, “Optimizing Database Performance: Strategies for Efficient Query Execution and Resource Utilization,” *International Journal of Computer Trends and Technology*, vol. 71, no. 7, pp. 15–21, Jul. 2023, doi: 10.14445/22312803/ijctt-v71i7p103.

[2] S. Samidi, D. Iskandar, M. Fachruroji, W. A. S. Wibowo, and A. Khaerani A, “Database Tuning in Hospital Applications Using Table Indexing and Query Optimization,” *Jurnal Pendidikan Tambusai*, vol. 6, no. 1, pp. 1960–1967, 2022, doi: 10.31004/jptam.v6i1.3241.

[3] D. ’ Ulhaq, A. Ramsi, A. A. Ertiansyah, and S. Mukaromah, “Prosiding Seminar Nasional Teknologi dan Sistem Informasi (SITASI) 2023 Surabaya,” 2023. [Online]. Available: https://www.kaggle.com/kazanova/sentiment140

[4] A. A. Chikkamannur, “Indexing Strategies for Performance Optimization of Relational Databases,” *International Research Journal of Engineering and Technology*, 2021, [Online]. Available: www.irjet.net

[5] Samidi, Fadly, Y. Virmansyah, R. Y. Suladi, and A. B. Lesmana, “Optimasi Database dengan Metode Index dan Partisi Tabel pada Aplikasi E-Commerce. Studi pada Aplikasi Tokopintar,” *Jurnal Pendidikan Tambusai*, vol. 6, no. 1, pp. 2094–2102, 2022, [Online]. Available: https://jptam.org/index.php/jptam/article/view/3241

[6] W. Khan, C. Zhang, B. Luo, T. Kumar, and E. Ahmed, “Robust Partitioning Scheme for Accelerating SQL Database,” in *2021 IEEE International Conference on Emergency Science and Information Technology (ICESIT)*, 2021, pp. 369–376. doi: 10.1109/ICESIT53460.2021.9696761.

p-ISSN: 2775-4227
e-ISSN: 2775-4219

[7] V. Šalgová and K. Matiaško, "The Effect of Partitioning and Indexing on Data Access Time," in *2021 29th Conference of Open Innovations Association (FRUCT)*, 2021, pp. 301–306. doi: 10.23919/FRUCT52173.2021.9435500.

[8] A. Anchlia, "Enhancing Query Performance Through Relational Database Indexing," *International Journal of Computer Trends and Technology*, vol. 72, no. 8, pp. 130–133, Aug. 2024, doi: 10.14445/22312803/IJCTT-V72I8P119.

[9] S. O. Eka Putri, N. A. Zahra, N. R. Kuslaila, and S. Mukaromah, "PERBANDINGAN TEKNIK INDEXING BITMAP DAN B-TREE PADA ORACLE DATABASE," *Prosiding Seminar Nasional Teknologi dan Sistem Informasi*, vol. 3, no. 1, pp. 439–446, Nov. 2023, doi: 10.33005/sitasi.v3i1.390.

[10] I. C. Saidu, M. Yusuf, F. C. Nemariyi, and A. C. George, "Indexing techniques and structured queries for relational databases management systems," *Journal of the Nigerian Society of Physical Sciences*, vol. 6, no. 4, Nov. 2024, doi: 10.46481/jnsps.2024.2155.

[11] T. Yu, Z. Zou, W. Sun, and Y. Yan, "Refactoring Index Tuning Process with Benefit Estimation," in *Proceedings of the VLDB Endowment*, VLDB Endowment, 2024, pp. 1528–1541. doi: 10.14778/3654621.3654622.

[12] M. Kvet, "Database Index Balancing Strategy," in *2021 29th Conference of Open Innovations Association (FRUCT)*, 2021, pp. 214–221. doi: 10.23919/FRUCT52173.2021.9435452.

[13] V. Šalgová, "Effect of indexes on DML operations," *Transportation Research Procedia*, vol. 55, pp. 1368–1372, May 2021, doi: 10.1016/j.trpro.2021.07.121.

[14] P. F. Tanaem, A. R. Tanaamah, and infraim oktofianus boymau, "Partition Table in STARS: Concept and Evaluations," Oct. 17, 2022. doi: 10.36227/techrxiv.21324264.v1.

[15] M. H. Rachman, S. Samidi, and E. Aprianto, "COMPARISON OF INDEX, PARTITION, AND MATERIALIZED VIEW METHODS ON THE ORACLE DATABASE STUDY ON CENTRAL GOVERNMENT FINANCIAL REPORTS (LKPP)," *Jurnal Teknik Informatika (Jutif)*, vol. 5, no. 4, pp. 1009–1014, Jul. 2024, doi: 10.52436/1.jutif.2024.5.4.1962.

[16] H. C. Oracle, "VLDB and Partitioning Guide," https://docs.oracle.com/en/database/oracle/oracle-database/23/vldbg/index.html.

[17] I. Oktofianus Boymau, P. Fiodinggo Tanaem, A. Rocky Tanaamah, and K. Satya Wacana, "Tabel Partisi Pada STARS: Konsep Dan Evaluasi (Studi Kasus STARS UKSW)," vol. 8, no. 2, 2023.