International Journal Software Engineering and Computer Science (IJSECS)
6 (3), 2026, 1142-1152
Published Online December 2026 in IJSECS (http://www.journal.lembagakita.com/index.php/ijsecs)
P-ISSN: 2776-4869, E-ISSN: 2776-3242. DOI: https://doi.org/10.35870/ijsecs.v6i3.8351.

RESEARCH ARTICLE Open Access

Performance Analysis of Oracle Database 23ai on SELECT Queries and VIEW Statements Under Intensive Iterations

Yosafat Stephen Suryadarma [1*], **Dian Widiyanto Chandra** [2]

[1*,2] Department of Informatics Engineering, Satya Wacana Christian University, Salatiga City, Central Java Province, Indonesia.

Article History:

Received: August 10, 2026.
Revised: August 11, 2026.
Accepted: August 27, 2026.
Available online: September 20, 2026.
Published: December 1, 2026.

***Corresponding Author:**
stephenn2465@gmail.com

Abstract: The rapid growth of data volume demands efficient data management techniques, where appropriate query writing strategies become a crucial factor in maintaining system performance. This study aimed to analyze the performance of SELECT and VIEW commands on Oracle Database 23ai, specifically under intensive benchmarking involving thousands of iterations and massive workloads. An experimental quantitative method was implemented using a Java-based application utilizing the Operating System and Hardware Information (OSHI) library to monitor system resource usage in real time. Testing was conducted on a simulation dataset generating 40,000 rows of joined query results derived from a 40,000-row orders table and a 10,000-row customers table (~10.44 MB), with workload variations ranging from 10 to 10,000 iterations. Performance metrics evaluated included response time as well as CPU, RAM, and disk utilization. The results demonstrated that the SELECT command achieved superior performance at low iteration scales (10–100 iterations) with faster response times. Conversely, the VIEW command demonstrated more optimal execution at high iteration scales (1,000–10,000 iterations), yielding lower average execution times than SELECT. In terms of hardware utilization, VIEW consistently maintained more efficient RAM management across all iteration levels. The performance stability of both commands was governed by the database buffer cache mechanism and the operational transition from hard parse to soft parse after the initial execution phase. This study concludes that selecting between SELECT and VIEW must strictly account for the execution scale: SELECT is recommended for low-scale queries requiring immediate response, whereas VIEW provides a more sustainable solution for enterprise systems executing repetitive, large-scale workloads.

Keywords: Oracle 23ai; SELECT; VIEW; Database Performance; Response Time.

1. Introduction

The exponential growth of data generated by modern enterprise information systems demands efficient data management techniques, particularly in terms of storage efficiency and retrieval throughput. This rapid data expansion poses significant challenges to maintaining low latency during high-volume data retrieval operations. While optimization techniques such as indexing substantially enhance search performance compared to unindexed structures (Thoib *et al.*, 2024), ongoing data accumulation continues to present risks of application throughput degradation. Consequently, beyond hardware provisioning, selecting appropriate query formulation strategies represents a decisive factor in sustaining system performance (Putra *et al.*, 2022).

Beyond query structure design, database optimization relies heavily on how the Database Management System (DBMS) manages its database buffer cache. The buffer cache retains frequently accessed data blocks in volatile memory to minimize physical disk I/O latency (Andriadi & Faqih, 2026). Within the Oracle architecture, this mechanism operates inside the System Global Area (SGA) and can be evaluated using administrative monitoring utilities. To maintain optimal throughput, the database engine tracks execution statistics to calculate the cache hit ratio, thereby preventing costly physical storage reads (Oracle, 2025). Two fundamental SQL structures utilized for relational data retrieval are the standalone SELECT query and the stored VIEW statement. A standard SELECT query retrieves records directly from one or more tables, offering dynamic flexibility in defining projections, join conditions, and filter predicates. In contrast, a VIEW functions as a stored virtual relation representing a pre-compiled query definition; it stores no physical data itself but dynamically evaluates the underlying query block when referenced (Aulia *et al.*, 2023). Previous empirical benchmarks have primarily evaluated query execution times across divergent database engines such as MySQL, PostgreSQL, MongoDB, and SQLite. However, rigorous evaluations examining the behavioral efficiency of VIEW queries remain sparse, particularly on enterprise-grade platforms such as Oracle Database. Furthermore, prior studies have rarely captured real-time hardware resource consumption—such as CPU overhead, RAM allocation, and disk I/O activity—during continuous execution cycles. Currently, empirical evaluations benchmarking the operational performance of SELECT versus VIEW structures on Oracle Database 23ai under intensive workloads exceeding thousands of iterations, combined with automated resource tracing via Java runtime environments, remain unexplored. To address this gap, this study investigates the performance divergence between SELECT queries and VIEW statements on Oracle Database 23ai under massive execution workloads scaling up to 10,000 iterations. Specifically, this research evaluates system resource consumption (CPU, RAM, and disk utilization) and response time stability within a large-scale inner join execution scenario. In addressing these objectives, this study hypothesizes that query execution efficiency in Oracle Database 23ai is governed by the workload scale. Specifically, standalone SELECT queries are expected to achieve faster response times under low-iteration workloads (10–100 iterations) due to lower initial structural resolution overhead, whereas VIEW queries are hypothesized to yield superior memory efficiency, lower disk utilization, and greater latency stability across repetitive, high-iteration workloads (1,000–10,000 iterations) by leveraging the shared SQL area and buffer cache optimization. Consequently, the performance superiority of VIEW queries is anticipated to be conditional, depending directly on the operational iteration scale.

2. Related Work

2.1 State-of-the-Art Research

A number of prior studies have examined query execution performance across relational database systems. Noviyanti *et al.* (2018) demonstrated that the use of VIEW yields faster execution times than the CROSS PRODUCT model in non-partitioned environments, particularly when evaluated on datasets ranging from 100 to 1,000 rows. In another study, Wathani *et al.* (2025) investigated the effectiveness of indexes in accelerating JOIN query execution in relational database systems, showing that indexing can improve execution efficiency by up to 50% in triple-join scenarios featuring conditional filters. Focusing on engine scalability, Nugraha and Susetyo (2023) conducted a comparative performance evaluation between DuckDB and SQLite for big data workloads, emphasizing the necessity of selecting database engines based on analytical volume requirements and processing architectures. Similarly, Putra *et al.* (2022) evaluated query execution times across MySQL, PostgreSQL, and MongoDB, reporting that PostgreSQL achieved superior throughput when executing SELECT commands on large-scale datasets. Furthermore, Abdi *et al.* (2021) compared retrieval latency between SQL and NoSQL engines across multiple query conditions, concluding that while NoSQL engines exhibit higher throughput for massive data searches, SQL engines remain superior for executing simpler, highly structured queries.

2.2 Comparison with Previous Studies

Beyond raw query execution metrics, internal memory management plays a fundamental role in database throughput optimization. The database buffer cache functions as a memory management layer designed to bridge the latency disparity between high-speed CPU execution and physical disk I/O. As explained by Gadupudi and Saha (2025), buffer management has evolved beyond standard Least Recently Used (LRU) algorithms into intelligent eviction schemes, such as Oracle's touch-count algorithm. This mechanism prevents sequential flooding—a condition where large-scale sequential table scans evict frequently referenced data blocks from memory. By tracking block access frequency, the database engine maintains data residency within the System Global Area (SGA), thereby maximizing the cache hit ratio. The efficiency of these memory management mechanisms is further governed by the underlying technical data structures utilized in the database. Putra and Samidi (2025) observed that utilizing specialized indexing structures, such as Bitmap Indexes, significantly improves memory resource utilization in Oracle databases. Benchmarking on datasets reaching 600 million rows demonstrated that the compact footprint of Bitmap Indexes minimizes buffer cache overhead compared to standard B-Trees. This structural efficiency reduces physical I/O activity while optimizing RAM and CPU consumption, thereby sustaining workload stability under high query frequencies. A comparative synthesis of these prior studies and the technical positioning of this research is summarized in Table 1.

Table 1. Synthesis of Prior Studies and the Novelty of This Research

Author & Year	DBMS Tested	Workload / Scale	Key Metrics Monitored	Automated Hardware Monitoring	Core Focus / Findings
Noviyanti *et al.* (2018)	MySQL	100–1,000 rows	Response time	No	Compared VIEW vs. CROSS PRODUCT in non-partitioned environments
Wathani *et al.* (2025)	Relational DBMS	Triple join with filter conditions	Execution efficiency	No	Evaluated effectiveness of indexes in accelerating JOIN query execution
Putra *et al.* (2022)	MySQL, PostgreSQL, MongoDB	Large-scale	Response time	No	Evaluated query speed across relational and NoSQL platforms, showing PostgreSQL superiority
Nugraha & Susetyo (2023)	DuckDB, SQLite	Big data workloads	Performance speed	No	Evaluated big data processing capability differences based on volume
Abdi *et al.* (2021)	SQL, NoSQL	Multi-scenario search	Data retrieval speed	No	Tested retrieval speeds, showing NoSQL faster for big data, SQL for simpler queries
This Study (2026)	Oracle Database 23ai	10–10,000 iterations (Macro & Micro)	Response time, CPU, RAM, disk utilization	Yes (Java application utilizing OSHI library in real time)	Analyzes the internal performance dynamics (hard/soft parse, SGA cache) of SELECT vs. VIEW under massive workloads

2.3 Research Positioning and Novelty

While previous investigations have extensively compared throughput across divergent DBMS platforms (Abdi *et al.*, 2021; Nugraha & Susetyo, 2023; Putra *et al.*, 2022) or evaluated VIEW operations under constrained workloads (Noviyanti *et al.*, 2018), empirical assessments examining the behavioral divergence between standalone SELECT queries and VIEW statements on Oracle Database 23ai under high-frequency iteration workloads remain absent. Furthermore, existing comparative studies have largely omitted automated, real-time hardware telemetry—specifically CPU utilization, RAM consumption, and disk I/O activity—during continuous execution loops. This study addresses this empirical gap by delivering an end-to-end performance benchmark of SELECT and VIEW commands on Oracle Database 23ai across workloads scaling from 10 to 10,000 iterations, coupling latency recording with real-time hardware monitoring via the OSHI library.

2.4 Review of Technologies, Frameworks, and Algorithms

Oracle Database 23ai incorporates intelligent performance optimization frameworks into its enterprise relational engine. Central to its execution architecture is the System Global Area (SGA), which houses the Database Buffer Cache to cache frequently accessed data blocks in volatile memory, thereby mitigating physical I/O latency (Oracle, 2025). The query compilation lifecycle comprises two principal parsing mechanisms: a Hard Parse, which validates statement syntax, evaluates object permissions, and calculates an execution plan; and a Soft Parse, which bypasses execution plan computation by reusing compiled execution paths stored in the Library Cache within the Shared Pool of the SGA. Oracle processes queries referencing a VIEW distinctly from ad-hoc standalone SELECT statements. A VIEW is maintained as a virtual object storing only the query specification within the database data dictionary. When a query references a VIEW, the relational engine invokes structural query transformations, such as select-project-join view merging, to combine the view definition with the containing query block and generate a unified execution plan (Ahmed *et al.*, 2020). This automated transformation minimizes the number of discrete query blocks evaluated by the query optimizer. As described by Kvet and Papan (2024), the optimizer determines the most cost-effective access paths for the merged structure, caching the resulting execution plan in the Shared Pool for subsequent re-execution. This synergy between structural query rewriting and memory-based plan caching enables VIEW statements to minimize repetitive compilation overhead during prolonged execution cycles. In contrast, ad-hoc SELECT queries involving multi-table joins require structural validation and metadata resolution during initial executions, incurring higher processing overhead (Hard Parse). Once execution plans are cached in the Library Cache, subsequent iterations of both SELECT and VIEW statements transition to a Soft Parse state. Under these intensive execution workloads, residency of data blocks in the Database Buffer Cache reduces physical disk reads, thereby stabilizing execution latency across high iteration scales. The benchmarking framework was developed using Java (JDK 17), utilizing System.currentTimeMillis() for execution latency recording. Operating system and hardware telemetry—including active CPU load, system RAM allocation, and storage disk throughput—was captured programmatically using the Operating System and Hardware Information (OSHI) library. This integrated architecture ensures automated, objective, and deterministic data acquisition throughout testing.

3. Methodology

This study employed an experimental quantitative method to evaluate the operational performance of SELECT and VIEW commands on Oracle Database 23ai. Benchmarking was performed in a local host environment running on the 64-bit Windows 11 operating system. The physical host hardware comprised an 11th Generation Intel Core i5 processor, 12 GB of DDR4 RAM, and a 512 GB NVMe ADATA LEGEND 710 solid-state drive. The database engine tested was Oracle Database 23ai Enterprise Edition installed locally, while the automated testing client was developed using Java version 17 (JDK 17) within the NetBeans IDE. Postman was utilized as an auxiliary interface to dispatch API endpoint requests to the Java application, ensuring deterministic communication between the execution harness and the database instance. Several experimental boundaries were established to scope the benchmark. Testing was restricted to Oracle Database 23ai, focusing strictly on a standalone SELECT query versus a virtual VIEW statement within an inner join context, without comparative evaluation across alternative DBMS engines. The processed data comprised synthetic relational records generated to simulate structured transaction workloads rather than production enterprise records. The execution environment was constrained to a single-host architecture, thereby eliminating network transmission latency and distributed server routing variables. Furthermore, no internal process isolation was enforced over operating system background activities; therefore, scheduling mechanisms such as OS context switching and system thread interrupts were preserved as external variables inherent to the testing environment. Experimental iterations from 10 to 10,000 runs were executed in a continuous, single-run sequential session. Performance data analysis was conducted using descriptive statistical averages calculated from execution timings recorded via System.currentTimeMillis() and telemetry gathered through the OSHI library, without conducting inferential statistical hypothesis testing. The experimental dataset comprised two relational tables: an orders table containing 40,000 records totaling approximately 10 MB, and a customers table containing 10,000 records totaling approximately 0.44 MB. To ensure experimental reproducibility, the precise query structures executed during testing are defined. The standalone SELECT query used to perform the inner join between the two entities is formulated as follows:

```
SELECT
    o.order id, o.order date, o.total amount, o.notes,
    c.customer id, c.name, c.city
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id;
```

Correspondingly, the virtual VIEW relation created in the Oracle Database 23ai data dictionary is defined as follows:

```
CREATE OR REPLACE VIEW orders customers view AS
SELECT
    o.order_id, o.order_date, o.total_amount, o.notes,
    c.customer_id, c.name, c.city
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id;
```

During subsequent benchmarking iterations, this view structure was queried using the following statement:

```
SELECT * FROM orders customers view;
```

Query complexity was centered on joining the orders and customers tables via the shared foreign key column (customer_id) without auxiliary filtering predicates (WHERE clauses). This isolation ensured that performance evaluations focused directly on the processing throughput of 40,000 joined result rows, representing an exhaustive many-to-one relationship evaluation and its impact on statement parsing mechanisms within Oracle Database 23ai. To isolate relational query execution and join processing efficiency, no specialized indexing schemes or physical caching structures were configured; the database instance relied entirely on default primary key B-tree indexes generated on order_id and customer_id. Furthermore, testing loops were executed starting from a cold cache state without applying database warm-up routines prior to execution, allowing the benchmarking harness to capture the internal transition from hard parse to soft parse. System-level context switching and background operating system overhead in Windows 11 were retained and evaluated as environmental variables rather than removed as statistical outliers. To evaluate macro-scale throughput alongside micro-level initialization dynamics, the experimental framework was organized into two distinct evaluation phases: Macro-testing and Micro-testing. The Macro-testing phase assessed database scalability and hardware consumption across four logarithmic workload tiers: 10, 100, 1,000, and 10,000 iterations. The Micro-testing phase captured transient latency spikes and initial parsing overhead at a higher sampling resolution, recording measurements at discrete intervals of 10 iterations from 10 up to 1,000 iterations. Micro-testing was capped at 1,000 iterations because query execution latency stabilized completely beyond this threshold, indicating that the Database Buffer Cache and library cache reuse had fully minimized transient execution fluctuations. During the execution cycle, the testing framework recorded multiple performance parameters, including individual query response time, cumulative response time, mean response time, and real-time hardware consumption across CPU load, RAM allocation, and storage disk throughput. Execution latency was captured via the native Java System.currentTimeMillis() function, while hardware telemetry was monitored using the Operating System and Hardware Information (OSHI) library. This monitoring design aligns with the instrumentation approach documented by Yang *et al.* (2025), who utilized OSHI as a high-precision sampler for capturing CPU metrics and memory states within a Java runtime environment. All telemetry was collected programmatically in real time without manual intervention. The resulting experimental data was evaluated quantitatively by computing average execution latencies across each workload scenario and assessing hardware utilization between the SELECT and VIEW commands. The consolidated metrics were subsequently structured into comparative tables and analytical graphs using Microsoft Excel to support evaluation.

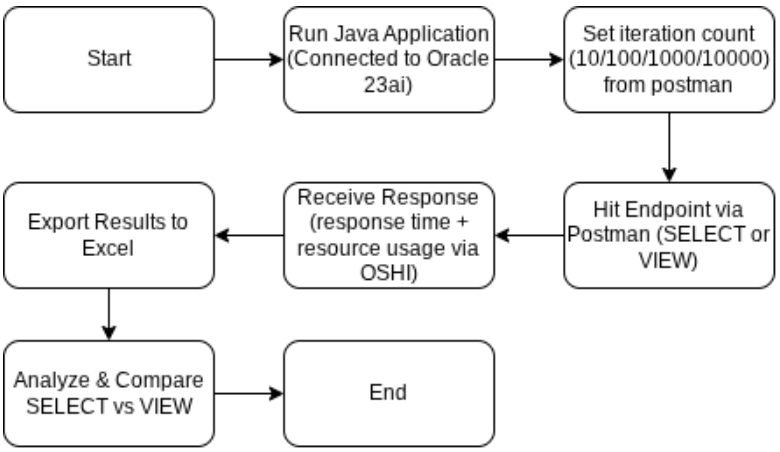

Figure 1. Testing Process Flow

Figure 1 outlines the operational testing process flow implemented in this study. The procedure begins by launching the Java benchmarking harness connected directly to the local Oracle Database 23ai instance. Subsequently, execution parameters are configured via Postman to define the target workload, designating either the Macro-testing tiers (10, 100, 1,000, or 10,000 iterations) or the Micro-testing sequence (10-iteration intervals up to 1,000 iterations). Once the workload configuration is initialized, the API endpoint is triggered in Postman by specifying the target query structure (SELECT or VIEW). During continuous execution, the Java application captures individual statement response times while concurrently gathering real-time CPU, RAM, and disk utilization through the OSHI library. Upon completing the designated iteration cycle, the compiled performance metrics are exported to Microsoft Excel for synthesis and comparative evaluation. To support experimental transparency and scientific reproducibility, the complete Java source code, OSHI monitoring scripts, execution documentation, and full raw datasets for both Macro-testing and Micro-testing phases are openly available in the GitHub repository at https://github.com/stephen11456/oracle-23ai-select-view-comparison.

4. Results and Discussion

4.1 Results

This study evaluated the operational performance of SELECT and VIEW commands on Oracle Database 23ai across two experimental tiers: Macro-testing (evaluating broad scalability trends across 10, 100, 1,000, and 10,000 iterations) and Micro-testing (capturing granular latency transitions from 10 to 1,000 iterations at 10-iteration intervals). The benchmark recorded four primary metrics: query response time, CPU utilization, RAM consumption, and disk activity. The empirical results are presented systematically below.

Table 2. Average Response Time of SELECT and VIEW Queries

Iteration	SELECT (Avg)	VIEW (Avg)
10	331 ms	372 ms
100	276 ms	329 ms
1000	440 ms	426 ms
10000	397 ms	320 ms

Based on the execution latency results summarized in Table 2, distinct performance variations emerged between the SELECT and VIEW commands across divergent workload scales. At low iteration tiers (10 and 100 iterations), the standalone SELECT query consistently demonstrated faster execution than VIEW, recording average response times of 331 ms and 276 ms, respectively. Conversely, at higher workloads (1,000 and 10,000 iterations), the VIEW statement exhibited superior execution efficiency, recording lower average response times of 426 ms at 1,000 iterations and 320 ms at 10,000 iterations.

Table 3. Average CPU Usage of SELECT and VIEW Queries

Iteration	SELECT (Avg)	VIEW (Avg)
10	27.2%	43.6%
100	21.9%	21.9%
1000	22.6%	22.8%
10000	20.6%	17.0%

As shown in Table 3, a substantial divergence in CPU consumption occurred at 10 iterations, where VIEW registered 43.6% utilization compared to 27.2% for SELECT. This elevated computational load directly corresponds to the higher initial response time recorded in Table 2. Technically, this surge reflects an intensive hard parse phase on VIEW, during which Oracle resolves and compiles the execution plan for the underlying multi-table join structure. However, this computational gap closed at 100 and 1,000 iterations as the execution transitioned to a soft parse state, stabilizing CPU consumption at approximately 21.9% for both commands. At 10,000 iterations, a definitive efficiency shift occurred: VIEW's CPU load decreased to 17.0%, notably lower than SELECT at 20.6%. This drop illustrates the cumulative benefit of soft parse utilization and memory caching within the SGA, where the processing overhead of resolving joined relations within a stored view diminishes substantially once initialization concludes.

Table 4. Average RAM Usage of SELECT and VIEW Queries

Iteration	SELECT (Avg)	VIEW (Avg)
10	82.7%	77.3%
100	76.4%	75.9%
1000	75.3%	72.8%
10000	72.2%	70.9%

Based on the measurements in Table 4, system RAM consumption peaked during the initial 10 iterations at 82.7% for SELECT and 77.3% for VIEW. This elevated initial memory allocation is driven by resource provisioning within the System Global Area (SGA) to facilitate hard parsing and populate the Database Buffer Cache from physical storage. For SELECT, higher memory demand was driven by intensive physical I/O throughput (6.40% in Table 5) required to scan raw data blocks, whereas VIEW's virtual relation model supported more compact metadata mapping from the onset. Across the transition phase (100 to 1,000 iterations), distinct memory release patterns were observed. RAM allocation for VIEW dropped by approximately 3.1% (from 75.9% to 72.8%), whereas SELECT exhibited a marginal reduction of roughly 1.1% (from 76.4% to 75.3%). The slower rate of memory reclamation for SELECT at 1,000 iterations aligns with the latency spike identified in Table 2. This behavior is attributed to operating system context switching and background thread interruptions within Windows 11. A comparable 3.1% memory reduction for SELECT was attained only at 10,000 iterations, after the runtime state fully stabilized under soft parsing. The sustained memory efficiency of VIEW (70.9%) relative to SELECT (72.2%) under maximum workload confirms that virtual structure caching in the SGA offers superior memory scalability for high-frequency execution cycles.

Table 5. Average Disk Usage of SELECT and VIEW Queries

Iteration	SELECT (Avg)	VIEW (Avg)
10	6.40%	0.08%
100	0.05%	0.02%
1000	0.10%	0.16%
10000	0.05%	0.07%

As presented in Table 5, physical disk activity displayed pronounced fluctuations during initial execution. At 10 iterations, SELECT generated a sharp I/O spike of 6.40%, whereas VIEW registered only 0.08%. This marked divergence indicates heavy physical I/O on SELECT as raw table records were initially scanned from physical storage and populated into volatile memory during the hard parse phase. Conversely, VIEW maintained minimal initial disk interaction because Oracle 23ai pre-resolved the relational schema within data dictionary metadata, thereby bypassing ad-hoc disk reads. At 100 iterations, disk activity for both statements dropped sharply below 0.1% (0.05% for SELECT and 0.02% for VIEW). This steep decline validates the effectiveness of the Database Buffer Cache in retaining accessed data blocks in RAM (SGA), eliminating subsequent disk reads. This migration from physical storage to memory-resident processing explains the stabilized response times reported in Table 2. A secondary variation emerged at 1,000 iterations, where disk activity rose slightly to 0.10% for SELECT and 0.16% for VIEW. While representing a notable relative increase, this shift reflects operating system context switching and background scheduler interruptions in Windows 11 rather than internal database engine degradation. Given that the compact dataset (~10.44 MB) executes within millisecond windows, execution timing is sensitive to host-level thread preemption. At 10,000 iterations, disk activity stabilized at 0.05% for SELECT and 0.07% for VIEW, confirming sustained reliance on soft parsing and high memory cache hit ratios. Although VIEW registered a minor increase in disk utilization at 10,000 iterations, it concurrently delivered faster response times (320 ms vs. 397 ms for SELECT), demonstrating that volatile memory optimization in VIEW exerts greater leverage over large-scale throughput than residual disk activity.

To evaluate transient execution behaviors and initialization anomalies at higher granularity, a Micro-testing benchmark was conducted at 10-iteration intervals from 10 to 1,000 iterations. The complete step-by-step dataset for this phase is accessible within the project's public repository under the /data/data-micro-testing/ directory. Experimental measurements revealed that the average execution latency of SELECT queries began increasing at iteration 30 (350 ms), climbing to a peak latency of 455 ms at iteration 70. Following this peak, SELECT response times declined progressively as the execution environment stabilized, settling at a consistent baseline of approximately 185 ms around iteration 200, with minor millisecond-scale variations persisting thereafter. The lowest average latency for SELECT was recorded at 185 ms during iteration 280. In comparison, VIEW queries exhibited initial latency spikes at iteration 10 (336 ms) and iteration 40 (367 ms), reaching an overall maximum latency of 369 ms at iteration 550. Subsequently, VIEW latencies trended downward, establishing consistent stability between iterations 600 and 650, and recording an absolute minimum of 181 ms at iteration 650. Under fully optimized cache residency, both query types achieved comparable steady-state performance. These empirical trends confirm that the Database Buffer Cache

activates effectively once initial storage reads conclude, migrating subsequent operations into volatile RAM. This transition substantially mitigates execution latency, ensuring consistent query throughput. This mechanism corroborates the architectural framework documented by Wagner, Nissan, and Rasin (2023), who established that relational database engines require data blocks to be loaded into I/O buffer cache structures in main memory prior to query evaluation. Residual micro-fluctuations observed following cache stabilization are driven by host-level operating system scheduling and context switching in Windows 11. Because the active dataset (~10.44 MB) is evaluated across millisecond intervals, runtime sampling is susceptible to CPU thread preemption. This behavior aligns with Huang *et al.* (2025), who demonstrated that standard operating system scheduling interrupts induce delivery latency variations across low-latency thread workloads. Furthermore, the initial latency surges observed during early iterations reflect statement compilation overhead within the Shared Pool. As documented by Oracle (2025), the database engine must execute a hard parse during initial execution to validate syntax, verify schema permissions, and calculate optimal access paths. This is consistent with Karri and Muntala (2023), who observed that execution plan calculations for joined relational structures represent computationally demanding operations that induce initial execution latency. The elevated CPU consumption during this query resolution phase generated the early latency peaks for SELECT (455 ms) and VIEW (367 ms). Once compilation concluded and execution blueprints were retained in the Library Cache, subsequent iterations transitioned entirely to soft parsing. Across the overall Micro-testing spectrum, VIEW statements yielded a lower aggregate average response time (235 ms) and a lower maximum peak (369 ms) than SELECT queries (average 268 ms, peak 455 ms). Nonetheless, SELECT established steady-state latency earlier (around iteration 200), whereas VIEW exhibited a prolonged stabilization phase before converging between iterations 600 and 650.

4.2 Discussion

The empirical findings demonstrate that the comparative performance of SELECT queries and VIEW statements on Oracle Database 23ai is governed by execution scale. At low workloads (10–100 iterations), SELECT delivered superior responsiveness, achieving lower latencies of 331 ms and 276 ms, whereas VIEW demonstrated superior efficiency across intensive workloads (1,000–10,000 iterations), recording lower execution times of 426 ms and 320 ms. These outcomes partially substantiate the research hypothesis: VIEW ensures superior memory efficiency and greater latency stability across prolonged iteration workloads, while standalone SELECT retains clear advantages under low-frequency execution. This operational crossover is governed by two internal database mechanisms: the progression from hard parse to soft parse and Database Buffer Cache caching dynamics. During initial runs, the database engine executes hard parse routines to validate query syntax, check object authorizations, and determine execution plans for multi-table joins. This process requires significant computational resources, particularly for VIEW statements where Oracle resolves virtual join structures into optimized execution paths. This structural resolution explains why VIEW incurred 43.6% CPU utilization at 10 iterations compared to 27.2% for SELECT (Table 3), contributing directly to VIEW's slower initial latency. Once execution paths are compiled and retained in the Library Cache, subsequent runs transition to soft parsing, eliminating repetitive optimization routines. Concurrently, the Database Buffer Cache populates relational data pages into RAM, eliminating repeated physical storage reads. This behavior is evidenced in Table 5, where SELECT disk utilization dropped from 6.40% at 10 iterations to 0.05% at 100 iterations. When both mechanisms reach steady state, VIEW's pre-compiled relational mapping enables streamlined metadata evaluation within the SGA, yielding consistently lower resource overhead at scale. The superiority of VIEW at high iteration volumes corroborates the findings of Noviyanti *et al.* (2018), who observed faster execution for VIEW queries compared to unoptimized cross products across datasets of 100–1,000 rows. This operational advantage is further contextualized by the limitations of physical caching structures. As noted by Ahmed *et al.* (2020), physically persisting precomputed queries via materialized views imposes substantial disk consumption and maintenance overhead, making frequent refresh cycles computationally prohibitive for high-velocity environments.

Consequently, memory-based logical query optimization, as highlighted by Kvet and Papan (2024), provides an effective mechanism for database acceleration. By leveraging virtual VIEW definitions alongside Oracle's touch-count buffer replacement policy (Gadupudi & Saha, 2025), the database instance sustains high cache hit ratios and memory efficiency (70.9% for VIEW vs. 72.2% for SELECT at 10,000 iterations) directly within volatile memory, avoiding the physical storage penalties detailed by Ahmed *et al.* (2020). Furthermore, the latency variations observed in this study corroborate Huang *et al.* (2025), confirming that host OS scheduling and background interrupts introduce execution jitter in millisecond-scale relational processing. These experimental insights offer concrete architectural guidelines for database administrators and software engineers deploying Oracle Database 23ai. For application modules executing ad-hoc or infrequent queries (<100 executions per session), standalone SELECT queries are recommended to minimize initial parsing and compilation overhead. Conversely, for systems characterized by high-frequency, repetitive execution—such as transaction reporting, batch ETL pipelines, and high-concurrency microservices—stored VIEW statements

provide superior long-term throughput, lowering CPU utilization (17.0% vs. 20.6% at 10,000 iterations) and maintaining tighter RAM allocation (70.9% vs. 72.2%). System administrators should anticipate initial resource spikes during the hard parse phase, specifically heightened disk I/O for SELECT and elevated CPU load for VIEW during initialization. Moreover, because millisecond-scale queries are sensitive to host-level context switching, production deployments should minimize non-essential host background processes to ensure uniform execution throughput. Several experimental constraints contextualize these findings. Testing was conducted on a synthetic dataset comprising 40,000 records in the orders table and 10,000 records in the customers table (~10.44 MB); enterprise workloads with complex relational schemas and multi-gigabyte footprints may exhibit different caching dynamics. Benchmarking was performed on a standalone local host running Windows 11, excluding distributed network transport latency and multi-tier connection pool dynamics. The query scope was restricted to a single inner join without auxiliary filtering or aggregation, which may behave differently under complex nested queries. Finally, performance evaluations were based on descriptive averages rather than formal inferential statistical tests (such as ANOVA or paired t-tests). Nevertheless, the experimental design successfully addressed the core research objective, delivering empirical evidence that validates how execution scale governs the performance divergence between SELECT and VIEW statements on Oracle Database 23ai.

5. Conclusion and Recommendations

This study demonstrates that the operational performance of SELECT queries and VIEW statements on Oracle Database 23ai is governed by the execution workload scale. At low-frequency execution tiers (10–100 iterations), standalone SELECT queries achieve superior responsiveness, recording average latencies of 331 ms and 276 ms alongside lower initial CPU allocation (27.2%). In contrast, under intensive execution scales (1,000–10,000 iterations), VIEW statements exhibit superior efficiency, attaining an average response time of 320 ms at 10,000 iterations, reduced CPU load (17.0%), and more efficient RAM allocation (70.9% versus 72.2% for SELECT). This behavioral crossover is driven by the internal progression from hard parse to soft parse routines and the steady-state caching of relational blocks within the Database Buffer Cache, which collectively allow the virtual metadata representation of VIEW statements to execute with minimal computational overhead inside the System Global Area once initialization concludes. The primary contribution of this research is delivering the first systematic empirical evaluation of SELECT and VIEW execution dynamics specifically on Oracle Database 23ai under workloads extending to 10,000 iterations. While existing literature has largely prioritized cross-engine comparisons across platforms such as MySQL, PostgreSQL, and MongoDB, this investigation deconstructs intra-engine resource consumption—integrating simultaneous telemetry of execution latency, CPU utilization, RAM residency, and physical disk I/O through an automated Java-based monitoring harness. Furthermore, the study identifies and contextualizes the empirical impact of operating system context switching and thread scheduling noise on millisecond-level database benchmarks, providing a technical baseline for performance variations that prior evaluations have typically left unaddressed within the Oracle 23ai ecosystem.

These conclusions should be interpreted within the defined constraints of the experimental setup. Benchmarks were conducted on a synthetic two-table dataset (~10.44 MB) restricted to a single inner-join relationship without auxiliary filtering predicates, group aggregations, or nested queries. In addition, measurements were gathered in a single-host local environment running Windows 11, omitting network transport overhead, multi-node clustering, and the process scheduling characteristics typical of server-grade Linux operating systems. The analysis was also grounded in descriptive performance metrics rather than formal inferential statistical tests. Future research should expand upon these findings by evaluating multi-node distributed database topologies to incorporate network transport and connection pool latencies, as well as testing against multi-gigabyte production enterprise datasets with complex schema architectures. Implementing formal inferential statistical evaluations, such as ANOVA and repeated-measures tests, would further validate the statistical significance of execution divergence across iteration tiers. In addition, broadening the query profile to encompass outer joins, subqueries, and analytical aggregation functions will offer a more comprehensive mapping of virtual relational optimization. Finally, replicating these benchmarks across enterprise Linux environments and conducting comparative evaluations across multiple Oracle release versions or alternative commercial enterprise engines will enhance the generalizability of these findings.

References

Abdi, M. F., Susanto, A., & Kusnawi, K. (2021). Perbandingan kecepatan pencarian data SQL dan NoSQL. *Jurnal Teknologi Informasi*, *5*(1), 7–11. https://doi.org/10.36294/jurti.v5i1.1696

Ahmed, R., Bello, R., Witkowski, A., & Kumar, P. (2020). Automated generation of materialized views in Oracle. *Proceedings of the VLDB Endowment*, *13*(12), 3046–3058. https://doi.org/10.14778/3415478.3415533

Andriadi, A. A., & Faqih, A. F. M. (2026). Penerapan teknik database tuning pada MySQL untuk mengoptimalkan sistem reservasi online berbasis web. *Journal of Integrated Engineering and Applied Technology (JIEAT)*, *1*(1), 19–26. https://doi.org/10.65487/jieat.v1i1.30

Aulia, C. P., Pratama, M. Y., & Dewi, H. L. (2023). Perbandingan performa query SELECT dasar, VIEW, dan stored procedure pada database MySQL. *Prosiding Seminar Nasional Teknologi Dan Sistem Informasi*, *3*(1), 456–464.

Gadupudi, P., & Saha, S. (2025). Evolution of buffer management in database systems: From classical algorithms to machine learning and disaggregated memory. *arXiv preprint*. https://doi.org/10.48550/arXiv.2512.22995

Huang, K., Zhou, J., Zhao, Z., Xie, D., & Wang, T. (2025). Low-latency transaction scheduling via userspace interrupts: Why wait or yield when you can preempt? *Proceedings of the ACM on Management of Data*, *3*(3), Article 182. https://doi.org/10.1145/3725319

Karri, N., & Muntala, P. S. R. P. (2023). Query optimization using machine learning. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(4), 109–117. https://doi.org/10.55524/ijetcsit.2023.4.4.21

Kvet, M., & Papan, J. (2024). Enhancing analytical select statements using reference aliases. *IEEE Access*, *12*, 28315–28330. https://doi.org/10.1109/ACCESS.2024.3366455

Noviyanti, P., Deolika, A., Hartinah, S., Haris, C. A., Maryana, T., & Sari, N. D. (2018). Perbandingan query response time pada model query VIEW dan cross product. *E-Jurnal JUSITI (Jurnal Sistem Informasi Dan Teknologi Informasi)*, *7*(2), 131–141. https://doi.org/10.36774/jusiti.v7i2.285

Nugraha, F. A., & Susetyo, Y. A. (2023). Analisis perbandingan performa database DuckDB dan SQLite pada pengolahan big data. *JIPI (Jurnal Ilmiah Penelitian dan Pembelajaran Informatika)*, *8*(3), 1052–1060. https://doi.org/10.29100/jipi.v8i3.3986

Oracle. (2025). *Oracle AI database: Database performance tuning guide (Release 26ai)* (Part No. G43584-01). Oracle Help Center.

Putra, A. E., & Samidi. (2025). Perbandingan kinerja teknik index bitmap dan B-tree dalam optimasi query pada database Oracle. *Jurnal Sistem dan Teknologi Informasi (JUSTIN)*, *13*(2), 245–252. https://doi.org/10.26418/justin.v13i2.81234

Putra, Y. Y., Purwaningrum, O., & Winata, R. H. (2022). Perbandingan performa respon waktu kueri MySQL, PostgreSQL, dan MongoDB. *Jurnal Sistem Informasi dan Bisnis Cerdas*, *15*(1), 39–48. https://doi.org/10.33005/sibc.v15i1.2789

Thoib, B., Candra, P., Sururi, N., & Nugraha, D. S. (2024). Perbandingan performa pencarian data berbasis teks dengan dan tanpa full-text index pada basis data MySQL. *INSOLOGI: Jurnal Sains dan Teknologi*, *3*(6), 663–673. https://doi.org/10.55123/insologi.v3i6.4385

Wagner, J., Nissan, M. I., & Rasin, A. (2023). Database memory forensics: Identifying cache patterns for log verification. *Forensic Science International: Digital Investigation*, *45*, Article 301567. https://doi.org/10.1016/j.fsidi.2023.301567

Wathani, M. R., Wijaya, E. S., Zaenuddin, Z., & Abidin, M. Z. (2025). Efektivitas indeks dalam meningkatkan performa query join di sistem basis data relasional. *Technologia: Jurnal Ilmiah*, *16*(2), 361–369. https://doi.org/10.31602/tji.v16i2.15284

Yang, S., Reichelt, D. G., Jung, R., Hansson, M., & Hasselbring, W. (2025). The Kieker observability framework version 2. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE Companion '25)* (pp. 1–5). Association for Computing Machinery. https://doi.org/10.1145/3710714.3713020