A Comparative Analysis of Resource Utilization Efficiency and I/O Performance in MySQL and PostgreSQL Running in Docker Containers

Indah Dwi Putri*[1], Maria Yuliana Erni Artika[2], Rosalia Ningsi Feu Uskenat[3], Edhy Sutanta[4], Suparyanto[5]
[1,2,3,4]Informatics Study Program, Akprind University
[5]Informatics Study Program, STMIK El Rahma
*e-mail: *[1]indah.dwi.putri@akprind.ac.id, [2]maria_yea@akprind.ac.id, [3]rosalia_nfu@akprind.ac.id, [4]edhy_sst@akprind.ac.id, [5]anto_suparyanto@stmikelrahma.ac.id*
*Correspondence author email: **

Abstract
The exponential growth of data demands highly efficient relational database management systems (RDBMS). This study evaluates resource utilization and input/output (I/O) performance differences between MySQL 8.0 and PostgreSQL 15 within a Docker container. A quantitative experiment was conducted using a constrained environment of 2 CPU cores and 2 GB RAM, utilizing a normalized sales schema with 100,000 records. Three query scenarios—a complex four-table join, an aggregation, and a single-table range filter—were each executed 30 times under cold and warm-start conditions, reporting statistical means, medians, standard deviations, and 95% confidence intervals. Memory was tracked via container-level resident memory (RSS) and engine-specific buffers (InnoDB buffer pool and PostgreSQL's shared_buffers), while I/O utilized precise before-after deltas. Under warm-start complex joins, PostgreSQL 15 averaged 0.0234 seconds compared to MySQL 8.0's 0.0442 seconds. PostgreSQL maintained a significantly lower container RSS (~54 MiB versus MySQL's ~533 MiB), though engine-specific buffer utilization remained closer (~27 MiB versus ~41 MiB). This indicates MySQL's default settings over-allocate container memory. Ultimately, PostgreSQL 15 demonstrates superior execution speed and resource efficiency for complex queries in limited-resource Docker environments under the default configurations tested here.

Keywords*—Docker Container, Database Benchmarking, MySQL, PostgreSQL, I/O Performance,*

1. INTRODUCTION

The development of information technology has increased the volume of data that must be managed by various organizations. This condition has led to an increasing need for Relational Database Management Systems (RDBMS). RDBMS is not only capable of storing very large amounts of data but also of processing and responding to requests from many users efficiently. Given its position as a core component in almost all software systems, its performance has a direct impact on the operational costs of service providers and end-user satisfaction. The selection of the right DBMS is a strategic decision, as indicated by the performance characteristics of different DBMS [1], which are influenced by workload characteristics, data volume, and configuration and software parameters [2].

MySQL and PostgreSQL are database management systems widely used in information systems development. Previous research has shown that the two DBMSs have different performance characteristics across various workload types. Several studies have reported that PostgreSQL produces lower query response times than MySQL for relational operations involving many relations, including in complex query-response-time stability testing scenarios [3]. This difference is essentially rooted in the memory management architecture of each system: MySQL, through InnoDB, allocates a large portion of memory to a dedicated buffer pool to store copies of table and index data [4]. PostgreSQL takes a different approach by relying on the operating system's page cache in addition to its internal shared buffers [5]. This difference in memory management philosophy can lead to different resource consumption patterns when the two systems are run in environments with tight memory allocation limits, such as Docker containers. Furthermore, several studies have shown that properti index design and how the

DBMS manages buffer memory are the two physical factors that most strongly determine query execution efficiency on the two systems [2].

As cloud computing and microservices architectures evolve, containerization technology has become a widely used approach in application deployment. Docker containers enable applications and supporting services to run in isolation with relatively lower resource requirements compared to conventional virtual machines. However, the presence of container layers and resource restrictions through the cgroup mechanism can impact memory management mechanisms and input/output (I/O) activities performed by the DBMS, particularly due to the emergence of additional overhead in file operation cycles at the host system kernel level [6]. This condition warrants further study, given that both DBMSs have different internal mechanisms for responding to memory limitations, as reflected in the differences in the buffer pool and shared buffer approaches mentioned earlier, so the impact on resource efficiency may be disproportionate between systems.

In this study, resource efficiency is defined as the DBMS's ability to complete workloads with optimal memory usage. Meanwhile, I/O performance is measuring by query execution time, Block I/O activity, and Network I/O during testing. This study focuses on typical deployment conditions in small to medium-scale cloud environments with resource limits of 2 CPUs and 2 GB of RAM.

Previous studies have compared MySQL and PostgreSQL in a live installation environment on the host operating system or without explicitly measuring resource constraints. Although the implementation and evaluation of RDBMS in a Docker Container environment with real-time monitoring instruments has begun to be explored [7], most of these studies tend to focus on a single metric, generally execution time or throughput, without simultaneously linking it to memory efficiency and I/O activity under the same resource ceiling conditions. Furthermore, the configurations used in these studies generally do not consider out-of-the-box conditions, namely the use of default parameters of both DBMSs without special adjustments (tuning). This condition actually represents a rapid deployment scenario commonly found in the early stages of small-to medium-scale information system development, including in startup environments and digital MSMEs. This gap is the focus of this study, coupled with the need for research that focuses on the accuracy of relational query plan calculations [8] to evaluate both DBMSs in a Docker Container environment. To compare these, the resource configuration is set to the same setting, so that a more objective comparison will be produced.

Based on these conditions, this study aims to analyze and compare the efficiency of resource usage and I/O performance between MySQL 8.0 and PostgreSQL 15 in a Docker Container environment using large sales transaction simulation data, with default (out-of-the-box) configurations on both systems to maintain equivalent test conditions and represent deployment scenarios that do not go through an advanced tuning process, evaluated across three query scenarios that differ in access pattern rather than a single complex query alone.

This research focuses on guiding application developers to avoid choosing and configuring the wrong DBMS in the container, which can degrade system performance or increase cloud infrastructure costs. Based on this background, the problem formulation in this research is (1) How does the memory usage efficiency compare between MySQL 8.0 and PostgreSQL 15 when running in a Docker Container with limited resources? (2) How does the I/O performance and execution time of complex relational queries compare between the two DBMS in the same environment? The purpose of this research is to quantitatively analyze the differences in resource usage efficiency and I/O performance between MySQL 8.0 and PostgreSQL 15 in a Docker Container configured with 2 CPU cores and 2GB of RAM, using simulated data from a normalized sales transaction scheme containing 100.000 records, with queries involving INNER JOIN operations. Testing was conducted using a quantitative experimental approach, in which each query was run 30 times per DBMS, with cold-start and warm-start conditions measured separately, to obtain statistically grounded execution time estimates including mean, median, standard deviation, and 95% confidence interval. The results of this study are expected to serve

as an empirical reference for application developers and system architects in selecting the appropriate DBMS for resource-constrained container-based infrastructure, especially during the initial deployment stage, which generally lacks specialized configuration tuning.

In this study, resource efficiency is evaluated through two container memory metrics, container-level RSS and engine-specific buffer memory, while I/O performance is evaluated based on query execution time and before-after Block I/O and Network I/O deltas obtained from the Docker Stats utility around each query execution.

2. MATERIALS AND METHODS

Testing Environment

This study uses a controlled quantitative experimental approach to compare resource efficiency and I/O performance between MySQL 8.0 and PostgreSQL 15 in a Docker Container environment. All tests were run on the same hardware (Table 1) to ensure that differences in results stem purely from the characteristics of each DBMS, rather than from differences in machine specifications.

Table 1. Host Hardware and Software Specifications

Component	*Specification*
Processor (CPU)	11th Gen Intel Core i5-11-45G7 @2.60GHz (8 CPUs)
Memory (RAM)	16 GB DDR 4
Storage Media	NVMe SSD 512GB
Host Operating System	Windows 11
Platform Containerisasi	Docker Desktop v28.4.0

To ensure fair testing, resource limits are enforced via the resource limits feature in Docker Compose, allowing each container to receive at most 2 CPUs and 2 GB of RAM. These limits were chosen to represent the conditions of a small to medium-sized cloud deployment and to serve as critical points for testing the memory management of each DBMS, MySQL, through its explicitly allocated InnoDB buffer pool within these limits, and PostgreSQL, which relies on the host operating system's page cache. Because the Linux page cache counts toward a container's cgroup memory limit [9], these two approaches have the potential to produce different memory usage and reclamation patterns when allocations are tightly constrained. This condition is the main focus of this study. In addition to resource constraints, both containers use the same Docker Bridge Network and Docker Volume to ensure consistent network communication and data storage mechanisms across both DBMSs, and both engines were kept at default (out-of-the-box) configuration throughout the study, without tuning of parameters such as innodb_buffer_pool_size, shared_buffers, or planner cost settings. The implications of this choice for how the results should be interpreted are discussed in the Research Limitations section.

Simulation Data and Seeding Process

Simulation data was generated using Faker to resemble the characteristics of real transaction data without involving sensitive user data. The number of 100,000 records was chosen to represent the transaction scale of a medium system, large enough to reveal differences in query optimizer behavior, cache management, and I/O activity between DBMSs, but still within the resource allocation limits of the container. The database schema was designed to be normalized with four interrelated relations, namely product, customer, transaction, and transaction detail. The number of records in each of these relations is 100, 1,000, 100,000, and >100,000, respectively. Each transaction has a varying number of items so that the data distribution is more realistic.

The same data is then fed into MySQL and PostgreSQL via a Python-based automated seeder script, which generates data for each relation sequentially and matches the relations to

ensure both DBMSs receive identical data structure and content before the benchmarking process is run.

The complete data definition language (DDL) used to create the four relations in both engines is shown in Table 2. All primary key and column type definitions are identical across MySQL and PostgreSQL to keep the schema equivalent between the two systems.

Table 2. Table Definitions (DDL)

Aspect	*Definition*
tbl_produk	id_produk INT PRIMARY KEY, nama_produk VARCHAR(255), harga INT, deskripsi TEXT
tbl_pelanggan	id_pelanggan INT PRIMARY KEY, nama VARCHAR(255), email VARCHAR(255), wilayah VARCHAR(255)
tbl_transaksi	id_transaksi INT PRIMARY KEY, id_pelanggan INT, tanggal_transaksi DATE, total_bayar INT
tbl_detail_transaksi	id_detail INT PRIMARY KEY, id_transaksi INT, id_produk INT, jumlah INT, subtotal INT

Four secondary indexes were added to both engines before benchmarking, listed in Table 3. These indexes were not present in the earlier version of this study and were added specifically to support the WHERE, JOIN, and ORDER BY clauses used in the benchmark queries described in section 2.4.

Table 3. Secondary Indexes Applied Before Benchmarking

Index name	*Table*	*Column*	*Purpose*
idx_transaksi_pelanggan	tbl_transaksi	id_pelanggan	join to tbl_pelanggan
idx_transaksi_tanggal	tbl_transaksi	tanggal_transaksi	ORDER BY / range filter
idx_detail_transaksi	tbl_detail_transaksi	id_transaksi	join to tbl_transaksi
idx_detail_produk	tbl_detail_transaksi	id_produk	join / group by tbl_produk

Evaluation Parameters and Measurement Tools

This study evaluates three aspects, resource usage efficiency, query performance, and I/O activity (Table 4).

Table 4. Research Evaluation Parameters

Aspect	*Parameter*	*Definition*
Resource Efficiency	Container RSS	Container memory consumption reported by docker stats
	Engine buffer bytes	InnoDB buffer pool data (MySQL) or shared_buffers content for the active database (PostgreSQL, via pg_buffercache
Query Performance	Execution time	Query execution time, measured over 30 repetitions per condition
I/O Performance	Block I/O	Disk read/write activity, measured as the before-after delta around each query
	Network I/O	Network data sending/receiving activity, measured as the before-after delta around each query

Data was collected using Docker stats, a built-in Docker feature for monitoring container health in real time without the need for additional monitoring tools. Two memory figures are reported instead of one. The first is container RSS, the resident memory figure reported directly by docker stats. The second is engine-specific buffer memory, obtained from each DBMS rather than from Docker. For MySQL this is Innodb_buffer_pool_bytes_data read through SHOW STATUS, and for PostgreSQL this is the byte count of the active database's pages currently held in shared_buffers, read through the pg_buffercache extension (enabled once before testing). If

pg_buffercache cannot be enabled, the script falls back to a database-level cache-hit ratio from pg_stat_database, flagged separately so it is never mixed with the byte-based figures. Reporting both memory figures side by side makes it possible to see whether a DBMS's container-level footprint is actually backed by data held in its own buffer, or reserved ahead of use. Block I/O and Network I/O are captured as a snapshot immediately before and immediately after each query execution, and the delta between the two snapshots is what is reported per iteration, rather than the cumulative counters used in the earlier version of this study.

Testing Scenarios and Procedures

Testing in the earlier version of this study used a single complex relational query. Following review feedback, this was expanded to three query scenarios that represent different access patterns commonly found in sales transaction systems, a complex join query with a range filter, ORDER BY, and LIMIT (Q1), an aggregation query summarizing sales per product (Q2), and a single-table range filter with ORDER BY and LIMIT (Q3). The three queries are shown in Figures 1 to 3.

The testing procedure was carried out in the following stages. Simulation data was generated and seeded into both MySQL and PostgreSQL. Table and index DDL were applied once, together with activation of the pg_buffercache extension on PostgreSQL. Cold-start testing was then performed by recreating both containers from a clean state (docker compose down -v followed by up -d) so no cache carried over from a previous run, then executing each of the three queries 30 times per DBMS, taking a docker stats snapshot immediately before and after each execution. Warm-start testing followed immediately afterward in the same container session, without any container restart in between, one untimed warm-up execution was run per query per DBMS before the 30 timed iterations, so that engine and OS caches were populated before measurement began. Container creation timestamps were recorded before cold-start and after warm-start to confirm that no restart occurred between the two conditions. EXPLAIN ANALYZE execution plans were captured for all three queries on both engines and are provided as supplementary material.

```
SELECT
    t.id_transaksi,
    p.nama,
    pr.nama_produk,
    d.jumlah,
    d.subtotal,
    t.tanggal_transaksi
FROM tbl_detail_transaksi d
INNER JOIN tbl_transaksi t
    ON d.id_transaksi = t.id_transaksi
INNER JOIN tbl_pelanggan p
    ON t.id_pelanggan = p.id_pelanggan
INNER JOIN tbl_produk pr
    ON d.id_produk = pr.id_produk
WHERE t.total_bayar > 100000
ORDER BY t.tanggal_transaksi DESC
LIMIT 5000;
```

Figure 1. Benchmark Query Q1 – Complex Join

```sql
SELECT
    pr.nama_produk,
    COUNT(*) AS jumlah_terjual,
    SUM(d.subtotal) AS total_omzet
FROM tbl_detail_transaksi d
INNER JOIN tbl_produk pr
    ON d.id_produk = pr.id_produk
GROUP BY pr.nama_produk
ORDER BY total_omzet DESC;
```

Figure 2. Benchmark Query Q2 – Aggregation

```sql
SELECT
    id_transaksi,
    id_pelanggan,
    tanggal_transaksi,
    total_bayar
FROM tbl_transaksi
WHERE total_bayar BETWEEN 500000 AND 1000000
ORDER BY tanggal_transaksi DESC
LIMIT 2000;
```

Figure 3. Benchmark Query Q3 – Single-table filter

3. RESULT AND DISCUSSION

Query Execution Time

Execution time was measured across three query types under both cold-start and warm-start conditions, with 30 repetitions per DBMS per condition. Table 5 summarizes the mean, median, standard deviation, and 95% confidence interval for each combination. Across all three queries and both start conditions, PostgreSQL 15 recorded a lower mean execution time than MySQL 8.0. On the complex join query (Q1), the warm-start mean was 0.0234 seconds for PostgreSQL against 0.0442 seconds for MySQL. The gap was largest on the aggregation query (Q2), where PostgreSQL averaged 0.0353 seconds against 0.3508 seconds for MySQL under warm-start, and smallest on the single-table filter (Q3), at 0.0079 seconds against 0.0205 seconds.

Compared to the previous version of this study, absolute execution times are considerably lower for both engines across the board, which follows from the addition of the four secondary indexes described in Table 3, the earlier benchmark ran against unindexed tables. The relative ordering between the two engines, however, PostgreSQL remains faster than MySQL on every query and every start condition tested.

Memory Usage

Memory usage is reported through the two metrics described in section 2.3, container RSS and engine-specific buffer bytes. Table 6 presents both figures for all three queries under cold-start and warm-start conditions. Under warm-start, PostgreSQL's container RSS averaged

around 54 MiB across the three queries, against roughly 533 MiB for MySQL, a gap of about 9.9 times. The engine-specific buffer figures tell a different story, MySQL's InnoDB buffer pool data averaged around 41 MiB, and PostgreSQL's shared_buffers content averaged around 27 MiB, a gap of roughly 1.5 times, far narrower than the RSS comparison suggests.

Table 5. Execution Time Statistics (n = 30 per conditions)

DBMS	*Query*	*Mode*	*Mean (s)*	*Median (s)*	*Standard Deviation*	*95% Confidence Interval*
MySQL 8.0	Q1-Complex Join	Cold	0.0461	0.0424	0.0077	[0.0434, 0.0489]
		Warm	0.0442	0.0429	0.0044	[0.0426, 0.0457]
PostgreSQL 15	Q1-Complex Join	Cold	0.0291	0.0217	0.0149	[0.0238, 0.0345]
		Warm	0.0234	0.0215	0.0059	[0.0213, 0.0255]
MySQL 8.0	Q2-Aggregation	Cold	0.3724	0.3473	0.0470	[0.3556, 0.3892]
		Warm	0.3508	0.3434	0.0188	[0.3441, 0.3575]
PostgreSQL 15	Q2-Aggregation	Cold	0.0382	0.0321	0.0152	[0.0327, 0.0436]
		Warm	0.0353	0.0304	0.0079	[0.0325, 0.0381]
MySQL 8.0	Q3-Single-table Filter	Cold	0.0385	0.0302	0.0209	[0.0310, 0.0460]
		Warm	0.0205	0.0183	0.0042	[0.0190, 0.0221]
PostgreSQL 15	Q3-Single-table Filter	Cold	0.0113	0.0094	0.0055	[0.0093, 0.0133]
		Warm	0.0079	0.0078	0.0006	[0.0077, 0.0081]

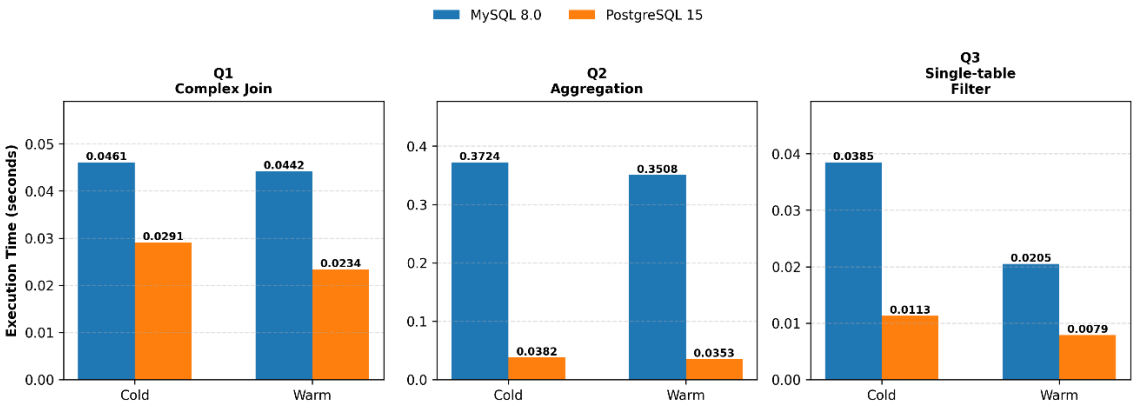

Figure 4. Mean Execution Time by Query, Start Condition, and DBMS

Table 6. Memory Usage Container RSS vs. Engine-Specific Buffer Bytes

DBMS	*Query*	*Mode*	*Container RSS (MiB)*	*Engine Buffer Bytes (MiB)*
MySQL 8.0	Q1-Complex Join	Cold	536.50	39.31
		Warm	533.04	40.95
PostrgreSQL 15	Q1-Complex Join	Cold	70.64	26.75
		Warm	53.84	26.85
MySQL 8.0	Q2-Aggregation	Cold	560.92	40.95
		Warm	532.86	40.98
PostgreSQL 15	Q2-Aggregation	Cold	70.81	26.78
		Warm	55.04	26.85
MySQL 8.0	Q3-Single-table Filter	Cold	562.26	40.95
		Warm	533.02	40.98
PostgreSQL 15	Q3-Single-table Filter	Cold	70.34	26.84
		Warm	54.48	26.85

I/O Activity

Block I/O and Network I/O were measured as the delta between a docker stats snapshot taken immediately before and immediately after each query execution, rather than as cumulative

counters. Table 7 presents the mean delta for both metrics across all three queries and both start conditions.

For the aggregation query (Q2), which reads the full detail-transaction table, Block I/O deltas were effectively zero for both engines under warm-start, consistent with the relevant pages already residing in cache after the warm-up run. Block I/O was more visible on Q1 for PostgreSQL, averaging 233,333 bytes under cold-start and 140,000 bytes under warm-start, higher than MySQL's near-zero figures on the same query, this is plausibly related to PostgreSQL fetching index and heap pages for the join that were not yet resident, rather than indicating less efficient storage access overall, since PostgreSQL's execution time on Q1 remained lower than MySQL's despite the higher Block I/O reading. Network I/O deltas were broadly similar in magnitude between the two engines across all three queries, and slightly higher for PostgreSQL on Q1 and Q3.

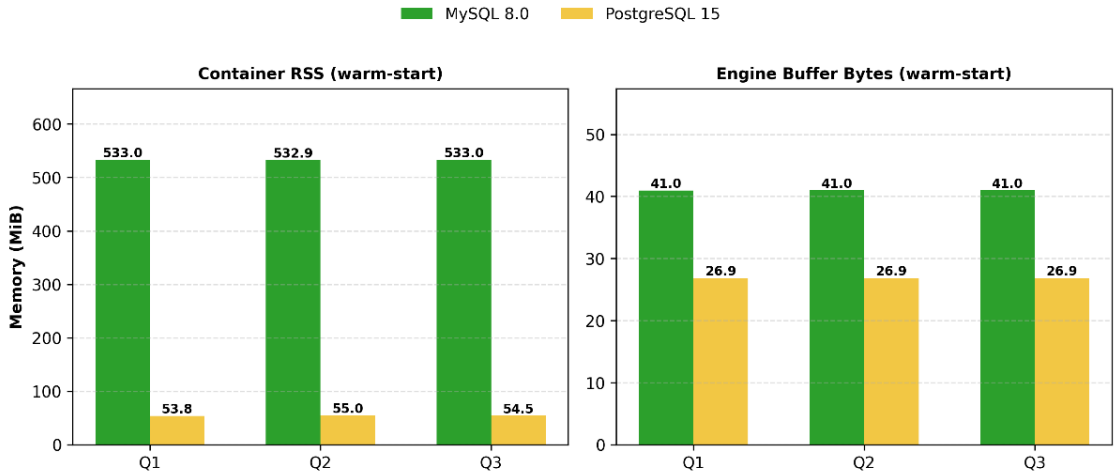

Figure 5. Warm-start Memory Usage: Container RSS vs. Engine-Specific Buffer Bytes

Table 7. Block I/O and Network I/O Deltas (mean, bytes)

DBMS	*Query*	*Mode*	*Block I/O (bytes)*	*Network I/O (bytes)*
MySQL 8.0	Q1-Complex Join	Cold	3,333	312,197
		Warm	0	322,333
PostgreSQL 15	Q1-Complex Join	Cold	233,333	426,667
		Warm	140,000	426,667
MySQL 8.0	Q2-Aggregation	Cold	20,000	3,667
		Warm	0	0
PostgreSQL 15	Q2-Aggregation	Cold	0	6,667
		Warm	0	0
MySQL 8.0	Q3-Single-table Filter	Cold	0	64,667
		Warm	0	67,333
PostgreSQL 15	Q3-Single-table Filter	Cold	0	93,333
		Warm	0	96,667

Query Execution Time Performance Analysis

The results in Table 5 show that PostgreSQL 15 consistently outperforms MySQL 8.0 in query execution time across all three query types and both start conditions tested. This confirms that in a Docker Container environment with equivalent resource allocation, PostgreSQL is able to process benchmark queries much more efficiently than MySQL, especially in queries involving many JOIN operations, where the accuracy of the cardinality estimation in the query optimizer is crucial for the efficiency of the resulting execution plan [8].

This finding is in line with the results of Salunke and Ouda [3], who proved that PostgreSQL consistently outperforms MySQL on SELECT operations with large data volumes, even reaching 9 to 13 times faster on a test of one million records. A similar pattern was also reported by Taipalus [2] in his systematic literature review, as well as by Kithulwatta et al. [10] who found similar performance advantages when both DBMSs were run on Docker. This agreement indicates that PostgreSQL's superiority is not a coincidental phenomenon in a single test scenario, but rather a pattern that tends to repeat itself across various DBMS benchmarking environments, and, with the addition of two further query types in this revision, appears consistent across join-heavy, aggregation, and simple filter workloads rather than being specific to complex joins alone.

The difference in execution time between cold-start and warm-start conditions is present but modest for both DBMSs across all three queries, most visible on Q3 for MySQL (0.0385 seconds cold against 0.0205 seconds warm) and comparatively small elsewhere. This suggests that caching contributes to the results but is not the dominant factor behind the gap between the two engines, the observed performance differences appear to stem more from the internal characteristics of each DBMS in processing relational queries, including index usage and query planning, than from cache state alone.

Memory Usage Efficiency Analysis

Memory efficiency shows two different pictures depending on which metric is examined, which is the reason both are reported in Table 6. On container RSS, PostgreSQL uses markedly less memory than MySQL, roughly 54 MiB against 533 MiB under warm-start. On engine-specific buffer bytes, the gap narrows substantially, roughly 27 MiB against 41 MiB. Read together, these figures indicate that MySQL's default configuration reserves a large amount of container memory for the InnoDB buffer pool independent of how much data it is actually caching for this workload, while PostgreSQL's RSS tracks its shared_buffers content more closely.

This is consistent with the different memory management philosophies of each system. InnoDB in MySQL by default allocates a buffer pool and keeps it resident in memory during the process [4]. PostgreSQL, with its much smaller shared_buffers setting, is designed to rely more heavily on the host operating system's page cache in addition to its own shared memory [2]. Because the page cache in Linux is included in the cgroup memory limit of a container [9], part of PostgreSQL's effective caching may not be visible in either the RSS or the shared_buffers figures reported here, which is a limitation of container-level measurement rather than evidence that PostgreSQL uses less memory overall than the numbers suggest.

This finding is also in line with the explanation that containerization is inherently lighter than conventional virtual machines because it utilizes the host kernel sharing mechanism [11]. This advantage is clearly reflected in the memory efficiency of PostgreSQL in this study, especially since other studies also show that the configuration of the integrated cloud and Docker environment also influences the overall resource utilization pattern [12].

It should still be emphasized that this study does not examine the internal memory management mechanisms of each DBMS beyond the two figures reported here. Both figures are container or session-level measurements taken through docker stats and DBMS system views respectively, not a trace of internal memory allocation behavior, and neither configuration was tuned away from its default. These results are best understood as a description of memory behavior under default configuration in this specific container setup, not a general claim about how MySQL and PostgreSQL manage memory internally under all conditions.

Block I/O Activity Analysis

Block I/O activity, now measured as a before-after delta rather than a cumulative value, shows a more query-dependent pattern than reported in the earlier version of this study. On the aggregation query (Q2), which scans the full detail-transaction table, Block I/O deltas were near zero for both engines under warm-start, since the relevant pages were already cached after the

warm-up execution. On the complex join query (Q1), PostgreSQL recorded a higher Block I/O delta than MySQL under both start conditions.

This pattern differs from the earlier finding that MySQL consistently generated higher Block I/O overall, and illustrates why measuring I/O as a delta around a specific query matters. A cumulative counter conflates all container activity since startup with the footprint of the query being tested, while a before-after delta isolates the query itself. The characteristics of the Docker image and the copy-on-write mechanism can still play a role in shaping access patterns [13], and index design together with the query optimizer's chosen execution path remain more significant performance-determining factors than hardware aspects alone [2], [8]. Because these values are derived from docker stats, they still represent activity at the container level and should be read as indicators of data access characteristics for this specific benchmark, not as an absolute measure of underlying storage media performance.

Network I/O Activity Analysis

Network I/O deltas were broadly comparable between the two engines across the three queries, with PostgreSQL somewhat higher than MySQL on Q1 and Q3 and both engines near zero on Q2 under warm-start. This finding aligns with Rehmann and Folkerts [14], who cited network transfer activity as a key determinant of data exchange in containers, as well as the impact of access patterns on storage container performance [13].

The Network I/O activity observed for PostgreSQL did not translate into a slower response, and PostgreSQL still showed a better execution time than MySQL on every query tested. This confirms that the magnitude of data transfer activity does not always directly correlate with decreased performance. In this study, query processing efficiency and index usage appear to be the more determining factors, not the volume of network communication by itself.

Relationship between Resource Efficiency and I/O Performance

Taken together, the three query scenarios point to a consistent pattern. PostgreSQL's advantage in execution time is not purchased through heavier memory or I/O usage. On engine-specific buffer bytes, its footprint is only somewhat smaller than MySQL's rather than dramatically so, and on Block I/O for the join query it is in fact higher, yet its execution time remains lower across the board. This suggests that the performance gap observed here is driven more by query planning and execution-path efficiency than by raw resource consumption.

This finding aligns with Rehmann and Folkerts [14] and [10], who both demonstrated that databases running on Docker Containers can achieve faster response times and better resource utilization than those on virtual machines.

Another factor contributing to these results is the use of datasets designed following the principles of normalization up to the third normal form (3NF), so that relationships between relations can be represented consistently and free from update anomalies and redundancy. This study [15] even found that higher levels of normalization, such as 3NF, can improve the success of query completion within a certain time limit despite increasing the number of relations in the database. This pattern aligns with the conditions in this study, where the normalized sales transaction schema allows the optimizer to work more optimally in processing complex relational queries. Thus, the PostgreSQL efficiency advantages observed in this study are also supported by the application of good database design principles from the schema design stage, making this benchmark process more representative of real transaction database structures.

Research Limitations

Despite the finding that PostgreSQL exhibits lower execution times and lower memory consumption than MySQL, these results must be interpreted within the scope of the study: a single hardware configuration, a single container configuration with 2 CPUs and 2 GB of RAM, a single dataset size of 100,000 records, and three benchmark queries covering join, aggregation, and single-table filter patterns. The data used is also simulated data from Faker, which, while meeting

the testing requirements in terms of structure and values, still differs in characteristics from actual production data. The resource measurements in this study also rely entirely on Docker stats, which only represent the overall condition of the container, not a direct measurement of the internal mechanisms of each DBMS.

A further limitation follows directly from the testing setup, both engines were run under default, out-of-the-box configuration throughout, with no tuning of parameters such as innodb_buffer_pool_size, shared_buffers, work_mem, or planner cost settings. The memory gap observed on container RSS in particular could narrow considerably under a tuned configuration, for example by right-sizing innodb_buffer_pool_size to the container's 2 GB limit instead of relying on MySQL's default sizing behavior. The conclusions drawn here therefore describe the behavior of both engines under default configuration in this specific environment, and should not be read as a general claim that PostgreSQL will always outperform MySQL, or use less memory than MySQL, once tuning is applied.

Considering these limitations, the results of this study cannot be used to conclude that PostgreSQL will always be superior to MySQL under all conditions. Future research is expected to expand the scope of testing by varying dataset sizes, tuned configurations, and hardware configurations to obtain a more comprehensive picture of the performance of both DBMSs across various deployment scenarios.

4. CONCLUSION

This study analyzed the resource utilization efficiency and I/O performance of MySQL 8.0 and PostgreSQL 15 in a Docker Container environment with a limited resource configuration of 2 CPUs and 2 GB RAM, across three query scenarios and 30 repetitions per condition with cold-start and warm-start measured separately. The results show that PostgreSQL 15 recorded lower mean execution time than MySQL 8.0 on every query tested and under both start conditions, with the gap ranging from around 1.9 times on the single-table filter query to around 10 times on the aggregation query under warm-start.

In terms of resource efficiency, PostgreSQL used substantially less container RSS than MySQL, around 54 MiB against 533 MiB under warm-start, though the gap in engine-specific buffer memory, InnoDB buffer pool data against PostgreSQL's shared_buffers content, was considerably narrower, around 27 MiB against 41 MiB. This distinction indicates that MySQL's larger container footprint under default configuration reflects buffer pool reservation more than a proportionally larger working set. Block I/O deltas were query-dependent rather than uniformly favoring one engine, with PostgreSQL recording higher Block I/O on the join query despite its faster execution time, and Network I/O deltas were broadly comparable between engines.

This research demonstrates that resource efficiency does not necessarily require intensive memory and I/O activity, and that the advantage is partly explained by the difference in memory management philosophy between InnoDB's buffer pool and PostgreSQL's shared_buffers approach, which relies more heavily on the operating system's page cache.

Within the scope of default configuration, the specific hardware, and the 100,000-record dataset used here, PostgreSQL 15 showed better execution time performance than MySQL 8.0 across all three query types tested. Application developers and system architects choosing a DBMS for container-based infrastructure, particularly in small to medium-sized cloud environments and during early deployment stages before tuning is applied, may find this pattern relevant to their planning, while keeping in mind that tuned configurations could narrow or otherwise change the gaps reported here.

ACKNOWLEDGEMENT

This publication is funded by the Directorate General of Research and Development, Ministry of Higher Education, Science, and Technology through the Fundamental Research

scheme for the 2025 Fiscal Year with contract number SP DIPA-139.04.1.693320/2026, revision 01 dated December 24, 2025.

REFERENCES

[1] F. Wiedner, M. Helm, A. Daichendt, J. Andre, and G. Carle, "Performance evaluation of containers for low-latency packet processing in virtualized network environments,"Performance Evaluation, vol. 166, art. no. 102442, 2024, https://doi.org/10.1016/j.peva.2024.102442

[2] T. Taipalus, "Database management system performance comparisons: A systematic literature review," Journal of System and Software, vol. 208, art. no. 111872, 2024, https://doi.org/10.1016/j.jss.2023.111872

[3] S. V. Salunke and A. Ouda, "A performance benchmark for the PostgreSQL and MySQL databases," Future Internet, vol. 16, no. 10, art. no. 382, pp. 1-22, 2024, https://doi.org/10.3390/fi16100382

[4] Oracle Corporation, "17.5.1 Buffer Pool," MySQL 8.0 Reference Manual. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html. [Accessed: Jul. 8, 2026].

[5] The PostgreSQL Global Development Group, "19.4. Resource Consumption," PostgreSQL 15 Documentation. [Online]. Available: https://www.postgresql.org/docs/15/runtime-config-resource.html. [Accessed: Jul. 8, 2026].

[6] K. Lee, J. Kim, I.-H. Kwon, H. Park, and C.-H. Hong, "Impact of secure container runtimes on file I/O performance in edge computing," Applied Sciences, vol. 13, no. 24, art. no. 13329, 2023, https://doi.org/10.3390/app132413329

[7] T. E. Ali, F. I. Ali, S. Hekmat, F. Eyvazov, P. Dakić, and A. D. Zoltan, "A comparative evaluation of PostgreSQL, MongoDB, and MySQL for Ethiopian migrant data management," in Proc. SQAMIA 2025: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Maribor, Slovenia, Sep. 10-12, 2025, CEUR-WS.org, vol. 4077.

[8] J. Ba and M. Rigger, "CERT: Finding performance issues in database systems through the lens of cardinality estimation," in Proc. 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), Lisbon, Portugal, Apr. 14-20, 2024, https://doi.org/10.1145/3597503.3639076

[9] T. Zussman, I. Zarkadas, J. Carin, A. Cheng, H. Franke, J. Pfefferle, and A. Cidon, "Cache is King: Smart page eviction with eBPF," arXiv preprint arXiv:2502.02750, 2025. [Online]. Available: https://arxiv.org/abs/2502.02750

[10] W. M. C. J. T. Kithulwatta, K. P. N. Jayasena, B. T. G. S. Kumara, and R. M. K. T. Rathnayaka, "Performance analysis of docker-based database management systems compared to virtual machine-based systems: A Comparative Study," International Journal of Advanced Computer Science and Applications, vol. 14, no. 11, pp. 22-31, 2023, https://doi.org/10.64701/ijrc/345/6635

[11] L. Baresi, G. Quattrocchi, and N. Rasi, "A qualitative and quantitative analysis of container engines," Future Generation Computer Systems, vol. 136, pp. 1-16, 2022, https://doi.org/10.1016/j.jss.2024.111965

[12] S. Kiran, V. C. S. Rao, S. Venkatramulu, M. S. B. Phridviraj, S. Pratapagiri, and S. Madugula, "Database patterns for the cloud and docker integrated environment using open-source machine learning," in Proceedings of the International Conference on Electronics and Renewable Systems (ICEARS 2022), IEEE, pp. 1909-1911, 2022, https://doi.org/10.1109/ICEARS53579.2022.9751894

[13] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. K. Paul, K. Chen, and A. R. Butt, "Large-scale analysis of docker images and performance implications for container storage systems," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 4, pp. 918-932, 2021, https://doi.org/10.1109/TPDS.2020.3034517

[14] K. T. Rehmann and E. Folkerts, "Performance of containerized database management systems," in Proceedings of the ACM International Workshop on Testing Database Systems (DBTest '18), Houston, TX, USA, pp. 1-6, 2018, https://doi.org/10.1145/3209950.3209953

[15] M. Fotache, M. I. Cluci, T. Taipalus, and G. Talaba, "The effects of database normalization on decision support system performance," Information Systems, vol. 136, art. no. 102636, 2024, https://doi.org/10.1016/j.is.2025.102636