Enhancing query performance in graph databases through physical design

Mejora del rendimiento de las consultas en bases de datos orientadas a grafos mediante el diseño físico

Ana Aguilera[1, 2*] https://orcid.org/0000-0003-0726-1759
Marlene Goncalves[3, 4] https://orcid.org/0000-0003-4843-0774
Francisco Castillo[1] https://orcid.org/0009-0002-3843-2928

Recibido 22 de octubre de 2025, aceptado 3 de diciembre de 2025
Received: October 22, 2025 *Accepted: December 3, 2025*

ABSTRACT

Graph-based databases have become increasingly valuable for modeling relationships in complex systems and social networks. Nonetheless, critical aspects of these systems-particularly concerning query optimization and data storage-have yet to be thoroughly investigated. It is well known that physical design is a critical aspect of a database, as it impacts the performance and functionality of the supported applications. The Linked Data Benchmark Council hosts the Social Network Benchmark project, which seeks to promote the development of novel physical design strategies through a challenge comprising two groups of queries: an Interactive Workload group and a Business Intelligence Workload group. This paper presents proposed and validated physical design strategies that optimize data search in graph-oriented databases. The proposed strategies are Query Rewriting, Path Materialization, and Index Creation, which were implemented for the 25 queries of the SNB Business Intelligence Workload group. The effectiveness and scalability of the proposed physical design strategies were validated through experimentation. The Neo4j engine was used to assess the strategies, with 1, 10, and 100 GB workloads. The results show, on average, reductions of 59% for the first design strategy, 77% for the second, and 45% for the third in execution time.

Keywords: Databases, physical design, query processing, Benchmarks.

RESUMEN

Las bases de datos basadas en grafos se han vuelto cada vez más valiosas para modelar relaciones en sistemas complejos y redes sociales. Sin embargo, algunos aspectos críticos de estos sistemas, en particular los relacionados con la optimización de consultas y el almacenamiento de datos, aún no se han investigado a fondo. Es bien sabido que el diseño físico es un aspecto crítico de una base de datos, ya que afecta al rendimiento y la funcionalidad de las aplicaciones que soporta. El Linked Data Benchmark Council acoge el proyecto Social Network Benchmark, que busca promover el desarrollo de nuevas estrategias de diseño físico a través de un desafío que comprende dos grupos de consultas: un

[1] Universidad de Valparaíso. Escuela de Ingeniería Informática. Facultad de Ingeniería. Valparaíso, Chile. E-mail: ana.aguilera@uv.cl; francisco.castillo@alumnos.uv.cl
[2] Universidad de Valparaíso. Centro Interdisciplinario MEDING. E-mail: ana.aguilera@uv.cl
[3] Universidad Internacional de Valencia. Valencia, España. E-mail: marlene.goncalves@universidadviu.com
[4] Universidad Simón Bolívar. Departamento de Computación y Tecnología de la Información. Caracas, Venezuela. E-mail: mgoncalves@usb.ve
* Autor de correspondencia: ana.aguilera@uv.cl

DOI: 10.64966/ingeniare.v33.23.

grupo de carga de trabajo interactiva y un grupo de carga de trabajo de inteligencia empresarial. Este artículo presenta estrategias de diseño físico propuestas y validadas que optimizan la búsqueda de datos en bases de datos orientadas a grafos. Las estrategias propuestas son la reescritura de consultas, la materialización de rutas y la creación de índices, que se implementaron para las 25 consultas del grupo de carga de trabajo inteligencia empresarial del SNB. La eficacia y la escalabilidad de las estrategias de diseño físico propuestas se validaron mediante experimentos. Se utilizó el motor Neo4j para evaluar las estrategias, con cargas de trabajo de 1, 10 y 100 GB. Los resultados muestran una reducción del tiempo de ejecución, en promedio, del 59% para la primera estrategia de diseño, del 77% para la segunda y del 45% para la tercera.

Palabras clave: Bases de datos, diseño físico, procesamiento de consultas, Benchmarks.

INTRODUCTION

Currently, there is a special interest in graph databases due to their ability to model and query complex networks. Graph databases replace relational tables with structured relational graphs of interconnected key-value pairs [1]. Applications include representation and traversal of social networks, generating recommendations (e.g., upsell or cross-sell suggestions), conducting forensic investigations (e.g., pattern detection), geospatial and logistics applications, and simulation of contagion networks (e.g., covid-19 and others [2]. These databases are better suited to connected data, for which mature graph algorithms can be used directly for analysis and computations [1]. The increased focus on graph databases is driven by two forces: the massive commercial success of companies such as Google, Facebook, and Twitter, all of whom have centered their business models on their proprietary graph technologies; and the introduction of general-purpose graph databases into the technology landscape [3]. In general, graph databases are useful when the focus is on relationships rather than the data itself [2]. In these cases, it is important to optimize access to these relationships for both traversing and querying data to achieve good performance.

Query optimization is a critical component of NoSQL graph databases, as it directly affects query performance and efficiency. This optimization is a complex, multifaceted problem that requires careful consideration of various factors such as data structure, query nature, and the underlying execution engine. Cost-based query optimization is widely used in relational databases, but its application in graph databases poses unique challenges. The vast plan space and the difficulty of gathering accurate statistics make traditional cost-based approaches less effective [4]. Relational databases do not efficiently handle massive volumes of data, which has driven the adoption of NoSQL databases in the industry. These databases are developed to process large amounts of data quickly. However, the methods used to design physical structures in relational databases cannot necessarily be transferred to the NoSQL environment [5]. The physical design –defined as the set of techniques and mechanisms employed to ensure efficient storage and query performance– has been extensively studied in the context of relational databases. The physical design stage in relational databases involves creating indexes, partitioning data, and selectively materializing data, focusing on how data is stored and accessed on disk to improve query execution time [6]. Therefore, developers must understand the specific details of each system to achieve optimal results when executing their queries. The growing volume of data and the need for efficiency drive the need for physical database design, as significant performance improvements can be critical for applications ranging from everyday tasks to high-impact business decisions [5].

Traditional database systems achieve high performance through tabular data structures, normalized schemas, and cost-based query optimizers. In contrast, emerging technologies such as NoSQL databases –and graph databases in particular– present new challenges and opportunities for physical design. This domain remains comparatively underexplored despite its critical importance [7]. Graph databases like Neo4j represent information as nodes, relationships, and properties, offering a natural, semantically rich model for complex, highly interconnected domains. However, this data

model introduces non-trivial challenges related to data layout, access patterns, and execution planning [8]. In this context, physical design plays a pivotal role in query optimization by determining how data is physically stored, indexed, and accessed at runtime. As such, adopting well-founded, empirically validated physical design strategies is essential to ensuring optimal performance in production environments that rely on graph database management systems. Motivated by these considerations, this work proposes a set of physical design guidelines tailored to graph-oriented database management systems. A benchmarking challenge defined by the Linked Data Benchmark Council's Social Network Benchmark (LDBC SNB) [10] was used to validate the effectiveness of the proposed strategies, with Neo4j as the underlying graph database platform.

RELATED WORK

This section explores key strategies and techniques for optimizing queries in graph databases, drawing insights from recent research. Table 1 summarizes the reviewed works.

Jota, Goncalves, and Parra [11], propose physical design guidelines for Neo4j graph databases, focusing on indexes, path materialization, and query rewriting to enhance query execution time. Empirical studies showed significant improvements in query performance and database hit efficiency using these guidelines. An empirical study is conducted on 13 queries from LDBC SNB.

De Virgilio, Maccioni, and Torlone [12], discuss a model-driven methodology for designing graph databases, focusing on translating a conceptual Entity-Relationship model into an intermediate representation that minimizes data access operations, ultimately leading to an efficient physical design suitable for various GDBMSs. Methodology improves query performance over naive approaches.

Yang, Pang, and Zou [4], propose a Graph Cost-Based Optimizer (gCBO) that employs a hybrid plan enumerator based on dynamic programming. This enumerator generates potential execution plans while considering the structural properties of graph queries. Additionally, gCBO uses cost models that capture the characteristics of different joins types, such as node joins and edge joins, to estimate execution costs more accurately. The gCBO incorporates a sampling-based cardinality estimation strategy to collect the necessary statistics

Table 1. Comparison of query optimization approaches in graph databases.

Reference	Approach	Key Features
Yang *et al.* [4].	gCBO.	Cost-based optimizer with dynamic programming and sampling-based cardinality estimation.
Jachiet *et al.* [13].	mu-RA.	Relational algebra with fixpoint operator for recursive Queries.
Sharma *et al.* [14].	Schema-Based Optimization.	Type inference mechanism for enriching recursive graph Queries.
Baumstark *et al.* [15].	Adaptive Compilation.	Integration of interpretation and compilation for efficient query execution.
Dörpinghaus and Stefan[16].	Large-Scale Optimization.	Classification-based optimization approaches for large knowledge graphs.
Kotiranta *et al.* [17], Khan *et al.* [18], Asplund and Sandell [19], and Alyas *et al.* [20].	Performance Comparison.	Comparison of graph and relational databases for complex queries.
Jota *et al.* [11].	Physical Design Guidelines.	Proposes physical design guidelines for Neo4j, including indexing, path materialization, and query rewriting; empirically validated on 13 LDBC SNB queries with significant performance gains.
De Virgilio *et al.* [12].	Model-Driven Physical Design.	Presents a conceptual-to-physical methodology for designing graph databases based on ER-to-intermediate mapping, reducing data access operations and improving performance.

on the fly. This strategy allows the optimizer to adapt to the dynamic nature of graph data without requiring precomputed statistics. The gCBO also includes an interactive component that allows users to receive optimized execution plans with detailed information and to generate their execution plans. This interactive feature makes gCBO more user-friendly and enables incorporating user expertise into the optimization process.

Jachiet, Genevès, Gesbert, and Layaida [13], propose mu-RA (mu-Relational Algebra), a variant of the Relational Algebra equipped with a fixpoint operator for expressing recursive relational queries, which notably supports unions of conjunctive regular path queries. The fixpoint operator in mu-RA enables algebraic transformations that generate new execution plans, which cannot be obtained with traditional approaches. These transformations lead to significant performance improvements for evaluating recursive queries over graphs. The mu-RA approach also includes rewrite rules specifically devised to optimize recursive queries. These rules leverage the properties of the fixpoint operator to transform recursive terms into more efficient forms. Experimental results demonstrate that the newly generated plans can provide substantial performance improvements for evaluating recursive queries over graphs.

Sharma, Genevès, Gesbert, and Layaïda [14], introduce a type inference mechanism that enriches recursive graph queries with relevant structural information from a graph schema, thereby optimizing query performance. It demonstrates that the schema information can improve the performance of evaluating acyclic recursive graph queries, while also proving that the method is sound and complete, ensuring the preservation of the query's semantics during the schema-enrichment process.

Baumstark, Jibril, and Sattler [15], present an adaptive approach that integrates graph query interpretation and compilation, allowing queries to use an interpreter while compilation and code generation processes run in the background. This method enables the system to switch to the compiled code once it is ready, improving overall query performance. The assessment of the approach demonstrates that autonomously switching execution modes improves runtime for short-running and complex queries, effectively hiding compilation time and mitigating additional latencies associated with the underlying storage.

Dörpinghaus and Stefan [16] propose a multi-step optimization approach for graph queries, in which an external algorithm communicates with the graph database to execute only elementary queries. This method focuses on typical questions such as neighborhood, paths, and relations, thereby optimizing the query execution process. A polyglot persistence approach is suggested for handling trivial queries, with other data sources used to execute simple requests. This approach aims to improve efficiency by offloading basic queries that can be managed by common relational or specialized databases, thus enhancing the overall performance of graph database interactions.

Kotiranta, Junkkari, and Nummenmaa [17], compare the query performance of a graph-based database system (Neo4j) and relational database systems (MySQL and MariaDB) by executing typical information needs that would be in the chosen test databases, focusing on simple and complex queries to assess efficiency differences. The effects of indexing were examined by indexing specific columns and properties used in queries across all databases and by analyzing their performance, with particular attention to the impact of indexing on relational-database efficiency relative to graph databases for queries of varying complexity.

In Khan, Ahmad, Luo, and Ahmed [18], physical tuning is first performed on an Oracle relational database, improving its performance by up to 50%, and then comparing it with the Neo4j NoSQL graph database, concluding that the latter performed better in all evaluated scenarios; the experimental study was conducted on a 406 MB database. In this work, tuning is performed only on the relational database. Likewise, Asplund and Sandell [19] compared the performance of three database systems: MySQL with two graph databases, Neo4j and ArangoDB experimentally, using the Novabench benchmark to measure query time, CPU usage, memory, power consumption, and server temperature. The results showed that Neo4j was the fastest at executing queries, and ArangoDB was the slowest. In a cloud environment, Alyas, Alzahrani, Alsaawy, Alissa, Abbas, and

Tabassum [20], focus on improving database performance through compression and query optimization to reduce resource consumption and execution time. The performance of MySQL and Neo4j was compared, with a particular focus on memory usage and query speed on the dew cloud servers, using an ERP dataset from an educational institution with an approximate size of 5.7 MB.

LDBC SNB CHALLENGE

An important element to consider in the study is selecting an appropriate benchmark for conducting the experiments. The LDBC SNB used in our study satisfies the set of basic guidelines for selecting a Graph Database benchmark proposed by [9]. LDBC [10] is an European Union program that aims to develop solid industry benchmarks for graph data management systems and related Resource Description Framework components. Within this program, the SNB project raises two points of reference in a common data set through a data generator (Datagen] that simulates a synthetic social network with a structure based on the potential law (power-law) [10]. The original data are obtained from forum conversations on DBpedia.

The specifications published by LDBC [10] include 39 queries, divided into two benchmarks: the Interactive Workload (IW) with 14 queries and the Business Intelligence Workload (BIW) with 25 queries. IW tests the performance of a system with relatively simple queries and simultaneous updates. IW could be classified as an Online Transaction Processing (OLTP) workload. However, while queries typically access only a small portion of a database, this workload can involve hundreds of thousands of data points. BIW consists of complex, structured queries to support the analysis of the users' online behavior for marketing purposes. This problem is dedicated exclusively to query execution and optimization. Unlike IW, BIW queries involve a greater amount of data as the database grows. LDBC SNB specified that IW queries involve a considerable volume of data, though confined to the immediate adjacency of a single node. Conversely, BIW queries traverse the complete graph structure to extract relevant information. Since the objective of our work is to test the validity of physical design strategies, the selected queries must cover the entire graph or the largest possible area, which corresponds to the properties of BIW queries.

Furthermore, to test the scalability of these strategies, the queries are executed at different workloads of 1, 10, and 100 GB scale. Figure 1 presents the LDBC [10] data schema, which defines the data structure used in the benchmarks, including entities and their relationships. The data describe the activity of a social network during a defined period of time. The data is available on GitHub [10].

Physical design guidelines

In this section, we will define three physical design guidelines for databases on Neo4j: Query Rewriting, Path Materialization, and Index Creation. It should be mentioned that the minimum query guideline, which is part of query rewriting, along with the second and third guidelines, was introduced by [11].

Definition 1 (GR1. Query Rewriting): Rewriting should only be applied to queries involving multi-hop traversals and complex filter conditions. Rewritten queries must leverage existing indexes or materialized paths to ensure performance gain. Violation of this guideline may increase computational overhead without improving execution time.

Application 1 (GR1. Query Rewriting): Analyze complex queries using the database's execution planner. If query plans involve multiple relationships or filters, rewrite them using intermediate nodes, subqueries, or pattern projections. Validate that rewritten forms activate indexes or match materialized patterns.

Definition 2 (GR1.1. Minimal Query): In a graph-oriented database, data relationships allow the construction of paths that the execution engine must traverse to retrieve the data required by a user query. This sub-guideline eliminates redundant or semantically equivalent paths that differ only in variable naming. A path is considered duplicated if it consists of nodes with identical labels and relationships but uses different variable identifiers.

Application 2: Analyze the query for pattern repetition, and refactor paths to a single representation when they match on label and relationship semantics, reducing the execution time by minimizing graph traversal overhead.

Definition 3 (GR1.2. Query Decomposition): This sub-guideline recommends relocating selection

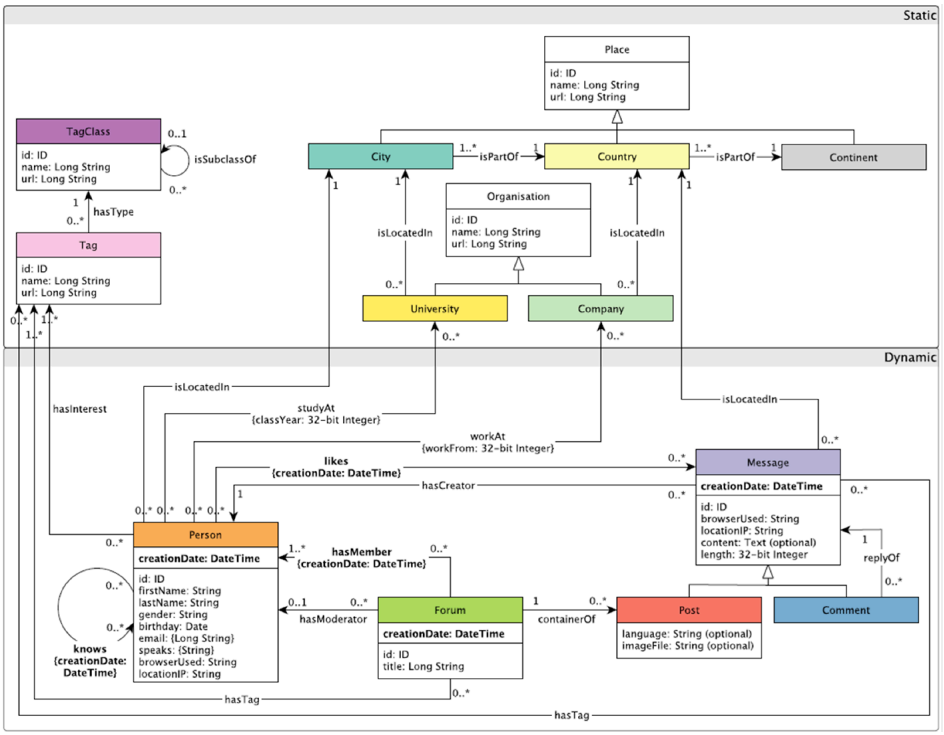


Figure 1. LDBC SNB data schema [10].

and aggregation operations to the portions of the query where the relevant nodes or relationships are accessed. Doing so ensures that data is filtered at the earliest opportunity, thus reducing the volume of intermediate results processed by the engine.

Application 3: Move filtering predicates into the MATCH or OPTIONAL MATCH clauses, and position aggregations in WITH clauses closer to their data origin. This restructuring ensures tighter control over intermediate data and avoids unnecessary computation.

Definition 4 (GR1.3. Limited Properties): This sub-guideline addresses query efficiency when ORDER BY and LIMIT clauses are used. Rather than accessing all node properties before limiting the result set, it proposes reordering the query to restrict the number of rows first, and then to access the required properties. This approach also encourages inlining aggregations and calculations directly where needed.

Application 4: Introduce WITH clauses before ORDER BY and LIMIT to pre-filter results. Move any property access or aggregation from RETURN to WITH and apply calculations only after reducing the data volume. Replace unnecessary intermediate variables with inline expressions to minimize memory usage and improve response time.

Definition 5 (GR2. Path Materialization): Paths selected for materialization must occur in at least 30% of executed queries and exhibit high traversal cost during runtime. Eligible paths must also involve stable subgraph structures with low update frequency. Ignoring this rule may result in unnecessary memory consumption and reduced efficiency.

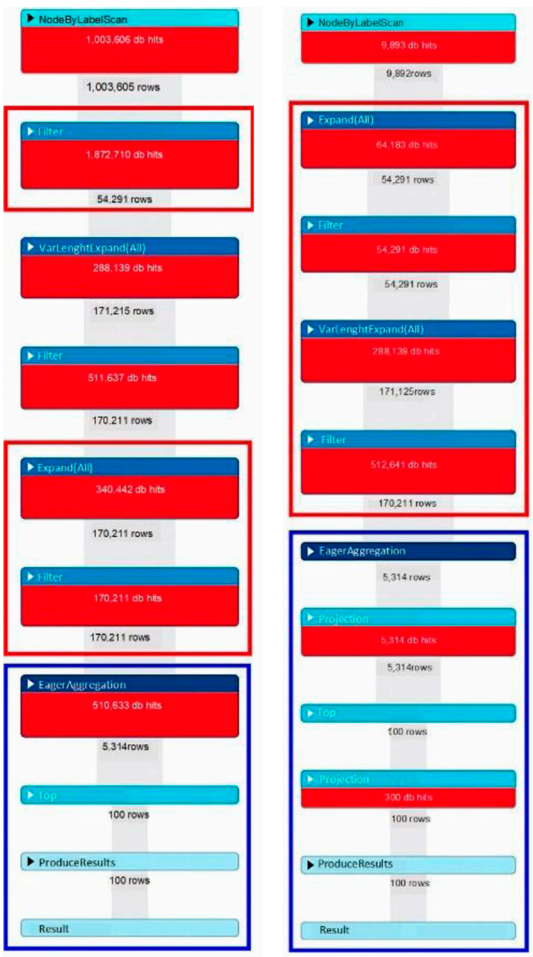
Application 5: Use query profiling tools to detect frequent traversal paths that involve 2+ hops. Evaluate their computational cost, and if they are stable (not frequently updated), store them in materialized views or summary nodes. Avoid materializing volatile or rarely used paths.

Definition 6 (GR3. Index Creation): Index attributes, which are declared for optimization purposes, must directly correspond to properties frequently used in MATCH or WHERE clauses. These attributes

should be selected to match the workload's access patterns. Failure to adhere to this guideline may lead to full graph scans and degraded query performance.

Application 6: During schema design, identify properties that appear in equality or range predicates of most queries. Create composite or full-text indexes on these properties using database-specific mechanisms (e.g., CREATE INDEX in Neo4j). Avoid indexing attributes that are rarely queried or frequently updated.

The BI14 query from the Social Network Benchmark, presented in Algorithm 1, illustrates several of the proposed guidelines. For this query, the Path Materialization and Limited Properties guidelines will be used. Figure 2 shows the implementation Plan Query BI14. The Initial Plan corresponds to the query plan generated by Neo4j by default, without applying any physical design guidelines. The Optimized Plan is the one produced by Neo4j when the proposed scheme is executed. In Figure 2a, we will apply Path Materialization to the first two red boxes and Limited Properties to the blue box.



a) Initial Plan b) Optimized Plan

Figure 2. Implementation Plan Query BI14.

Algorithm 1 Original Code Query BI14

```
1  MATCH.
2  (person:Person)<- [:HAS_CREATOR]-
   (post:Post) <- [:REPLY_OF*0..]-(reply:Message).
3  WHERE.
4  post.creationDate >= 20120531220000000.
5  AND post.creationDate <= 20120630220000000.
6  AND reply.creationDate >= 20120531220000000.
7  AND reply.creationDate <= 20120630220000000.
8  RETURN.
9  person.id, person.firstName, person.lastName,
   count(DISTINCT post) AS threadCount.
10 count(DISTINCT reply) AS messageCount.
11 ORDER BY.
12 messageCount DESC, person.id ASC.
13 LIMIT 100.
```

The blue box in Figure 2a contains an operator, EagerAggregation, with a high DB Hits cost. Algorithm 3 shows a portion of the query BI14 that corresponds to the operator EagerAggregation in Figure 2a, where the Limited Properties guideline will be applied. In this code, two count functions are the reason for the high cost of the EagerAggregation operator. However, this cost can be reduced by relocating these two functions to a WITH statement and modifying the code as shown in Algorithm 4. The first red box contains the filters on creationDate belonging to publications. In algorithm 2, route materialization has been applied to the HAS_CREATOR relationship between the Person and Post nodes, since the filters are applied to the creationDate property.

Figure 2b shows the result of applying the Path Materialization guideline, where the critical point corresponding to the operator Filter was directly eliminated, and reduced the execution plan's processing cost shown in Figure 2a. The result of applying limited properties is also observed, leading to a reduction of up to 10% of the original cost of the EagerAggregation operator. Finally, Algorithm 5 contains the code resulting from applying the physical design guidelines.

Algorithm 2 Implementation Strategy Road Materialization

```
MATCH
(person:Person)<- [:HAS_CREATOR]-(post:Post).
WHERE.
post.creationDate >= 20120531220000000.
AND post.creationDate <= 20120630220000000.
CREATE.
(post)- [:BI14_post_person]->(person).
```

Algorithm 3 Original Code Segment

```
RETURN.
person.id, person.firstName, person.lastName, count(DISTINCT post) AS threadCount.
count(DISTINCT reply) AS messageCount.
ORDER BY.
messageCount DESC, person.id ASC.
LIMIT 100.
```

Algorithm 4 Optimized Code Segment

```
WITH.
person, count(DISTINCT post) AS threadCount, count(DISTINCT reply) AS messageCount.
ORDER BY.
messageCount DESC, person.id ASC.
LIMIT 100.
RETURN.
person.id, person.firstName, person.lastName, threadCount, messageCount.
```

Algorithm 5 Optimized Code Query BI14

```
MATCH.
(person:Person)<- [:BI14_post_person]-(post:Post)<- [:REPLY_OF*0..]-(reply:Message)
WHERE.
reply.creationDate >= 20120531220000000.
AND reply.creationDate <= 20120630220000000.
WITH.
person, count(DISTINCT post) AS threadCount, count(DISTINCT reply) AS messageCount.
ORDER BY.
messageCount DESC, person.id ASC.
LIMIT 100.
RETURN.
person.id, person.firstName, person.lastName, threadCount, messageCount.
```

Experimental Study

Performance was measured as *query execution time*, defined as the elapsed time, in seconds, between the submission of a query to the Neo4j engine and its delivery of results. *Database Hits (DB Hits)* were also measured; these are units of measurement defined by Neo4j to quantify the actions performed by the execution engine. This unit allows the assessment of the number of actions required by the execution engine to fulfill query requirements. The *DB Hits* metric represents the number of accesses to storage primitives required by the execution plan. It is deterministic for a given query and database state, making it an objective, reproducible metric for comparing the efficiency of different query strategies. Although the Hits/Miss metrics could have been used, these were not considered because they measure the effectiveness of the physical memory layer, not the query's logical efficiency. Their values depend on previously executed operations, not on the query's structure. In this sense, the decision to use *DB Hits* instead of Hits/Miss is based on the distinction between the query evaluation at the logical level and the physical management of memory.

Initially, the experimental databases were generated in accordance with the prescribed workload specifications. In this work, workloads of 1 GB, 10 GB, and 100 GB were used. Using Datagen [10], workloads were generated and converted to the Neo4j query language, known as Cypher Query Language (CQL), as defined by LDBC SNB. Once the workloads were generated, the DBMS was configured to load the databases. Each query was executed with a hot cache to compile workload statistics. The performance of a hot cache is measured by ignoring the first execution of a test and averaging subsequent executions, without restarting the database. This cache was specifically configured in Neo4j by assigning 44.4 GB to the pagecache variable to store graph data and indexes.

Due to RAM constraints, this same value was used for the heap size, and it was also necessary to implement 100 GB of virtual memory to handle the query load. The database is only restarted to clear the cache when testing a new configuration. In this work, the number of repetitions was set at 25, 10, and 5, respectively, for the three different workloads (1GB, 10GB, 100GB). Subsequently, the execution times and DB Hits accesses for each query were collected, without applying the physical design. Finally, the collection of statistics

was repeated after applying the physical design guidelines. 86,400,000 ms (24 hours) was defined as the timeout limit per transaction.

The current experimental study was conducted on a server with an Intel(R) Core(TM) i7-8700 CPU, comprising six main processors and twelve logical processors, 32 GB of RAM, a main hard disk of 256 GB SSD, and a secondary hard disk of 4 TB HDD. Given the RAM restriction, it was decided to implement virtual memory with a size of 100 GB to be able to cover the largest number of queries. The operating system used is Microsoft Windows 10 Pro, with Neo4j Desktop 1.2.9 installed and Neo4j Browser 3.5.18 configured on the secondary disk. Based on the intended usage, principally local experimentation with an individual user, Neo4j Desktop fulfilled our requirements.

Other advantages include no registration fee and a free development license for the Enterprise Edition, allowing us to use Neo4j Enterprise on our local desktop for application development. Neo4j Desktop is for local development only and is not intended or licensed for deployment or use as a server version. For these reasons, it has been selected for the current experiment over other possibilities, such as a Neo4j Server, which is more suitable for scenarios where the database is intended to be hosted on a dedicated server, making it available to remote clients or with other characteristics like clustering, high availability, and security options that are required or more suitable for enterprise-grade applications. Another option is Neo4j on AWS, which is not expensive for research purposes but always incurs a fee based on the number of hours used and the region.

RESULTS

The execution times for each query across each workload are shown in Figure 3, where the x-axis shows the query number (with the prefix BI) and the y-axis corresponds to the percentage of improvement in execution time. Similarly, the percentage of improvement for the DB Hits metric is presented in Figure 4. We have made sure that our data presentation aligns with best statistical practices, acknowledging that the precision of our measures may vary.

The Shapiro-Wilk test [21] was conducted to statistically verify that execution times in the physically designed database convey an enhancement over those in the database without physical design, thereby confirming that both sets of execution times follow a normal distribution. This test implies that the sample follows a normal distribution when the p-valor is greater than $\alpha = 0.05$. According to the results of this test, found in Table 2, it can be concluded that the samples do not follow a normal distribution due to values less than 0.05.

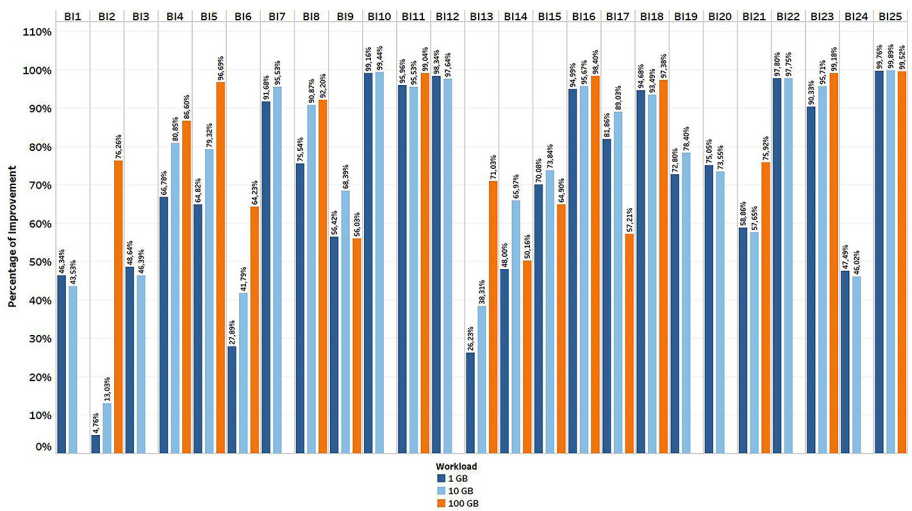


Figure 3. Comparison of execution time improvement percentages by workload for each query.

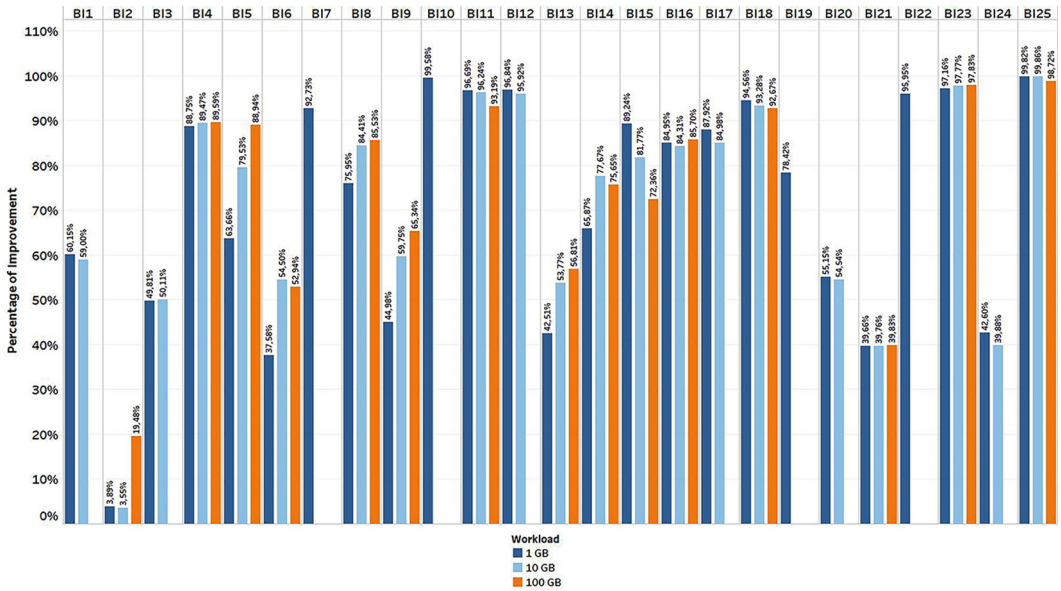


Figure 4. Comparison of DB Hits improvement percentages by workload for each query.

Table 2. *p-value* obtained from the Shapiro-Wilk. test.

Query	*p-value* (original)	*p-value* (physical design)
BI1	0.218	0.6483
BI2	**3.78x10**−6	**1.25x10**−5
BI3	0.5586	**4.58x10**−3
BI4	**1.87x10**−4	**4.47x10**−4
BI5	0.7952	**5.13x10**−3
BI6	**2.85x10**−7	**9.92x10**−3
BI7	0.03439	0.09587
BI8	**2.36x10**−3	0.01625
BI9	0.09111	**4.49x10**−7
BI10	0.6551	0.07871
BI11	0.05071	**9.62x10**−5
BI12	0.0971	**4.63x10**−4
BI13	**6.80x10**−4	**1.92x10**−4
BI14	0.09982	**1.77x10**−3
BI15	**6.36x10**−3	**4.05x10**−6
BI16	0.00143	0.02321
BI17	**2.14x10**−4	**1.06x10**−3
BI18	0.5242	**3.33x10**−4
BI19	0.01486	0.01307
BI20	0.9565	0.3918
BI21	0.1157	**4.55x10**−5
BI22	**1.61x10**−3	0.04222
BI23	**3.96x10**−7	**2.85x10**−4
BI24	**3.21x10**−4	**3.20x10**−4
BI25	0.5799	**2.72x10**−7

Since the results do not follow a normal distribution, another test must be performed. The Wilcoxon signed-rank paired test [22] is a non-parametric test that allows us to compare the averages of the execution times of all the queries for the database without physical design (μ0) and the database with physical design (μ1). The assumption to be tested is $\mu 0 > \mu 1$. After applying this test, the results shown in Table 3 were obtained. Based on statistical tests, it can be validated that the times obtained using the database without physical design are considerably higher than those obtained with physical design, confirming that $\mu 0 > \mu 1$.

In this study, the concept of threads of validity is employed as a guiding framework to ensure methodological rigor across all phases of the experimental design and analysis involving Neo4j and the LDBC Social Network Benchmark (SNB). Regarding internal validity, strict control over

Table 3. *p-value* obtained from the Wilcoxon signed-rank paired test.

Workload	*p-value*	Significance Level
1 GB	0.0000000596	$\alpha = 0.05$
10 GB	0.0000000596	$\alpha = 0.05$
100 GB	0.0000305200	$\alpha = 0.05$

the benchmarking environment was maintained to ensure that observed performance differences were attributable to the physical design strategies applied –such as indexing, path materialization, and query rewriting– and not to external confounding variables. The use of standardized workloads and repeated trials under identical system configurations strengthens this thread.

For external validity, the selection of the LDBC SNB benchmark ensures representativeness of real-world social network workloads, supporting the generalizability of the findings to other graph database applications with similar structural complexity and query patterns. In terms of construct validity, the study operationalizes key performance concepts, such as "query efficiency," using well-established metrics, including average response time, and database size. These metrics are directly aligned with theoretical definitions of performance optimization in graph data management. Lastly, the conclusion validity thread is maintained by ensuring logical consistency between the data analysis and the study's claims. All reported improvements are statistically supported and tied explicitly to the implemented physical design interventions, avoiding overgeneralizations or unsupported inferences. Together, these threads provide an integrated framework of methodological consistency and validity, reinforcing the reliability of the results and their contribution to the physical design of graph database systems.

DISCUSSION

Analysis of the results shown in Figure 3 reveals that most exhibit stable behavior. The fields with no information are due to a timeout. Queries BI6, BI13, BI18, and BI25 work only with Path Materialization, while queries BI11, BI16, and BI23 use Limited Properties. On the one hand, all of these queries show stability across their three workloads, with no significant variation; even the most considerable difference in improvement percentage is only 10% for query BI23.

On the other hand, we have queries BI2, BI4, BI5, BI6, BI8, BI13, and BI21 that suffered a high impact as the workload increased. Among this set of queries, the first one, which implements Query Decomposition, showed low improvement percentages in its 1 GB and 10 GB workloads. This query also reveals the most significant variation, with an increase of approximately 72% for the 100 GB workload. Query BI21 that uses exclusively Limited Properties achieves a 17% increase, highlighting that both strategies improve performance as the dataset size increases. The remaining queries in this set use the combination of Path Materialization and Limited Properties, achieving an average increase of 30%.

The other pattern identified in Figure 3 are queries BI1, BI3, BI7, BI10, BI12, BI19, BI20, BI22, and BI24. It can be observed that, across these queries, the percentages are balanced, highlighting query BI1, for which an index was created. This query shows the most significant decrease in improvement percentage within this set of queries, at only 2%. Finally, the last set of queries is analyzed, which includes those that presented a decrease in their statistics: BI9, BI14, BI15, and BI17. Across these four queries, different factors have been identified that could provide insight into their behavior. The first factor is that the queries have a high number of operations. Another factor is the use of conditions in the WHERE statement, which accesses the nodes, resulting in a high processing cost. Furthermore, if the query lacks a statement that limits records, it cannot reduce processing costs when handling such a large number of records.

Additionally, Figure 4 shows the improvement percentages of DB Hits per query for each workload. First, we notice that there are even fewer results than in Figure 3 due to the memory restrictions that caused the queries to throw buffer overflow. Most queries show even greater stability than the results obtained with execution-time improvements (Figure 3). However, some queries underwent some positive and negative variations. In the first set, which includes all queries except BI15, queries BI5 and BI9 show improvements of approximately 25% and 20%, respectively. Query BI15 is the only query that shows a sustained decrease in its improvement percentage, from 89% to 72%; this may be due to the growth of the data set, as the query's structure increased complexity when manipulating the records and performing the calculations. Finally, we would like to highlight query BI2, which in the first two workloads maintained an average of 3%, and exhibited a significant increase of 19% in the 100 GB dataset, an increase of almost 16%.

The advantages and disadvantages of implementing each physical design strategy are described below. Query Rewrite is the most recommended guideline because it incurs no storage cost. However, extreme caution is advised when implementing it, as incorrect application can cause query equivalence to be lost. Path Materialization is the guideline that presents the best results. It should be emphasized that the way this guideline was applied reduced its implementation cost.

Neo4j does not implement algorithms to update materialized relationships when new records are added automatically; this creates a constant concern for the database administrator, who must manually update them.

It is important to mention that Path Materialization was implemented for the benchmark queries exactly as specified in the documentation. While a more generalized path might be feasible, it is important to emphasize that this does not undermine the validity of the results obtained through our chosen strategy. The decision to adhere to the specified parameters was deliberate and aimed at aligning with the defined criteria for benchmark evaluation. It allowed us to provide a clear and consistent assessment of our system's performance within the established scope. Index creation is recommended when query conditions are highly selective.

However, the number of records to be indexed should be evaluated in advance against the associated storage costs, given that memory constraints may sometimes require discarding this strategy altogether. This strategy is commonly used in relational databases and has been shown to be effective. This study has shown that effectiveness is also maintained in graph databases. Although indexes are a commonly used strategy and do not constitute an innovation, they are considered essential to ensure the strategy's comprehensiveness. Despite their familiarity with the field, their inclusion ensures that no relevant aspect is overlooked and guarantees a comprehensive and all-encompassing approach. The incorporation of indices reflects a commitment to rigor, the consideration of all relevant factors in this strategic approach, and adherence to conventional practices.

CONCLUSIONS

In this work, three physical design guidelines have been presented: Query Rewriting, Path Materialization, and Index Creation. For the first one, three additional sub-strategies: Minimal Query, Query Decomposition, and Limited Properties were considered. First, Query Rewriting is recommended over the other strategies since it does not incur an extra storage cost. Second, the Index Creation strategy is recommended because the graph database engine incorporates in into its platform via optimized algorithms. Finally, Path Materialization is the least recommended strategy because graph database engines do not directly supported it. Therefore, it can be complex for non-technical users to apply, and its constant re-implementation can be expensive. In addition, this last strategy entails additional storage costs. Among the three strategies, Path Materialization offers the best execution times. The results show an average reduction in execution time of 77% when using this guideline, 59% for Query Rewriting, and 45% for Index Creation. These percentages show a reduction in execution time when comparing queries run with and without applying guidelines.

Finally, the proposal in this paper goes beyond the state of the art by focusing on three physical design guidelines that could have a positive impact on query performance and support developers in their design tasks. These guidelines should be considered before assessing and executing queries. Furthermore, the experimental results show the efficiency and feasibility of implementing these guidelines.

As future work, the implementation of these guidelines for query performance improvement in an automatic recommender system, including mathematical specifications of the proposed guidelines, will be conducted in subsequent work for this project. The initial implementation of Path Materialization focused on specific parameter values in the queries. This approach can indeed limit the effectiveness of Path Materialization when the same query is issued with different parameter values, as the pre-materialized paths may no longer be applicable. The original intention in suggesting Path Materialization was to decrease query times and enhance overall database performance for common query patterns. One potential approach is to adopt a more dynamic, parameterized path materialization

system that intelligently adjusts materialization based on the specific query parameters. Thus, it can be guaranteed that Path Materialization remains valuable for a broader range of queries without significantly increasing database size.

ACKNOWLEDGEMENT

We would like to thank Javiera Soto for her English proofreading support. A. Aguilera thanks to Regular Fondecyt No. 1250344, ANID, Chile.

REFERENCES

[1] A. Oussous, F.Z. Benjelloun, A.A. Lahcen, and S. Belfkih, Comparison and classification of NoSQL databases for big data.

[2] A.B.M. Moniruzzaman and S.A. Hossain, "NoSQL database: New era of databases for big data analytics - classification, characteristics and comparison," *International Journal of Database Theory and Application*, vol. 6, no. 4, pp. 1-14, 2013.

[3] I. Robinson, J. Webber, and E. Eifrem, *Graph databases: New opportunities for connected data,* 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2015, pp. 1-10.

[4] L. Yang, L. Yang, Y. Pang, and L. Zou, "gCBO: a cost-based optimizer for graph databases," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, Atlanta, GA, USA, Oct. 17-21, 2022, pp. 5054-5058, doi: 10.1145/3511808.3557197.

[5] M. Mior, "Physical design for non-relational data systems," Ph.D. dissertation, University of Waterloo, Ontario, Cánada, 2018. [Online] Available: https://dspacemainprd01.lib.uwaterloo.ca/server/api/core/bitstreams/5f5f6b63-e6a9-4ed1-8e5c-85bbcdb70b4e/content

[6] S.S. Lightstone, "Physical database design for relational databases," in *Encyclopedia of Database Systems*. L. Liu and M.T. Özsu, Eds., Boston, MA, USA: Springer, 2009, pp. 2108-2114, doi: 10.1007/978-0-387-39940-9_644.

[7] M. Mozaffari, A. Dignös, J. Gamper, and U. Störl, "Self-tuning database systems: A systematic literature review of automatic database schema design and tuning," *ACM Computing Surveys*, vol. 56, no. 11, Art. no. 277, pp. 1-37, 2024, doi: 10.1145/3665323.

[8] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys*, vol. 40, no. 1, pp. 1-39, 2008, doi: 10.1145/1322432.1322433.

[9] D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta, and J.L. Larriba-Pey, "A discussion on the design of graph database benchmarks," *in Performance Evaluation, Measurement and Characterization of Complex Systems*, vol. 6417, R. Nambiar, M. Poess, Eds., Berlin, Germany: Springer, 2010, pp. 25-40, doi: 10.1007/978-3-642-18206-8_3.

[10] R. Angles *et al.*, "The LDBC social network benchmark," 2020, arXiv:2001.02299.

[11] M. Jota, M. Goncalves, and R. Parra, "A physical design strategy on a NoSQL DBMS," *in Data Science. Theory, Analysis and Applications*, Q.A. Memon, S.A. Khoja, Eds., Boca Raton, FL, USA: CRC Press, 2019, pp. 69-93, doi: 10.1201/9780429263798-4.

[12] R. De Virgilio, A. Maccioni, and R. Torlone, "Model-Driven design of graph databases," *in Conceptual Modeling. Lecture Notes in Computer Science*, vol. 8824, E. Yu, G. Dobbie, M. Jarke, and S. Purao, Eds., Cham, Switzerland: Springer, 2014, pp. 172-185, doi: 10.1007/978-3-319-12206-9_14.

[13] L. Jachiet, P. Genevès, N. Gesbert, and N. Layaida, "On the optimization of recursive relational queries: Application to graph queries," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, Portland, OR, USA, Jun. 14-19, 2020, pp. 681-697, doi: 10.1145/3318464.3380567.

[14] C. Sharma, P. Genevès, N. Gesbert, and N. Layaïda, "Schema-Based Query Optimisation for Graph Databases," in *Proceedings of the ACM on Management of Data*, vol. 3, no. 1, Art. no. 72, pp. 1-29, 2025, doi: 10.1145/3709722.

[15] A. Baumstark, M.A. Jibril, and K.-U. Sattler, "Adaptive query compilation in graph databases," *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW), Chania, Greece, 2021, pp. 112-119,* doi: 10.1109/ICDEW53142.2021.00027.

[16] J. Dörpinghaus and A. Stefan, "Optimization of retrieval algorithms on large scale knowledge graphs," in *Federated Conference on Computer Science and Information Systems*, vol. 21, 2020, pp. 227-236, doi: 10.15439/2020F2.

[17] P. Kotiranta, M. Junkkari, and J. Nummenmaa, "Performance of graph and relational databases in complex queries," *Applied Sciences*, vol. 12, no. 13, Art. no. 6490, 2022, doi: 10.3390/app12136490.

[18] W. Khan, W. Ahmad, B. Luo, and E. Ahmed, "SQL database with physical database tuning technique and NoSQL graph database comparisons," in *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, Chengdu, China, 2019, pp. 110-116, doi: 10.1109/ITNEC.2019.8729264.

[19] E. Asplund and J. Sandell, "Comparison of graph databases and relational databases performance," Degree project dissertation, Department of Computer and Systems Sciences, Stockholm University, Stockholm, Sweden, 2023.

[20] T. Alyas, A. Alzahrani, Y. Alsaawy, K. Alissa, Q. Abbas, and N. Tabassum, "Query optimization framework for graph database in cloud dew environment," *Computers, Materials & Continua*, vol. 74, no. 1, pp. 2317-2330, 2023, doi:10.32604/cmc.2023.032454.

[21] S.S. Shapiro and M.B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591-611, 1965, doi: 10.2307/2333709.

[22] F. Wilcoxon, "Individual Comparisons by Ranking Methods," in *Breakthroughs in Statistics. Springer Series in Statistics*, S. Kotz, N.L. Johnson, Eds., New York, NY, USA: Springer, 1992, doi: 10.1007/978-1-4612-4380-9_16.