UNIVERSIDAD TÉCNICA DEL NORTE

Facultad de Ingeniería en Ciencias Aplicadas

Carrera de Software

Tema:

"Comparativa de eficiencia de acceso de datos entre bases de datos relacionales y no relacionales medido desde una arquitectura REST y GraphQL"

Trabajo de grado previo a la obtención del título de Ingeniero de Software presentado ante la ilustre Universidad Técnica del Norte.

Autor:

Brayan Alexis Velasco Rosero

Director:

PhD. José Antonio Quiña Mera

Ibarra – Ecuador

2025

UNIVERSIDAD TÉCNICA DEL NORTE

BIBLIOTECA UNIVERSITARIA

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

DATOS DE CONTACTO			
CÉDULA DE IDENTIDAD:	1050259348		
APELLIDOS Y NOMBRES:	VELASCO ROSERO BRAYAN ALEXIS		
DIRECCIÓN:	PRIOTATO-IBARRA		
EMAIL:	brayanvlsc@gmail.com		
TELÉFONO FIJO:	N/A	**TELÉFONO MÓVIL:**	0986971170

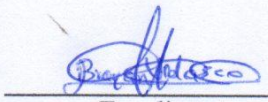
DATOS DE LA OBRA	
TÍTULO:	**Comparativa de eficiencia de acceso de datos entre bases de datos relacionales y no relacionales medido desde una arquitectura REST y GraphQL**
AUTOR(ES):	Brayan Alexis Velasco Rosero
FECHA:	31 de julio del 2025
PROGRAMA:	PREGRADO
TÍTULO POR EL QUE OPTA:	INGENIERO DE SOFTWARE
DIRECTOR:	PhD. José Antonio Quiña Mera
ASESOR 1:	Msc. Carpio Agapito Pineda Manosalvas

2. CONSTANCIAS

El autor manifiesta que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto, la obra es original y que es el titular de los derechos patrimoniales, por lo que asume la responsabilidad sobre el contenido de la misma y saldrá en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, a los 28 días del mes de agosto de 2025

EL AUTOR:



Estudiante

Brayan Alexis Velasco Rosero

1050259348

CERTIFICACIÓN DIRECTOR

Ibarra 31 de julio del 2025

CERTIFICACIÓN DIRECTOR DEL TRABAJO DE TITULACIÓN

Por medio del presente yo, **José Antonio Quiña Mera**, certifico que el Sr. **Brayan Alexis Velasco Rosero** portador de la cedula de ciudadanía número **1050259348**, ha trabajado en el desarrollo del proyecto de grado **"Comparativa de eficiencia de acceso de datos entre bases de datos relacionales y no relacionales medido desde una arquitectura REST y GraphQL",** previo a la obtención del Título de Ingeniero en Software realizado con interés profesional y responsabilidad que certifico con honor de verdad.

Es todo en cuanto puedo certificar a la verdad

Atentamente

JOSE ANTONIO QUIÑA MERA

Firmado digitalmente por JOSE ANTONIO QUIÑA MERA
Nombre de reconocimiento (DN): c=EC, sn=QUIÑA MERA, givenName=JOSE ANTONIO, serialNumber=IDCEC-1002322384, cn=JOSE ANTONIO QUIÑA MERA, 2.5.4.97=TINEC-1002322384001

PhD. José Antonio Quiña Mera

DIRECTOR DE TRABAJO DE GRADO

DEDICATORIA

Este trabajo va dedicado para mi familia, en especial a mis padres, ustedes quienes han sido mi inspiración para seguir adelante y no dejarme vencer a pesar de las adversidades. Gracias papá por la responsabilidad que me enseñaste y a pesar de la distancia siempre me apoyaste y me diste un amor incondicional, gracias mamá por el tiempo y toda la paciencia que has tenido conmigo, por todas tus enseñanzas y la humildad que siempre me has inculcado, ustedes han sido mi pilar fundamental en todo este camino que he recorrido y espero lo sigan siendo por muchos años más.

Brayan Alexis Velasco Rosero

AGRADECIMIENTO

Siempre estaré agradecido con mis padres por todo lo que han hecho por mí, gracias a ustedes he podido alcanzar esta nueva meta en mi vida, su apoyo en todo momento fue importante en todo el transcurso de este proyecto y es por eso que este logro alcanzado es por ustedes y para ustedes.

De la misma manera, me encuentro agradecido con la Universidad Técnica del Norte, donde he vivido momentos extraordinarios y siempre me he llevado una gran enseñanza gracias a sus docentes en la carrera de Software y a todos quienes fueron mis docentes en el transcurso de este proceso. Un agradecimiento especial al PhD. Antonio Quiña quien fue mi tutor de tesis y me enseñó sus conocimientos y me brindó su tiempo en el desarrollo de este proyecto.

También estoy agradecido con todos mis hermanos, Anderson, Andrés, Maykel, Robin y la pequeña Samara, gracias por sus consejos y por darme el valor de seguir adelante. Asimismo, gracias Alex, has sido alguien muy importante en todo este transcurso y un ejemplo a seguir.

Gracias también a todos mis amigos y compañeros que he hecho a lo largo de este camino, a los que por cuestiones del destino se han alejado de mi vida y a los que aún permanecen conmigo.

Por último, pero no menos importante, quiero agradecer a una persona que ha marcado mi vida y ha sido de gran importancia en este proceso, gracias por tu tiempo y los consejos que me diste para seguir adelante y no perder el rumbo, sin ti no habría logrado esto, gracias Carolina.

Brayan Alexis Velasco Rosero

TABLA DE CONTENIDOS

Tabla de contenido

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

El presente documento se encuentra conformado por tres capítulos, en el cual se detalla todo el proceso para realizar el Trabajo de Grado: **"Comparativa de eficiencia de acceso de datos entre bases de datos relacionales y no relacionales medido desde una arquitectura REST y GraphQL".**

La sección de introducción aborda los antecedentes, la situación actual, la prospectiva, el planteamiento del problema, los objetivos general y específicos, así como el alcance y la justificación del estudio.

En el capítulo 1 se detalla el marco teórico de la investigación, abordando los fundamentos de REST, GraphQL, bases de datos relacionales y no relacionales. Además, se analizan las tendencias tecnológicas en el ámbito de las bases de datos y se incluyen conceptos relacionados con la norma ISO/IEC 25023, enfocándose en la métrica de tiempo de respuesta.

El capítulo 2 detalla la planificación del proyecto de investigación, incluyendo la descripción del entorno de ejecución del experimento computacional, la definición de las consultas y su distribución según cada caso de uso, así como la estructura y el proceso de ejecución del experimento.

En el capítulo 3, se muestran los resultados obtenidos en la ejecución del experimento computacional y se realiza el análisis correspondiente para responder a la pregunta de investigación.

Finalmente se encuentra las conclusiones, recomendaciones y referencias bibliografías

Palabras clave: Api, REST, GraphQL, ISO, experimento, tiempo de respuesta, eficiencia, bases de datos, arquitectura, consultas.

ABSTRACT

This document consists of three chapters, which detail the entire process for completing the thesis project: "Comparison of data access efficiency between relational and non-relational databases measured from a REST and GraphQL architecture."

The introduction section addresses the background, the current situation, the outlook, the problem statement, the general and specific objectives, as well as the scope and justification of the study.

Chapter 1 details the theoretical framework of the research, addressing the fundamentals of REST, GraphQL, relational and non-relational databases. In addition, it analyzes technological trends in the field of databases and includes concepts related to the ISO/IEC 25023 standard, focusing on the metric of response time.

Chapter 2 details the planning of the research project, including a description of the execution environment of the computational experiment, the definition of the queries and their distribution according to each use case, as well as the structure and execution process of the experiment.

Chapter 3 shows the results obtained in the execution of the computational experiment and performs the corresponding analysis to answer the research question.

Finally, the conclusions, recommendations, and bibliographic references are presented.

Keywords: API, REST, GraphQL, ISO, experiment, response time, efficiency, databases, architecture, queries.

INTRODUCCIÓN

Tema

Comparativa de eficiencia de acceso de datos entre bases de datos relacionales y no relacionales medido desde una arquitectura REST y GraphQL

Problema

La selección de la base de datos y la arquitectura de acceso de datos es un aspecto crítico en el desarrollo de aplicaciones web modernas debido a la diversidad de opciones disponibles. Las bases de datos relacionales, como Amazon RDS, ofrecen consistencia y soporte para consultas estructuradas, mientras que las no relacionales, como Amazon DynamoDB, proporcionan flexibilidad y escalabilidad para datos dinámicos. La elección depende de factores como el modelo de datos, los patrones de acceso y las necesidades de escalabilidad, donde decisiones inadecuadas pueden impactar negativamente el rendimiento, la escalabilidad y los costos operativos (AWS, 2025b).

Antecedentes

Las bases de datos relacionales, como Amazon RDS, organizan datos en tablas con esquemas predefinidos, utilizando SQL para consultas y transacciones complejas, ideales para aplicaciones que requieren consistencia, como sistemas empresariales. Amazon RDS, gestionado por AWS, soporta motores como MySQL y PostgreSQL, automatizando tareas como copias de seguridad y escalado vertical (AWS, 2025c). En contraste, las bases de datos NoSQL, como DynamoDB, están diseñadas para datos no estructurados y alta escalabilidad, siendo adecuadas para aplicaciones modernas como comercio electrónico o IoT (AWS, 2025a).

Las arquitecturas de acceso a datos, como REST y GraphQL, facilitan la interacción con

estas bases de datos. REST, basado en recursos identificados por URLs, es eficiente para operaciones CRUD, aunque requiere documentación externa para navegar la API (Ahmedou Yassin, 2023). GraphQL, introducido en 2015, permite consultas personalizadas, reduciendo el sobre o subfetching, lo que es ideal para interfaces dinámicas y bases NoSQL (Amazon Web Service, 2024).

Situación Actual

Amazon RDS es preferido para aplicaciones que requieren consistencia y consultas SQL avanzadas, con características como Multi-AZ para alta disponibilidad y réplicas de lectura para mejorar el rendimiento (AWS, 2025c). Sin embargo, su escalabilidad horizontal es limitada, lo que puede generar latencias en cargas de trabajo intensivas comparado con bases NoSQL como DynamoDB (Bytebase, 2025).

DynamoDB destaca por su modelo sin servidor, escalando automáticamente con latencias de milisegundos, ideal para aplicaciones de alto tráfico como juegos o IoT. Su diseño sin esquema y características como índices secundarios globales permiten flexibilidad, aunque requiere un modelado de datos optimizado (AWS, 2025a).GraphQL, integrado con AWS AppSync, mejora la eficiencia de consultas en DynamoDB, mientras que REST sigue siendo común por su simplicidad, aunque menos flexible en escenarios dinámicos (Amazon Web Service, 2024).

Prospectiva

DynamoDB continuará ganando relevancia en aplicaciones modernas, especialmente con integraciones como AWS AppSync para GraphQL y soporte para casos de uso como IA generativa, gracias a su escalabilidad y flexibilidad (AWS, 2025a). GraphQL probablemente

superará a REST en aplicaciones con datos dinámicos, ya que reduce la carga de red y permite consultas personalizadas (Amazon Web Service, 2024).

Las bases de datos relacionales, como RDS, evolucionarán con opciones sin servidor como Aurora Serverless para mejorar la escalabilidad, manteniendo su ventaja en transacciones críticas (AWS, 2025c). Las arquitecturas híbridas, combinando relacionales y NoSQL, serán comunes para optimizar rendimiento y costos, con GraphQL liderando en personalización y REST en aplicaciones más simples (Bytebase, 2025).

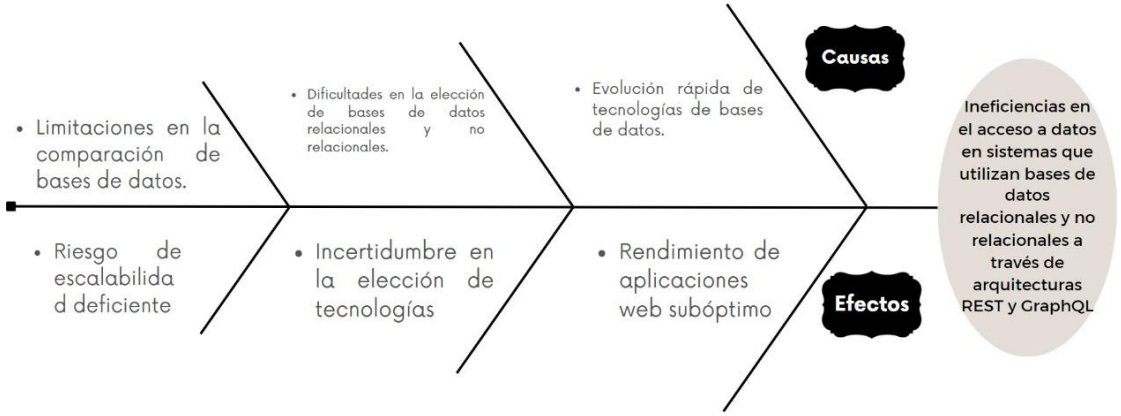
Planteamiento del problema

La elección de la base de datos y la arquitectura de acceso de datos es un desafío creciente para el desarrollo de aplicaciones web modernas. Las bases de datos relacionales y no relacionales tienen sus propias ventajas y desventajas, por lo que la selección de la tecnología adecuada depende de los requisitos específicos de la aplicación. La evolución constante de las tecnologías de bases de datos y las arquitecturas web crea un entorno altamente dinámico, en el que las decisiones incorrectas pueden tener consecuencias significativas en términos de rendimiento, escalabilidad y costo (AWS, 2025b).

En la figura 1 se encuentra detallado el diagrama de Ishikawa, donde están las causas y efectos correspondientes para el problema actual.

Figura 1

Diagrama de Ishikawa



Objetivos

Objetivo General.

- Comparar la eficiencia de las arquitecturas REST y GraphQL en el acceso de datos de bases de datos relacionales y no relacionales, utilizando la métrica de tiempo de respuesta de la ISO/IEC 25023.

Objetivos Específicos

A. Revisión de la literatura y estudios relevantes para identificar las tres bases de datos relacionales y no relacionales más usadas con las arquitecturas REST y GraphQL.

B. Desarrollar un experimento computacional para comparar la eficiencia del acceso de datos en bases de datos relacionales y no relacionales con las arquitecturas REST y GraphQL, utilizando la métrica de tiempo de respuesta de la ISO/IEC 25023.

C. Analizar los resultados obtenidos del experimento computacional para determinar la base de datos y arquitectura más eficiente en el acceso de datos.

Alcance

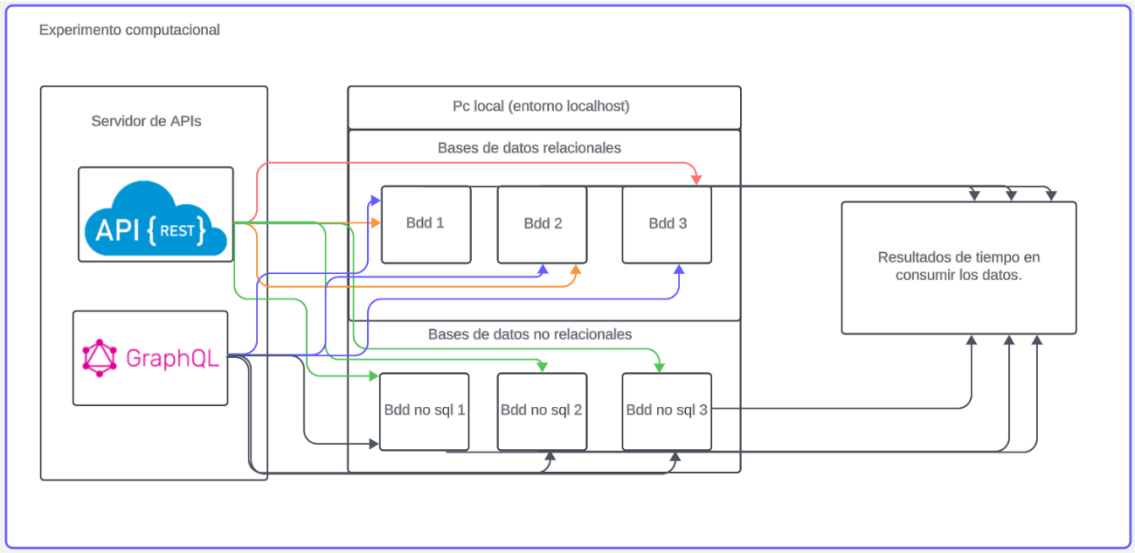
El proyecto se enfocará en la comparación de la eficiencia de acceso de datos entre bases de datos relacionales y no relacionales en el contexto de las arquitecturas REST y GraphQL. Se seleccionarán bases de datos específicas relevantes para aplicaciones web modernas. Para esto se aplicará una investigación para así conocer cuáles son estas bases de datos más usadas, se planea buscar toda la información posible para que de esta manera se tenga claro el contexto con lo que se va a trabajar.

GraphQL es un lenguaje de consulta de la API, así como un tiempo de ejecución para responder a esas consultas con los datos existentes. También viene equipado con potentes herramientas para manejar incluso las consultas más complejas. La característica central de GraphQL es su capacidad para solicitar y recibir sólo los datos específicos solicitados, nada más. Esto hace que sea mucho más sencillo escalar tus APIs junto con tu aplicación. La parte más emocionante de GraphQL es su capacidad para proporcionarte todos los datos en un punto final (Hernández Matías, 2022).

Se medirán métricas de rendimiento clave basándose en la ISO 25023, como el tiempo medio de respuesta, ese tiempo se obtendrá de manera sencilla restando el tiempo final menos el tiempo inicial. Como se muestra en la figura 2 el proyecto se basa en una arquitectura cliente servidor

Figura 2

Arquitectura del proyecto



Metodología

Para cumplir el primer objetivo de la investigación, se realizará una revisión de la literatura existente sobre bases de datos relacionales y no relacionales. Esta revisión incluirá tanto fuentes primarias, como artículos científicos y revistas, como fuentes secundarias, como foros y documentación menos formal. Para realizar esta revisión, se utilizarán bases de datos académicas y literatura especializada en el desarrollo web. Además, se utilizarán términos clave para realizar búsquedas específicas relacionadas con las tendencias actuales y bases de datos. Para cumplir el segundo objetivo de la investigación, se realizará un experimento computacional para comparar el rendimiento de acceso de datos en bases de datos relacionales y no relacionales. El experimento se realizará utilizando la guía de Wohlin, que se especializa en la realización de experimentos en entornos controlados. La métrica de tiempo de respuesta de la ISO/IEC 25023 se utilizará para medir la eficiencia de las arquitecturas.

Para cumplir el tercer objetivo de la investigación, se analizarán los resultados del experimento computacional para determinar qué base de datos es más eficiente para el acceso de datos. Los resultados se tabularán y graficarán para facilitar el análisis.

Justificación

Este trabajo de investigación contribuye al objetivo de desarrollo sostenible número 9 de la ONU, que es "promover la industria, la innovación y la infraestructura". Esto se debe a que el trabajo tiene como objetivo evaluar y comparar la eficiencia del acceso de datos en bases de datos relacionales y no relacionales. Al proporcionar información sobre la eficiencia de estas bases de datos, el trabajo puede ayudar a los desarrolladores a elegir la base de datos adecuada para sus proyectos web. Esto, a su vez, puede ayudar a promover la innovación en el desarrollo web y a impulsar la utilización de infraestructuras tecnológicas eficientes y sostenibles (Naciones Unidas, 2023).

Este proyecto también está en línea con el Plan de Creación de Oportunidades 2021-2025 de Ecuador, que tiene como objetivo promover la transformación digital y el acceso a las tecnologías de la información y la comunicación (TIC). Al analizar tecnologías vanguardistas como GraphQL y REST, este proyecto contribuye a actualizar y fortalecer la base tecnológica del país, y también a promover la implementación de soluciones digitales avanzadas (Secretaría Nacional de Planificación, 2021).

Justificación Tecnológica. – La evolución constante de las tecnologías de bases de datos y las arquitecturas web ha dado lugar a un entorno altamente dinámico, en el que las aplicaciones web deben ser capaces de adaptarse a cambios rápidos y repentinos. Las bases de datos no relacionales, como MongoDB, ofrecen una serie de ventajas que las hacen atractivas para las aplicaciones web, como su flexibilidad, escalabilidad y rendimiento. Sin embargo, el debate

sobre la eficiencia de las bases de datos relacionales y no relacionales es complejo y depende de una serie de factores (Campoverde Henry, 2012).

CAPÍTULO 1: Marco Teórico

El presente capítulo proporciona los fundamentos conceptuales necesarios para comprender el desarrollo del experimento propuesto. Se abordan las tecnologías involucradas, como los sistemas de gestión de bases de datos relacionales y no relacionales, así como las arquitecturas REST y GraphQL utilizadas para el acceso a datos. Además, se presentan conceptos relacionados con la calidad del software y métricas de rendimiento, en particular aquellas definidas por la norma ISO/IEC 25023. Este marco teórico permite contextualizar la importancia de evaluar el comportamiento de diferentes tecnologías en entornos reales y justificar la metodología empleada en el estudio.

1.1. Fundamentación conceptual

1.1.1. Introducción

En el tejido esencial de la era digital, las bases de datos desempeñan un papel vital a la hora de facilitar el almacenamiento, la organización y la recuperación eficientes de la información. Estas estructuras informáticas actúan como repositorios fundamentales para una gran variedad de aplicaciones, desde sistemas empresariales hasta plataformas web y aplicaciones móviles. Existen varios tipos de bases de datos diseñadas para satisfacer necesidades específicas. Las bases de datos relacionales, arraigadas en el modelo propuesto por Edgar Codd, estructuran los datos en tablas interconectadas, ideales para aplicaciones que requieren integridad y coherencia.

Las bases de datos no relacionales, conocidas como NoSQL, proporcionan flexibilidad y escalabilidad para gestionar datos dinámicos y heterogéneos, utilizando modelos como clave-valor, documentos, columnares y grafos. Estas bases de datos destacan por su capacidad para manejar grandes volúmenes de datos no estructurados o semiestructurados, adaptándose a

entornos con requisitos cambiantes y ofreciendo escalabilidad horizontal sin interrupciones. La elección entre los diferentes tipos de bases de datos NoSQL depende de factores como la estructura de los datos, la complejidad de las consultas y las necesidades de rendimiento del sistema. En este sentido, cada modelo aborda desafíos específicos, como el procesamiento en tiempo real, la gestión de relaciones complejas o el análisis de grandes conjuntos de datos, siendo ideales para aplicaciones modernas como redes sociales, análisis en tiempo real y sistemas de recomendación (Winkler et al., 2023).

1.1.1 Bases de datos relacionales

Las bases de datos relacionales organizan los datos en tablas compuestas por filas y columnas, donde cada tabla se estructura mediante un esquema definido. Estas tablas se interconectan mediante claves primarias y foráneas, que establecen relaciones entre los datos, permitiendo modelar estructuras complejas a través de diferentes modelos de datos relacionales. Los analistas emplean consultas SQL para integrar y analizar datos, facilitando la generación de informes, la optimización de procesos empresariales y la identificación de oportunidades estratégicas. Este enfoque relacional sigue siendo fundamental en aplicaciones que requieren consistencia, integridad de datos y consultas estructuradas, como sistemas de gestión empresarial y aplicaciones financieras (Elmasri & Navathe, 2022).

Las bases de datos relacionales permiten estructurar datos en tablas interconectadas, como una tabla de clientes que contiene información a nivel de cuenta (por ejemplo, ID de cliente, nombre de la empresa, dirección, industria) y una tabla de transacciones que registra detalles de operaciones individuales (como fecha de transacción, ID de cliente, monto, método de pago). Estas tablas se vinculan mediante claves comunes, como el ID de cliente, lo que facilita la integración de datos para generar informes analíticos, como ventas por industria o

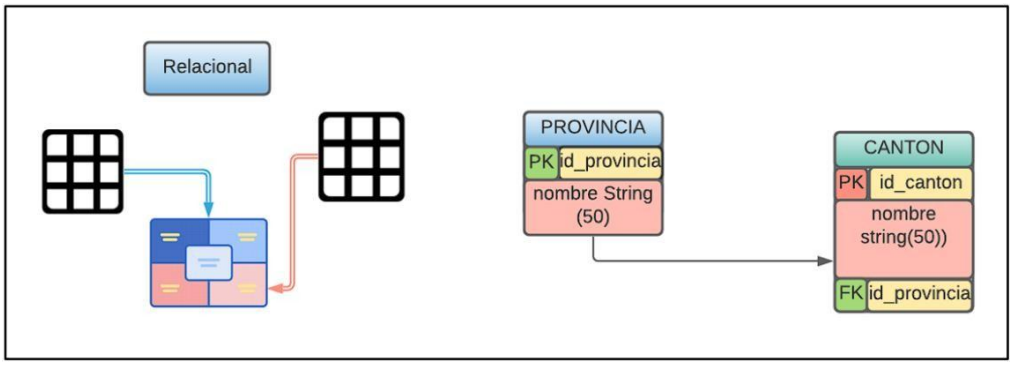
empresa. Este enfoque relacional, soportado por consultas SQL, permite a las organizaciones extraer información valiosa, optimizar estrategias de negocio y personalizar comunicaciones con clientes, siendo esencial en sistemas de gestión y análisis de datos (Silberschatz, 2023).

En la figura 3 se puede observar cómo es el diseño de la estructura de una base de datos relacional.

Figura 3

Diseño de la estructura de una base de datos relacional

Fuente: (Quinche Moran, 2021)



El crecimiento exponencial de los datos transmitidos por Internet, impulsado por aplicaciones como redes sociales, IoT y comercio electrónico, ha desafiado las capacidades de las bases de datos relacionales (RDBMS) para procesar grandes volúmenes de datos semiestructurados o no estructurados de manera eficiente. Las RDBMS, diseñadas para datos estructurados y esquemas rígidos, enfrentan limitaciones en términos de escalabilidad y velocidad cuando se manejan cargas de trabajo dinámicas que requieren distribución en múltiples servidores. En este contexto, las bases de datos NoSQL ("No solo SQL") han surgido como una solución robusta, ofreciendo flexibilidad para gestionar datos heterogéneos y escalabilidad

horizontal sin comprometer el rendimiento. Estas bases de datos, que incluyen modelos como clave-valor, documentos, columnares y grafos, se adaptan a las necesidades de aplicaciones modernas que priorizan rapidez y adaptabilidad (Arroyo, 2024).

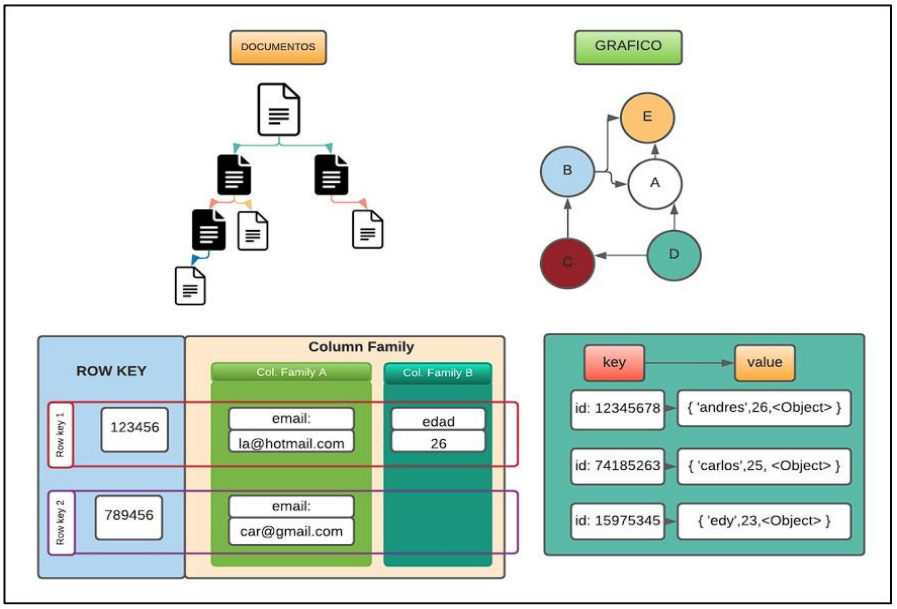
Entre los modelos NoSQL, las bases de datos orientadas a documentos, como MongoDB o Amazon DocumentDB, destacan por su capacidad de almacenar datos en formatos flexibles como JSON o BSON, utilizando estructuras de pares clave-valor que facilitan la gestión y recuperación de datos semiestructurados. Este enfoque permite un procesamiento más rápido y simplifica la distribución de datos en entornos distribuidos, lo que las hace ideales para aplicaciones que requieren alta disponibilidad y escalabilidad, como sistemas de recomendación o análisis en tiempo real. Además, las bases de datos NoSQL ofrecen ventajas en términos de costos operativos y facilidad de integración con arquitecturas nativas de la nube, lo que las convierte en una opción preferida para desarrolladores que buscan optimizar el rendimiento en entornos dinámicos (AWS, 2025b).

En la figura 4 Se puede observar cómo es el diseño de la estructura de una base de datos no relacional.

Figura 4

Diseño de la estructura de una base de datos no relacional

Fuente: (Quinche Moran, 2021)



1.1.2 Bases de datos más usadas en los últimos años

Para saber esto se realizó una investigación en conjunto en las bases de datos bibliográficas que nos ofrece la Universidad Técnica del Norte como son:

- IEEE XPLORE
- ScienceDirect
- Scopus

Así como también, en diferentes bases de datos bibliográficas de internet, como son:

- DBLP
- Google Scholar

Obteniendo los siguientes resultados:

- Bases de datos relacionales

MySQL es la más usada en este tipo de bases de datos, seguida por **Postgres** y en tercer lugar **SQL Serve**

- Bases de datos no relacionales

En este tipo de bases de datos se encontró que la más usada es **MongoDB**, seguida de **Neo4j** y en tercer lugar **Cassandra.**

Para corroborar esta información se hizo uso de Platzi una destacada plataforma de educación en línea en América Latina en los últimos años, ganando una considerable popularidad gracias a su amplia variedad de cursos que contribuyen al desarrollo profesional en áreas como tecnología, marketing, diseño, entre otros. La elección de Platzi como fuente se fundamenta en la calidad de sus contenidos y la experiencia de sus instructores, quienes son profesionales activos en empresas líderes como Google, Microsoft y Amazon Web Services. En uno de sus cursos sobre bases de datos, Platzi destaca las principales tecnologías utilizadas en la industria, presentando ejemplos de bases de datos relacionales como MySQL, PostgreSQL y SQL Server, así como no relacionales como MongoDB y Redis, explicando sus casos de uso y ventajas en aplicaciones modernas (Platzi, 2020).

Los resultados que arroja esta plataforma son los siguientes: Dentro de las bases de datos relacionales están:

- **MySQL:** Es ampliamente adoptado, especialmente en el stack LAMP (Linux, Apache, MySQL y PHP), siendo esta combinación la preferida debido a su extenso uso en la mayoría de las operaciones en internet.

- **PostgreSQL:** De acuerdo con Platzi, esta base de datos proporciona un rendimiento

superior y cuenta con características específicas, como la capacidad de almacenar archivos y objetos en formato JSON.

- **SQL Server:** Esta base de datos es más utilizada dentro del ámbito bancario y corporativo (Platzi, 2020).

Dentro de las bases de datos no relacionales se obtuvo estos resultados:

- **MongoDB:** Esta base de datos no relacional tiene un enfoque orientado a documentos, donde los datos se almacenan en estructuras de tipo BSON (Binary JSON).
- **Redis:** Se trata de una base de datos no relacional comúnmente utilizada para funciones de caché, y la mayoría de los frameworks de desarrollo web brindan soporte y compatibilidad con esta base de datos (Platzi, 2020).

En este caso concuerdan las bases de datos relacionales, sin embargo, las bases de datos no relacionales solamente MongoDB es la que concuerda, esto puede deberse al año de la publicación del ranking.

1.2. GraphQL

1.2.1. Introducción

GraphQL es un lenguaje de consulta para implementar arquitecturas de servicios web. Se desarrolló internamente en Facebook como solución a varios problemas a los que se enfrentaban al utilizar estilos arquitectónicos estándar, como REST. En 2015, Facebook publicó la definición e implementación de GraphQL. Como resultado, el lenguaje comenzó a ganar impulso y ahora es compatible con las principales API web, incluidas las proporcionadas por

GitHub, Airbnb, Netflix y Twitter (Brito & Valente, 2020).

Una de las ventajas de GraphQL es que permite a los desarrolladores crear consultas para extraer datos de múltiples fuentes en una sola llamada a la API. Los esquemas de GraphQL se componen de tipos de objetos, que definen lo que se puede solicitar y sus campos. A medida que se introducen las consultas, GraphQL las aprueba o rechaza basándose en el esquema, y luego ejecuta las validadas. Nuestros resultados muestran que GraphQL requiere menos esfuerzo para implementar consultas de servicios remotos en comparación con REST (9 frente a 6 minutos, tiempos medios) (Brito & Valente, 2020).

1.2.2. Lenguaje de consultas

El lenguaje de consulta GraphQL se caracteriza por su intuición y flexibilidad, siendo concebido para facilitar la creación de aplicaciones cliente al ofrecer un sistema para describir los requisitos de datos e interacciones. En su versión más básica, GraphQL implica solicitar campos particulares de objetos, asegurando que las consultas siempre arrojen resultados predecibles. Las aplicaciones que emplean GraphQL destacan por su rapidez y estabilidad, ya que tienen el control sobre los datos que obtienen en lugar de depender del servidor (GraphQL, 2022).

1.2.3. Estructura de consultas

GraphQL es un lenguaje de consulta de datos diseñado para implementar servicios web que se centran en abstracciones de alto nivel, tales como esquemas, tipos, consultas y mutaciones. A través de GraphQL, los clientes tienen la capacidad de especificar con precisión los datos que desean solicitar. Utilizando esta tecnología, los clientes pueden definir de manera exacta los datos necesarios por parte de los proveedores de servicios (Brito & Valente, 2020).

- **Campos**

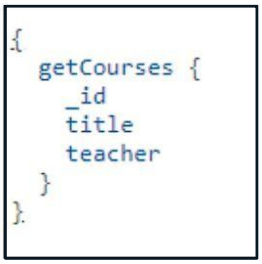
Las consultas en GraphQL están organizadas en términos de tipos y campos, no de endpoints. En su forma más simple, GraphQL se trata de solicitar campos específicos en objetos. Las consultas siempre devuelven resultados predecibles (GraphQL, 2022).

En la figura 4 se muestra un ejemplo de campos para una consulta en GraphQL.

Figura 5

Campos de una consulta GraphQL

Fuente: (Quinche Moran, 2021)



- **Argumentos**

Los campos de consulta pueden ser interpretados como "funciones" implementadas en el backend que retornan valores. Similar a las "funciones", tienen la capacidad de recibir argumentos para modificar su comportamiento o los valores que retornan (Vazquez-Ingelmo et al., 2017).

En la figura 6 se muestra el uso del argumento “id” para realizar una consulta GraphQL.

Figura 6

Argumento en una consulta GraphQL

```
{
getCourse (Id: "5f933d4b688a843113dde188”) {
        _id
        title
        teacher
        }
}
```

Esta es la forma en que se emplea un argumento en una consulta de GraphQL: "id" actúa como el argumento que especifica qué curso en particular solicita el usuario. Por supuesto, en el backend deben estar implementados mecanismos que permitan filtrar los objetos de datos a través de un identificador, facilitando así el uso del argumento "id".

1.2.4. Ejecución

GraphQL realiza una consulta que produce un resultado que refleja la estructura de la consulta solicitada, generalmente en formato JSON (GraphQL, 2022).

1.2.4.1. Resolvers

Para atender las consultas, el desarrollador de un servidor GraphQL debe implementar una función denominada "resolver" para cada consulta declarada en el tipo de consulta. Estas funciones son invocadas cada vez que el motor del servidor GraphQL requiere recuperar un tipo de objeto especificado en una consulta (Brito & Valente, 2020).

En la figura 7 se puede observar el ejemplo de cómo está estructurado un resolver hecho en GraphQL.

Figura 7

Ejemplo de un resolver

```
Query: {
  human(obj, args, context, info) {
    return context.db.loadHumanByID(args.id).then(
      userData => new Human(userData)
    )
  }
}
```

Se encuentran disponibles resolvers asincrónicos que posibilitan la carga de datos desde una base de datos, devolviendo una Promise. Las promesas se emplean para manejar valores de manera asíncrona. Cuando la base de datos proporciona un dato, podemos construir y retornar un nuevo objeto (GraphQL, 2022). En la siguiente imagen se puede observar un ejemplo de un resolver asincrónico que está cargando los datos desde una base de datos no relacional, en este caso MongoDB.

```
getCourses: async() => {

    let courses = []
    try {
      db = await connectDB()
      courses = await db.collection('course').find().toArray()
      return courses
      }catch(error){
      console.error(error)
    }
},
```

1.2.4.2.Resultado

El resultado de este resolver se presenta en la figura 8. Como podemos observar el resultado se muestra en un formato JSON.

Figura 8

Resultado de una consulta GraphQL

1.3. REST

1.3.1. Introducción

REST es un estilo arquitectónico diseñado para construir sistemas hipermedia distribuidos, basándose en el empleo de URLs para identificar y acceder a los recursos disponibles en la web. Los APIs REST son ampliamente utilizados en la actualidad debido a su capacidad para crear interfaces sencillas, flexibles y escalables para aplicaciones web, móviles y dispositivos. Estos APIs hacen uso de métodos, cabeceras y otros elementos del protocolo HTTP, junto con formatos de datos como JSON, XML o HTML. Para cumplir con los principios fundamentales, los APIs REST deben adherirse a seis restricciones: interfaz uniforme, separación cliente-servidor, operaciones sin estado, caché de recursos, sistema por capas y código bajo demanda (Husar, 2022).

Las APIs REST ofrecen interfaces simples y uniformes al posibilitar la disponibilidad de datos, contenido, algoritmos, medios y otros recursos digitales a través de URLs web. Para que un servicio de API sea considerado RESTful, es necesario cumplir con seis restricciones fundamentales: utilizar una interfaz uniforme, basarse en un modelo cliente-servidor, operar de manera sin estado, implementar almacenamiento en caché de recursos RESTful, seguir un

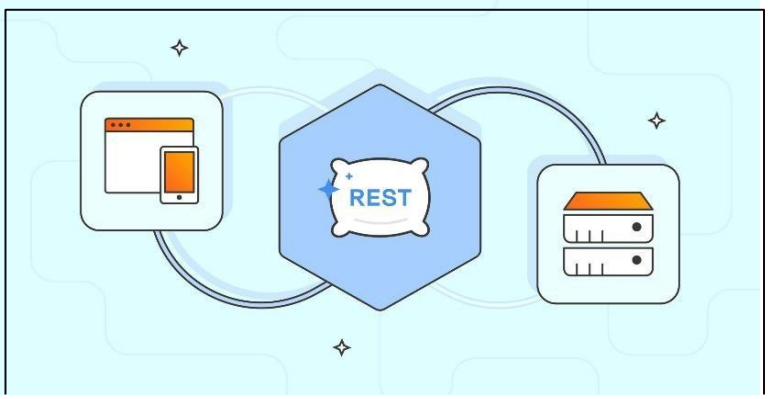
sistema en capas y permitir el código bajo demanda (Postman, 2020).

En la figura 9 se puede apreciar como es la estructura de la arquitectura REST.

Figura 9

Arquitectura REST

Fuente: (Postman, 2023)



1.3.2. Lenguaje de consultas

El lenguaje de consulta REST se fundamenta en la utilización de verbos HTTP para llevar a cabo operaciones sobre los recursos. Los verbos más usuales son GET, POST, PUT y DELETE. GET se emplea para obtener datos, POST para enviarlos, PUT para actualizarlos y DELETE para eliminarlos. Estas operaciones posibilitan que los clientes interactúen con los recursos del servidor de manera estandarizada (Rock, 2021).

1.3.3. Estructura de consultas

Las consultas SQL en una API REST se organizan mediante peticiones HTTP que interactúan con una base de datos. Estas peticiones pueden abarcar operaciones CRUD (Create, Read, Update, Delete), las cuales se asocian con los métodos POST, GET, PUT y DELETE,

respectivamente. A modo de ejemplo, una solicitud GET puede emplearse para recuperar datos de la base de datos mediante una consulta SQL (Microsoft, 2021).

- **GET**

Este verbo es usado para consultar los registros de un recurso en específico. Cabe aclarar que solo se encarga de consultar, esto quiere decir que, no se va a modificar o crear un nuevo recurso.

- **POST**

Este verbo se lo usa para crear nuevos recursos en el servidor que se esté usando. Cada una de las llamadas a este verbo significará la creación de un nuevo recurso.

- PUT

Este verbo nos ayuda a modificar un recurso existente en el servidor. En caso de no existir el recurso que se quiere modificar este verbo lo crea.

- DELETE

Este es el último de los 4 verbos más usados en REST y sirve para eliminar un recurso del servidor. Para realizar esta acción o la del verbo PUT se debe especificar un identificador, como puede ser un "ID".

1.3.4. Ejecución

La ejecución REST se basa en la interacción entre un cliente y un servidor a través de una API RESTful (Red Hat, 2023).

Estos son algunos de los pasos de la ejecución de REST:

- Cliente-servidor

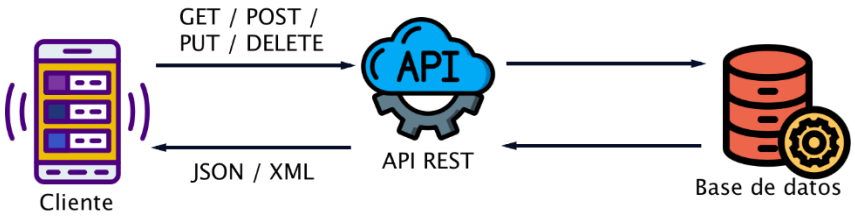
En el marco del modelo cliente-servidor, el cliente busca recursos o lleva a cabo operaciones en los datos, mientras que el servidor es la entidad encargada de proporcionar o procesar los datos según la solicitud del cliente (HubSpot, 2021).

En la figura 10 Se puede observar cómo es la arquitectura del cliente-servidor, así como también, el uso de los verbos principales y de la manera que se trabaja en REST, el cliente envía sus peticiones y todo se procesa en API REST.

Figura 10

Arquitectura cliente-servidor

Fuente: (Gutiérrez, 2024)



- Comunicación sin estado

Las ejecuciones de la API deben llevarse a cabo sin considerar la condición actual del cliente, las solicitudes previas o cualquier indicador almacenado que pueda afectar su comportamiento (HubSpot, 2021).

- URL como identificador único del recurso

Cada recurso en una API RESTful posee una URL única, lo que simplifica su identificación y manejo (HubSpot, 2021).

- Hipermedia

Al realizar una consulta a un recurso, este debe incluir enlaces o hipervínculos a acciones

o recursos adicionales que lo complementen (HubSpot, 2021).

- Formato de los datos

La información se transmite mediante HTTP en uno de los siguientes formatos: JSON (JavaScript Object Notation), HTML, XML, Python, PHP o texto sin formato (Red Hat, 2023).

- Encabezado y parámetros

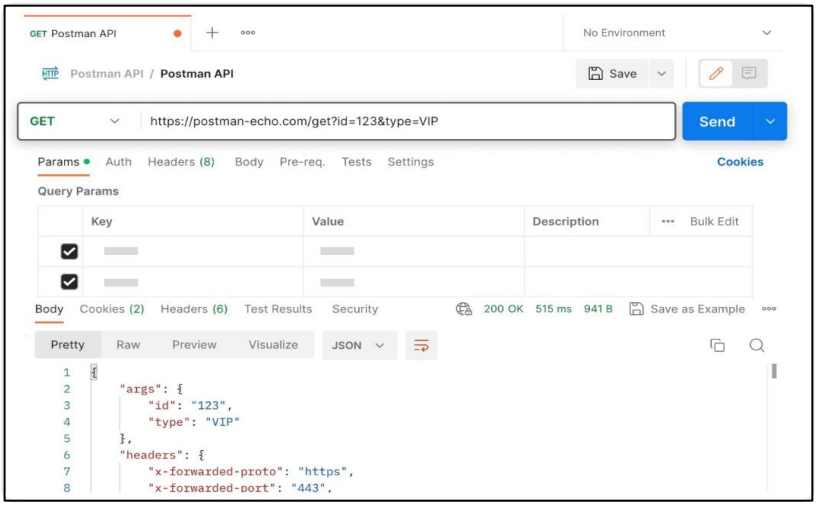
Las cabeceras y los parámetros son elementos significativos en los métodos HTTP de una solicitud en una API RESTful, ya que contienen información identificativa crucial relacionada con metadatos, autorización, identificador uniforme de recursos (URI), almacenamiento en caché, cookies y otros aspectos de la solicitud (Red Hat, 2023).

En la figura 11 se muestra un ejemplo de una herramienta la cual ayuda al cliente a desarrollar APIs, así como probar su correcto funcionamiento. Es de gran ayuda y facilita el trabajo de los desarrolladores al tener una interfaz gráfica.

Figura 11

Ejemplo de la herramienta Postman

Fuente: (Postman, 2020)



1.4. Revisión de la literatura acerca de las bases de datos relacionales y no relacionales

1.4.1. Tendencias tecnológicas y criterios de selección

Para seleccionar las bases de datos se realizó una ardua investigación en diferentes bases de datos bibliográficas de la UTN como son: Sience Direct, Scopus y IEEE Xplore, además de otras bases de datos como Google Scholar y DBLP. Para cada una de estas bases de datos bibliográficas se tomó los mismos parámetros para coincidir en los mismos resultados que buscamos. Los principales parámetros fueron: el año de publicación (a partir del año 2020), el idioma inglés o español, arquitectura REST, arquitectura GraphQL, API, ingeniería, ciencias de la computación, articulo científico, ponencia.

En la Tabla 1 se presentan los resultados obtenidos para cada motor de base de datos, evaluados bajo las arquitecturas REST y GraphQL.

Tabla 1

Resultados de la búsqueda

SQL	**NoSQL**
PostgreSQL	MongoDB
MySQL	CouchDB
SqLite	Neo4j

1.4.2. Bases de datos relacionales seleccionadas

Para examinar el desempeño y la conducta de las consultas en entornos GraphQL y REST, se eligieron tres sistemas de gestión de bases de datos relacionales basándose en un estudio previo acerca de su popularidad y uso recurrente en aplicaciones reales. Se seleccionaron PostgreSQL, MySQL y SQLite debido a que son las más comúnmente empleadas en proyectos

que aplican estas arquitecturas de acuerdo con las tendencias actuales. Además de su prestigio en la comunidad de desarrollo, estas bases de datos presentan variadas características en términos de complejidad, eficiencia y modelos de implementación, lo que facilita una comparación representativa y justa.

PostgreSQL

PostgreSQL es un sistema de gestión de bases de datos relacional (RDBMS) de código abierto, conocido por su robustez, cumplimiento del estándar SQL (SQL:2011 y posteriores) y soporte para características avanzadas como tipos de datos personalizados, transacciones ACID completas y extensibilidad. Es ampliamente utilizado en sistemas complejos y de misión crítica debido a su capacidad para manejar grandes volúmenes de datos y consultas sofisticadas. Además, su compatibilidad con herramientas como PostGraphile y Hasura lo hace ideal para integrar con GraphQL, permitiendo la generación automática de APIs a partir de esquemas de bases de datos (PostgreSQL Global Development Group, 2023).

Ventajas clave:

1. **Alto rendimiento en consultas complejas**: PostgreSQL optimiza consultas avanzadas mediante índices avanzados (como GIN y GiST) y soporte para operaciones como búsquedas full-text y consultas geoespaciales (con PostGIS). Según la documentación oficial, "PostgreSQL ofrece un planificador/optimizar de consultas sofisticado que puede manejar consultas complejas con múltiples uniones y subconsultas" (PostgreSQL Global Development Group, 2023).

2. **Integridad referencial fuerte**: Garantiza consistencia de datos mediante claves foráneas, restricciones de unicidad y triggers. "PostgreSQL implementa un sistema robusto de

integridad referencial que asegura la consistencia de los datos en transacciones complejas" (PostgreSQL Global Development Group, 2023).

3. **Compatibilidad con JSON y búsquedas avanzadas**: Soporta tipos de datos JSON/JSONB, permitiendo consultas avanzadas en datos semiestructurados, comparable a bases de datos NoSQL. "El soporte para JSONB permite realizar consultas indexadas en documentos JSON, lo que lo hace competitivo con bases NoSQL" (Kerstiens, 2020).

   Aplicaciones comunes: Sistemas empresariales (ERP, CRM), analítica de datos (data warehouses), APIs complejas (especialmente con GraphQL), y sectores como fintech y banca, donde la consistencia y fiabilidad son críticas.

   PostgreSQL es conocido por su robustez comprobada, fiabilidad y conjunto de características que lo hacen adecuado para aplicaciones de misión crítica (PostgreSQL Global Development Group, 2023).

MySQL

MySQL es un RDBMS de código abierto ampliamente utilizado, especialmente en aplicaciones web debido a su velocidad, simplicidad y compatibilidad con múltiples frameworks y ORMs (como Django, Laravel y Sequelize). Es mantenido por Oracle y conocido por su rendimiento en operaciones de lectura intensiva y facilidad de configuración. Aunque históricamente se centraba en velocidad más que en características avanzadas, versiones recientes (como MySQL 8.0) han añadido soporte para JSON, expresiones de tabla común (CTEs) y transacciones mejoradas (Oracle, 2023).

Ventajas clave:

1. **Alto rendimiento en operaciones de lectura concurrente**: MySQL está optimizado para entornos con alta concurrencia de lecturas, como sitios web de alto tráfico. "MySQL utiliza el motor de almacenamiento InnoDB para ofrecer alto rendimiento en operaciones de lectura y escritura concurrentes" (Oracle, 2023).

2. **Integración directa con múltiples ORMs y frameworks web**: Es compatible con herramientas populares como Hibernate, Sequelize y Laravel Eloquent, facilitando el desarrollo rápido de aplicaciones web. "MySQL es una opción popular para aplicaciones web debido a su facilidad de integración con frameworks modernos" (Percona, 2021).

3. **Facilidad de uso y administración**: MySQL ofrece herramientas como phpMyAdmin y MySQL Workbench, que simplifican la gestión de bases de datos para desarrolladores. "La configuración y administración de MySQL son intuitivas, lo que lo hace ideal para equipos pequeños o proyectos rápidos" (Oracle, 2023).

   Aplicaciones comunes: Aplicaciones web (WordPress, Drupal), plataformas de comercio electrónico (Magento, WooCommerce), SaaS y blogs, donde la velocidad de lectura y la simplicidad son prioritarias.

   MySQL es una base de datos relacional líder en el mercado, diseñada para ofrecer velocidad y escalabilidad en aplicaciones web modernas (Oracle, 2023).

SQLite

SQLite es una base de datos relacional embebida, ligera y sin servidor, diseñada para operar directamente dentro de una aplicación sin necesidad de un proceso servidor separado. Es ideal para entornos con recursos limitados, como dispositivos móviles, aplicaciones de escritorio

o pruebas locales. Aunque no está diseñada para alta concurrencia o escalabilidad en producción, su simplicidad y portabilidad la hacen popular para prototipos y aplicaciones embebidas (SQLite Documentation, 2023).

Ventajas clave:

1. **Extremadamente ligera y portátil**: SQLite almacena la base de datos en un solo archivo, lo que facilita su transporte y uso en entornos sin infraestructura pesada. "SQLite es una base de datos autocontenida que no requiere configuración ni administración, ideal para aplicaciones embebidas" (SQLite Documentation, 2023).

2. **No requiere instalación ni configuración de servidor**: SQLite funciona sin dependencias externas, lo que lo hace ideal para desarrolladores que buscan simplicidad. "La ausencia de un servidor hace que SQLite sea perfecto para aplicaciones con requisitos mínimos de infraestructura" (SQLite Documentation, 2023).

   Aplicaciones comunes: Aplicaciones móviles (Android, iOS), dispositivos IoT, prototipos, pruebas unitarias y aplicaciones de escritorio (como navegadores web, donde SQLite es usado para almacenamiento local).

Comparación general

En la Tabla 2 se muestra una comparación de todas las bases de datos SQL que se usará en el desarrollo de este proyecto, se muestran sus puntos fuertes y varias características importantes de cada una.

Tabla 2

Comparación general de bases de datos SQL

Base de datos	Característica	Modelo de despliegue	Soporte JSON	Transacciones	Escalabilidad	Compatibilidad REST/GraphQL
PostgreSQL		Cliente-servidor	Avanzado	Completo (ACID)	Alta	Alta (PostGraphile, Hasura)
MySQl		Cliente-servidor	Parcial	Parcial (depend. engine)	Media	Alta
SqLite		Embebido	Básico	Parcial	Baja	Media (requiere adaptación)

1.4.3. Bases de datos NoSQL seleccionadas

Durante el proyecto, CouchDB mostró limitaciones significativas en inserciones masivas, con tiempos de escritura elevados y degradación del rendimiento al aumentar el volumen de datos (hasta 100,000 registros). Su arquitectura documental y modelo de replicación, junto con una gestión ineficiente de concurrencia e indexación, resultaron inadecuados para cargas intensivas. Por ello, se migró a Cassandra, una base de datos NoSQL columnar que ofrece escalabilidad lineal, alta disponibilidad y escrituras optimizadas mediante LSM Trees. Su diseño descentralizado mejora la tolerancia a fallos y el rendimiento, haciéndola ideal para procesar consultas jerárquicas en los cinco niveles funcionales definidos (usuarios, publicaciones,

comentarios, respuestas y reacciones).

Para complementar el análisis comparativo en entornos REST y GraphQL, se seleccionaron tres sistemas de gestión de bases de datos no relacionales (NoSQL): **MongoDB**, **Neo4j** y **Cassandra**. La elección se fundamenta en su alta adopción en aplicaciones modernas, según el **Stack Overflow Developer Survey 2024**, donde MongoDB aparece como la base de datos NoSQL más utilizada (28.6% de los desarrolladores), seguida por Cassandra en aplicaciones distribuidas, y Neo4j destacando en el nicho de bases de datos de grafos (Stack Overflow, 2024). Estas bases representan modelos NoSQL distintos: **documentos** (MongoDB), **grafos** (Neo4j) y **columnas distribuidas** (Cassandra), permitiendo un análisis robusto del rendimiento en diferentes estilos de consulta.

Cada tecnología ofrece ventajas específicas para necesidades particulares de almacenamiento, escalabilidad y modelado de datos, enriqueciendo la evaluación comparativa en entornos REST y GraphQL.

MongoDB

MongoDB es una base de datos orientada a documentos que utiliza el formato **BSON** (Binary JSON), conocida por su flexibilidad y escalabilidad. Según el informe de (MongoDB Inc, 2024) es la base de datos NoSQL más adoptada, utilizada por el **30% de los desarrolladores** en aplicaciones modernas, especialmente por su integración con frameworks como **Node.js** y herramientas GraphQL como **Apollo Server** y **Mongoose** (MongoDB Inc, 2024).

Ventajas clave:

- **Modelo flexible sin esquemas rígidos**: Permite cambios dinámicos en la estructura de datos, ideal para desarrollo ágil (MongoDB Inc, 2024).
- **Soporte nativo para documentos anidados**: Simplifica el manejo de estructuras complejas, eliminando la necesidad de uniones costosas en la mayoría de los casos (Percona, 2024).
- **Alta escalabilidad horizontal y replicación**: Su arquitectura de clústeres soporta grandes volúmenes de datos, con tiempos de respuesta promedio de **2-5 ms** para consultas optimizadas en configuraciones estándar (Percona, 2024).
- **Aplicaciones comunes**: APIs RESTful, microservicios, plataformas de comercio electrónico, y sistemas de gestión de contenido en tiempo real.

MongoDB ha mejorado su compatibilidad con GraphQL mediante integraciones con **Apollo Federation**, lo que permite construir APIs federadas con mayor eficiencia (Apollo, 2024).

Neo4j

Neo4j es una base de datos orientada a grafos que modela datos como **nodos** y **relaciones**, optimizada para estructuras altamente conectadas. Según un informe de **Gartner (2024)**, Neo4j lidera el mercado de bases de datos de grafos, con un crecimiento en adopción del **27% desde 2023**, impulsado por su integración con GraphQL a través de la **Neo4j GraphQL Library** (Gartner, 2024). Su lenguaje de consulta, **Cypher**, permite explorar relaciones complejas de manera eficiente.

Ventajas clave:

- **Consultas optimizadas para relaciones complejas**: Cypher ofrece un rendimiento hasta **100 veces superior** a bases relacionales en escenarios de redes complejas, como análisis de grafos profundos (Neo4j Inc, 2024).
- **Ideal para exploración de grafos y jerarquías**: Eficiente en casos como análisis de redes sociales, detección de fraudes, y sistemas de recomendación.
- **Alta integración con GraphQL**: La Neo4j GraphQL Library genera esquemas GraphQL automáticamente a partir de modelos de grafos, reduciendo el tiempo de desarrollo de APIs (Neo4j Inc, 2024).
- **Aplicaciones comunes**: Redes sociales, sistemas de recomendación (e.g., Netflix, Airbnb), detección de fraudes, y gestión de conocimiento en IA.

Cassandra

Descripción: Cassandra es una base de datos NoSQL distribuida orientada a columnas, diseñada para manejar grandes volúmenes de datos en entornos distribuidos sin un único punto de fallo. Según un informe de **DataStax (2024)**, Cassandra es utilizada por el **22% de las empresas Fortune 500** en aplicaciones de alta disponibilidad, como IoT, sistemas bancarios, y análisis en tiempo real, gracias a su arquitectura descentralizada (DataStax, 2024).

Ventajas clave:

- **Alto rendimiento en operaciones de escritura**: Soporta hasta **12,000 escrituras por segundo** en clústeres distribuidos, ideal para aplicaciones de gran escala (DataStax,

2024).

- **Escalabilidad horizontal prácticamente ilimitada**: Permite añadir nodos sin interrupciones, con latencias promedio de **1-3 ms** en configuraciones optimizadas (Apache Cassandra, 2024).
- **Arquitectura sin nodo maestro**: Garantiza tolerancia a fallos y alta disponibilidad en despliegues globales.
- **Aplicaciones comunes**: Telecomunicaciones, IoT, sistemas bancarios distribuidos, y análisis de logs en tiempo real.

"Apache Cassandra's decentralized architecture and linear scalability make it the go-to solution for mission-critical applications requiring high write throughput and fault tolerance across distributed environments" (DataStax, 2024).

Cassandra ha mejorado su compatibilidad con GraphQL mediante integraciones con DataStax Astra DB, que ofrece soporte nativo para APIs GraphQL en entornos serverless (DataStax, 2024)

1.5. Métricas de la norma ISO/IEC 25023

La serie de Normas Internacionales SQuaRE (Requisitos y Evaluación de la Calidad de Sistemas y Software) proporciona un marco global para medir y evaluar la calidad de productos y sistemas de software. Estas normas, parte de la familia ISO/IEC 2502n, incluyen estándares específicos diseñados para abordar diversos aspectos de la calidad, tales como:

• ISO/IEC 25020: Modelo y guía de referencia para la medición de la calidad.

• ISO/IEC 25021: Definición de elementos de medida de la calidad.

• ISO/IEC 25022: Evaluación de la calidad en el uso del software.

• ISO/IEC 25023: Medición de la calidad de sistemas y productos de software.

• ISO/IEC 25024: Evaluación de la calidad de los datos.

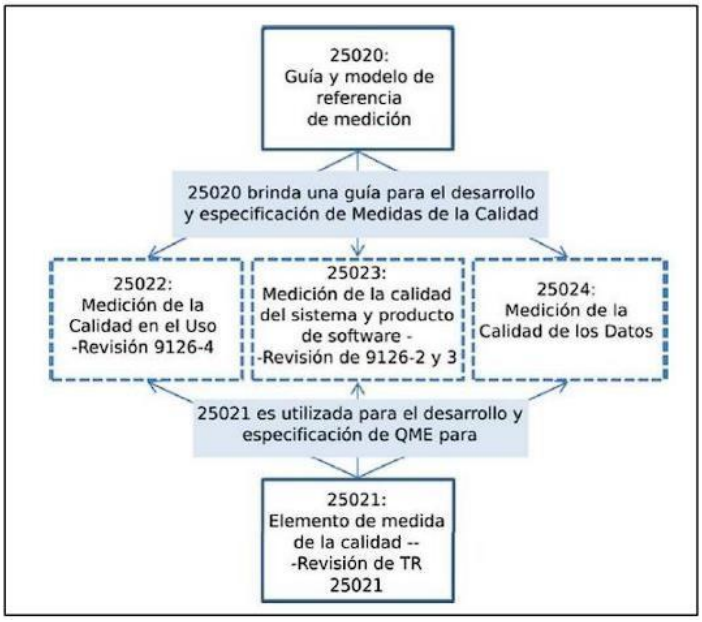
Estas normas ofrecen directrices estructuradas para garantizar que los productos de software cumplan con requisitos de calidad consistentes y medibles (ISO, 2024).

A continuación, en la figura 12 se muestra un cuadro donde se presentan todas las relaciones entre estas normas:

Figura 12

Estructura de la División de la Medición de la Calidad

Fuente: (Quinche Moran, 2021)



1.5.1. Medidas de eficiencia del desempeño

Para llevar a cabo esta investigación, se seleccionó la métrica de tiempo de respuesta de la norma ISO/IEC 25023, que forma parte de las medidas de eficiencia de desempeño definidas

en la serie SQuaRE (Requisitos y Evaluación de la Calidad de Sistemas y Software). Esta métrica evalúa el rendimiento de un sistema o producto de software en función del tiempo de respuesta y procesamiento bajo condiciones específicas, considerando el uso eficiente de recursos como CPU, memoria, almacenamiento y componentes de red. La métrica de tiempo de respuesta permite analizar cómo los recursos de hardware y software, incluyendo la configuración del sistema, impactan el desempeño, proporcionando una base cuantitativa para optimizar aplicaciones en entornos con demandas variables (ISO, 2023).

CAPÍTULO 2: DESARROLLO

En eta etapa del proyecto se llevó a cabo un experimento computacional con el propósito de medir la eficiencia en el acceso de datos entre bases de datos relacionales y no relacionales. El experimento se estructuró en dos componentes principales: por un lado, una API (Interfaz de Programación de Aplicaciones), y por otro, un cliente consumidor de dicha API. Dado que se evalúan dos tecnologías distintas de comunicación REST y GraphQL fue necesario implementar dos APIs independientes, una para cada tecnología, No obstante, el cliente desarrollado es único y está diseñado para gestionar manualmente la selección de la tecnología y base de datos que se utilizará en cada ejecución del experimento, lo que permite comparar de forma controlada el desempeño de ambas arquitecturas frente a diferentes motores de bases de datos.

2.1 Instalación y entorno de ejecución

Para la ejecución del experimento computacional se trabajó en un mismo entorno controlado, todas las bases de datos seleccionadas se instalaron en una maquina con las siguientes características:

- Marca: Lenovo
- Procesador: AMD Ryzen 7 7840 Hs
- Memoria RAM: 16 GB
- Sistema operativo: Windows 11
- Almacenamiento: SSD 512 GB

Sabiendo esto y con las bases de datos relacionales y no relacionales ya elegidas se

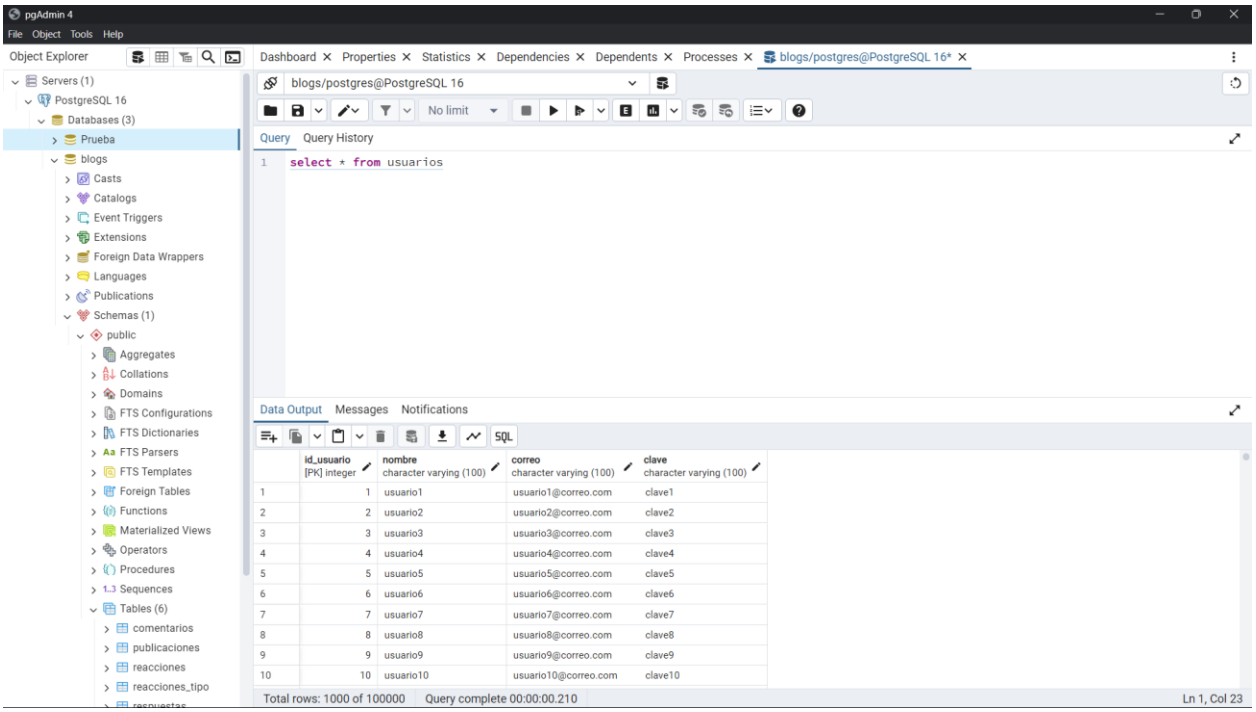
procedió a su respectiva instalación desde sus páginas oficiales.

- **PostgreSQL**
  - Fuente de descarga: https://www.postgresql.org/download/
  - Versión usada: PostgreSQL 16
  - Herramienta de cliente: pgAdmin 4

La Figura 13 muestra la interfaz del sistema de gestión de bases de datos PostgreSQL, evidenciando su instalación exitosa y disponibilidad operativa en el equipo.

Figura 13

Interfaz de PostgreSQL



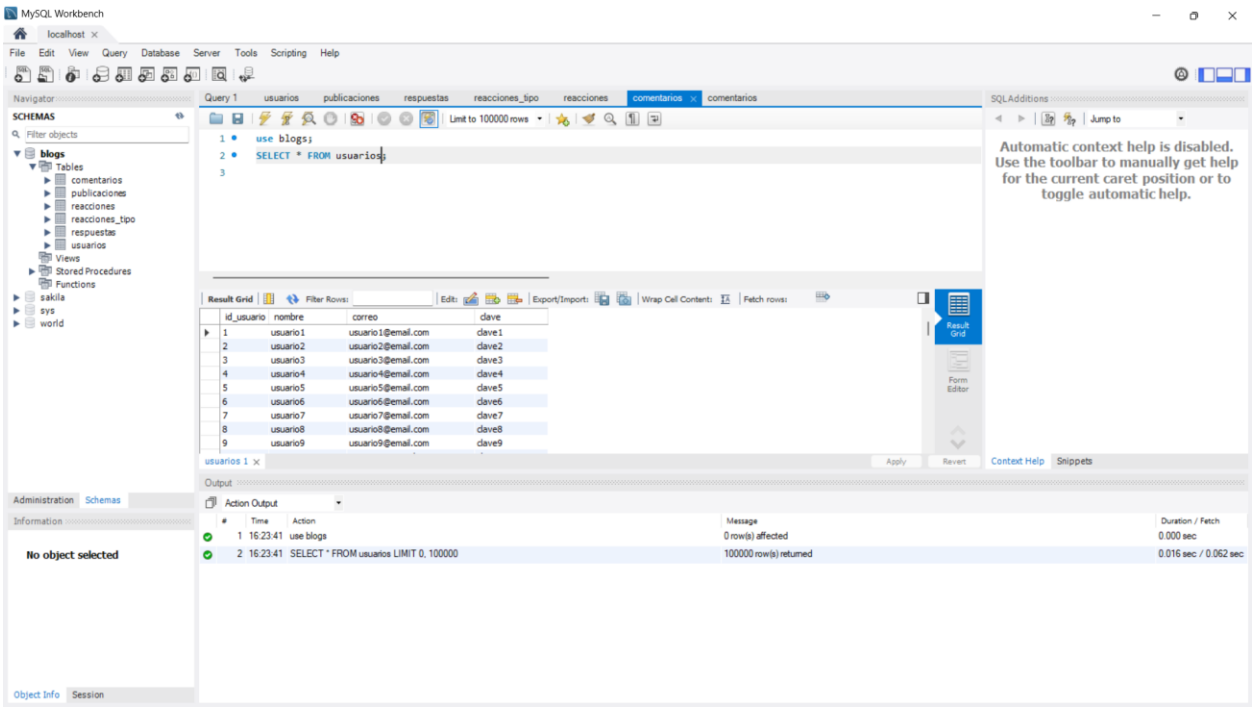
- **MySql**
  - Fuente de descarga: https://dev.mysql.com/downloads/

- Versión usada: MySql 8.0
- Herramienta de cliente: MySql Workbench

La Figura 14 muestra la interfaz del sistema de gestión de bases de datos MySQL, evidenciando su instalación exitosa y disponibilidad operativa en el equipo.

Figura 14

Interfaz de MySQL



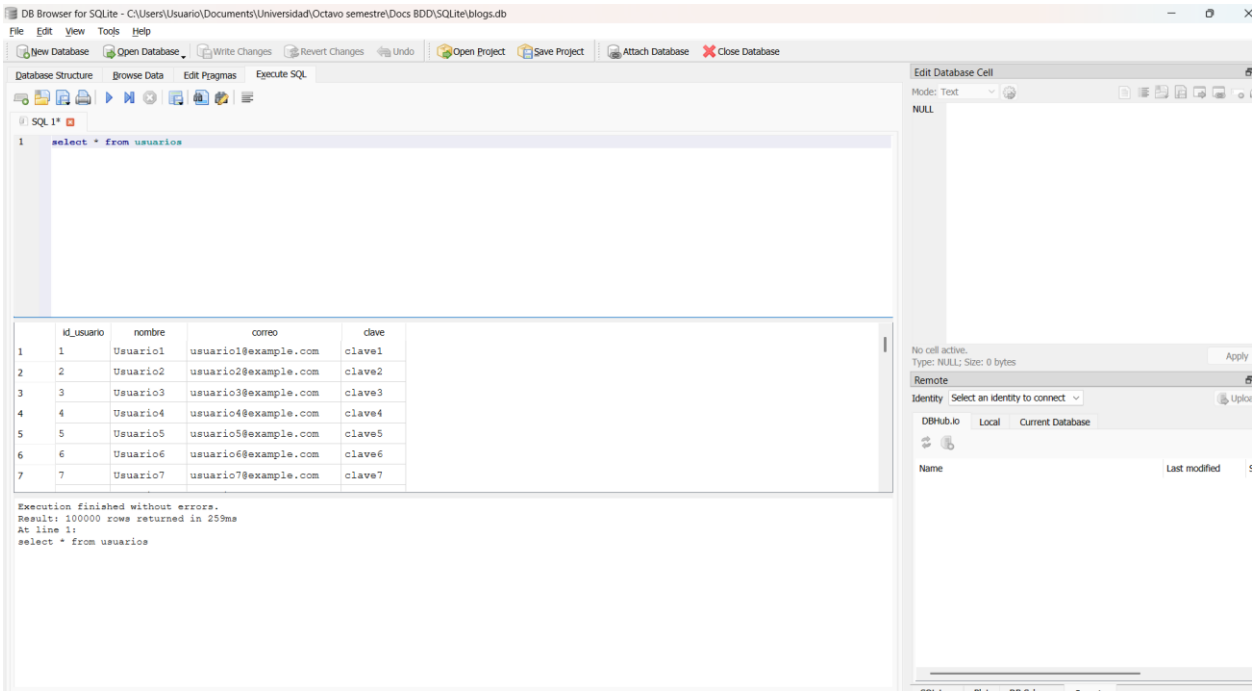
- **SQLite**
  - Fuente de descarga: https://sqlite.org/download.html
  - Versión usada: 3.13.1
  - Herramienta de cliente: DB Browser for SQLite.

La Figura 15 muestra la interfaz del sistema de gestión de bases de datos SqLite,

evidenciando su instalación exitosa y disponibilidad operativa en el equipo.

Figura 15

Interfaz de SqLite



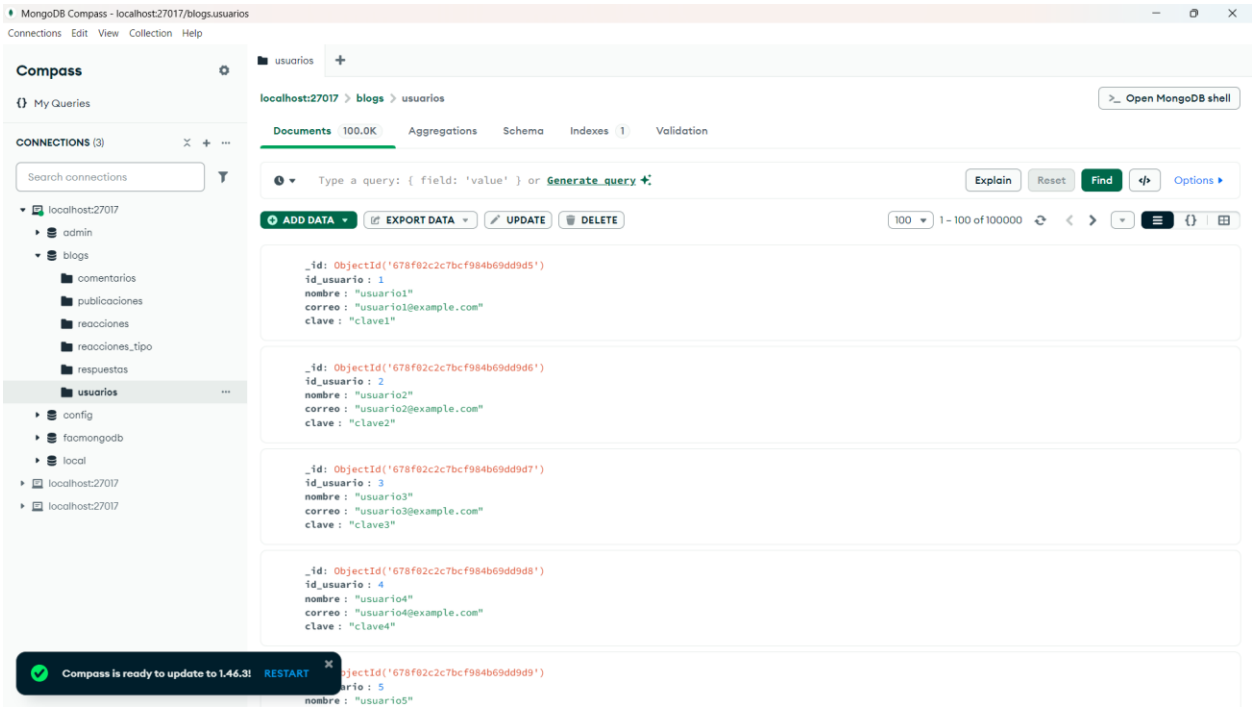
- **MongoDB**

    - Fuente de descarga: https://www.mongodb.com/try/download/community
    - Versión usada: 1.46.6
    - Herramienta de cliente: MongoDB Compass

La Figura 16 muestra la interfaz del sistema de gestión de bases de datos MongoDB, evidenciando su instalación exitosa y disponibilidad operativa en el equipo.

Figura 16

Interfaz de MongoDB



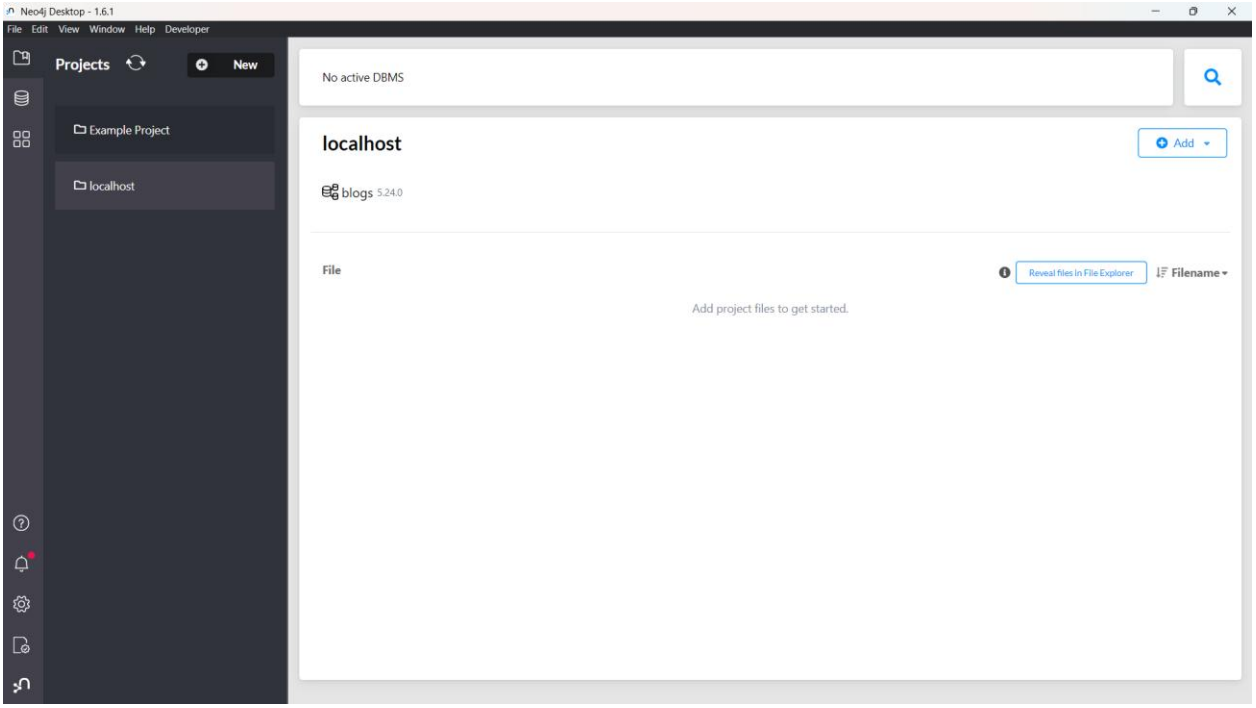
- **Neo4j**

    - Fuente de descarga: https://neo4j.com/download/
    - Versión usada: 1.6.2
    - Herramienta de cliente: Neo4j Browser

La Figura 17 muestra la interfaz del sistema de gestión de bases de datos Neo4j, evidenciando su instalación exitosa y disponibilidad operativa en el equipo.

Figura 17

Interfaz de Neo4j



- **Cassandra**

Para la instalación de esta base de datos se hizo la excepción de descargarla desde su fuente oficial, ya que, debe ser usada en conjunto con una versión estable de Python y Java.

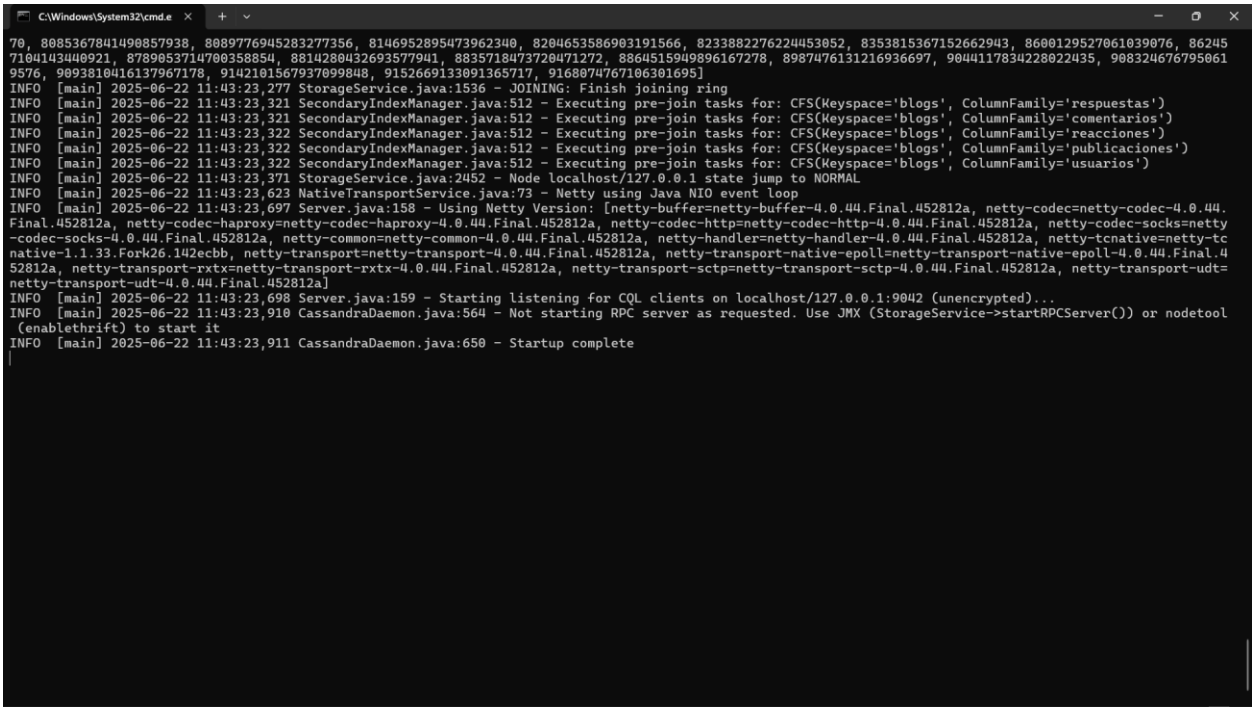
- **Fuente de descarga de Cassandra**: https://drive.google.com/file/d/1PhST8jGAR1D-SF-Q0yL4ggcC1oeIYuzw/view?pli=1
- **Fuente de descarga de Python:** https://drive.google.com/file/d/1m7iflcdoc7ueZGgPcqGT6rJKNt_tCrJN/view
- **Fuente de descarga del JDK:** https://drive.google.com/file/d/1_Je2fQG8w2ndFX0uMOOlUqz715HeqKV6/view
- **Versión de Cassandra usada**: Apache Cassandra 3.11.10
- **Versión de Pyhton usada:** Python 2.7

- **Versión de JDK usada**: 1.8.251
- Herramienta de cliente: CQLSH

La Figura 18 muestra la interfaz del sistema de gestión de bases de datos Cassandra, evidenciando su instalación exitosa y disponibilidad operativa en el equipo.

Figura 18

Interfaz de Cassandra



Para la API REST se utilizó **Node.js (v20.11)** con Express.js, mientras que la API GraphQL fue construida utilizando **Apollo Server** sobre el mismo entorno de ejecución. Ambas APIs fueron ejecutadas localmente en el mismo equipo, conectadas a cada motor mediante drivers oficiales y configuradas con una variable dbType que permitía identificar el motor objetivo para ser usada en el cliente.

2.2 Definición de las consultas.

2.2.1 Diseño del modelo de datos común

Para asegurar una evaluación equitativa y comparable entre las bases de datos seleccionadas, tanto relacionales como no relacionales, se definió un modelo de datos unificado que ayudará a realizar las consultas de manera similar para que los resultados sean lo más acertados.

Este modelo de datos simula un sistema de "blogs" el cual fue el más recomendado para el tipo de consultas que se buscaba realizar, cada base de datos creada en los distintos sistemas de gestión de base de datos tiene las mismas tablas con los mismos atributos, tipo de datos y cantidad de datos. A continuación, se describe las 6 entidades principales de las bases de datos:

- Usuarios: usuarios que representan a los autores principales.
- Publicaciones: contenido que es creado por cada usuario.
- Comentarios: interacciones escritas por diferentes usuarios a las publicaciones.
- Respuestas: son respuestas especificas a un comentario.
- Reacciones: son interacciones específicas como Me gusta, Me enoja, Me entristece, Me divierte, Me enoja
- Reacciones_tipo: entidad que almacena las 5 reacciones posibles.

La Tabla 3 presenta la estructura general de las entidades definidas para cada base de datos, detallando el nombre de las tablas junto con sus respectivos atributos. Esta organización permite identificar claramente la distribución de la información y la relación entre las entidades dentro del modelo de datos.

Tabla 3

Estructura de las entidades

Tabla	Atributos
usuarios	id_usuario (PK), nombre, correo, clave
publicaciones	id_publicacion (PK), id_usuario (FK), titulo, contenido, fecha_publicacion
comentarios	id_comentario (PK), id_publicacion (FK), texto, fecha_comentario
respuestas	id_respuesta (PK), id_comentario (FK), texto, fecha_respuesta
reacciones_tipo	id_tipo_reaccion (PK), nombre
reacciones	id_reaccion (PK), id_respuesta (FK), id_tipo_reaccion (FK), fecha_reaccion

Justificación del modelo

El modelo se seleccionó teniendo en cuenta el tipo de consultas que se realizarían durante el experimento computacional. Se estableció un flujo jerárquico, donde las búsquedas siempre comienzan en la entidad principal del usuario y descienden ordenadamente por los niveles inferiores: primero se accede a las publicaciones vinculadas a un usuario, luego a los comentarios sobre dichas publicaciones, luego a las respuestas a dichos comentarios y, finalmente, se obtienen las reacciones a cada respuesta. Esta disposición garantiza que todas las consultas tengan al menos cinco niveles de profundidad, lo que permite simular la navegación y la extracción de información en entornos reales en sistemas grandes y distribuidos, así como evaluar eficazmente las capacidades de cada motor de base de datos.

Por otro lado, esta arquitectura de datos representa con precisión un patrón común en aplicaciones sociales y plataformas de contenido, donde cada acción o consulta depende de una relación contextual previa. Esta secuencia descendente de relaciones no solo favorece el diseño de consultas anidadas en GraphQL, sino también la ejecución eficaz de uniones o relaciones en REST. Esta estrategia garantiza la equivalencia funcional entre las consultas implementadas en diferentes sistemas de bases de datos, lo que permite una evaluación más precisa de su rendimiento al gestionar estructuras de datos altamente relacionales o interconectadas. Esta configuración también facilita una adaptación consistente a los diversos paradigmas de almacenamiento utilizados, como documentos, gráficos y columnas.

2.2.2 Tipos de consultas definidas

Para evaluar la eficiencia del acceso a datos entre diferentes motores de bases de datos que utilizan arquitecturas REST y GraphQL, se estableció un formato de consulta estándar que reflejaba las operaciones típicas en sistemas de gestión de contenido y plataformas sociales. Estas consultas se implementaron mediante dos API simultáneas (una basada en REST y otra en GraphQL), lo que permite medir el rendimiento de cada tecnología con una funcionalidad equivalente.

Las consultas se organizaron en una jerarquía de cinco niveles que representa las entidades del modelo de datos: usuarios, publicaciones, comentarios, respuestas y reacciones. Cada nivel se creó para simular el acceso real a datos anidados. Por ejemplo, la consulta getUsuariosCantidad permite obtener un número específico de usuarios, mientras que getPostsCantidad accede a las publicaciones de un usuario, getComentariosCantidad recupera los

comentarios vinculados a una publicación, y así sucesivamente hasta el nivel de reacción.

También se introdujo una variable esencial en ambas API, denominada dbType, que permite la identificación dinámica del motor de base de datos que ejecutará la consulta (PostgreSQL, SQLite, MySQL, MongoDB, Neo4j o Cassandra). Esta variable fue clave para garantizar una lógica de ejecución consistente, facilitando así una comparación directa entre las distintas tecnologías bajo los mismos parámetros operativos.

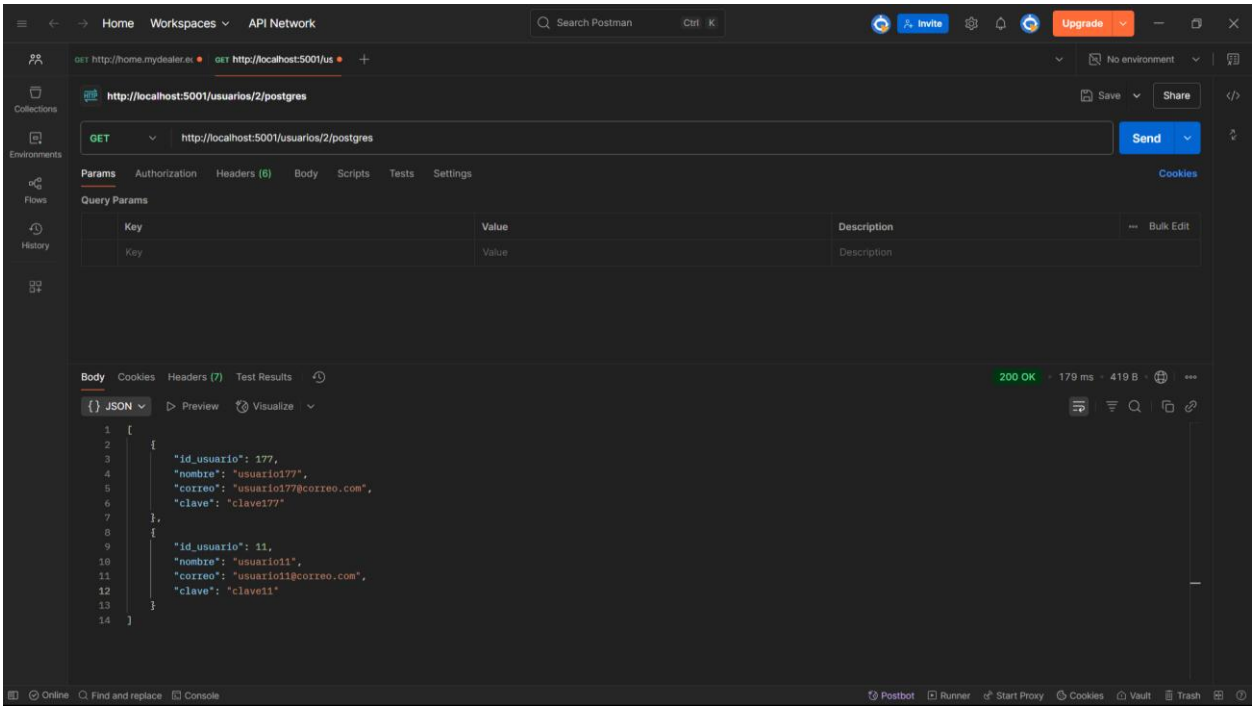
Para evaluar las consultas definidas en el experimento computacional, se empleó Postman, una herramienta especializada en la realización de consultas a interfaces de programación de aplicaciones (API). En este caso, se diseñó un ejemplo de consulta dirigida a la tabla de usuarios, siguiendo un procedimiento estructurado: primero, se describió la estructura de la tabla; posteriormente, se determinó el número total de usuarios; y, finalmente, se especificó la base de datos objetivo para la consulta. La estructura de la consulta se configuró de la siguiente manera:

/usuarios/:numUsr/:dbType

La Figura 19 muestra un ejemplo de la ejecución del método getUsuariosCantidad utilizando Postman, una herramienta ampliamente empleada para el consumo y prueba de APIs, se evidencia una respuesta exitosa, reflejada en la correcta obtención de los datos correspondientes a los usuarios consultados. Esta validación confirma la funcionalidad del método dentro de la arquitectura REST.

Figura 19

Postman, Ejecucion de getUsuariosCantidad en REST



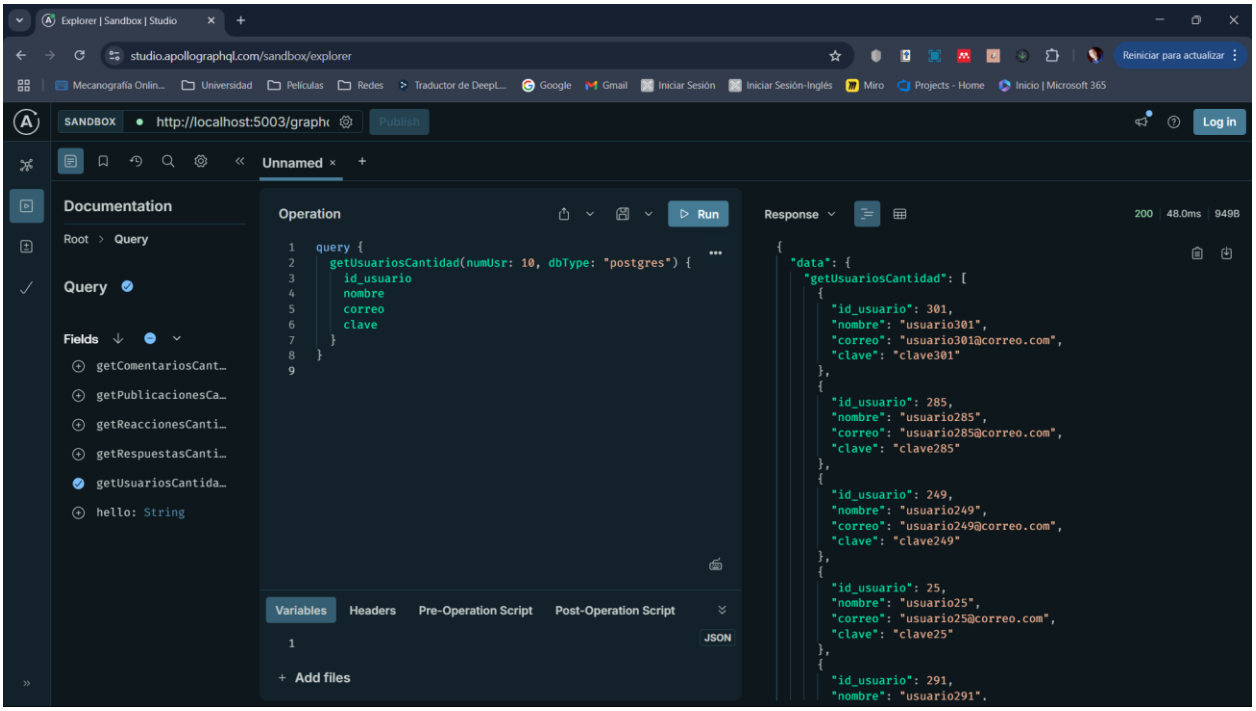
De manera similar, se implementaron consultas en la API GraphQL utilizando el servidor Apollo como plataforma para su ejecución. Para la consulta de usuarios, se empleó el método getUsuariosCantidad, definido en la API, especificando los parámetros requeridos: número de usuarios, que indica la cantidad de usuarios a consultar, y dbType, un parámetro obligatorio en todas las consultas que determina el tipo de base de datos (relacional o no relacional) a utilizar. La estructura de la consulta se configuró de la siguiente manera:

getUsuariosCantidad(numUsr, dbType)

La Figura 20 presenta un ejemplo de la ejecución del método getUsuariosCantidad mediante Apollo Server en el entorno de GraphQL. Se observa una respuesta satisfactoria, reflejada en la obtención precisa de los datos de los usuarios consultados. Esta ejecución valida el correcto funcionamiento del método dentro de la arquitectura GraphQL implementada.

Figura 20

Ejecución de getUsuariosCantidad en GraphQL usando Apollo Server



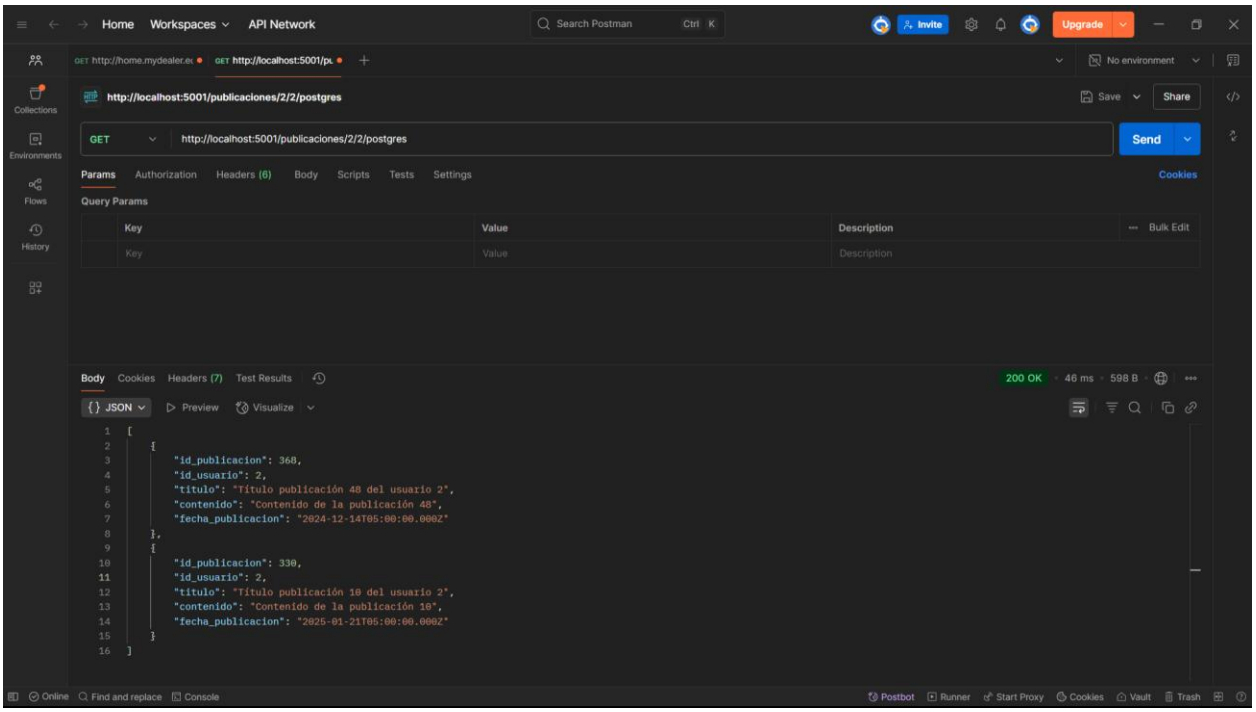
Para las siguientes consultas ya sea en REST o en GrapQL las consultas son parecidas y solamente se debe cambiar los parámetros de la consulta, como por ejemplo en el siguiente nivel sería de esta manera:

/publicaciones/:numUsr/idPub/:dbType

La Figura 21 muestra un ejemplo de la ejecución del método getPublicacionesCantidad utilizando Postman, se evidencia una respuesta exitosa, reflejada en la correcta obtención de los datos correspondientes a los usuarios consultados. Esta validación confirma la funcionalidad del método dentro de la arquitectura REST.

Figura 21

Postman, Ejecucion de getPublicacionesCantidad en REST

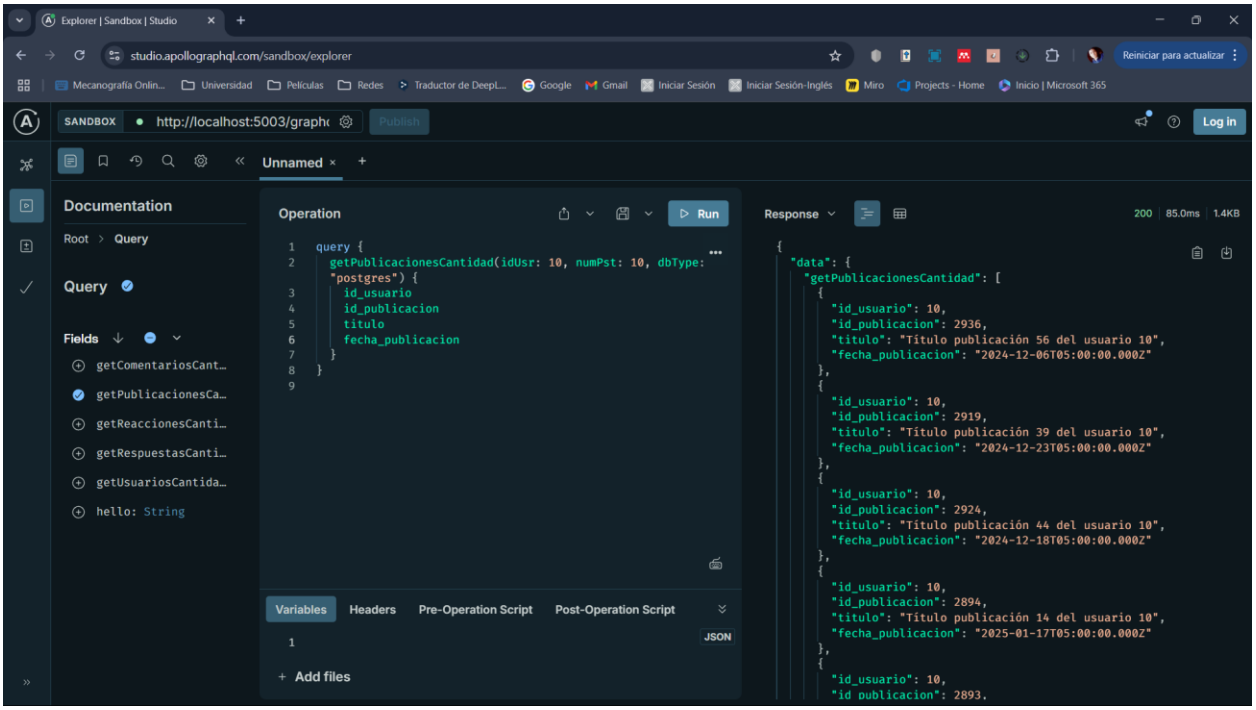


Y en GraphQL de esta manera:

La Figura 22 presenta un ejemplo de la ejecución del método getPublicacionesCantidad mediante Apollo Server en el entorno de GraphQL. Se observa una respuesta satisfactoria, reflejada en la obtención precisa de los datos de los usuarios consultados. Esta ejecución valida el correcto funcionamiento del método dentro de la arquitectura GraphQL implementada.

Figura 22

Ejecución de getPublicacionesCantidad en GraphQL usando Apollo Server



La Tabla 4 presenta las consultas definidas para cada nivel jerárquico del modelo de datos, desde usuarios hasta reacciones, implementadas en ambas arquitecturas: REST y GraphQL. Se incluyen los endpoints REST y las funciones equivalentes en GraphQL, cada una con sus parámetros para especificar la cantidad de registros y el contexto de la base de datos. La descripción resume la finalidad de cada consulta, enfocada en la obtención de datos asociados a cada entidad del modelo.

Tabla 4

Estructura de consultas

Nivel	REST	GraphQL	Descripción
1	/usuarios/:numUsr/:dbType	getUsuariosCantidad(numUsr, dbType)	Obtener N usuarios
2	/publicaciones/:idUsr/:numPst/:dbType	getPostsCantidad(idUsr, numPst, dbType)	Obtener publicaciones de un usuario
3	/comentarios/:idPub/:numCmt/:dbType	getComentariosCantidad(idPub,numCmt, dbType)	Obtener comentarios de una publicación
4	/respuestas/:idCmt/:numRes/:dbType	getRespuestasCantidad(idCmt, numRes, dbType)	Obtener respuestas de un comentario
5	/reacciones/:idRes/:numReac/:dbType	getReaccionesCantidad(idRes, numReac, dbType)	Obtener reacciones de una respuesta

2.2.3 Criterios de equivalencia entre consultas

Para garantizar una comparación objetiva y equitativa entre las arquitecturas REST y GraphQL en el experimento computacional, se establecieron criterios rigurosos de equivalencia funcional y lógica en las consultas implementadas en ambas APIs. El objetivo fue asegurar que, independientemente de la arquitectura o el motor de base de datos utilizado (relacional o no relacional), las consultas respondieran a la misma lógica funcional y devolvieran datos equivalentes en estructura y contenido.

Equivalencia Funcional entre APIs

El primer criterio definido fue la equivalencia funcional entre las consultas de ambas tecnologías. Cada endpoint REST tiene una contraparte exacta en la API GraphQL. Por ejemplo, la ruta REST GET /comentarios/:idPub/:numCmt/:dbType realiza la misma operación que la consulta GraphQL getComentariosCantidad(idPub, numCmt, dbType), recuperando un número

específico de comentarios asociados a una publicación determinada. Este principio de paridad se aplicó a los cinco niveles de consulta definidos —desde la obtención de usuarios hasta las reacciones— permitiendo una comparación directa entre ambas arquitecturas.

Consistencia en Parámetros y Lógica de Selección

Para garantizar la consistencia de los resultados, se emplearon los mismos parámetros de búsqueda en ambas APIs, independientemente del motor de base de datos. Por ejemplo, para consultar comentarios se utilizaron los parámetros idPublicacion (identificador de la publicación), numCmt (número de comentarios a recuperar) y dbType (tipo de base de datos). El parámetro dbType desempeña un papel clave al especificar el motor de base de datos (por ejemplo, relacional como PostgreSQL, o no relacional como MongoDB o Neo4j) que ejecutará la consulta, permitiendo una abstracción que minimiza las variaciones en la implementación y asegura que las diferencias en rendimiento se atribuyan exclusivamente a las características de cada motor.

Asimismo, se mantuvo una equivalencia en la lógica de selección de datos. En ambas APIs, se implementaron técnicas comparables para filtrar y limitar resultados, como ORDER BY RANDOM () en bases de datos relacionales, $sample en MongoDB, o LIMIT combinado con selección aleatoria en Neo4j. Esta coherencia aseguró que el volumen y la estructura de los datos devueltos fueran comparables entre motores y arquitecturas.

Implementación de Consultas

Las consultas REST se probaron utilizando Postman, una herramienta especializada para interactuar con APIs, mientras que las consultas GraphQL se ejecutaron mediante el servidor

Apollo. Por ejemplo, para consultar la cantidad de usuarios, se empleó el método getUsuariosCantidad(numeroUsuarios, dbType) en ambas APIs, especificando el número de usuarios a recuperar y el tipo de base de datos. La estructura de las consultas se diseñó para garantizar que los resultados fueran idénticos en contenido y formato, independientemente de la tecnología utilizada.

Validación de Resultados

A lo largo del desarrollo, se realizaron pruebas manuales y automáticas para validar la consistencia entre las respuestas de las APIs REST y GraphQL. Estas pruebas verificaron que las respuestas fueran equivalentes en contenido, estructura y comportamiento, cumpliendo con los requisitos para un experimento computacional justo, replicable y robusto. La utilización de la variable dbType en ambas APIs facilitó la reutilización de la lógica de consulta, minimizando las diferencias en la implementación y asegurando la comparabilidad de los resultados.

2.2.4 Consultas preparadas y optimizadas por motor

Para garantizar un rendimiento excelente y una comparación justa entre las consultas implementadas en APIs REST y GraphQL, se realizaron ajustes en las operaciones según las particularidades de cada tipo de motor de base de datos, tanto relacional como no relacional. El objetivo fue maximizar la eficiencia de cada sistema de gestión, reduciendo cuellos de botella sin sacrificar la lógica funcional de las consultas.

En lo que respecta a las bases de datos relacionales, se implementaron tácticas de optimización personalizadas para cada motor. En PostgreSQL y MySQL, se establecieron índices en tablas con grandes volúmenes de datos, como reacciones y respuestas, enfocándose en

campos clave como id_respuesta e id_reaccion. Esto no solo aceleró las búsquedas, sino que también disminuyó el tiempo de ejecución de las consultas comunes. Se aplicaron cláusulas como LIMIT, ORDER BY RANDOM () y WHERE ... IS NOT NULL, calibradas cuidadosamente para reducir la carga computacional. En el caso de SQLite, dada su naturaleza como base de datos embebida, se priorizaron consultas ligeras que se ejecutan en memoria para asegurar una evaluación justa de su rendimiento.

En MongoDB, se desarrollaron agregaciones eficientes utilizando operadores como $match y $sample, aprovechando su capacidad para manejar grandes volúmenes de datos en estructuras no relacionales. Se generaron índices en las tablas de reacciones y respuestas sobre campos como id_respuesta y fecha_reaccion, evitando escaneos completos de las colecciones.

En el caso de Neo4j, las consultas fueron optimizadas mediante el uso eficaz de patrones en el lenguaje Cypher, priorizando relaciones directas y estableciendo índices en los nodos reacciones y respuestas, específicamente en atributos como id_respuesta y fecha_reaccion. Se evitaron operaciones costosas, como OPTIONAL MATCH o UNWIND innecesarios, y se formularon patrones de búsqueda con rutas claras en el grafo. Además, se implementó la cláusula LIMIT para simular situaciones reales de paginación o muestreo, mejorando el rendimiento en consultas complejas.

En Cassandra, la optimización se centró en la creación de tablas con claves primarias compuestas, como (id_usuario, id_publicacion), adaptadas a su arquitectura orientada a columnas. Se descartaron operaciones no soportadas como JOINs o filtros WHERE sobre columnas no indexadas sin ALLOW FILTERING. En las tablas más pesadas (reacciones y respuestas), se generaron índices secundarios sobre campos como fecha_respuesta y

fecha_reaccion con fines experimentales, evaluando su efecto en el rendimiento. Este enfoque permitió mantener la coherencia en la lógica de las consultas, aprovechando al mismo tiempo las capacidades inherentes del motor.

Todas las consultas fueron verificadas en entornos de prueba para asegurar que no existieran errores, tiempos de respuesta inadecuados o discrepancias en los resultados. Este procedimiento garantizó que cada sistema de gestión de bases de datos operara en condiciones ideales, permitiendo una comparación justa y reproducible durante el experimento. La indexación selectiva de las tablas más grandes (reacciones y respuestas) y la estandarización de la lógica de las consultas aseguraron que las variaciones en el rendimiento reflejaran únicamente las particularidades de cada motor, sin alteraciones causadas por la implementación.

2.3 Ejecución del experimento.

2.3.1 Cliente del experimento

El cliente del experimento fue desarrollado como una aplicación Node.js que tiene como objetivo centralizar, ejecutar, controlar y registrar las pruebas sobre las distintas combinaciones entre motores de base de datos y arquitecturas REST y GraphQL. Este cliente actúa como **orquestador del experimento computacional**, asegurando la coherencia en la ejecución y facilitando la recolección automática de resultados.

El cliente está compuesto por tres módulos principales:

- **index.js**: Controlador maestro que coordina manualmente la ejecución de todos los casos de uso.
- **experimentoComputacional_Rest.js** y **experimentoComputacional_Graphql.js**:

Módulos encargados de realizar las consultas de cada nivel para REST y GraphQL, respectivamente. Incluyen funciones como obtenerDatosTercerNivel() y obtenerDatosQuintoNivel(), que acceden secuencialmente a los datos, respetando la jerarquía usuarios → publicaciones → comentarios → respuestas → reacciones.

Cada función de estos módulos sigue un mismo patrón de ejecución:

1. Se registra un **tiempo inicial (tInicial)** antes de iniciar las consultas.
2. Se recorren los identificadores obtenidos en el nivel anterior y se construyen URLs dinámicas.
3. Se ejecutan consultas individuales por cada ID (evitando cuellos de botella por grandes uniones).
4. Se registra el **tiempo final (tFinal)** y se calcula el tiempo total de respuesta (tFinal - tInicial), cumpliendo así con la métrica de la norma **ISO/IEC 25023**.

El archivo index.js es el responsable de invocar las funciones de cada caso de uso con distintos volúmenes de datos. Por ejemplo:

```
await guardarDatos(await CasodeUso1G(urlEnUso, 1, 'GraphQL', baseDeDatosEnUSo));
```

Se especifica el caso de uso que se va a usar, así como también la letra “G” que hace referencia a Graphql, también se especifica que tecnología se va a usar y una variable que me indica qué motor de base de datos se usará para dicha prueba. Este patrón se repite para los cinco niveles, aumentando progresivamente la cantidad de datos por nivel (1, 10, 100, 1000, 10000 y 100000.). Para cada consulta ejecutada, la función guardarDatos() abre un archivo Excel (Matriz_TiemposRespuesta.xlsx) y registra automáticamente los tiempos de respuesta y la

arquitectura utilizada (REST o GraphQL), con ayuda de la librería xlsx-populate.

Características técnicas destacadas del cliente:

- **Medición precisa de tiempo de respuesta**, aplicando new Date() antes y después del bloque de ejecución, conforme a la ISO/IEC 25023.
- **Automatización completa** de los cinco casos de uso en ambas arquitecturas (REST y GraphQL).
- **Control de arquitectura y motor de base de datos** gracias al parámetro dbType, que permite ejecutar las mismas consultas sobre diferentes motores sin alterar la lógica del cliente.
- **Registro automatizado en Excel**, con identificación de arquitectura, iteración y volumen de datos procesado.
- **Iteración y escalamiento**, permitiendo observar cómo varía el rendimiento según la cantidad de datos y profundidad de las consultas.

2.3.2 Consulta de datos masivos

Con el propósito de evaluar el desempeño de los sistemas bajo condiciones de alta demanda, el experimento computacional se diseñó para ejecutar consultas sobre grandes volúmenes de datos, emulando escenarios reales en los que los motores de bases de datos deben procesar información masiva de manera eficiente. Este enfoque permitió analizar el comportamiento de cada motor frente a cargas de trabajo escalables y complejas, proporcionando una visión clara de su rendimiento en contextos operativos exigentes.

La ejecución de las consultas se controló mediante un script centralizado implementado

en el archivo index.js. Este script orquesta múltiples iteraciones para cada caso de uso, incrementando progresivamente el volumen de datos recuperados. Se definieron cargas de prueba con tamaños de 1, 10, 100, 1000, 10 000 y hasta 100 000 registros, lo que permitió evaluar cómo varía el rendimiento en función de la cantidad de datos procesados y la complejidad de las consultas. Para garantizar la reproducibilidad, cada consulta se ejecutó varias veces por carga, registrando métricas como el tiempo de respuesta y el consumo de recursos, lo que facilitó un análisis estadístico robusto de los resultados.

El experimento abarcó cinco niveles jerárquicos de datos: usuarios, publicaciones, comentarios, respuestas y reacciones. Para cada nivel, se diseñaron combinaciones específicas de consultas que procesaran aproximadamente 100 000 registros, asegurando una carga de trabajo representativa y comparable entre niveles. La distribución de datos se estructuró en una arquitectura piramidal descendente, donde los niveles superiores (como usuarios) contenían menos entidades que los niveles inferiores (como reacciones), reflejando un modelo realista de aplicaciones sociales o de interacción masiva.

La Tabla 5 presenta la distribución de datos consultados según cada caso de uso. Como se observa, la cantidad de datos varía en función del nivel jerárquico; a medida que se avanza en los casos de uso, se consultan menos registros, pero provenientes de un mayor número de tablas o entidades relacionadas.

Tabla 5

Resumen de distribución

Nivel de Consulta	Usuarios	Publicaciones	Comentarios	Respuestas	Reacciones	Total aproximado
Nivel 1	100 000	/	/	/	/	100 000
Nivel 2	316	316	/	/	/	99 856
Nivel 3	46	46	47	/	/	99 452
Nivel 4	18	18	18	17	/	99 144
Nivel 5	10	10	10	10	10	100 000

Esta estructura piramidal permitió equilibrar el volumen de datos entre niveles, asegurando que las consultas fueran representativas de escenarios reales mientras se mantenía una carga computacional manejable para los motores de bases de datos. Además, se optimizaron las consultas en los niveles inferiores (respuestas y reacciones) mediante la creación de índices en campos clave, como se describió anteriormente, para mitigar el impacto de procesar grandes volúmenes de datos.

2.3.3 Ejecución de consultas

La ejecución de las consultas se gestionó de forma centralizada mediante la clase index.js, la cual orquesta todo el proceso experimental. Esta clase es responsable de recorrer los cinco niveles definidos (usuarios, publicaciones, comentarios, respuestas y reacciones), que se encuentran implementados como **casos de uso individuales**, tanto para la arquitectura REST como para GraphQL.

Cada **caso de uso** se prueba con **seis tamaños de carga diferentes**: 1, 10, 100, 1 000, 10 000 y 100 000 registros. De este modo, se obtiene una visión clara de cómo escala el rendimiento de cada combinación de base de datos y arquitectura frente al aumento del volumen de datos.

Para cada combinación de arquitectura (REST y GraphQL), base de datos (PostgreSQL,

MySQL, SQLite, MongoDB, Cassandra y Neo4j) y tamaño de carga, **se realizaron tres iteraciones** consecutivas con el objetivo de reducir el efecto de posibles fluctuaciones en el entorno de ejecución. La clase index.js ejecuta cada iteración y posteriormente llama a la función guardarDatos(), que se encarga de almacenar los resultados en un documento de Excel estructurado.

El documento generado, Matriz_TiemposRespuesta.xlsx, contiene:

- Hojas diferenciadas para REST y GraphQL.
- Registros ordenados por caso de uso, volumen de datos, arquitectura y motor de base de datos.
- Los tiempos de respuesta de cada iteración (en milisegundos).
- Todos los parámetros utilizados en la consulta.

Cada fila en el archivo Excel representa una ejecución concreta, lo que permite realizar análisis comparativos detallados, tanto en forma horizontal (entre arquitecturas) como vertical (entre motores de base de datos y niveles de profundidad).

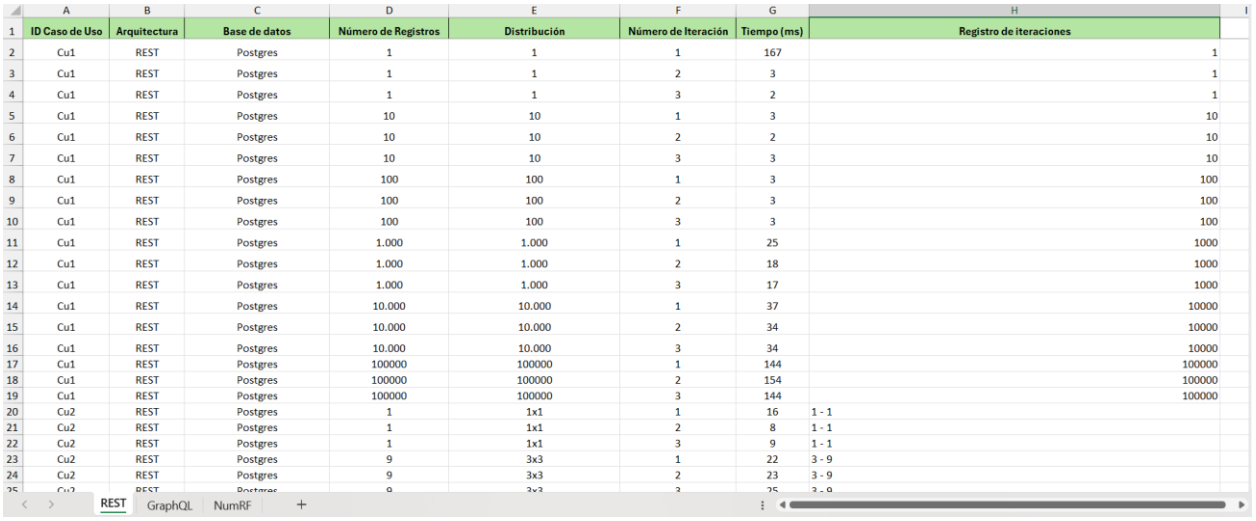
Este enfoque estructurado asegura que:

- Las pruebas son **reproducibles**.
- Los datos recolectados tienen **validez estadística** mínima.
- El experimento cumple con los requisitos planteados en los objetivos de la tesis, particularmente con el uso de la métrica de **tiempo de respuesta** definida en la norma **ISO/IEC 25023**.

En la figura 23 se evidencia todos los resultados que se han guardado en el documento de Excel, con los resultados para cada base de datos usando la arquitectura REST.

Figura 23

Hoja de Excel con los resultados de REST



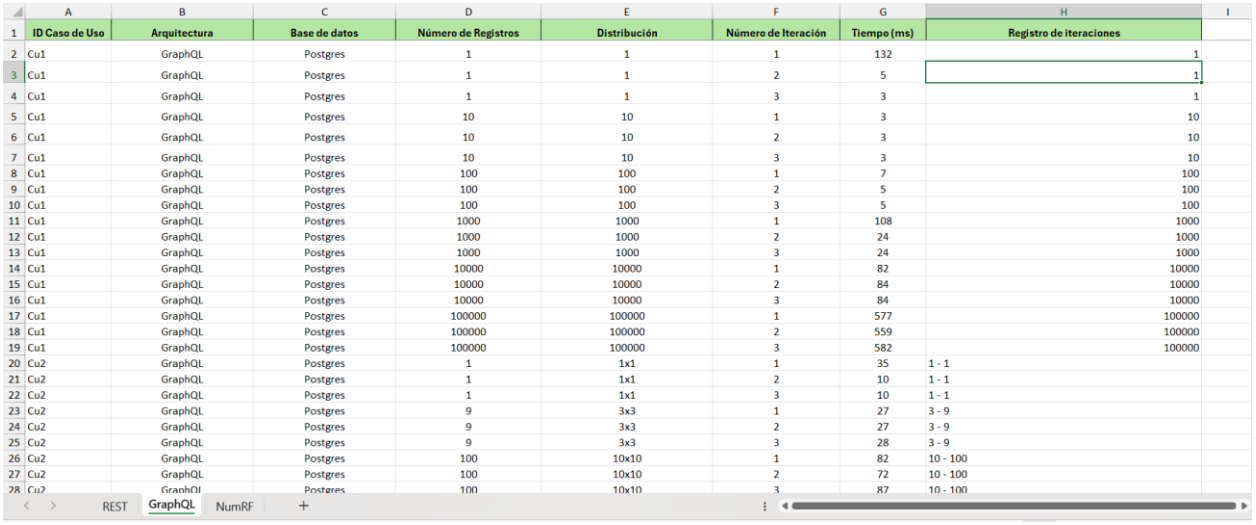
ID Caso de Uso	Arquitectura	Base de datos	Número de Registros	Distribución	Número de Iteración	Tiempo (ms)	Registro de iteraciones
Cu1	REST	Postgres	1	1	1	167	1
Cu1	REST	Postgres	1	1	2	3	1
Cu1	REST	Postgres	1	1	3	2	1
Cu1	REST	Postgres	10	10	1	3	10
Cu1	REST	Postgres	10	10	2	2	10
Cu1	REST	Postgres	10	10	3	3	10
Cu1	REST	Postgres	100	100	1	3	100
Cu1	REST	Postgres	100	100	2	3	100
Cu1	REST	Postgres	100	100	3	3	100
Cu1	REST	Postgres	1.000	1.000	1	25	1000
Cu1	REST	Postgres	1.000	1.000	2	18	1000
Cu1	REST	Postgres	1.000	1.000	3	17	1000
Cu1	REST	Postgres	10.000	10.000	1	37	10000
Cu1	REST	Postgres	10.000	10.000	2	34	10000
Cu1	REST	Postgres	10.000	10.000	3	34	10000
Cu1	REST	Postgres	100000	100000	1	144	100000
Cu1	REST	Postgres	100000	100000	2	154	100000
Cu1	REST	Postgres	100000	100000	3	144	100000
Cu2	REST	Postgres	1	1x1	1	16	1 - 1
Cu2	REST	Postgres	1	1x1	2	8	1 - 1
Cu2	REST	Postgres	1	1x1	3	9	1 - 1
Cu2	REST	Postgres	9	3x3	1	22	3 - 9
Cu2	REST	Postgres	9	3x3	2	23	3 - 9

REST GraphQL NumRF +

En la figura 24 se evidencia todos los resultados que se han guardado en el documento de Excel, con los resultados para cada base de datos usando la arquitectura GraphQL.

Figura 24

Hoja de Excel con los resultados de GraphQL



ID Caso de Uso	Arquitectura	Base de datos	Número de Registros	Distribución	Número de Iteración	Tiempo (ms)	Registro de iteraciones
Cu1	GraphQL	Postgres	1	1	1	132	1
Cu1	GraphQL	Postgres	1	1	2	5	1
Cu1	GraphQL	Postgres	1	1	3	3	1
Cu1	GraphQL	Postgres	10	10	1	3	10
Cu1	GraphQL	Postgres	10	10	2	3	10
Cu1	GraphQL	Postgres	10	10	3	3	10
Cu1	GraphQL	Postgres	100	100	1	7	100
Cu1	GraphQL	Postgres	100	100	2	5	100
Cu1	GraphQL	Postgres	100	100	3	5	100
Cu1	GraphQL	Postgres	1000	1000	1	108	1000
Cu1	GraphQL	Postgres	1000	1000	2	24	1000
Cu1	GraphQL	Postgres	1000	1000	3	24	1000
Cu1	GraphQL	Postgres	10000	10000	1	82	10000
Cu1	GraphQL	Postgres	10000	10000	2	84	10000
Cu1	GraphQL	Postgres	10000	10000	3	84	10000
Cu1	GraphQL	Postgres	100000	100000	1	577	100000
Cu1	GraphQL	Postgres	100000	100000	2	559	100000
Cu1	GraphQL	Postgres	100000	100000	3	582	100000
Cu2	GraphQL	Postgres	1	1x1	1	35	1 - 1
Cu2	GraphQL	Postgres	1	1x1	2	10	1 - 1
Cu2	GraphQL	Postgres	1	1x1	3	10	1 - 1
Cu2	GraphQL	Postgres	9	3x3	1	27	3 - 9
Cu2	GraphQL	Postgres	9	3x3	2	27	3 - 9
Cu2	GraphQL	Postgres	9	3x3	3	28	3 - 9
Cu2	GraphQL	Postgres	100	10x10	1	82	10 - 100
Cu2	GraphQL	Postgres	100	10x10	2	72	10 - 100

REST GraphQL NumRF +

CAPÍTULO 3: RESULTADOS

En este capítulo, se describe un experimento controlado diseñado conforme a los principios metodológicos establecidos por Wohlin (2012) en Experimentation in Software Engineering, con el propósito de responder a la pregunta de investigación: **¿Qué sistema de gestión de bases de datos ofrece un mejor rendimiento en función de la arquitectura utilizada (REST o GraphQL)?** El experimento se centró en evaluar el desempeño de seis motores de bases de datos ampliamente adoptados en la industria del software: PostgreSQL, MySQL, SQLite, MongoDB, Neo4j y Cassandra. La métrica principal de evaluación fue el tiempo de respuesta, alineada con la norma ISO/IEC 25023, que define los criterios de rendimiento como un atributo clave de la calidad del software.

3.1 Análisis de resultados

Para el análisis de los resultados, se utilizó un documento en Excel que recopiló los tiempos de respuesta registrados para cada caso de uso definido en el experimento computacional. Con el fin de garantizar la precisión de las mediciones y la calidad de los datos, se implementó un proceso automatizado que ejecutó tres iteraciones por cada caso de uso. Este enfoque permitió obtener promedios robustos de los tiempos de respuesta, minimizando variaciones derivadas de factores externos, como la carga del sistema, y asegurando la fiabilidad estadística de los resultados.

3.1.1 Arquitectura REST

- SQL

Para el análisis de los resultados en esta sección, se evaluaron los motores de bases de datos relacionales (PostgreSQL, MySQL y SQLite) considerando la estructura del experimento

computacional. Este experimento se diseñó con cinco casos de uso por motor, cada uno ejecutado en tres iteraciones automatizadas para garantizar la fiabilidad de las mediciones. Se calculó el promedio de los tiempos de respuesta de las tres iteraciones para cada caso de uso, que permite una comparación precisa del rendimiento entre los motores SQL.

La Tabla 6 presenta los resultados de la ejecución de los casos de uso sobre la base de datos PostgreSQL bajo la arquitectura REST, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 6

Resultados de PostgreSQL en REST

Nro.	**Caso de uso**	**Número de registros/tiempo (ms)**						**Promedio (s)**
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	57	3	3	20	35	147	0,44
2	CU-2	11	23	73	211	625	2504	5,75
3	CU-3	12	21	74	178	640	3023	6,58
4	CU-4	14	28	68	264	1034	6945	13,92
5	CU-5	20	35	105	308	1463	11175	21,84

La Tabla 7 presenta los resultados de la ejecución de los casos de uso sobre la base de datos MySQL bajo la arquitectura REST, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 7

Resultados de MySql en REST

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	12	3	4	62	79	158	0,53
2	CU-2	6	10	28	76	227	888	2,06
3	CU-3	8	14	53	155	559	3204	6,66
4	CU-4	9	24	64	283	1245	8587	17
5	CU-5	18	43	126	419	1831	12421	24,76

La Tabla 8 presenta los resultados de la ejecución de los casos de uso sobre la base de datos SqLite bajo la arquitectura REST, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 8

Resultados de SqLite en REST

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	14	2	2	69	57	595	1,23
2	CU-2	94	34	107	316	1110	3646	8,85
3	CU-3	18	30	88	205	655	2937	6,56
4	CU-4	16	37	67	267	947	6075	12,35
5	CU-5	16	37	101	279	1245	9473	18,59

- NoSql

La Tabla 9 presenta los resultados de la ejecución de los casos de uso sobre la base de datos MongoDB bajo la arquitectura REST, incluyendo el promedio general calculado a partir de

tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 9

Resultados de MongoDB en REST

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	10	100	1000	10000	100000	
1	CU-1	55	53	66	52	172	494	1,49
2	CU-2	49	54	79	129	299	1581	3,65
3	CU-3	51	66	99	194	663	3786	8,10
4	CU-4	52	65	101	339	1363	8532	17,42
5	CU-5	65	87	181	490	2391	16667	33,14

La Tabla 10 presenta los resultados de la ejecución de los casos de uso sobre la base de datos Neo4j bajo la arquitectura REST, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 10

Resultados de Neo4j en REST

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	10	100	1000	10000	100000	
1	CU-1	20	21	14	178	141	634	1,68
2	CU-2	15	29	68	176	457	3518	7,11
3	CU-3	11	25	113	329	1541	5912	13,22
4	CU-4	9	31	80	382	1772	11470	22,91
5	CU-5	14	37	133	464	2615	18600	36,44

La Tabla 11 presenta los resultados de la ejecución de los casos de uso sobre la base de datos Cassandra bajo la arquitectura REST, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 11

Resultados de Cassandra en REST

Nro.	**Caso de uso**	**Número de registros/tiempo (ms)**						**Promedio (s)**
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	13	20	151	1431	1151	415	5,301
2	CU-2	8	20	57	157	524	2146	4,85
3	CU-3	10	26	115	338	1649	7686	16,45
4	CU-4	13	42	121	658	3051	20241	40,21
5	CU-5	18	61	228	914	4815	34659	67,82

3.1.2 Arquitectura GraphQL

- SQL

La Tabla 12 presenta los resultados de la ejecución de los casos de uso sobre la base de datos PostgreSQL bajo la arquitectura GraphQL, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 12

Resultados de PostgreSQL en GraphQL

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	47	3	6	52	83	573	1,27
2	CU-2	18	27	80	241	765	3227	7,26
3	CU-3	17	27	98	270	996	4759	10,28
4	CU-4	15	36	87	363	1469	9933	19,84
5	CU-5	21	41	135	410	2006	14277	28,15

La Tabla 13 presenta los resultados de la ejecución de los casos de uso sobre la base de datos MySQL bajo la arquitectura GraphQL, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 13

Resultados de MySql en GraphQL

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	25	3	6	119	111	547	1,35
2	CU-2	32	17	37	110	362	1718	3,79
3	CU-3	10	17	78	230	914	4272	9,20
4	CU-4	9	30	78	360	1804	10133	20,69
5	CU-5	16	52	142	436	2232	15631	30,85

La Tabla 14 presenta los resultados de la ejecución de los casos de uso sobre la base de datos SqLite bajo la arquitectura GraphQL, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 14

Resultados de SqLite en GraphQL

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	18	3	5	71	100	1033	2,05
2	CU-2	35	55	139	358	1192	4126	9,84
3	CU-3	16	34	100	247	989	4463	9,75
4	CU-4	16	43	81	353	1485	8961	18,23
5	CU-5	18	46	122	385	2001	13887	27,43

- NoSQL

La Tabla 15 presenta los resultados de la ejecución de los casos de uso sobre la base de datos MongoDB bajo la arquitectura GraphQL, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 15

Resultados de MongoDB en GraphQL

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	68	44	47	48	198	893	2,17
2	CU-2	52	56	80	155	412	2113	4,78
3	CU-3	73	68	137	386	1631	7871	16,02
4	CU-4	50	77	119	431	1762	10809	22,08
5	CU-5	67	102	176	484	2370	17288	34,14

La Tabla 16 presenta los resultados de la ejecución de los casos de uso sobre la base de datos Neo4j bajo la arquitectura GraphQL, incluyendo el promedio general calculado a partir de

tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 16

Resultados de Neo4j en GraphQL

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	17	5	16	133	153	1074	2,33
2	CU-2	10	16	43	124	382	3347	6,54
3	CU-3	10	19	91	286	1445	6958	14,68
4	CU-4	11	37	101	560	2553	16378	32,73
5	CU-5	17	48	179	639	3328	23125	45,56

La Tabla 17 presenta los resultados de la ejecución de los casos de uso sobre la base de datos Cassandra bajo la arquitectura GraphQL, incluyendo el promedio general calculado a partir de tres iteraciones realizadas. Esta información permite evaluar el desempeño y consistencia del sistema bajo diferentes escenarios de consulta.

Tabla 17

Resultados de Cassandra en GraphQL

Nro.	Caso de uso	Número de registros/tiempo (ms)						Promedio (s)
		1	**10**	**100**	**1000**	**10000**	**100000**	
1	CU-1	13	16	151	1424	1140	892	6,06
2	CU-2	9	20	62	184	621	2659	5,93
3	CU-3	13	33	120	432	2075	9281	19,92
4	CU-4	17	57	152	995	3873	24510	49,34
5	CU-5	45	87	363	1424	6395	42504	84,70

3.2 Resumen de resultados

Basándose en los resultados obtenidos en la sección 3.1, se elaboró un resumen comparativo para evaluar el rendimiento de las arquitecturas REST y GraphQL, así como de los motores de bases de datos relacionales (SQL: PostgreSQL, MySQL, SQLite) y no relacionales (NoSQL: MongoDB, Neo4j, Cassandra). Este análisis sintetizó los tiempos de respuesta promedio de las consultas ejecutadas en los cinco casos de uso definidos, permitiendo identificar las diferencias de desempeño entre ambas arquitecturas y tipos de bases de datos bajo condiciones de carga variable.

3.2.1 REST vs GraphQL

Para iniciar el análisis comparativo, se presentan gráficas que ilustran el rendimiento de cada motor de base de datos en función de las arquitecturas REST y GraphQL utilizadas en el experimento. Estas visualizaciones, generadas a partir de los tiempos de respuesta promedio registrados en el documento Excel, permiten comparar el desempeño de cada motor bajo los cinco casos de uso definidos, destacando las diferencias entre ambas arquitecturas.

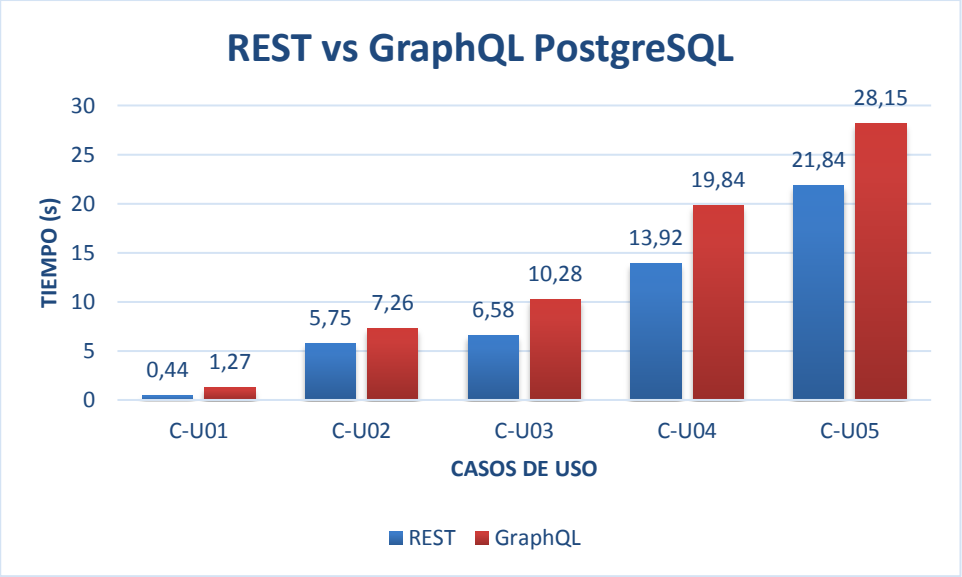
Postgres

A continuación, se detallan las gráficas correspondientes a la base de datos PostgreSQL, mostrando una comparación directa de los resultados obtenidos con las APIs REST y GraphQL.

La Figura 25 muestra una gráfica comparativa de los tiempos de respuesta obtenidos con las arquitecturas REST y GraphQL al ejecutar los casos de uso sobre la base de datos PostgreSQL. Esta visualización permite analizar el rendimiento relativo de ambas arquitecturas en distintos niveles de consulta.

Figura 25

Resultados de REST vs GraphQL en PostgreSQL



En el motor de base de datos PostgreSQL, la arquitectura REST mostró un rendimiento promedio un 34.88% superior al de GraphQL al ejecutar consultas jerárquicas distribuidas en los cinco niveles funcionales definidos. La mayor diferencia se observó en consultas con volúmenes de datos pequeños (1 a 100 registros), donde REST superó a GraphQL hasta en un 65%, debido a su menor sobrecarga en la resolución de consultas. A medida que el volumen de datos aumentó (hasta 100 000 registros), la brecha de rendimiento se redujo; sin embargo, REST mantuvo una ventaja consistente en todos los niveles, reflejando una mayor eficiencia en el procesamiento de consultas en este motor relacional.

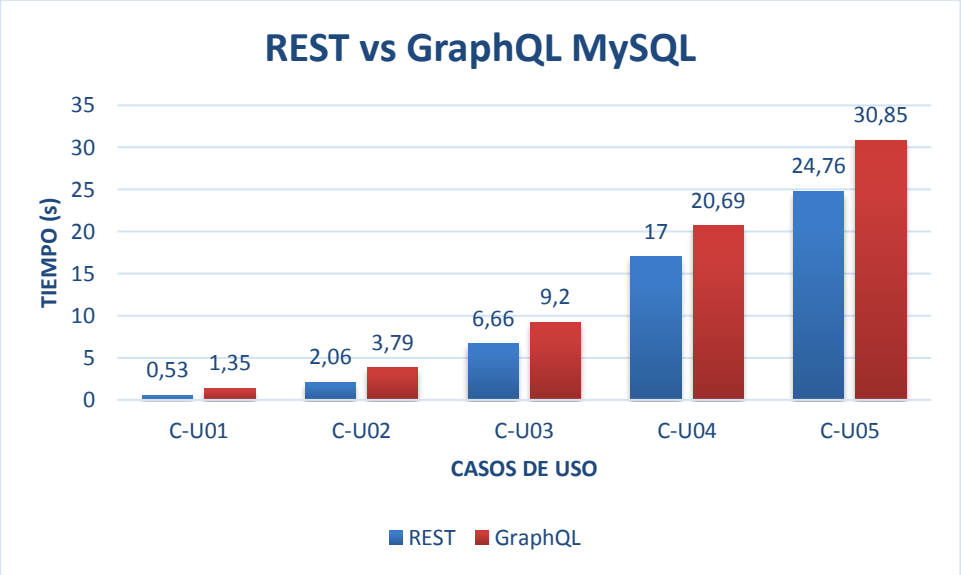
MySQL

A continuación, se detallan las gráficas correspondientes a la base de datos MySQL, mostrando una comparación directa de los resultados obtenidos con las APIs REST y GraphQL.

La Figura 26 muestra una gráfica comparativa de los tiempos de respuesta obtenidos con las arquitecturas REST y GraphQL al ejecutar los casos de uso sobre la base de datos MySQL. Esta visualización permite analizar el rendimiento relativo de ambas arquitecturas en distintos niveles de consulta.

Figura 26

Resultados de REST vs GraphQL en MySQL



En el motor de base de datos MySQL, la arquitectura REST exhibió un rendimiento promedio un 34.31% superior al de GraphQL al ejecutar consultas jerárquicas distribuidas en los cinco niveles funcionales definidos. La ventaja de REST fue particularmente notable en consultas con volúmenes de datos pequeños (1 a 100 registros), alcanzando hasta un 60% de mayor rapidez en el nivel de usuarios, debido a su menor sobrecarga en el procesamiento de consultas simples. A medida que el volumen de datos aumentó (hasta 100 000 registros) y se avanzó a niveles jerárquicos superiores, la diferencia de rendimiento se redujo, aunque REST mantuvo una ventaja consistente en todos los niveles, destacando su eficiencia en este motor relacional.

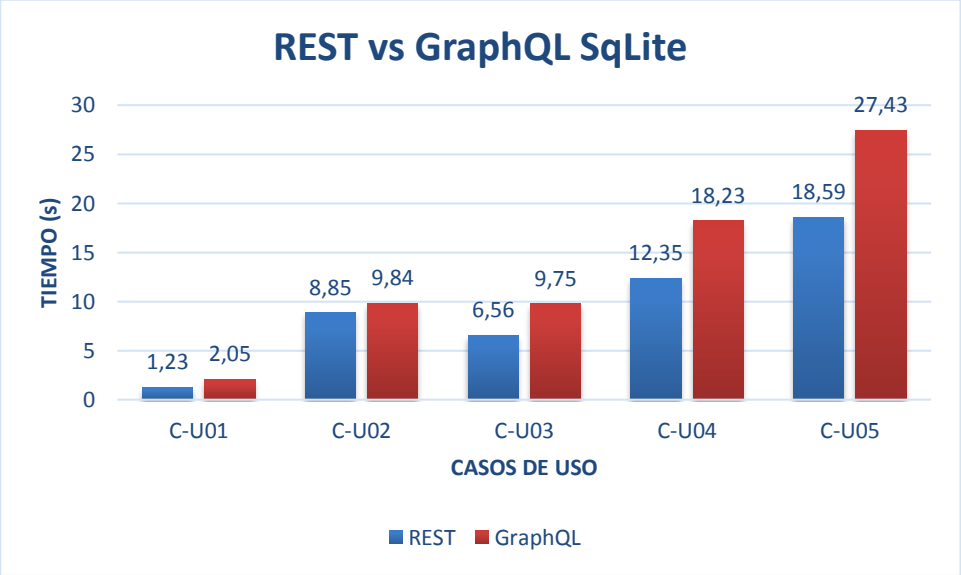
SqLite

A continuación, se detallan las gráficas correspondientes a la base de datos SqLite, mostrando una comparación directa de los resultados obtenidos con las APIs REST y GraphQL.

La Figura 27 muestra una gráfica comparativa de los tiempos de respuesta obtenidos con las arquitecturas REST y GraphQL al ejecutar los casos de uso sobre la base de datos SqLite. Esta visualización permite analizar el rendimiento relativo de ambas arquitecturas en distintos niveles de consulta.

Figura 27

Resultados de REST vs GraphQL en SqLite



En el motor de base de datos SQLite, la arquitectura REST mostró un rendimiento promedio un 29.85% superior al de GraphQL al ejecutar consultas jerárquicas distribuidas en los cinco niveles funcionales definidos. La ventaja de REST fue particularmente significativa en niveles con alta carga de datos (tercer, cuarto y quinto niveles: comentarios, respuestas y reacciones), donde superó a GraphQL en más de un 32%, atribuible a la optimización de consultas ligeras en la arquitectura embebida de SQLite. En contraste, en el segundo nivel (publicaciones), con volúmenes de datos intermedios (100 a 1000 registros), la diferencia se redujo a un 10%, lo que indica que GraphQL mantuvo una eficiencia competitiva en escenarios

de complejidad moderada debido a su capacidad para manejar consultas anidadas de manera más flexible.

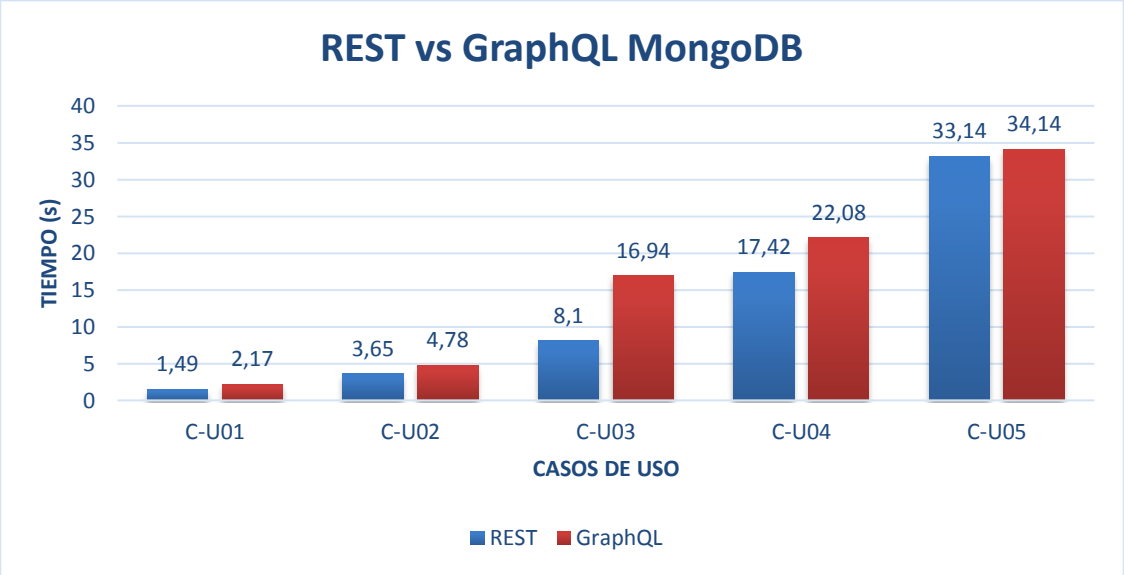
MongoDB

A continuación, se detallan las gráficas correspondientes a la base de datos MongoDB, mostrando una comparación directa de los resultados obtenidos con las APIs REST y GraphQL.

La Figura 28 muestra una gráfica comparativa de los tiempos de respuesta obtenidos con las arquitecturas REST y GraphQL al ejecutar los casos de uso sobre la base de datos MongoDB. Esta visualización permite analizar el rendimiento relativo de ambas arquitecturas en distintos niveles de consulta.

Figura 28

Resultados de REST vs GraphQL en MongoDB



En el motor de base de datos **NoSQL orientado a documentos MongoDB**, la arquitectura **REST** obtuvo un rendimiento promedio superior al de **GraphQL** en la ejecución de

consultas jerárquicas, con un tiempo medio de **12.36 segundos** frente a **16.02 segundos** para GraphQL. Esto representa una diferencia de **22.85% a favor de REST**, lo que refleja una ventaja significativa en términos de eficiencia en este entorno.

REST demostró un mejor comportamiento en los cinco niveles evaluados, especialmente en los niveles 3 y 4, donde el acceso a documentos anidados fue más eficiente gracias a la simplicidad y control explícito que REST proporciona. MongoDB, al ser una base de datos basada en documentos JSON, se adapta bien a consultas directas y específicas, lo cual favorece a REST al evitar la sobrecarga de resolver estructuras anidadas típicas de GraphQL.

Por su parte, **GraphQL** presentó tiempos consistentemente más altos, con diferencias notables en el **nivel 3 (comentarios)** y el **nivel 4 (respuestas)**, donde las resoluciones anidadas de los campos implicaron múltiples accesos a colecciones relacionadas. Esta sobrecarga es más visible en bases de datos donde no existen relaciones estrictamente definidas como claves foráneas, lo que requiere más lógica de resolución del lado del servidor.

Esta diferencia puede explicarse por:

Costos de resolución dinámica en GraphQL: MongoDB no implementa relaciones nativas como en bases relacionales, lo que obliga a GraphQL a realizar múltiples consultas internas (resolvers) para emular una estructura jerárquica.

Ventaja de REST en acceso directo: Las rutas explícitas de REST permiten un acceso más predecible y segmentado a los documentos, lo cual es especialmente ventajoso en estructuras NoSQL donde el control de las claves y subdocumentos está del lado del desarrollador.

En conjunto, los resultados evidencian que en **MongoDB**, la arquitectura REST es más eficiente que GraphQL en consultas jerárquicas de varios niveles, debido a su menor

complejidad operativa y mayor afinidad con el modelo de documentos. Esto sugiere que, para aplicaciones que requieren alto rendimiento en lectura jerárquica sobre bases NoSQL, **REST sigue siendo la opción preferida** frente a GraphQL en este tipo de entornos.

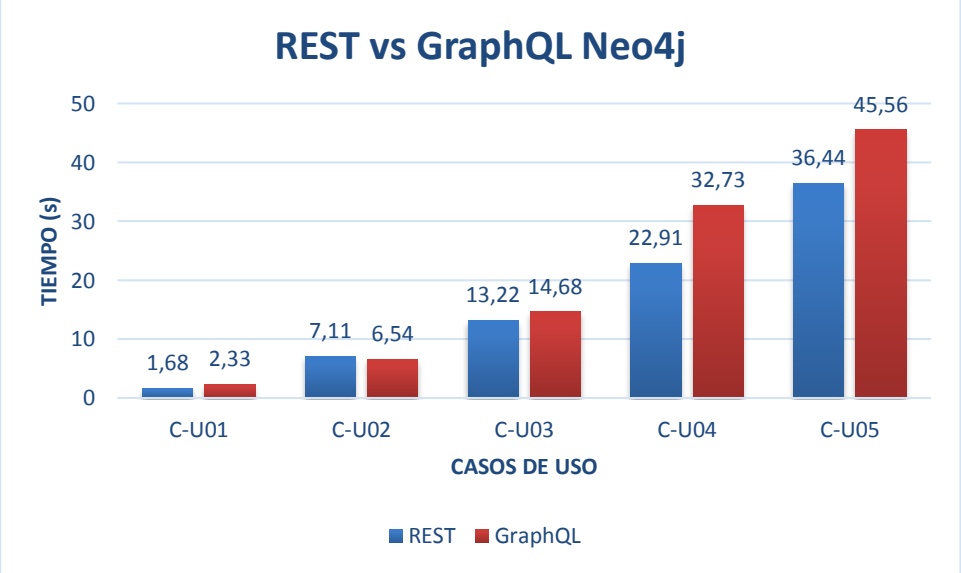
Neo4j

A continuación, se detallan las gráficas correspondientes a la base de datos Neo4j, mostrando una comparación directa de los resultados obtenidos con las APIs REST y GraphQL.

La Figura 29 muestra una gráfica comparativa de los tiempos de respuesta obtenidos con las arquitecturas REST y GraphQL al ejecutar los casos de uso sobre la base de datos Neo4j. Esta visualización permite analizar el rendimiento relativo de ambas arquitecturas en distintos niveles de consulta.

Figura 29

Resultados de REST vs GraphQL en Neo4j



En el motor de base de datos orientado a grafos Neo4j, la arquitectura REST mostró un

rendimiento promedio un 15.82% superior al de GraphQL al ejecutar consultas jerárquicas distribuidas en los cinco niveles funcionales definidos. A diferencia de los motores relacionales y NoSQL previamente analizados, las diferencias de desempeño en Neo4j fueron menos pronunciadas, lo que indica una competencia relativamente equilibrada entre ambas arquitecturas en este entorno de grafos.

REST destacó por su eficiencia en la mayoría de los niveles, con ventajas significativas en el cuarto nivel (respuestas, 29.96%) y el quinto nivel (reacciones, 20.02%), donde la simplicidad y el control directo de las consultas REST optimizaron el acceso a relaciones profundas en el grafo. Esta eficiencia se atribuye a la capacidad de REST para ejecutar consultas Cypher con menor sobrecarga en comparación con los resolvers de GraphQL. Sin embargo, en el segundo nivel (publicaciones), GraphQL superó a REST por un 8.72%, posiblemente debido a:

- Optimización de resolvers en GraphQL: La implementación de resolvers en GraphQL pudo haber sido más eficiente para manejar relaciones simples en el segundo nivel, donde las consultas requieren menos anidamiento y complejidad.
- Sobrecarga en REST: La implementación de las consultas REST podría haber introducido una ligera ineficiencia, como una configuración menos optimizada de los patrones Cypher o un manejo menos flexible de los resultados.

En conjunto, Neo4j exhibió un desempeño competitivo entre REST y GraphQL, con una ligera ventaja para REST en los niveles más complejos, donde la profundidad de las relaciones en el grafo beneficia la simplicidad de las consultas directas frente a la sobrecarga inherente a los resolvers de GraphQL. Estos resultados sugieren que la elección entre ambas arquitecturas en Neo4j puede depender de la naturaleza específica de las consultas y la complejidad de las relaciones en el grafo.

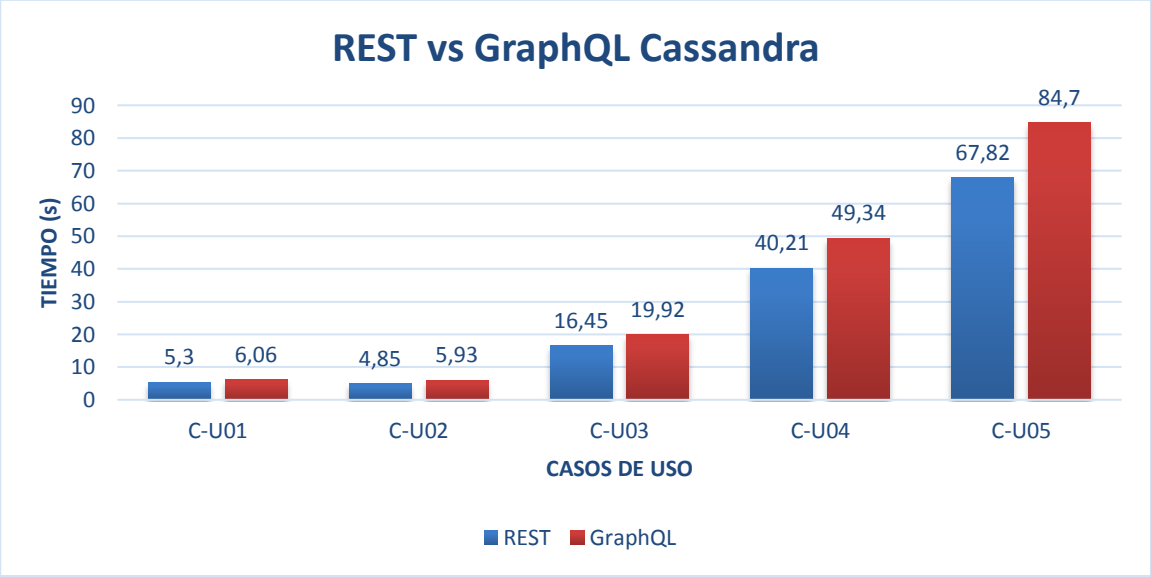
Cassandra

A continuación, se detallan las gráficas correspondientes a la base de datos Cassandra, mostrando una comparación directa de los resultados obtenidos con las APIs REST y GraphQL.

La Figura 30 muestra una gráfica comparativa de los tiempos de respuesta obtenidos con las arquitecturas REST y GraphQL al ejecutar los casos de uso sobre la base de datos Cassandra. Esta visualización permite analizar el rendimiento relativo de ambas arquitecturas en distintos niveles de consulta.

Figura 30

Resultados de REST vs GraphQL en Cassandra



En el sistema de gestión de bases de datos distribuidas **Cassandra**, la arquitectura **REST** evidenció un rendimiento promedio un **20.06% superior** al de **GraphQL** al ejecutar las consultas jerárquicas definidas en los cinco niveles funcionales del experimento computacional. Esta diferencia posiciona a REST como la opción más eficiente en este entorno NoSQL orientado a grandes volúmenes de datos y alta disponibilidad.

REST destacó principalmente en los niveles intermedios y altos de la jerarquía (niveles 3 a 5), donde el control secuencial de las llamadas permitió una ejecución más directa y menos costosa computacionalmente. La eficiencia alcanzada se debe, en parte, a la naturaleza de Cassandra como base de datos columnar, donde las consultas simples y direccionadas como las que ofrece REST se ajustan mejor a su modelo distribuido de lectura rápida por clave.

Por el contrario, GraphQL presentó tiempos de respuesta más elevados, especialmente en los niveles más profundos de consulta. Esto se atribuye a:

- **Sobrecarga de resolvers**: Cada nivel de profundidad en GraphQL introduce resolvers anidados que, en el caso de Cassandra, pueden aumentar la latencia al requerir múltiples accesos distribuidos a nodos del clúster.
- **Desajuste con el modelo de datos**: El enfoque de GraphQL, que prioriza la estructuración jerárquica y agregada de los datos, no se adapta de forma tan eficiente a la arquitectura distribuida de Cassandra, que favorece lecturas rápidas pero planas y predefinidas.

En conjunto, los resultados sugieren que, en el contexto de Cassandra, la arquitectura **REST ofrece ventajas claras en eficiencia temporal**, especialmente cuando se realizan múltiples accesos a datos relacionados en profundidad. Por tanto, para entornos que priorizan el rendimiento bajo grandes cargas de lectura y una arquitectura distribuida, REST se posiciona como una opción más apropiada que GraphQL.

La Tabla 18 presenta un resumen consolidado de los resultados de rendimiento para cada motor de base de datos (PostgreSQL, MySQL, SQLite, MongoDB, Neo4j y Cassandra) bajo las arquitecturas REST y GraphQL. Este resumen se basa en el promedio de los tiempos de respuesta de las consultas jerárquicas ejecutadas en los cinco casos de uso definidos (usuarios,

publicaciones, comentarios, respuestas y reacciones), adicionalmente, se determina un porcentaje que refleja el nivel de eficiencia relativa entre ambas arquitecturas, indicando en este caso cuánto más eficiente es REST frente a GraphQL, esto proporciona una visión integral que permite responder a la pregunta de investigación: **¿qué sistema de gestión de bases de datos ofrece un mejor rendimiento en función de la arquitectura utilizada?**

La Tabla 18 presenta los resultados finales entre la comparación de rendimiento entre las arquitecturas REST y GraphQL en cada uno de los motores de base de datos evaluados. Se muestran los tiempos promedio de respuesta en segundos obtenidos para cada tecnología, junto con el porcentaje de mejora registrado a favor de REST, expresado en la columna "Ventaja de REST sobre GraphQL (%)". Estos valores permiten identificar en qué medida REST resultó más eficiente en términos de tiempo de ejecución frente a GraphQL, dependiendo del sistema gestor utilizado, además presenta un promedio total de que tan eficiente es REST sobre GraphQL.

Tabla 18

Resumen final de resultados

Base de datos	REST	GraphQL	Ventaja de REST sobre GraphQL (%)
PostgreSQL	9,71 s	13,36 s	27,32%
MySQL	10,20 s	13,18 s	22,61%
SqLite	9,51 s	13,46 s	29,35%
MongoDB	12,76 s	16,02 s	20,35%
Neo4j	16,27 s	20,37 s	20,13%
Cassandra	26,93 s	33,19 s	18,86%
Promedio total	-	-	23,10%

El análisis de los promedios de tiempo de respuesta obtenidos en el experimento computacional permitió identificar **tendencias claras de rendimiento** entre las seis bases de

datos evaluadas (PostgreSQL, MySQL, SQLite, MongoDB, Neo4j y Cassandra) bajo las arquitecturas **REST y GraphQL**, conforme a la norma ISO/IEC 25023. Todas las pruebas se ejecutaron en un entorno local controlado, garantizando consistencia en la comparación.

En todos los casos, **la arquitectura REST demostró un desempeño superior** al de GraphQL, con **diferencias porcentuales que van del 18,86% al 29,35%** a favor de REST. La base de datos más eficiente en REST fue **SQLite (9,51 s)**, seguida de **PostgreSQL (9,71 s)** y **MySQL (10,20 s)**. En contraste, el sistema con mayor tiempo en ambas arquitecturas fue **Cassandra**.

Los resultados del experimento evidencian que, en un entorno local y bajo condiciones controladas, **la arquitectura REST ofrece un mejor rendimiento general en todos los sistemas de gestión de bases de datos evaluados**. REST presentó tiempos de respuesta más bajos de manera consistente en las seis bases de datos probadas, con una diferencia promedio del **23,10%** respecto a GraphQL.

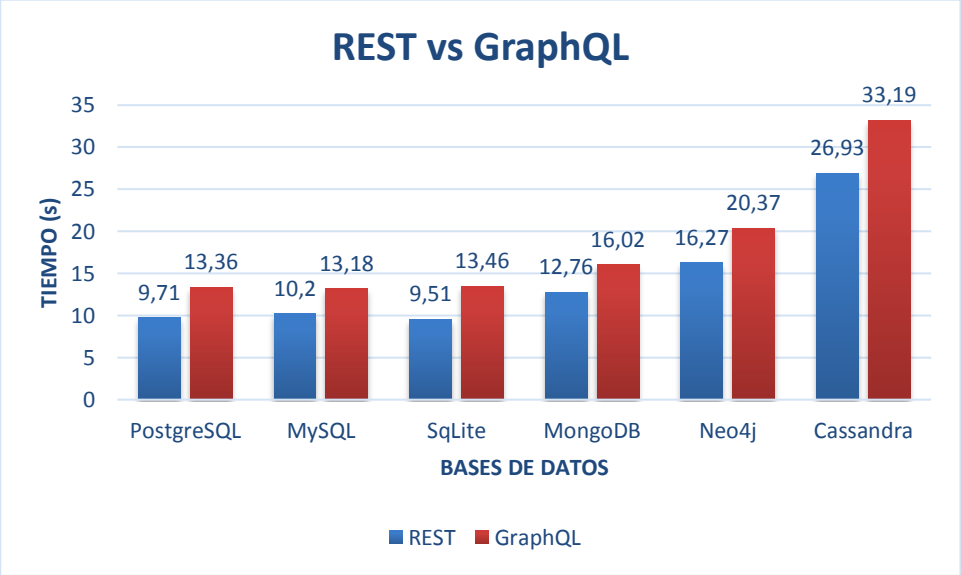
Entre las bases de datos, **SQLite y PostgreSQL fueron las más eficientes bajo REST**, mientras que **Cassandra**, si bien presentó los tiempos más altos, mantuvo una diferencia moderada entre ambas arquitecturas, lo que indica una relativa estabilidad en contextos de alto volumen de datos.

Según estos resultados, podemos decir que **REST es la arquitectura más adecuada para escenarios donde el tiempo de respuesta es crítico**, especialmente en entornos que implican consultas jerárquicas de gran volumen. No obstante, la elección también debe considerar otros factores como la flexibilidad en la estructura de los datos, la escalabilidad, y la complejidad de las relaciones, donde GraphQL podría ser ventajoso a pesar de su mayor costo en tiempo.

La Figura 31 presenta una gráfica comparativa final de los tiempos de respuesta obtenidos en cada motor de base de datos, evaluando el desempeño de las arquitecturas REST y GraphQL. Esta visualización permite identificar con claridad las diferencias de rendimiento entre ambas tecnologías según el sistema gestor utilizado.

Figura 31

Comparación tiempos de respuesta REST y GraphQL



CONCLUSIONES

- La revisión bibliográfica consolidó un marco teórico robusto sobre REST, GraphQL y arquitecturas orientadas a servicios, proporcionando una base sólida para el diseño experimental. Asimismo, el análisis de las bases de datos relacionales más relevantes (PostgreSQL, MySQL, SQLite) y no relacionales (MongoDB, Neo4j, Cassandra) permitió fundamentar de manera precisa y efectiva la configuración del experimento computacional.
- Con base en los conocimientos adquiridos, se diseñó un experimento computacional que incluyó dos APIs equivalentes (REST y GraphQL) conectadas a diferentes bases de datos. Se desarrolló un cliente en Node.js para consumir ambas arquitecturas y medir sus tiempos de respuesta. Esta infraestructura facilitó la realización de pruebas controladas y uniformes, alineadas con la métrica de eficiencia establecida por la norma ISO/IEC 25023.
- El análisis de los resultados experimentales demostró que la arquitectura REST supera consistentemente a GraphQL en tiempos de respuesta, con una eficiencia promedio 23,10% superior. SQLite, PostgreSQL y MySQL destacaron como las bases de datos más rápidas bajo REST, mientras que Cassandra mostró los tiempos más elevados en ambas arquitecturas. Estos hallazgos, obtenidos en un entorno local controlado, indican que **REST** con **SqLite** son los más eficientes para el acceso a datos en los sistemas evaluados, especialmente en escenarios donde el rendimiento es crítico.
- En respuesta a la pregunta de investigación: ***¿Qué sistema de gestión de bases de datos ofrece un mejor rendimiento en función de la arquitectura utilizada (REST o GraphQL) ?***, los resultados obtenidos demuestran que la arquitectura REST ofrece un

mejor rendimiento general en todos los sistemas evaluados. Entre las bases de datos, SQLite y PostgreSQL destacaron por su eficiencia bajo REST, mientras que Cassandra fue la menos eficiente en ambos casos. Por tanto, en contextos donde el tiempo de respuesta es un factor crítico, REST combinado con bases de datos como SQLite o PostgreSQL representa la opción más adecuada.

RECOMENDACIONES

- Ampliar el estudio incorporando métricas adicionales, como latencia, throughput y operaciones por segundo (OPS), para una evaluación más integral del rendimiento de REST y GraphQL. Esto sería particularmente relevante en escenarios de alta disponibilidad y procesamiento intensivo, permitiendo identificar limitaciones y optimizaciones potenciales.
- Evaluar el desempeño de REST y GraphQL en operaciones CRUD completas (creación, lectura, actualización y eliminación), dado que este estudio se centró en consultas de lectura. Incluir estas operaciones podría revelar diferencias de rendimiento y escalabilidad entre ambas arquitecturas.
- Incorporar una capa de autenticación y control de acceso en las pruebas experimentales para simular entornos de producción reales. Esto permitiría analizar el impacto de los mecanismos de seguridad en el rendimiento de REST y GraphQL, así como en su interacción con las bases de datos.

BIBLIOGRAFÍA

Amazon Web Service. (2024). API sin servidor de GraphQL y de publicación o suscripción – AWS AppSync – Amazon Web Services. https://aws.amazon.com/es/appsync/

Apache Cassandra. (2024). Cassandra Performance Metrics. https://cassandra.apache.org/doc/latest/

Apollo. (2024). GraphQL Federation with MongoDB. https://www.apollographql.com/docs/graphos/schema-design/federated-schemas/federation

Arroyo, J. (2024). (PDF) Bases de Datos NoSQL: Un Análisis Integral. https://www.researchgate.net/publication/382077712_Bases_de_Datos_NoSQL_Un_Analisis_Integral

AWS. (2025a). Base de datos clave-valor NoSQL rápida: Amazon DynamoDB, Amazon Web Services. https://aws.amazon.com/es/dynamodb/

AWS. (2025b). Guía de decisiones para bases de datos en AWS. https://aws.amazon.com/es/getting-started/decision-guides/databases-on-aws-how-to-choose/

AWS. (2025c). ¿Qué es Amazon Relational Database Service (Amazon RDS)? - Amazon Relational Database Service. 2025a. https://docs.aws.amazon.com/es_es/AmazonRDS/latest/UserGuide/Welcome.html

Brito, G., & Valente, M. T. (2020). REST vs GraphQL: A controlled experiment. Proceedings - IEEE 17th International Conference on Software Architecture, ICSA 2020, 81–91. https://doi.org/10.1109/ICSA47634.2020.00016

Bytebase. (2025). RDS vs. DynamoDB: a Complete Comparison in 2025. https://www.bytebase.com/blog/rds-vs-dynamodb/

DataStax. (2024). Cassandra Adoption Trends 2024. 2024. https://www.datastax.com/resources

Elmasri, R., & Navathe, S. B. (2022). FUNDAMENTALS OF FourthEdition DATABASE SYSTEMS. http://www.aw.com/cs

GraphQL. (2022). GraphQL | A query language for your API. https://graphql.org/

Gutiérrez, S. (2024). ¿Qué es una api rest? ¿Cómo funciona? ¿En que tipo de web utilizarlas? https://dossetenta.com/que-es-una-api-rest/

HubSpot. (2021). REST APIs: How They Work and What You Need to Know. https://blog.hubspot.com/website/what-is-rest-api

Husar, A. (2022). How to Use REST APIs – A Complete Beginner's Guide. https://www.freecodecamp.org/news/how-to-use-rest-api/

ISO. (2023). ISO/IEC 25023:2016 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality. https://www.iso.org/standard/35747.html

ISO. (2024). ISO/IEC TS 21419:2024 - Information technology — Cross-jurisdictional and societal aspects of implementation of biometric technologies — Use of biometrics for identity management in healthcare. https://www.iso.org/standard/80580.html

Kerstiens, C. (2020). PostgreSQL: The World's Most Advanced Open Source DatabaseData. https://www.citusdata.com/blog/authors/craig-kerstiens/

Microsoft. (2021). Ejemplo de datos de consulta de la API web - Power Apps |

Microsoft Learn. https://learn.microsoft.com/es-es/power-apps/developer/data-platform/webapi/web-api-query-data-sample

MongoDB Inc. (2024). MongoDB Annual Report 2024. https://investors.mongodb.com/

Neo4j Inc. (2024). Neo4j Graph Database Performance Report. https://neo4j.com/resources/

Oracle. (2023). MySQL Documentation. https://dev.mysql.com/doc/

Percona. (2021). MySQL vs. PostgreSQL: Which is Better? https://www.percona.com/blog/

Percona. (2024). MongoDB Performance Benchmarks 2024. https://www.percona.com/resources

Platzi. (2020). Bases de datos desde cero. https://platzi.com/ruta/base-de-datos/

PostgreSQL Global Development Group. (2023). PostgreSQL: Documentation. https://www.postgresql.org/docs/

Postman. (2020). REST Client | Postman API Platform [Free Download]. https://www.postman.com/product/rest-client/

Postman, T. (2023). What Is a REST API? Examples, Uses & Challenges | Postman Blog. https://blog.postman.com/rest-api-examples/

Quinche Moran, L. A. (2021). DESARROLLO DE UNA API-GRAPHQL PARA ANALIZAR LA EFICIENCIA DEL CONSUMO DE DATOS ENTRE BASE DE DATOS RELACIONALES Y NO RELACIONALES UTILIZANDO LAS MÉTRICAS DE LA NORMA ISO/IEC 25023.

Red Hat. (2023). ¿Qué es una API de REST? https://www.redhat.com/es/topics/api/what-is-a-rest-api

Rock, C. (2021). API Rest: ¿qué es y cómo funciona ese recurso? [con ejemplos]. https://pingback.com/es/resources/api-rest/

Silberschatz, A. (2023). Database System Concepts - 7th edition. https://db-book.com/

SQLite Documentation. (2023). SQLite Documentation. https://www.sqlite.org/docs.html

Stack Overflow. (2024). Stack Overflow Developer Survey 2024 . https://survey.stackoverflow.co/2024/

Vazquez-Ingelmo, A., Cruz-Benito, J., & García-Penalvo, F. J. (2017). Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL. ACM International Conference Proceeding Series, Part F132203. https://doi.org/10.1145/3144826.3145437;CSUBTYPE:STRING:CONFERENCE

Winkler, D., Lindner, F., & Meyer-Ross, K. K. (2023). Inklusion durch informationstechnische Assistenzsysteme - Gelingensbedingungen digitaler Lernszenarien mit Hilfe von Augmented Reality am Beispiel hörbeeinträchtigter oder gehörloser Menschen in der technischen Bildung. Communities in New Media. Inclusive Digital: Forming Community in an Open Way Self-Determined Participation in the Digital Transformation - Proceedings of 26th Conference GeNeMe 2023, 55–63. https://doi.org/10.25368/2024.273

ANEXOS