Memoria - Práctica 1 Bases de datos, Análisis de rendimiento y perfilado

Adam Maltoni, Ibón de Mingo

October 2024

Contents

1 Introducción

En el contexto actual del desarrollo de software y análisis de datos, el manejo eficiente de grandes volúmenes de información es esencial para garantizar un rendimiento óptimo en aplicaciones y servicios. Esta práctica tiene como objetivo la comparación y análisis del rendimiento de varios sistemas gestores de bases de datos (SGBD), empleando operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para medir el uso de recursos como el tiempo de ejecución y la memoria en escenarios de ingesta masiva de datos.

Para este análisis, se trabajará con cuatro sistemas de bases de datos: SQLite3, DuckDB, PostgresSQL y MongoDB. Cada uno de estos sistemas presenta diferentes características, ventajas y desventajas que se explorarán mediante el desarrollo de un conjunto de herramientas que automatizan la generación de datos sintéticos y el perfilado de rendimiento. Adicionalmente, se evaluará el impacto de mecanismos de caché e indexación para optimizar el acceso a la información.

El objetivo final es proporcionar un análisis comparativo sobre el rendimiento de los SGBD mencionados, permitiendo extraer conclusiones que guíen la elección de un sistema adecuado para diferentes aplicaciones de datos. A lo largo de este documento, se explicará la metodología empleada, los resultados obtenidos y las conclusiones derivadas del perfilado de rendimiento.

2 Generación y vertido de datos

Comenzamos con la generación de datos para lo cual hemos utilizado la librería Faker para crear dos conjuntos de datos. El primero contiene información de usuarios (nombre, DNI, email, teléfono móvil, fijo, dirección, etc.) y el segundo contiene datos de vehículos (matrícula, número de bastidor, fabricante, año, modelo y categoría). Ambos conjuntos están relacionados mediante el DNI del usuario, que es propietario del coche. Posteriormente, se crea un tercer conjunto de datos al cual llamamos mix, en el que dentro de los datos de cada usuario, en formato JSON, se incluye una lista de coches en su propiedad. Esta tabla se utilizará para facilitar las consultas en mongodb.

El código se estructura en tres clases: UserProvider, VehicleProvider y MasterProvider. Cada clase se encarga de la generación de un conjunto de datos de esta forma conseguimos separar responsabilidades lo cual facilita la escalabilidad y mantenibilidad del código.

La clase UserProvider es la responsable de generar los datos relacionados con los usuarios, como nombres, DNIs únicos, correos electrónicos, teléfonos y direcciones. Es de suma importancia que los DNIs sean únicos ya que serán las claves primarias y foráneas de las tablas de las bases de datos relacionales. Como buscamos que los datos sintéticos se asemejen lo máximo posible a la realidad, también nos hemos asegurado que otros campos como el teléfono o el email sean únicos. En cuanto a la dirección de cada usuario, se ha procurado que los códigos postales vayan acordes al municipio y que el municipio se encuentre en la provincia generada. Para la obtención de los códigos postales se ha utilizado el dataset CSV indicado en las referencias del enunciado de la práctica.

La clase VehicleProvider se encarga de generar datos de vehículos, como matrículas, números de bastidor y características de los coches. En la generación de las matrículas de los coches, el código ajusta el formato de la matrícula (antiguas:antes o durante 2000; modernas: posterior al 2000) en función del año de fabricación del vehículo, replicando el formato real de las matrículas españolas. Al igual que para los usuarios, se ha tratado de generar datos realistas, por lo que algunos atributos como el número de bastidor o VIN (clave primaria en vehículos) y la matrícula son valores únicos en el conjunto de datos. Para modelos, fabricantes (marcas) y años, nos hemos basado en otro CSV de ejemplo que hemos generado a partir de unas 300 observaciones de coches populares.

Finalmente, la clase MasterProvider unifica la generación de usuarios y vehículos, creando internamente una instancia de UserProvider y otra de VehicleProvider. Mediante el método 'generate', se forman las listas de usuarios y vehículos conjuntamente.

Una vez generados los datos, se pasan a CSV y JSON, este paso es clave ya que nos permitirá insertar los datos en los distintos SGBD. Pero antes de ello, creamos la función 'merge_data'. Con esta función unimos en una misma lista los usuarios y sus vehículos asignados como una lista de documentos anidados. Esta función es crucial para el manejo de los datos en mongodb ya que facilita las querys en las que se necesite hacer 'JOINS'.

Para la creación de los CSV se utilizan los datos de los vehículos y usuarios y se crean dos CSVs independientes con la información de cada uno. Para llevar a cabo esta tarea, se emplean funciones privadas para cada archivo. Los datos en CSV serán insertados en las bases de datos relacionales.

Por otro lado, se generan los archivos JSON. Utilizamos una función interna '__data_to_json', con esta decisión de diseño encapsulamos la funcionalidad de escritura en JSON. En JSON generamos tres archivos distintos: usuarios, vehículos y mix (datos de la función 'merge_data'). Estos archivos se insertarán en MongoDB ya que es una base de datos con un modelo de datos basado en documentos.

3 Creación de las tablas y operaciones CRUD sobre las BBDD

Tras haber generado los diferentes archivos con los datos, se lleva a cabo la ingesta de datos dentro de los distintos SGBD. Para las bases de datos relacionales es necesario definir previamente el esquema de los datos, en cambio, mongo al ser una base de datos documental, es más flexible y admite cualquier esquema de datos por lo que haremos la ingesta directamente.

Para crear la estructura de las bases de datos relacionales diseñamos una función llamada 'create_tables'. Lo primero que hace esta función es asegurarse de que las tablas que se van a crear no existen. Para ello mediante una función interna, se eliminan las tablas que se van a generar en caso de que existan. Para Postgresql utilizamos el método 'CASCADE' para asegurarnos de que la operación se lleva a cabo con éxito. Finalmente se crea la estructura de las tablas definiendo los tipos de cada atributo, indicando las primary keys (dni y número de bastidor), definiendo los atributos que deben ser unique y referenciando el DNI de los usuarios propietarios de coches en la tabla vehículos con el atributo 'dni_usuario'. Cabe destacar que es importante eliminar antes vehículos que usuarios para no violar la integridad referencial en el proceso.

Con la estructura definida en las bases de datos relacionales ya podemos empezar con la ingesta de datos. Para ello creamos una función llamada 'insert_data'. Esta función permite insertar los datos de tres distintas maneras: chunk, at_once y one. El método 'chunk' inserta todos los datos en 'batches' de un tamaño predeterminado en el SGBD, que el usuario también puede configurar con un tamaño personalizado. Con 'at_once' se insertan los datos directamente en el SGBD, aunque en algunos casos como mongodb (solo 100k por inserción) tienen un límite máximo de inserciones. Finalmente el método 'one' realiza la inserción registro a registro abriendo y cerrando conexión por cada inserción. También la función permite la creación de índices en cada SGBD, existe un parámetro en la función llamado use_index: bool, que activará la creación de índices. En todos los casos, es necesario separar las bases de datos relacionales con MongoDB ya que la conexión con la base de datos es diferente.

Como a lo largo de los tests y la práctica realizaremos múltiples operaciones de inserción de datos requerimos una función para eliminar los registros de las bases de datos. En esta función, simplemente se eliminan los registros de cada SGBD con DELETE. Diferenciamos entre mongodb y el resto de SGBD para la implementación de la lógica.

A continuación, diseñamos funciones para poder realizar operaciones de lectura y actualizaciones en los distintos SGBD. Cabe mencionar que en todas estas funciones se separa la lógica de mongodb con el resto de bases de datos. Tanto la conexión como la forma de ejecutar las querys en mongodb son muy diferentes al resto de SGBD, por ello se necesita implementar una lógica distinta. En las funciones de lectura (SELECTS),

hemos creado dos funciones, una que ejecuta cualquier query de lectura y otra que selecciona todos los registros de la base de datos. La función de actualización es capaz de ejecutar cualquier query de actualización (UPDATE). Estas funciones las emplearemos posteriormente para diseñar tests que nos permitan comparar las diferentes operaciones en cada SGBD.

4 Cachés e índices

A pesar de que las bases de datos, para el almacenamiento, son los componentes centrales de los SGBD, no son los únicos. Cabe destacar otros, como en este caso son los índices de búsqueda, una estructura de datos que puede mejorar la velocidad de las operaciones almacenando un puntero único a cada fila de la tabla (tipo PK), o las cachés, un componente o área de almacenamiento de datos de alta velocidad (como la RAM) que se utiliza para almacenar temporalmente información que se accede con frecuencia.

Sin embargo, ambos presentan o pueden presentar desventajas. Además, los índices pueden ralentizar las operaciones de escritura (como inserciones, actualizaciones y eliminaciones), debido a que cada vez que se modifica un dato en una columna indexada, el índice también debe actualizarse para reflejar el cambio, añadiendo una carga adicional al proceso de escritura. Por otro lado, una caché puede volverse obsoleta si los datos dentro de la BBDD cambian con frecuencia, además de que la RAM es rápida pero es costosa.

Para esta práctica, hemos hecho pruebas con ambas, creando índices mediante sentencias 'CREATE INDEX' (relacional) o 'collection.createIndex' (MongoDB, desde Python), y utilizando dos cachés distintas, la de Memcached (ref. en ¡Bibliografía¿) y una caché sencilla que funciona por detrás con un diccionario para comparar y realizar las pruebas con ambas, implementada con la misma interfaz que Memcached).

5 Testing. Resultados y conclusiones

Nota Para este apartado, se han hecho uso de distintas funciones de perfilado de memoria y tiempo proporcionadas en el propio enunciado de la práctica. Algunas de estas han sufrido ligeras adaptaciones para el funcionamiento correcto de todo el programa, que quedan detalladas en Decisiones de diseño.

5.1 Testing sobre la generación de datos

Comenzamos haciendo tests para la generación de datos. En ellos vamos a medir el tiempo que tardan en ejecutar las distintas funciones de generación de datos y el espacio que ocupan en la memoria. Para ello utilizaremos las funciones de perfilado mencionadas con anterioridad. Vamos a medir las funciones para generar usuarios, generar vehículos y para relacionar ambas tablas.

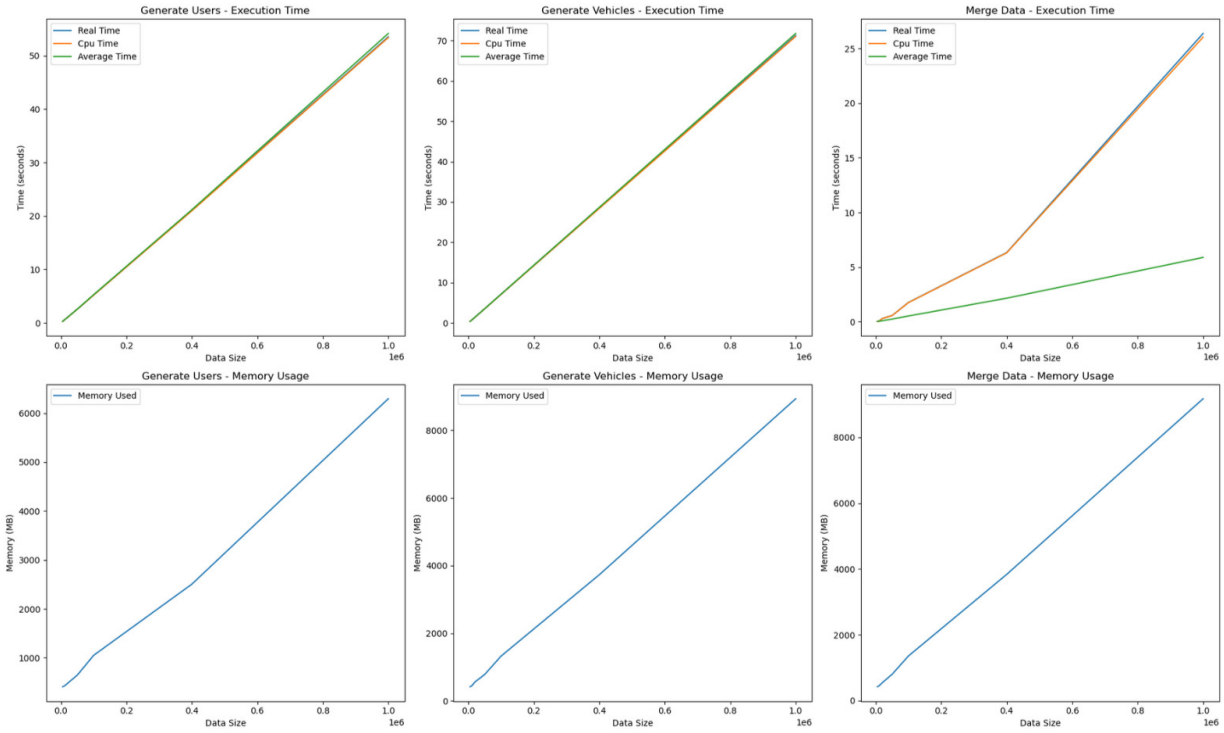

Figure 1: Plot del testeo de generación de datos

En las gráficas observamos que tanto en tiempos como en memoria en todas las funciones el crecimiento es lineal a medida que aumenta el tamaño de datos. En la gráfica de merge data podemos observar que el tiempo promedio es inferior al tiempo real, posiblemente se debe a factores externos que hayan provocado un pico en la medición de tiempos inicial. BBDD cambian con frecuencia, además de que la RAM es rápida pero es costosa.

Creemos que esto es lógico ya que en nuestro código hacemos uso de un bucle secuencial, es decir, cada registro (usuario o vehículo) se genera dentro de un bucle for, que itera un número de veces igual al tamaño de los datos (número de entradas). Por lo tanto, si duplicamos el tamaño, el código hará el doble de iteraciones. Cabe mencionar que cada operación para generar un DNI, nombre, dirección, teléfono o cualquier campo tiene un coste constante, por lo tanto, si se generan más registros, el tiempo total aumenta proporcionalmente.

5.2 Testing de operaciones CRUD sobre las distintas BBDD

A continuación realizaremos tests para medir los tiempos tanto en cpu como reales en los distintos SGBD. Comenzaremos haciendo una comparación conjunta de todos los métodos CRUD a la vez, sin tener en cuenta el SGBD, de tal forma que podemos observar que tipo de operación consume más tiempo a nivel general.

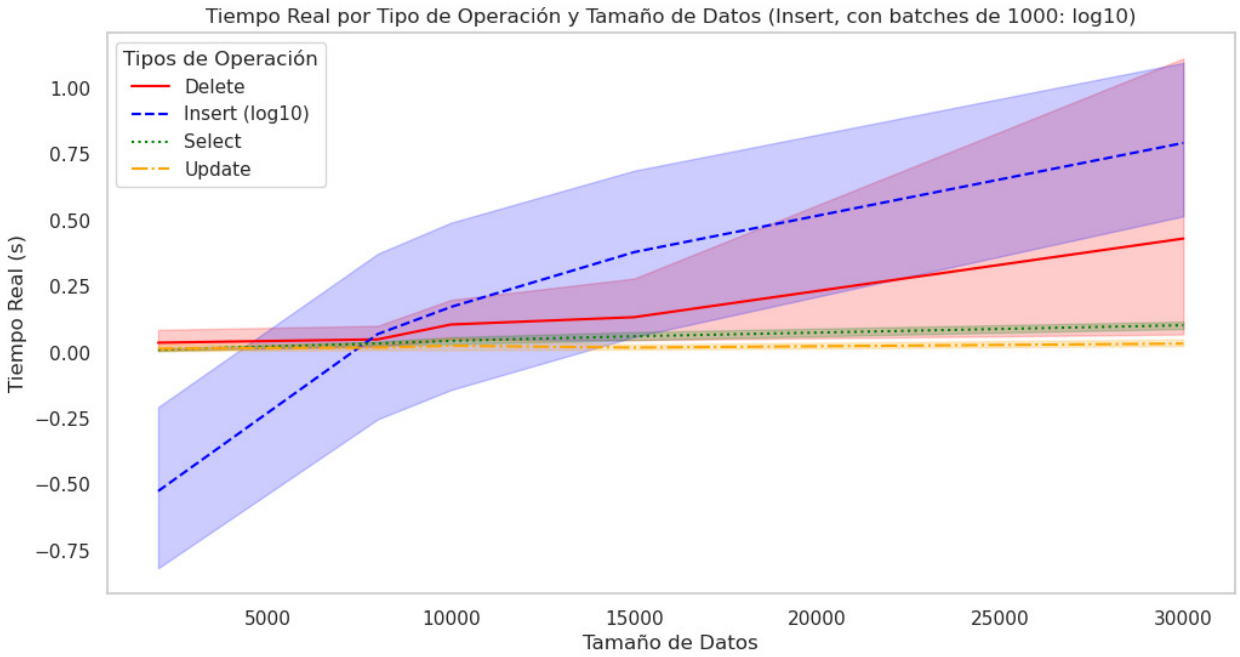

Figure 2: Lineplot comparativo de los costes temporales de diversas operaciones CRUD

Para la realización de esta gráfica se ha requerido aplicar el log10 a los tiempos medidos en insert ya que, claramente, es la operación que emplea más tiempo. El tiempo de las inserciones es lineal ya que en la gráfica, con el logaritmo aplicado, se ve una representación logarítmica. Consideramos que es realista porque cada operación de inserción individual tarda el mismo tiempo que las demás y el hecho de que haya muchos o pocos datos en la base de datos no afecta al rendimiento. La siguiente operación que consume más tiempo es la eliminación de registros de la base de datos.

Esto creemos que sucede porque los DELETE se realizan en dos bases de datos y necesitan revisar claves primarias para mantener la integridad referencial. Es por este mismo motivo por el que se elimina primero la tabla de vehículos y luego la de usuarios, como mencionado anteriormente.

En la siguiente figura, además de comparar las distintas operaciones hemos decidido enfocarnos en la inserción ya que ha sido la más lenta en el primer caso. Para ello, en los tests hemos decidido diferenciar por el método de inserción para ver más en detalle su comportamiento.

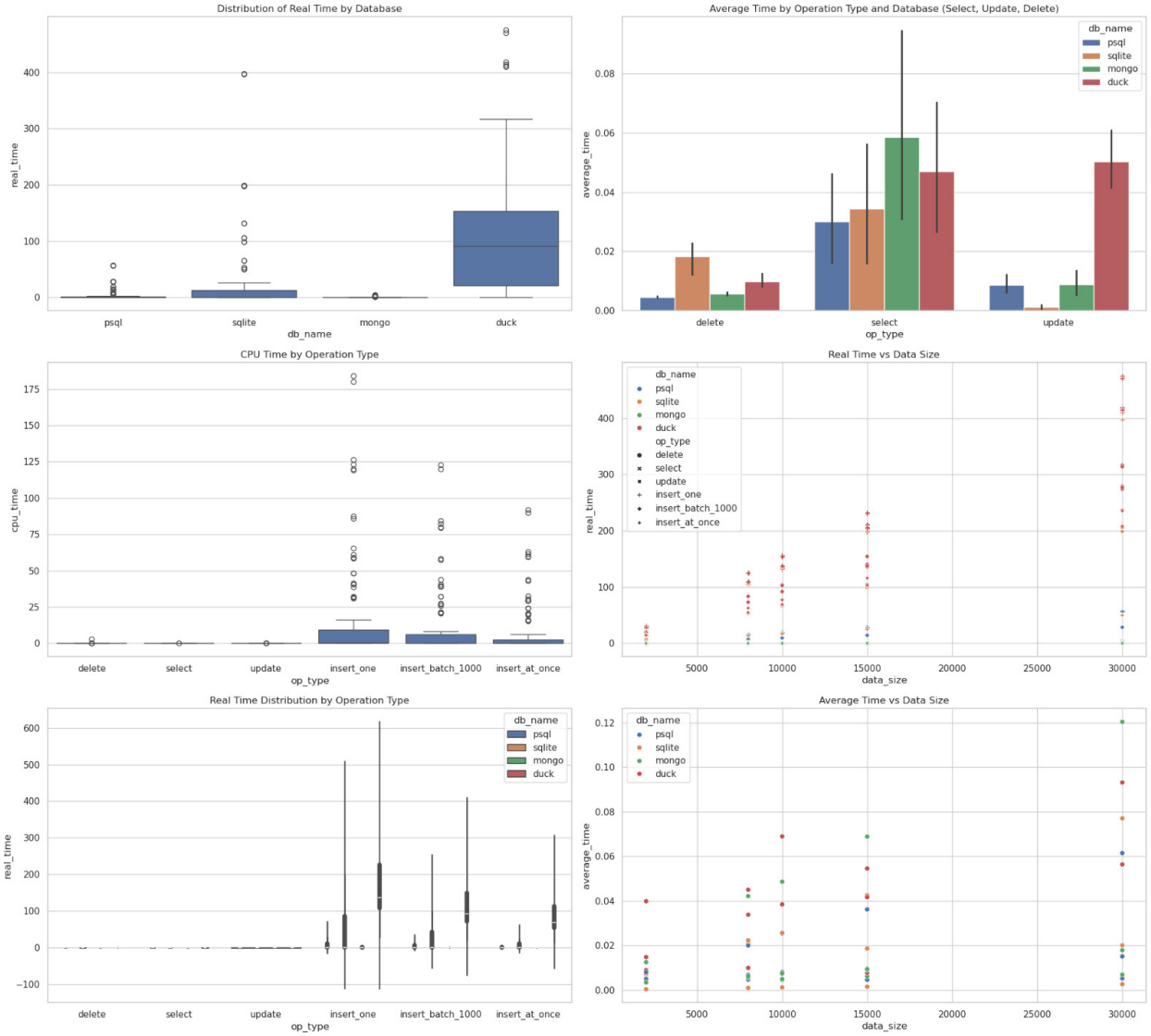

Figure 3: Subplots varios comparativos de varias operaciones CRUD y distintos SGBD

Nota Cabe destacar que, en esta otra figura, el plot de arriba a la izquierda obtiene mayores tiempos para los SELECTS. Esto es debido a que lo hemos ejecutado con una query algo más compleja para esta gráfica que la de antes (SELECT *).

En las distintas gráficas se aprecia claramente que el método de inserción más lento es el método 'one' ya que va abriendo y cerrando conexión cada vez que inserta un dato. Por el contrario el método más rápido es 'at once' ya que inserta los datos directamente. Dejando de lado los inserts hemos podido observar que las operaciones de lectura más lentas son en mongo. Creemos que sucede porque hemos hecho los tests con relaciones referenciadas lo cual es más lento que habiendo utilizado documentos incrustados (tabla 'mix' que contiene la información de usuarios y vehiculos junta) y es más lento que los INNER JOINS es las bases de datos relacionales.

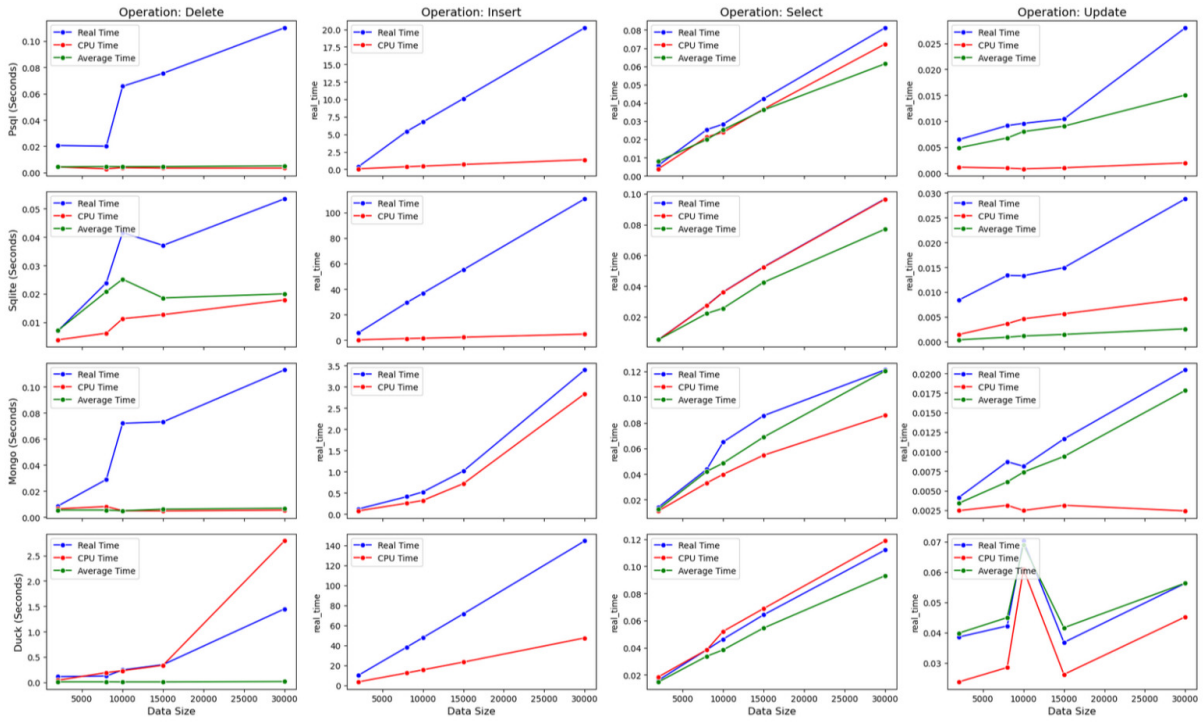

Figure 4: Otro subplot varios comparativos de varias operaciones CRUD y distintos SGBD

5.3 Testing de consultas de selección y efectividad de caché

En este apartado nos enfocaremos en las operaciones de lectura, analizando tiempos con y sin cache, viendo la influencia de los índices en estas operaciones y analizando la diferencia entre los dos métodos para llevar a cabo las operaciones de lectura: 'one' y 'at once'. En ambos casos se leen todos los registros de la base de datos, la diferencia es que en el método one se abre y cierra conexión por cada registro mientras que en at once la conexión se mantiene activa hasta el final del proceso. Estos tests han sido ejecutados con distintos tamaños de datos y en múltiples ocasiones, siempre observando los resultados y teniéndolos en cuenta para la redacción de este informe.

Para realizar pruebas iniciales diseñamos una caché propia, la cual hemos decidido mantener para comparar con Memcached. En teoría hemos aprendido que Memcached utiliza una arquitectura cliente-servidor y durante la práctica hemos tenido que conectarnos a un servidor. Es posible que la conexión con el servidor haya podido ralentizar el proceso de la caché y por eso en algunos casos hemos obtenido mejores rendimientos con la caché propia ya que trabaja de manera local.

5.3.1 Consultas sobre PKs

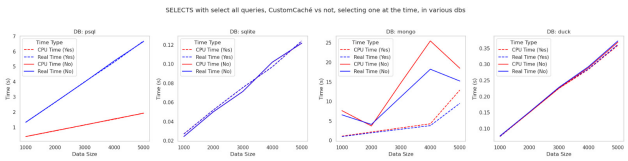

Figure 5: Subplots por DB, con CustomCaché vs sin, one, sobre SELECT ALL queries

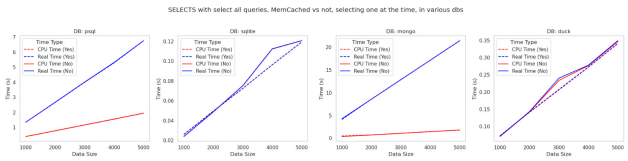

Figure 6: Subplots por DB, con Memcached vs sin, one, sobre SELECT ALL queries

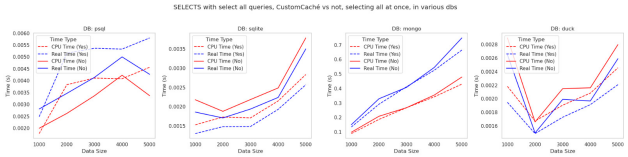

Figure 7: Subplots por DB, con CustomCaché vs sin, at_once, sobre SELECT ALL queries

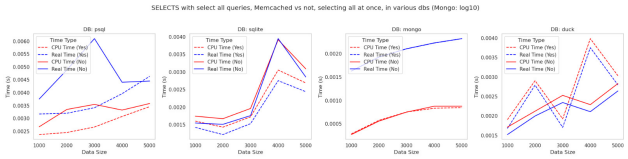

Figure 8: Subplots por DB, con Memcached vs sin, at_once, sobre SELECT ALL queries

Realizamos dos tipos de operaciones de lectura: Joins y búsqueda de claves primarias. Las operaciones de Join suponen realizar un INNER JOIN en las bases de datos relacionales (postgresql,sqlite y duckdb), mientras que en mongodb se analizará la diferencia entre a el uso de relaciones incrustadas (usando documentos embebidos) y relaciones referenciadas (conectando colecciones diferentes por medio de referencias). En las búsquedas de claves primarias buscaremos la clave primaria (dni) de la tabla usuarios en cada caso.

Hemos observado que claramente el método 'at once' es mucho más rápido que el método 'one' ya que el hecho de abrir y cerrar constantemente la conexión hace que sea mucho más lento.

5.3.2 Consultas con JOIN

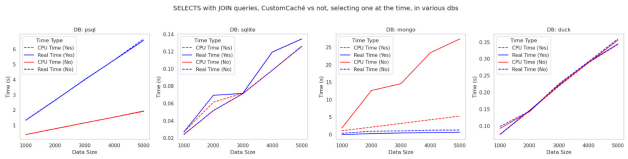

Figure 9: Subplots por DB, con CustomCaché vs sin, one, sobre JOIN queries

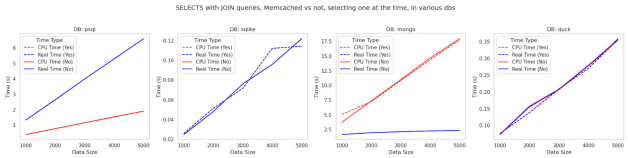

Figure 10: Subplots por DB, con Memcached vs sin, one, sobre JOIN queries

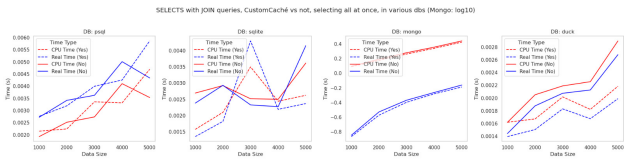

Figure 11: Subplots por DB, con CustomCaché vs sin, at_once, sobre JOIN queries

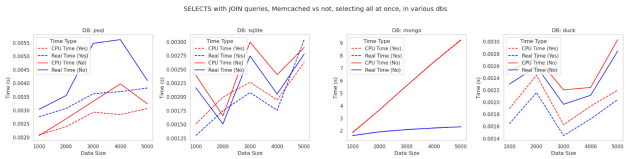

Figure 12: Subplots por DB, con Memcached vs sin, at_once, sobre JOIN queries

Como hemos explicado con anterioridad realizamos los tests con una consulta de join que busca los coches asociados a cada persona que se encuentra en la tabla de usuarios. En las bases de datos relacionales hemos podido observar que esta operación está bastante bien optimizada y en mongodb hemos observado que no está diseñado para ser rápido en operaciones de lectura con relaciones referenciadas.

En PostgresSQL los INNER JOINs generalmente funcionan rápido. Investigando, hemos descubierto que este utiliza un optimizador de consultas avanzado, que elige el mejor plan de ejecución para realizar las uniones, esto concuerda con nuestras observaciones ya que hemos podido comprobar que en postgresql los JOIN son bastante rápidos en comparación con otros SGBD como mongo. Con índices en el JOIN se mejora significativamente el rendimiento, reduciendo el tiempo de búsqueda.

En sqlite para consultas con tamaños de datos pequeños (como en la gráfica), las operaciones de lectura son rápidas, en cambio, a medida que aumentamos el tamaño de los datos perdía eficiencia progresivamente.

En duckdb generalmente los tiempos de lectura en JOINS es bajo ya que como ha sido mencionado en teoría, duckdb está diseñado para manejar grandes volúmenes de datos y procesar uniones de manera eficiente. Es por ello que creemos que es el SGBD más rápido a la hora de realizar lecturas. Por otro lado, el uso de índices en duckdb no mejora el rendimiento de la consulta ya que duckdb utiliza otros métodos para la optimización.

Tras realizar las mediciones en mongodb, hemos llegado a la conclusión de que mongodb no es recomendable para realizar relaciones referenciadas. Su rendimiento con $lookup es bastante más lento que los INNER JOINs de las bases de datos SQL. Obtenemos mucho mejor rendimiento cuando realizamos la búsqueda utilizando relaciones incrustadas ya que no es necesaria la presencia de $lookup y toda la información está contenida en el mismo documento. Sin embargo, hemos observado que los índices en $lookup han mejorado el rendimiento de la búsqueda haciéndola significativamente más rápida.

Todas estas operaciones han sido realizadas usando caché y sin usar caché. Hemos podido comprobar que el uso de la caché en la mayoría de los casos ha resultado beneficioso ya que al tener almacenada la búsqueda no ha sido necesario realizar la consulta con el JOIN lo cual hace que crezcan los tiempos.

5.3.3 Consultas sobre PKs

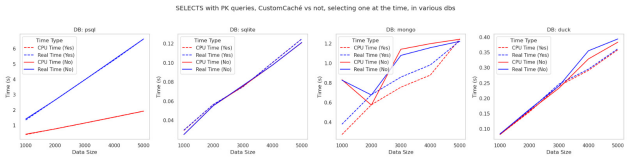

Figure 13: Subplots por DB, con CustomCaché vs sin, one, sobre PK queries

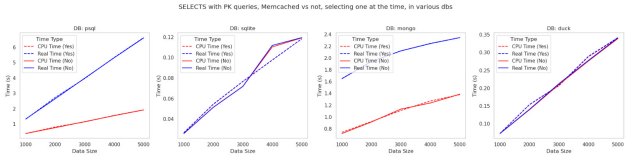

Figure 14: Subplots por DB, con Memcached vs sin, one, sobre PK queries

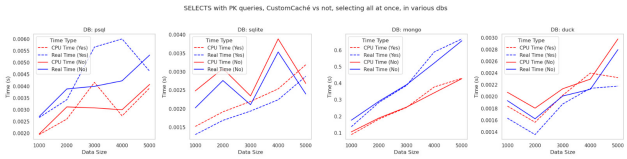

Figure 15: Subplots por DB, con CustomCaché vs sin, at_once, sobre PK queries

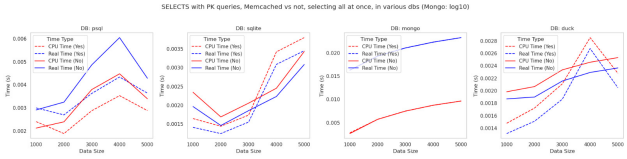

Figure 16: Subplots por DB, con Memcached vs sin, at_once, sobre PK queries

En sqlite y postgresql la búsqueda de las claves primarias ha resultado bastante rápida, ya que tiene mecanismos para optimizar la búsqueda en claves primarias que crean índices automáticamente sobre la clave primaria. Por ello el rendimiento creando índices manualmente no mejora la eficiencia.

No existe una clave primaria realmente pero tiene una clave única automática llamada _id, que actúa como una clave primaria. MongoDB crea automáticamente un índice en este campo _id, lo que hace que las búsquedas en ese campo sean extremadamente rápidas. Por otro lado, como esta columna ya está indexada no tiene sentido crear un índice sobre la clave primaria.

Duckdb no cuenta con un sistema de claves primarias por ello al realizar esta consulta se obtienen los mismos resultados que seleccionando cada documento. En este caso el uso de índices sí que mejora el rendimiento ya que no se crean automáticamente con la clave primaria (no existe).

5.3.4 Consultas sobre PKs

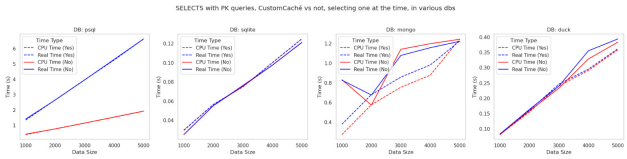

Figure 17: Subplots por DB, con CustomCaché vs sin, one, sobre PK queries

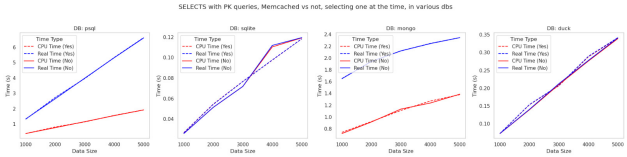

Figure 18: Subplots por DB, con Memcached vs sin, one, sobre PK queries

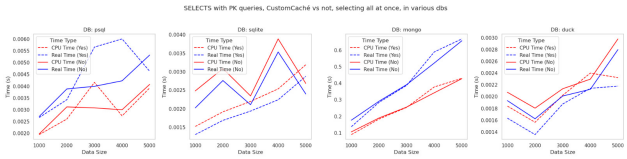

Figure 19: Subplots por DB, con CustomCaché vs sin, at_once, sobre PK queries

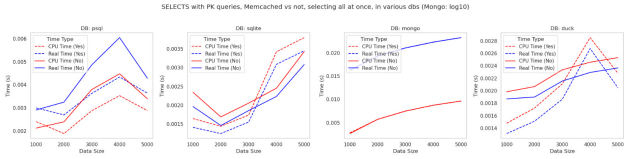

Figure 20: Subplots por DB, con Memcached vs sin, at_once, sobre PK queries

En sqlite y postgresql la búsqueda de las claves primarias ha resultado bastante rápida, ya que tiene mecanismos para optimizar la búsqueda en claves primarias que crean índices automáticamente sobre la clave primaria. Por ello el rendimiento creando índices manualmente no mejora la eficiencia.

No existe una clave primaria realmente pero tiene una clave única automática llamada _id, que actúa como una clave primaria. MongoDB crea automáticamente un índice en este campo _id, lo que hace que las búsquedas en ese campo sean extremadamente rápidas. Por otro lado, como esta columna ya está indexada no tiene sentido crear un índice sobre la clave primaria.

Duckdb no cuenta con un sistema de claves primarias por ello al realizar esta consulta se obtienen los mismos resultados que seleccionando cada documento. En este caso el uso de índices sí que mejora el rendimiento ya que no se crean automáticamente con la clave primaria (no existe).

5.4 Testing de consultas de actualización (updates)

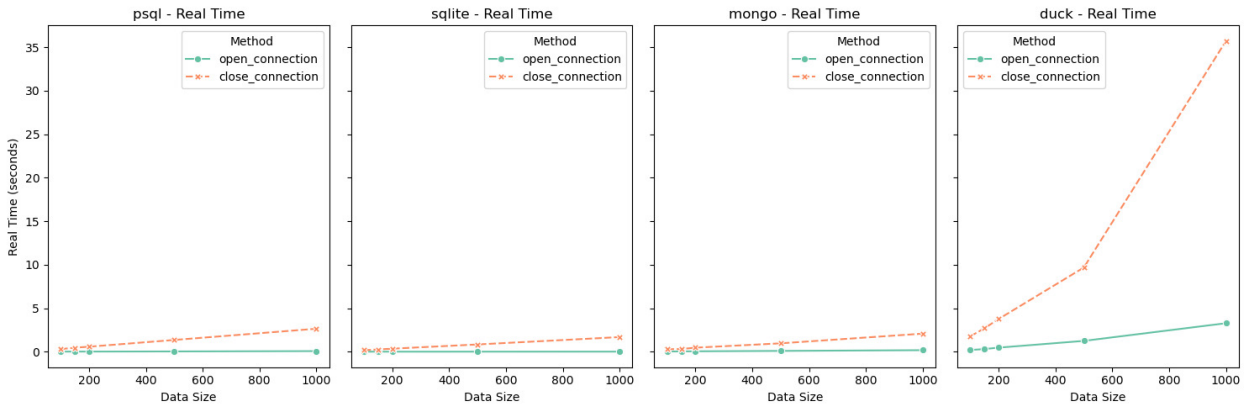

Figure 21: Subplots comparativos de operaciones de update, abriendo y cerrando conexión, en distintos SGBD

Las actualizaciones de los distintos registros en las bases de datos se realizan con los mismos métodos que en las operaciones de lectura: 'one' y 'at once'. Como hemos relatado previamente 'one' es mucho más lento que 'at once'.

En teoría hemos estudiado que mongo es una base de datos orientada a documentos , por tanto, el rendimiento de las actualizaciones depende de la estructura de los documentos. Las actualizaciones son a nivel de documento, lo que significa que todo el documento debe ser reescrito en disco cuando se modifica. La operación de UPDATE por lo tanto es relativamente rápida comparada con las bases de datos relacionales (sqlite y postgresql) si solo se modifica un documento, sin embargo, si el dato a modificar esta en varios documentos de forma referenciada y se requiere un $lookup puede ser muy lento como ya hemos comprobado en las operaciones de lectura.

En sqlite hemos descubierto que las actualizaciones bloquean la base de datos entera (hemos tenido que solucionar errores de este tipo), por ello creemos que puede afectar al rendimiento y ser más lento que postgresql. El uso de índices puede mejorar el rendimiento ya que facilita encontrar el registro que se desea actualizar.

Hemos comprobado que en postgresql las operaciones UPDATE tienen un buen rendimiento. Creemos que puede ser por la capacidad de manejo de transacciones que tiene incorporada. El uso de índices hace que los procesos de actualización sean más eficientes.

Como hemos visto en las clases teóricas, duckdb está diseñado para transacciones OLAP y no para transacciones OLTP. Aún así, soporta operaciones de actualización, pero su rendimiento no está optimizado para transacciones de alto volumen, como en MongoDB o postgresql. Duckdb no es una base de datos orientada a índices como PostgreSQL o MongoDB. Por tanto, las actualizaciones no suelen verse afectadas por índices, ya que el acceso a los datos se gestiona más a nivel de bloque columnar.

En general, los índices ayudan a mejorar la velocidad para encontrar el registro que vamos a actualizar. Sin embargo, si el campo actualizado está indexado, el índice también necesita ser modificado. En el caso de actualización de la ciudad los índices agilizan la operación ya que el campo ciudad no estaba indexado, en cambio si modificamos el DNI será más costoso porque tendrá que actualizar el índice.

6 Automatización. Pipeline

Una vez realizados y probados todos los tests por separado, es hora de automatizar el proceso de testing para poder replicarlo con distintos parámetros y bases de datos. Para ello, diseñamos la clase AutoTester, que implementará un método run_tests() que ejecutará todo lo desarrollado anteriormente de manera secuencial. Esta clase tendrá como parámetros el path de entrada de datos (CSV de códigos postales y coches), los paths de salida de datos (los resultados de los tests se mostrarán por pantalla, pero también se guardarán en formato CSV y los plots de tiempo que se generen en los tests), así como otros parámetros de configuración generales como las iteraciones por cada test, los SGBD a considerar, el proveedor de datos o las cachés (punteros a objetos creados, si no se crean automáticamente en la clase), los tamaños de datos de prueba (que se pasarán en un diccionario, justificado aquí.

Una vez creada, simplemente iniciamos una instancia de esto, AutoTester(), que podemos incluso inicializar sin parámetros (todos presentan valores predeterminados), para a continuación invocar el método run_tests() y ver cómo se van obteniendo distintos outputs a los tests.

7 Decisiones de diseño y arquitectura del código

Para diseñar toda la práctica, se ha optado por una arquitectura modular, buscando la mayor independencia entre componentes posible. Esto es fundamental debido a que, para códigos o sistemas con algo más de tamaño o complejidad, es muy fácil acabar entrelazando numerosas funciones, lo que hace que el código acabe siendo imposible de debuggear o mantener.

Por ello, se ha separado el código en los módulos util, con funciones de uso general y constantes como las queries a ejecutar, data_generation, que contiene las clases de proveedores, data_schema, con la definición del esquema de datos (tablas, en MongoDB no lo necesita al tener schema-on-read y no on-write como lo presentan psql, sqlite3 y DuckDB), db_operations, que contiene funciones para ejecutar operaciones CRUD sobre los 4 SGBD estudiados, testing y testing_pipeline, que implementan las funciones para testar y la definición del proceso de manera secuencial, y cache_implementation, que implementa las funciones y clases relativas a MemCached y a la caché propia. Técnicamente hay un último módulo, connection, pero que contiene simplemente los parámetros de conexión a cada BdD para que resulte más sencillo cambiarlo al compartir el código del proyecto entre distintos programadores.

El flujo principal del programa se desarrolla en un notebook de IPython, main.ipynb, al que se importan las funciones desarrolladas y se intenta mantener el código lo más corto y conciso posible, para presentar de manera más clara los outputs de los tests y los resultados concluidos.

En cuanto a las decisiones de diseño, cabe destacar que en DuckDB podríamos haber utilizado tanto una conexión a un fichero 'db' o a la propia memoria del ordenador. Hemos optado por manipular ficheros en el desarrollo de esta práctica.

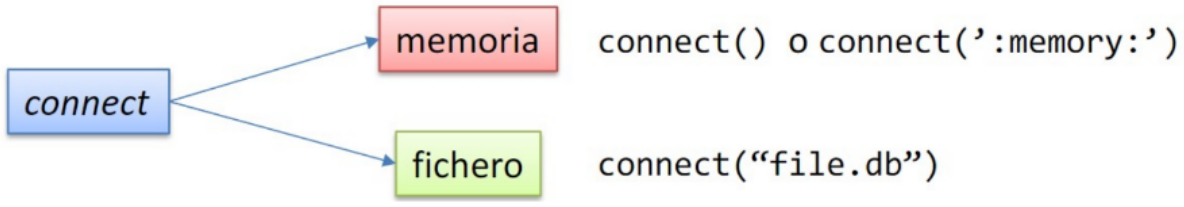

Figure 22: Diagrama posibles dos usos de DuckDB

Otra decisión de diseño que hemos tomado, y como es lógico, es que los métodos de la Caché diseñada

por nosotros tenga los mismos métodos y parámetros (es decir, una interfaz equivalente) a la que proporciona Memcached, ya que esto permite reutilizar funciones y procedimientos para las dos, disminuyendo la redundancia en el código y la por ende la posibilidad de errores.

En cuanto a las funciones de perfilado de memoria y tiempo proporcionadas en teoría, se han modificado ligeramente, añadiendo algún parámetro opcional para el correcto funcionamiento del flujo (por ejemplo, both:bool = False, ya que no se puede hacer la prueba de inserción varias veces si no es con datos distintos, por restricciones de clave primaria y unicidad, sin borrar en medio).

Por último y en el caso del pipeline de testeo, hemos decidido que el parámetro de tamaños no sea una simple List[int], si no que represente un Dict[str, List[int], en el que las claves ('huge', big', 'small', 'ultra_small') se asocian a listas de tamaños. Esto se ha hecho con el propósito de hacer práctico el testeo, ya que, por ejemplo, la generación de datos se puede hacer perfectamente con 1e6 datos (unos 50segs), mientras que una consulta con JOINS sobre todos los registros de una colección MongoDB tardaría varios horas de hacerse con esas cantidades de observaciones.

8 Conclusiones de la práctica

En conclusión, el análisis comparativo de los sistemas de gestión de bases de datos SQLite, DuckDB, PostgreSQL y MongoDB revela que cada uno posee un modelo de datos y características únicas que los hacen adecuados para diferentes tipos de aplicaciones. Por ejemplo, hemos observado en apartados anteriores que MongoDB es muy eficiente en lo que respecta la inserción de datos, mientras que para consultas complejas como JOINS ¡ref pto 5.3.1¿ tiene un rendimiento pésimo comparado con sus contrapartes relacionales.

Al considerar mecanismos de caché e indexación, es posible mejorar aún más la gestión de datos, pero no tiene porque hacerlo siempre, como hemos podido comprobar en ciertas pruebas de consultas ¡ref pto 5.4¿. Concluimos que la elección del sistema de bases de datos adecuado es, por tanto, fundamental para optimizar el rendimiento, ya que factores como el tipo de datos, el volumen de información y los requisitos de acceso pueden influir en la eficiencia del sistema.

9 Bibliografía

References

[1] PostgreSQL Global Development Group. *PostgreSQL Documentation*. https://www.postgresql.org/docs/.

[2] MongoDB, Inc. *MongoDB Manual*. https://docs.mongodb.com/manual/.

[3] SQLite Consortium. *SQLite Documentation*. https://www.sqlite.org/docs.html.

[4] DuckDB Community. *DuckDB Documentation*. https://duckdb.org/docs/.

[5] Memcached Team. *Memcached: A distributed memory object caching system*. https://memcached.org/.

[6] Sam Smith. *Faker Documentation*. https://faker.readthedocs.io/en/master/.

[7] The Pandas Development Team. *Pandas Documentation*. https://pandas.pydata.org/docs/.

[8] Travis E. Olliphant et al. *NumPy Documentation*. https://numpy.org/doc/.

[9] The Python Software Foundation. *os — Miscellaneous operating system interfaces*. https://docs.python.org/3/library/os.html.

[10] The Python Software Foundation. *sys — System-specific parameters and functions*. https://docs.python.org/3/library/sys.html.

[11] The Python Software Foundation. *time — Time access and conversions*. https://docs.python.org/3/library/time.html.

[12] Stack Overflow. *Stack Overflow Community*. https://stackoverflow.com/.