Práctica 1

Grado en Ciencia e Ingeniería de Datos

Bases de Datos: integración y arquitecturas 2024-25

Análisis de rendimiento y perfilado

Pareja 02 → Héctor Tablero Díaz y Álvaro Martínez Gamo.

Apartado 1: Generación de datos sintéticos

Para la generación de datos sintéticos, se ha implementado un script en Python que utiliza la biblioteca `Faker` junto con proveedores personalizados para crear datos realistas de usuarios y vehículos. El código se estructura de la siguiente manera:

Clases de proveedores personalizados

Se han creado dos clases principales: `CustomUserProviders` y `CustomCarProviders`. Estas clases extienden la funcionalidad de Faker para generar datos específicos como DNI, matrículas de vehículos, y otros datos relevantes para el contexto español. Hemos optado por crear una sola clase por tipo de dato en lugar de una clase para cada función (DNI, teléfono, ubicación, modelo de coche, matrícula, vin…) por simplicidad y para evitar duplicar los datos importados.

Además se usan datos reales para tratar de hacer más fidedignos los datos generados por `Faker`:

- `CustomUserProviders`: Almacenamos una lista de ciudades con sus provincias correspondientes. También tenemos una lista de prefijos telefónicos que corresponden con dichas provincias.
- `CustomCarProviders`: Guardamos modelos de coche con sus respectivos tipos y fabricantes. Para las matrículas antiguas (generadas con una probabilidad del 15%) hay una lista con códigos provinciales. En cuanto a las matrículas, se obtiene su fecha de matriculación aproximada a partir de una lista de últimas matrículas generadas en cada año según *https://www.seisenlinea.com/edad-matriculas/*.

Funciones de generación de datos

Se han implementado funciones como `get_user_entry()` y `get_car_entry(dnis : list[str])` para crear entradas individuales de usuarios y vehículos respectivamente. Estas funciones utilizan los proveedores personalizados para generar datos coherentes y realistas.

Funciones de escritura de datos

Se han desarrollado funciones para escribir los datos generados en archivos `json` y `csv`. Estas funciones incluyen mecanismos para manejar grandes volúmenes de datos, escribiendo en chunks para optimizar el rendimiento.

Paralelización del proceso

La parte final del código implementa un enfoque paralelo para la generación y escritura de datos, motivada por la lentitud en el momento de generar datos y los *crashes* del ordenador (pantalla azul) que sucedían cuando se llenaba la memoria al intentar crear 10 millones de datos en un solo diccionario. Utiliza el módulo `multiprocessing` de Python para distribuir la carga de trabajo entre múltiples procesos, aprovechando así los núcleos disponibles del CPU. Para asegurar el funcionamiento fluido del ordenador, solo se usan el 75% de los threads disponibles, dejando el resto para tareas del sistema operativo. Características clave de esta implementación incluyen:

- Uso de un `Pool` de procesos para ejecutar la generación de datos en paralelo.
- Implementación de un mecanismo de bloqueo (`Lock`) para garantizar la escritura segura en archivos compartidos desde múltiples procesos.
- División ajustada del trabajo entre los procesos disponibles, para poder manejar tamaños de datos que no son divisibles perfectamente entre los hilos disponibles. Por ejemplo, si se dividiesen 10 datos en 3 hilos, se encargarían de $[4, 3, 3]$ respectivamente.
- Generación de diferentes tamaños de conjuntos de datos $(10^3, 10^4, 10^5, 10^6, 10^7)$ para permitir un análisis de rendimiento considerando la escalabilidad.
- Escritura a disco desde el mismo thread, eliminando la necesidad de unificar los datos.

Este enfoque paralelo permite una generación de datos significativamente más rápida, especialmente para grandes volúmenes de datos, aprovechando eficientemente los recursos del sistema.

La paralelización añade una complejidad extra, ya que asegurarse de que no se repiten elementos únicos (`dni` en **users** y `vin, plate` en **cars**). Esta ha sido la evolución del método de generación:

1. No había control en cuanto a los datos generados, y comenzaron a salir errores al subir los datos a sus respectivas bases.
2. Se compartieron los datos mencionados previamente entre threads usando el `Manager` de `multiprocessing`. Este método no nos resultó satisfactorio ya que, a mayor número de datos generados, más frecuentes eran los conflictos y la necesidad de rehacer los datos, causando que el ritmo de generación bajase más de un 75% a partir de 1 millón de datos.
3. Se le dio una `id` a cada thread, con la que pueden delimitar la forma en la que generan sus datos. En cuanto a las implementaciones específicas:
    a. **Users**: se crean `num_processes` intervalos de longitud $\frac{99999999}{num\_processes}$ y cada thread selecciona el suyo según la `id` (`thread_n`).
    b. **Cars**: de forma similar, se calcula $\frac{624}{num\_processes}$. Este número se obtiene de considerar las 25 letras usadas y tomar 2 caracteres, lo cual supone 25^2 opciones (se resta 1 ya que python comienza a contar en 0). Dentro del rango se obtiene un número aleatorio para `plate`, el cual se convierte a 2 letras que se añaden al final de las matrículas (por su cuenta en matrículas antiguas y después de otra letra aleatoria válida en las actuales). En el caso de `vin`, se generan 15 letras aleatorias sin restricciones y se añaden 2 al final siguiendo el mismo método de `plate`.

Ya que este método vuelve a permitir que los threads puedan generar datos sin comunicarse entre sí y también reduce la probabilidad de choques con claves existentes, es tremendamente veloz y eficiente. A continuación se muestra una tabla de tiempos de generación:

Tabla\Cantidad*	1000	10000	100000	1000000	10000000
Users	0s	0s	1s	15s	2m39s
Cars	0s	0s	0s	2s	37s

**Nota: Estos tests se han realizado mientras el ordenador realizaba otras tareas, y no tienen en cuenta el tiempo de escritura en disco.*

Almacenado de datos

Hemos decidido implementar el código para subir los datos a los diferentes gestores en otro archivo (`Ejs 2 y 3.ipynb`) en lugar de en `data.py` para evitar tener código duplicado de forma innecesaria. Los detalles se pueden leer en el Apartado 2.

Diferencia entre los modos de almacenado

En esta práctica estamos trabajando con 2 formas de guardar los datos: estructurados (`SQL`: `SQLite3, DuckDB, PostgreSQL`) y no estructurados (`MongoDB`). Las ventajas principales de SQL son que funciona mucho más rápido y es más robusto dado que obliga a que los datos se introduzcan de cierta forma. Por otro lado, los datos no estructurados (en `BSON` como `Mongo`) hacen que el desarrollo de aplicaciones sea mucho más rápido y flexible, ya que no hay problemas al añadir o quitar campos a menos que los quieras convertir en obligatorios.

Apartado 2: Perfilado de tiempo y memoria

Para implementar el perfilado de tiempo y memoria de las operaciones de inserción, lectura y actualización de datos en diferentes bases de datos, se ha desarrollado un sistema orientado a objetos que permite una generalización eficiente para todas las bases de datos consideradas. Este enfoque facilita la extensibilidad y el mantenimiento del código, permitiendo añadir nuevas bases de datos con facilidad.

La implementación se basa en una estructura de clases que incluye:

- Una clase abstracta `DB` que define la interfaz común para todas las bases de datos.
- Clases específicas para cada base de datos (`MongoDB, PostgresqlDB, Sqlite3DB, DuckDB`) que heredan de la clase base e implementan sus métodos abstractos.
- Una clase `Measurements` que se encarga de realizar las mediciones de tiempo y memoria para las diferentes operaciones en cada base de datos.

La clase base `DB` proporciona métodos abstractos para las operaciones comunes como inserción, lectura y actualización. Las clases específicas de cada base de datos implementan estos métodos de acuerdo con las particularidades de cada sistema de gestión de bases de datos. Además, `DB` gestiona internamente la creación de tablas, el uso de la caché y el ciclo de vida de las conexiones. Sobrescribe el método `__del__(self)` de Python para asegurarse de que los elementos en caché relacionados con la base de datos específica sean eliminados garantizando que la conexión quedará cerrada cuando se eliminen todas las referencias al objeto. Esto funciona aún si el programa termina de forma inesperada. Además, la clase utiliza un atributo `oid` gestionado de forma automática por `Measurements` para asegurarse de que no se realicen operaciones por duplicado, ya que este fue un problema que enfrentamos debido al módulo de medición de memoria.

La clase `Measurements` contiene métodos para:

- Leer los datos de prueba desde archivos `json` y `csv`.
- Medir el tiempo de ejecución (tanto tiempo real como tiempo de CPU) y el uso de memoria para cada operación.
- Realizar mediciones con y sin caché, así como con y sin índices.
- Generar gráficos comparativos de rendimiento entre las diferentes bases de datos y operaciones.

Para medir el tiempo de ejecución, se utiliza la biblioteca `time` de Python, mientras que para medir el uso de memoria se emplea la función `memory_usage` de la biblioteca `memory_profiler`.

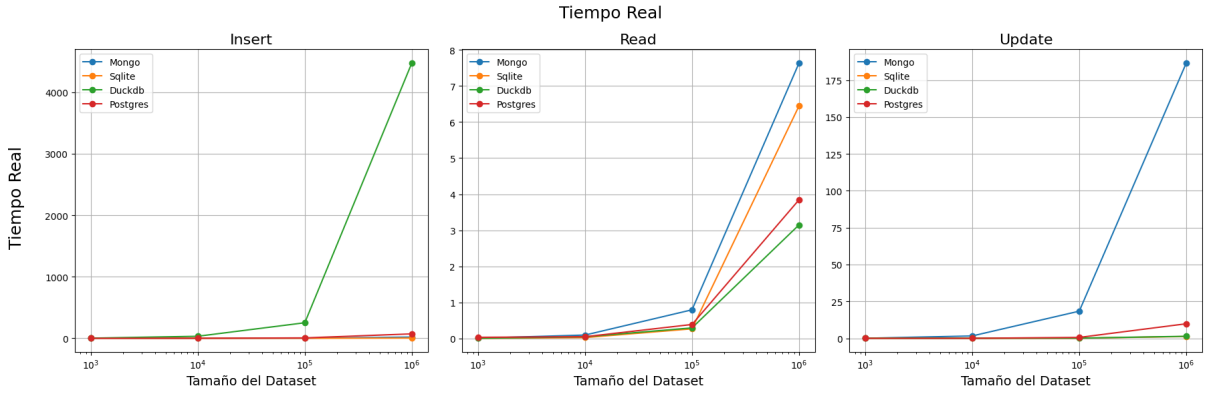
El sistema permite realizar pruebas con diferentes tamaños de conjuntos de datos, mencionados en el apartado anterior, lo que facilita el análisis de la escalabilidad de cada base de datos. Además, se implementa un mecanismo de caché para evaluar su impacto en el rendimiento de las operaciones de lectura.

Los resultados de las mediciones se visualizan mediante gráficos generados con la biblioteca `matplotlib`, lo que permite una comparación clara y directa del rendimiento entre las diferentes bases de datos y operaciones.

Este enfoque orientado a objetos y la automatización de las mediciones permiten realizar un análisis exhaustivo y sistemático del rendimiento de las diferentes bases de datos, facilitando la identificación de fortalezas y debilidades de cada sistema en distintos escenarios de uso.

Tiempo real

- Resultado de las mediciones para las operaciones de **insert**, **read** y **update** con tamaños desde 10^3 hasta 10^6:



El eje Y representa el tiempo en segundos, mientras que el eje X muestra los distintos tamaños de conjuntos de datos utilizados en las pruebas.

Observaciones clave:

- **Rendimiento general:** Se aprecia que `MongoDB` y `DuckDB` tienden a tener tiempos de ejecución más altos en comparación con `PostgreSQL` y `SQLite3`, especialmente para operaciones de actualización para `MongoDB` y escritura para `DuckDB`.
- **Escalabilidad:** Todas las bases de datos muestran un aumento en el tiempo de ejecución a medida que crece el tamaño del conjunto de datos, pero la tasa de incremento varía. `MongoDB` y `DuckDB` parecen escalar algo peor como se ha mencionado antes.

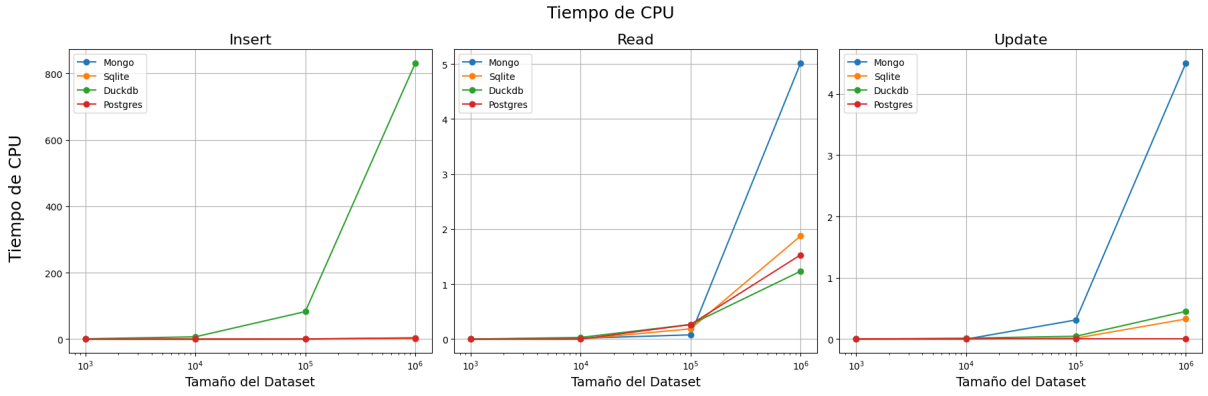
- **Operaciones específicas:**
  - **Inserción**: `MongoDB` destaca por su rápida inserción de datos, seguido de cerca por `SQLite3`. `DuckDB` destaca por su lenta inserción.
  - **Lectura**: Todas las bases de datos muestran un rendimiento excepcional en operaciones de lectura, manteniendo tiempos muy bajos incluso con grandes volúmenes de datos, siendo `DuckDB` la que menos tiempo tarda.
  - **Actualización**: Los tiempos de actualización son generalmente más altos que los de lectura para todas las bases de datos, con `MongoDB` mostrando tiempos particularmente elevados.
- **Consistencia:** `SQLite3` y `PostgreSQL` muestran un rendimiento más consistente entre las diferentes operaciones, generalmente con tiempos más bajos que `MongoDB` y `DuckDB`.

Estos resultados sugieren que para aplicaciones que requieren operaciones rápidas de inserción y lectura, especialmente con grandes volúmenes de datos, `MongoDB` podrían ser la opción más preferible. Sin embargo, si en un futuro tenemos intención de manipular esos datos, su rendimiento es inferior.

En general, `SQLite3` y `PostgreSQL` han mostrado un rendimiento estable y muy parecido para todas las operaciones, por lo que como un primer acercamiento para manipular datos sin saber muy bien cuál es nuestro objetivo con esos datos, nos van a ser de gran utilidad. Sin embargo, si lo que buscamos es una rápida inserción para una posterior lectura sin realizar modificaciones en los datos, `MongoDB` es mucho mejor opción.

Tiempo CPU

- Resultado de las mediciones para las operaciones de **insert**, **read** y **update** con tamaños de 10^3 hasta 10^6:



El eje Y representa el tiempo en segundos, mientras que el eje X muestra los distintos tamaños de conjuntos de datos utilizados en las pruebas.

Observaciones clave:

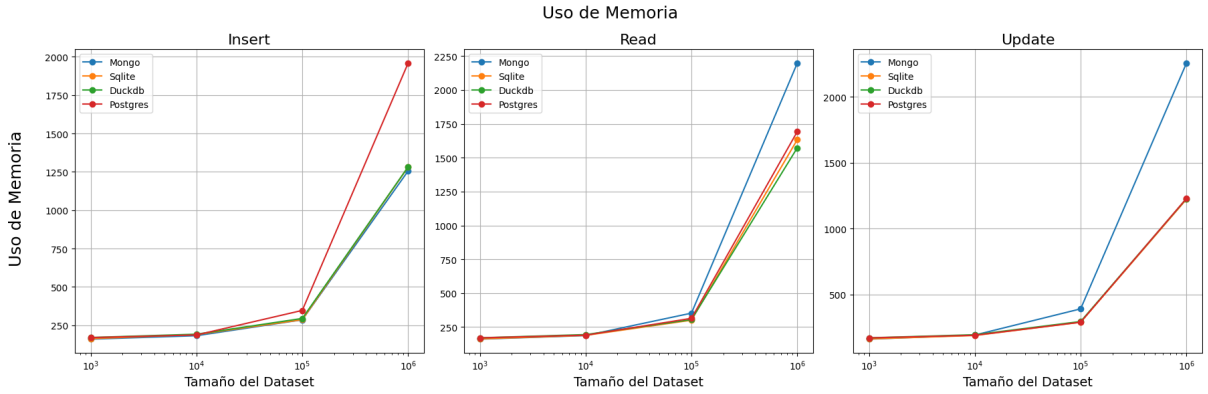
- **Rendimiento general:** El tiempo de CPU muestra patrones similares al tiempo real, con `MongoDB` y `DuckDB` generalmente requiriendo más tiempo de CPU, especialmente para operaciones de actualización y escritura respectivamente.
- **Escalabilidad:** Todas las bases de datos muestran un aumento en el tiempo de CPU a medida que crece el tamaño del conjunto de datos, con `MongoDB` y `DuckDB` mostrando una escalabilidad menos eficiente.

- **Operaciones específicas:**
  - **Inserción:** `DuckDB` requiere significativamente más tiempo de CPU que el resto de bases de datos, tal es así, que la gráfica muestra como todas menos `DuckDB` se pisan en tiempos.
  - **Lectura:** Todas las bases de datos muestran un rendimiento eficiente en operaciones de lectura, con tiempos de CPU muy bajos incluso para grandes volúmenes de datos, siendo `MongoDB` la que mas tarda.
  - **Actualización:** `MongoDB` muestra tiempos de CPU notablemente altos para actualizaciones, especialmente en comparación con las otras bases de datos.
- **Consistencia:** `SQLite3` y `PostgreSQL` muestran un uso de CPU más consistente y generalmente más bajo a través de las diferentes operaciones.

Estos resultados de tiempo de CPU refuerzan las observaciones hechas en el análisis de tiempo real. `MongoDB` es eficiente en CPU para inserciones, pero muestra un alto uso de CPU para lecturas y actualizaciones. `SQLite3` y `PostgreSQL` ofrecen un rendimiento de CPU más equilibrado en todas las operaciones, lo que las hace opciones sólidas para aplicaciones que requieren una variedad de operaciones. `DuckDB` muestra un alto uso de CPU para inserciones, lo que podría ser una consideración importante en escenarios donde la eficiencia de CPU es crucial.

Uso de memoria

- Resultado de las mediciones para las operaciones de **insert**, **read** y **update** con tamaños desde 10^3 hasta 10^6:



El eje Y representa el tamaño en MB, mientras que el eje X muestra los distintos tamaños de conjuntos de datos utilizados en las pruebas.

Observaciones clave:

- **Rendimiento general:** Se observa que `MongoDB` y `PostgreSQL` tienden a utilizar más memoria que `DuckDB` y `SQLite3` para todo tipo de operaciones.
- **Escalabilidad:** Todas las bases de datos muestran un aumento en el uso de memoria a medida que crece el tamaño del conjunto de datos, pero la tasa de incremento varía. `MongoDB` parece escalar peor en términos de uso de memoria, especialmente para operaciones de actualización, llegando a usar casi el doble de memoria.
- **Operaciones específicas:**
  - **Inserción:** `PostgreSQL` muestra un uso de memoria significativamente mayor para inserciones en comparación con el resto de bases de datos.

    - **Lectura:** Todas las bases de datos muestran un uso de memoria relativamente similar y constante para operaciones de lectura, con `DuckDB` mostrando el menor uso de memoria y `MongoDB` el mayor.
    - **Actualización:** `MongoDB` destaca por su alto uso de memoria en operaciones de actualización, especialmente con conjuntos de datos más grandes.
- **Consistencia:** `SQLite3` y `DuckDB` muestran un uso de memoria más consistente y generalmente más bajo a través de las diferentes operaciones y tamaños de datos.

Estos resultados sugieren que, para aplicaciones con restricciones de memoria, `SQLite3` y `DuckDB` podrían ser opciones más adecuadas, especialmente si se requieren operaciones frecuentes de actualización. `MongoDB` y `PostgreSQL`, aunque eficientes en ciertas operaciones, podrían no ser ideales para entornos con recursos de memoria limitados, particularmente cuando se manejan grandes volúmenes de datos o se realizan muchas operaciones de actualización.

Conclusiones finales

Basándonos en las discusiones previas sobre tiempo real, tiempo de CPU y uso de memoria, hemos extraído las siguientes conclusiones sobre las diferentes bases de datos y sus escenarios de uso más eficientes:

`MongoDB`

- **Puntos fuertes:** Rápida inserción de datos.
- **Consideraciones:** Alto uso de CPU y memoria en lecturas y actualizaciones, especialmente con conjuntos de datos grandes.

`PostgreSQL`

- **Puntos fuertes:** Rendimiento consistente en todas las operaciones, buen equilibrio entre velocidad y uso de recursos.
- **Consideraciones:** Uso de memoria relativamente alto, especialmente en inserciones.

`SQLite3`

- **Puntos fuertes:** Uso eficiente de memoria y CPU, rendimiento consistente en todas las operaciones.
- **Consideraciones:** Puede no ser la más rápida en operaciones individuales, pero ofrece un buen equilibrio general.

`DuckDB`

- **Puntos fuertes:** Excelente rendimiento en lecturas, uso eficiente de memoria.
- **Consideraciones:** Tiempos reales y de CPU de inserción excesivamente altos.

Escenarios mas eficientes

- Para aplicaciones con alta frecuencia de inserciones y lecturas en grandes volúmenes de datos, como sistemas de registro en tiempo real o aplicaciones IoT, `MongoDB` es una excelente opción.
- Si se requiere un rendimiento equilibrado en diversas operaciones con soporte para estructuras de datos complejas, como aplicaciones empresariales, sistemas de gestión de contenidos o plataformas de comercio electrónico, `PostgreSQL` es la elección ideal.

- Para aplicaciones con recursos limitados o que requieren portabilidad, como aplicaciones móviles, herramientas de escritorio o prototipos rápidos, `SQLite3` ofrece un rendimiento consistente y eficiente.
- En escenarios de aplicaciones analíticas que requieren consultas complejas y rápidas sobre grandes conjuntos de datos o escenarios de data warehousing para aplicaciones que priorizan la eficiencia en lecturas y el bajo uso de memoria, `DuckDB` destaca como la mejor opción.

Mejores combinaciones de bases de datos

- `MongoDB` y `DuckDB`: `MongoDB` permite una rápida inserción y almacenamiento de datos semi-estructurados, mientras que `DuckDB` facilita la realización de análisis complejos sobre grandes volúmenes de datos almacenados, combinando lo mejor de ambas en velocidad de inserción y eficiencia de consultas.
- `PostgreSQL` y `DuckDB`: `PostgreSQL` ofrece un soporte robusto para transacciones y estructuras complejas, mientras que `DuckDB` es excelente para optimizar el rendimiento en análisis de datos, proporcionando una combinación poderosa para sistemas que requieren transacciones y análisis rápidos.
- `SQLite3` y `MongoDB`: `SQLite3` es ideal para sistemas locales o móviles, y en combinación con `MongoDB`, que maneja grandes volúmenes de datos en la nube, permite un flujo eficiente entre almacenamiento ligero local y sincronización con datos masivos.
- `PostgreSQL` y `MongoDB`: Mientras que `MongoDB` se encarga de las inserciones rápidas y la flexibilidad del esquema, `PostgreSQL` puede ser utilizado para manejar datos estructurados complejos y garantizar integridad transaccional.

Apartado 3: Uso de mecanismos de caché e indexación de datos

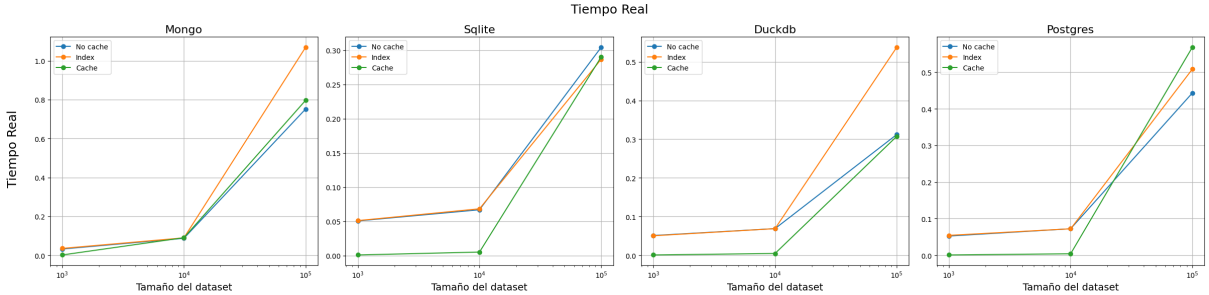
Implementación

Ya que previamente creamos la clase abstracta `DB`, la única modificación necesaria para añadir un sistema de caché a todas las clases es cambiarlo en `DB` antes de la llamada a las implementaciones particulares.

Para garantizar que la caché de una instancia de una de las implementaciones no perdure indefinidamente (lo cual afectaría a las mediciones subsecuentes) se guarda el registro de todas las queries ejecutadas y al eliminar la clase se ejecuta un `cache.delete(query)` para cada una.

También se eliminan todos los resultados cuando se ejecuta el método `insert` o `update` de la clase correspondiente, ya que existe una posibilidad de que los resultados almacenados dejen de coincidir con lo que devolvería la base de datos si la misma query se ejecutase de nuevo.

Resultados



El eje Y representa el tiempo en segundos, mientras que el eje X muestra los distintos tamaños de conjuntos de datos utilizados en las pruebas.

Se puede apreciar en los datos que la caché no ayuda tanto como cabría esperar, ya que tarda prácticamente lo mismo habilitándola que sin usarla, y a veces incluso más para tamaños de datos mayores. Esto indica que `pymemcached` es muy lento a la hora de des-serializar la información guardada.

Otra observación extraña y común en todas las gráficas es que no haya una diferencia notable entre operaciones con y sin índice, cuando el uso del índice debería acelerar la ejecución de las consultas. A través de las interfaces de los distintos SGBD hemos comprobado que los índices están siendo correctamente creados, y se puede comprobar en el código también.

Funcionamiento de la caché

Las cachés guardan información bajo una llave (`key`). Al tratar de acceder a dicha información, el sistema de caché comprueba si existe algún valor guardado para la `key` , y en caso positivo lo devuelve, mientras que si no existen datos le pide al programa que los obtenga de otra manera y los almacena para acelerar las siguientes peticiones.

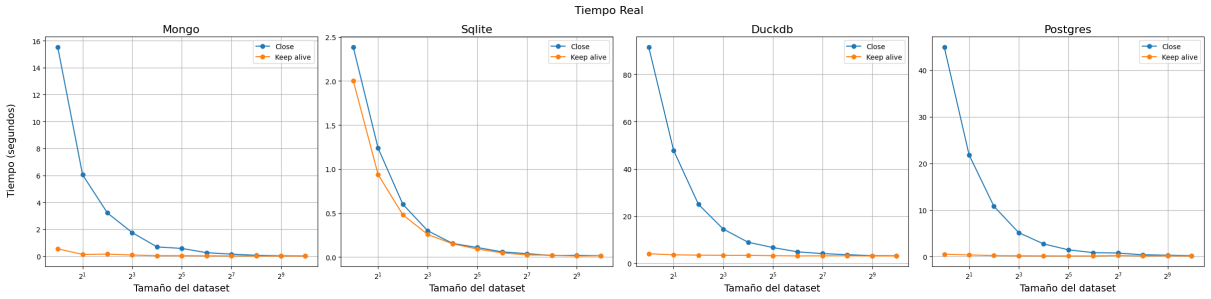
La ventaja principal es que los datos de las cachés se almacenan en RAM, por lo que suelen ser usables mucho más rápido que usando la base de datos directamente. Otra ventaja relacionada es que reduce la carga de trabajo de la base de datos, cosa que no afecta mucho en esta práctica pero sí puede afectar en aplicaciones reales con varias conexiones.

Posibles desventajas son que no hay garantías de que la información devuelta por la caché esté actualizada si se ha cambiado información en la base de datos directamente, por lo que hay que añadir protección contra estos casos desde el código, y que al consumir memoria puede hacer que otros programas o el sistema operativo se ralenticen.

Apartado 4: Automatizar el proceso de medida de rendimiento

Ya que este apartado solo consiste en unir piezas de ejercicios anteriores no vamos a mencionar nada de la implementación concreta. La única decisión de diseño que hemos tomado ha sido crear un nuevo archivo `Ej4gen.py` casi idéntico a `Ej 1.py` , ya que el apartado 1 solo pide generar datos y guardarlos en archivos en vez de devolverlos en dataframes y listas, por lo que hemos considerado oportuno dejar intacta dicha funcionalidad al tratarse esto de una práctica.

Inserción

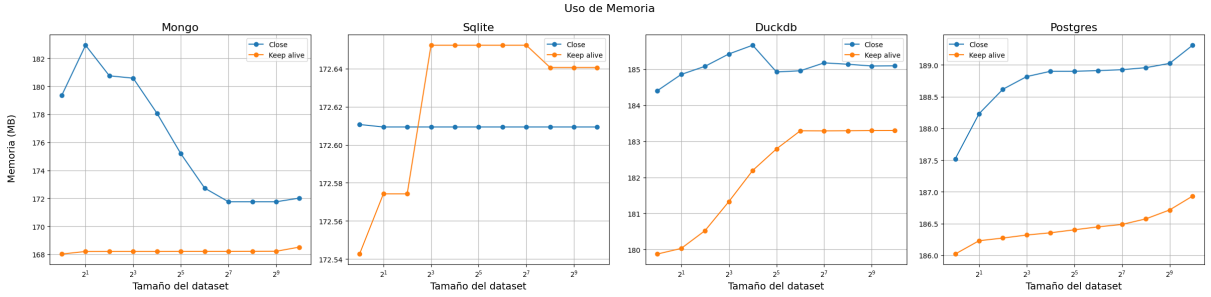


El eje Y representa el tiempo en segundos, mientras que el eje X muestra los distintos tamaños de batches utilizados en las pruebas.

Para comprobar el efecto de la inserción por lotes, además de subir los datos de uno en uno y todos de golpe, se han añadido puntos intermedios de tamaño $[1, 2, 4, \cdots, 2^n]$. Las líneas naranjas son aquellas en las que se ha mantenido una sola conexión para todas las operaciones, mientras que las azules son las que se reestablecían en cada operación.

En casi todas las bases de datos hay una diferencia notable entre mantener la conexión o cerrarla tras cada **insert** individual. Los puntos más destacables son:

- El mal rendimiento de `DuckDB` en **inserts**, como se había comentado en el apartado anterior, tardando más de 90 segundos para 1000 datos.
- El gran rendimiento de `QSLite3`, en el que apenas existe un overhead al cerrar y reabrir conexiones, como se puede ver por las dos líneas casi idénticas.



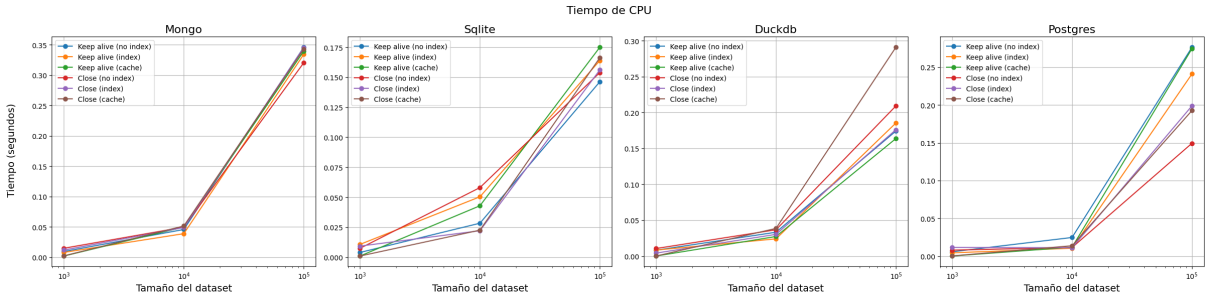
El eje Y representa el tamaño en MB, mientras que el eje X muestra los distintos tamaños de lotes utilizados en las pruebas.

Una tendencia clara entre todas las bases de datos es que cerrar la conexión y volver a abrirla para cada insert aumenta la memoria usada.

Mientras que la gráfica de `SQLite3` puede parecer un poco extraña, la diferencia total de los datos es de 0.1 MB (en lugar de los $10 \sim 15$ MB de las otras), por lo que puede interpretarse como si la memoria fuera constante.

Lectura

Resultados de la lectura registro a registro:



El eje Y representa el tiempo en segundos, mientras que el eje X muestra los distintos tamaños de conjuntos de datos utilizados en las pruebas.

Todos los tiempos siguen un aumento lineal según el tamaño del dataset (los ejes están alterados con `log10`).

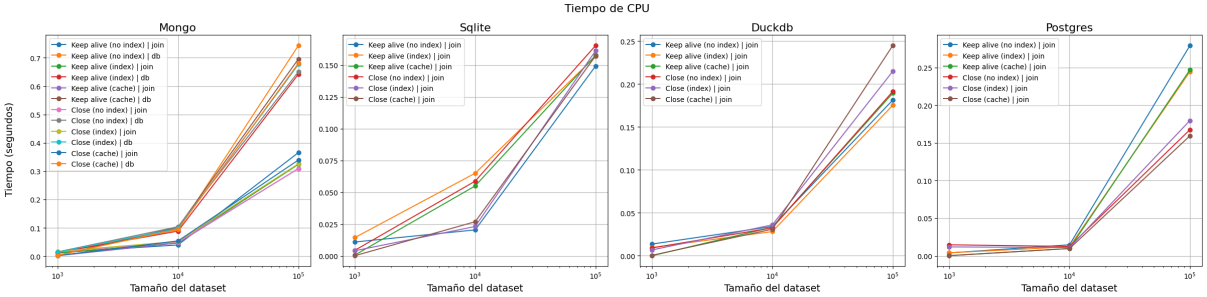
No se puede apreciar mucha diferencia entre ninguna de las estrategias usadas, pero estos han sido los resultados para cada Base de Datos:

- `MongoDB`: Cerrar (sin índice) $<$ Mantener (índice) $<$ Mantener (caché) $<$ Cerrar (caché) $<$ Cerrar (índice) $<$ Mantener (sin índice).
- `SQLite3`: Mantener (sin índice) $<$ Cerrar (sin índice) $<$ Cerrar (índice) $<$ Mantener (índice) $<$ Cerrar (caché) $<$ Mantener (caché).

- `DuckDB`: Mantener (caché) < Mantener (sin índice) < Cerrar (índice) < Mantener (índice) < Cerrar (sin índice) < Cerrar (caché).
- `PostgreSQL`: Cerrar (sin índice) < Cerrar (caché) < Cerrar (índice) < Mantener (índice) < Mantener (caché) < Mantener (sin índice).

Entre distintas bases de datos los resultados no son para nada consistentes, lo cual indica que hay mucho ruido en los resultados. Por ejemplo, se esperaría el mismo tiempo para la caché sea cual sea el estilo de la conexión, ya que no se llega a realizar ninguna operación, pero en `DuckDB` una variación de la caché es lo más rápido y la otra es lo más lento. Entre usar índice o no, y entre mantener o no la conexión no se puede ver ningún patrón.

Resultados con join:

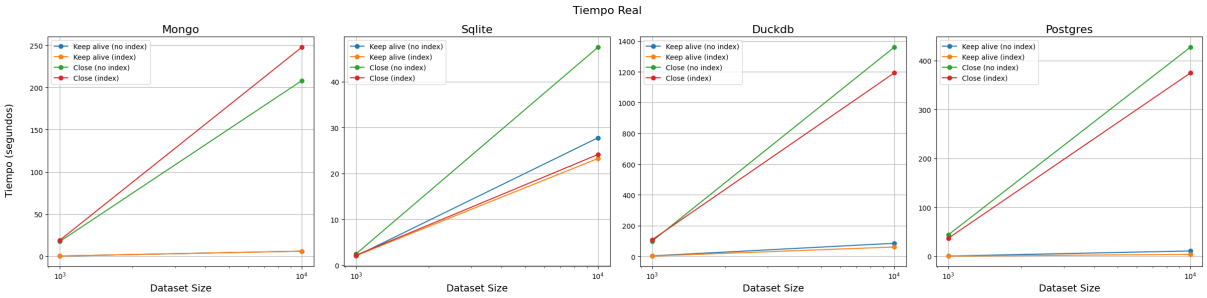


El eje Y representa el tiempo en segundos, mientras que el eje X muestra los distintos tamaños de conjuntos de datos utilizados en las pruebas.

En esta gráfica los resultados están más agrupados según la configuración. Comparando uno por uno:

- `MongoDB`: Algo a destacar es que hacer un join tarda considerablemente menos que leer de una tabla de documentos embebidos. De nuevo, es bastante extraño el comportamiento de la caché ya que se realiza la misma operación en todos los casos, y el tamaño de los datos devueltos es el mismo.
- `SQLite3`: Se puede apreciar claramente como la obtención de datos es mucho más rápida cuando se mantiene la conexión, y coincide con lo que cabría esperar.
- `DuckDB`: Aunque la diferencia es menos notable, se sigue el mismo patrón de la gráfica anterior.
- `PostgreSQL`: Aquí los resultados se han invertido y las operaciones en las que se cierran las conexiones son las más rápidas, al contrario de lo esperado.

Actualización



El eje Y representa el tiempo en segundos, mientras que el eje X muestra los distintos tamaños de conjuntos de datos utilizados en las pruebas.

Como se puede observar, todas las gráficas crecen de forma lineal, multiplicándose por $7 \sim 13$ al multiplicarse por 10 el tamaño del dataset. La única excepción a la norma es `SQLite3`, el cual muestra resultados muy parecidos entre los casos en los que se cierra la conexión antes de cada operación con los casos en los que se mantiene una sola conexión abierta. Como se podía ver en la gráfica de inserción por batches, esta base de datos tiene un overhead muy pequeño al abrir y cerrar conexiones.

Además, los resultados de las gráficas muestran que tardan más las operaciones en las que se cierra la conexión y también, salvo por `MongoDB`, es algo notable el incremento de tiempo si las tablas no tienen un índice.

Especificaciones del ordenador

- **RAM:** 32 GB
- **CPU:** Intel(R) Core(TM) i7-1360P 2.20 GHz
- **Sistema Operativo:** Windows 11 Pro (versión 22631.4317)
- **Versión de Python:** 3.12.4