Análisis procesamiento de grandes datos

Con el fin de comprender de manera práctica cómo distintos enfoques de programación influyen en el rendimiento de un sistema, se desarrollaron cuatro implementaciones que modelan la misma problemática: clases por valor, clases por punteros, estructuras por valor y estructuras por punteros.

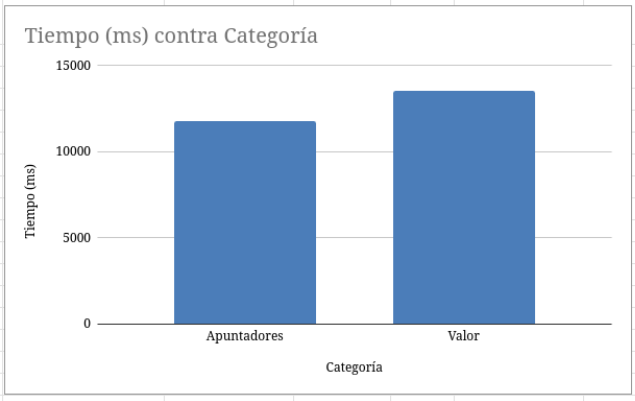
Este diseño experimental nos permite analizar no sólo cómo cambia el comportamiento según la forma de acceso a los datos (copias completas frente a referencias en memoria), sino también cómo la representación de dichos datos en la memoria (estructuras planas frente a clases con mayor complejidad) afecta los resultados.

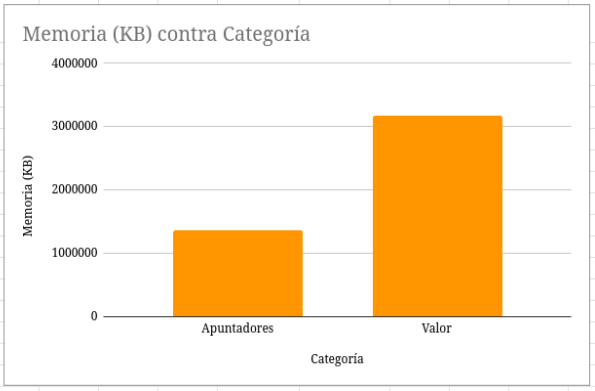
A través de pruebas con diez millones de registros y diversas operaciones de consulta, mediciones de tiempo de ejecución y consumo de memoria, fue posible identificar ventajas, desventajas en cada enfoque.

1. Apuntadores vs Valor

Cuando se manipulan datos en un programa se puede hacer por apuntadores, a través de su dirección de memoria por la que se accede al objeto, lo que optimiza memoria porque solo copia 8 bytes (Tamaño de la dirección de memoria). Sin embargo es necesario tener en cuenta la localidad de la memoria con apuntador, ya que los objetos pueden quedar dispersos en la heap (área de asignaciones dinámicas y puede quedar en cualquier lugar libre de la memoria).

Por otro lado, se manipula por valor, donde se hace una copia del contenido del objeto en otra variable cuando se va a modificar o trabajar con ella, esto permite independencia entre instancias pero implica un aumento en la memoria utilizada por la copia de los datos, que al trabajar con millones de datos implica un aumento significativo, que ni siquiera implica un beneficio así se almacenen de forma contigua en la memoria.



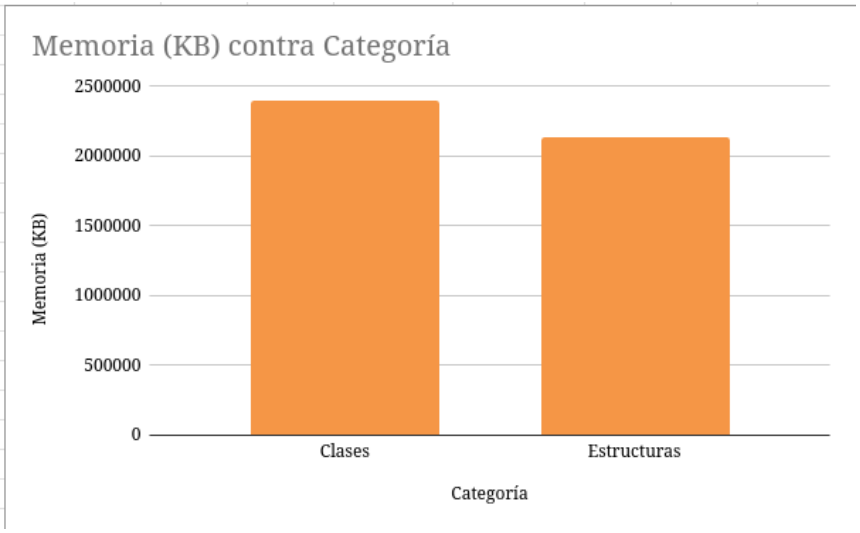


En las gráficas se evidencia como los apuntadores generan mejores métricas, tienen una disminución frente a los tiempos debido a que las operaciones predominantes en nuestro proyecto son de búsqueda y clasificación, sin embargo se evidencia que no es tan marcada como si lo viéramos opción a opción ya que hay otras implementaciones globales que no generan una diferencia tan marcada. En contraposición vemos el gasto de la memoria,

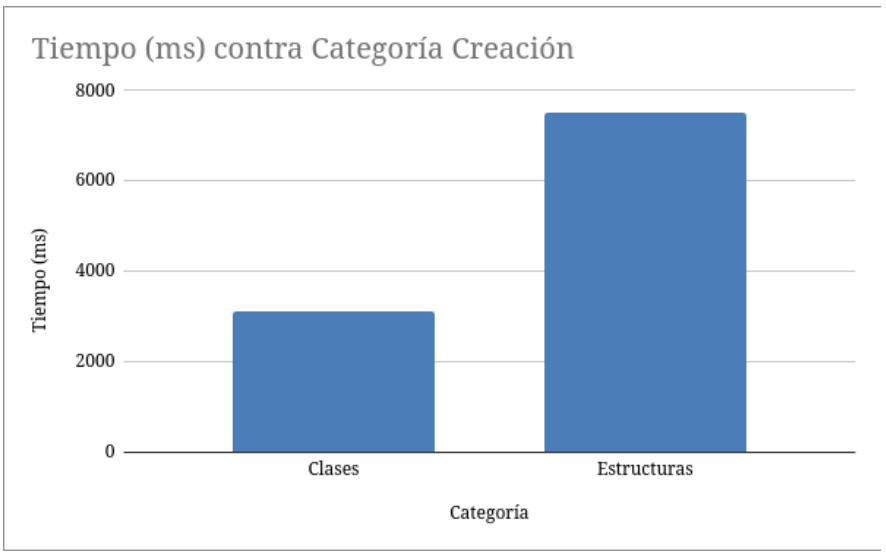
evidenciando una diferencia significativa de poco más del doble, debido a la copia de datos presentes en todas las operaciones que se realizan por valor.

2. Organización y representación en memoria: Clases vs Structs

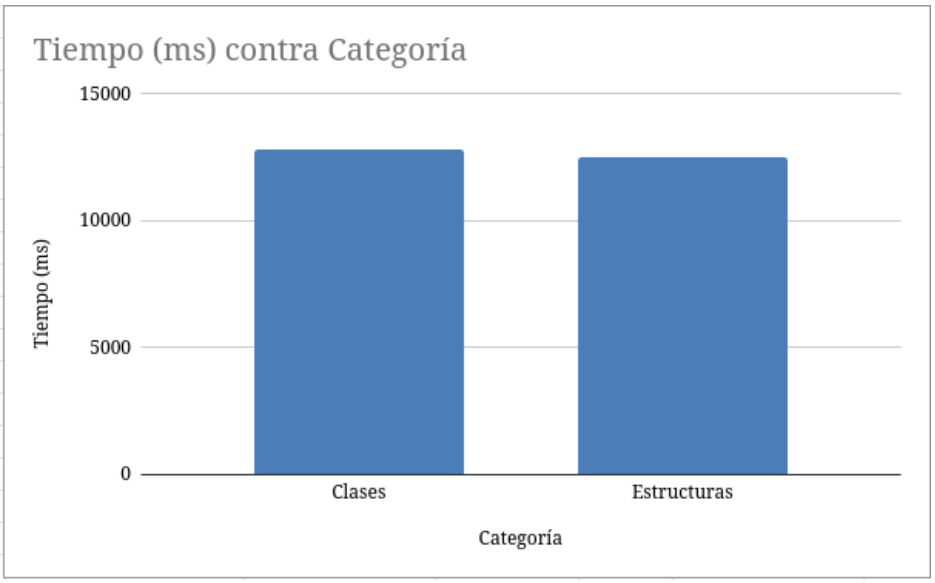
Los datos fueron representados en structs, almacenador “plano” de solo campos más simples, que son agregados en orden de declaración uno detrás de otro con posibles espacios de padding (alineación con la memoria, necesaria para acceder de forma correcta) y en Clases, almacenador más complejo al contar con métodos, constructores, herencias… lo que puede agregar un pequeño espacio adicional en la memoria (overhead), pero que puede permitir más eficiencia en la gestión de operaciones (agregar, mover…) y mejora en la alineación de los datos.



Sin embargo se evidencia que los resultados a la hora de la creación entre clases y estructuras son similares 1970688 KB vs 2018816 KB respectivamente, esto se debe a que almacenan el mismo número de elementos con los mismos campos y para ese momento el gasto de memoria depende del tamaño del objeto por el número de registros ya que no se han hecho operaciones adicionales, la diferencia que se evidencia puede ser por la alineación (padding) interno entre cada una.

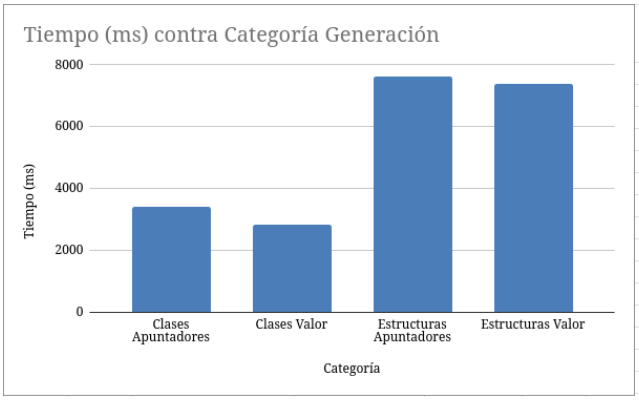


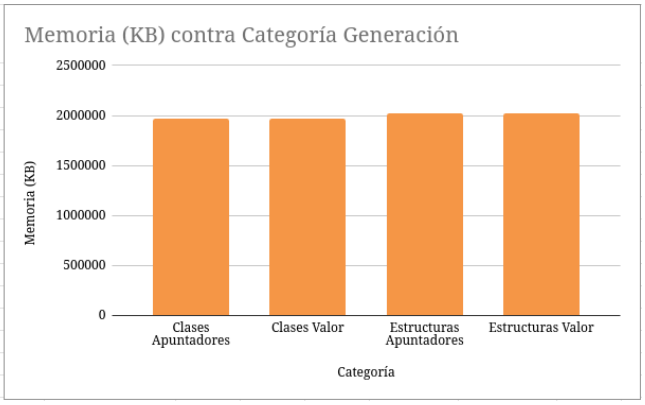
Ahora bien, si analizamos los tiempos de creación 3412 ms vs 7395 ms respectivamente evidenciamos que si hay una diferencia significativa, esto se debe a que las clases tienen sus constructores por defecto y los compiladores están optimizados para ellos, sin embargo la diferencia significativa se da por la implementación de la generación e iniciación de ambos,como acceden a los campos y los modifican.



Si bien en la creación de datos el tiempo es un aspecto diferenciador, al hacer un promedio entre las operaciones totales se evidencia que el tiempo tiende a converger, lo que significa que no depende del tipo de almacenamiento en el que están los datos la rapidez de respuesta, sino de la forma en la que se accede a la información como se evidenciará en los siguientes puntos.

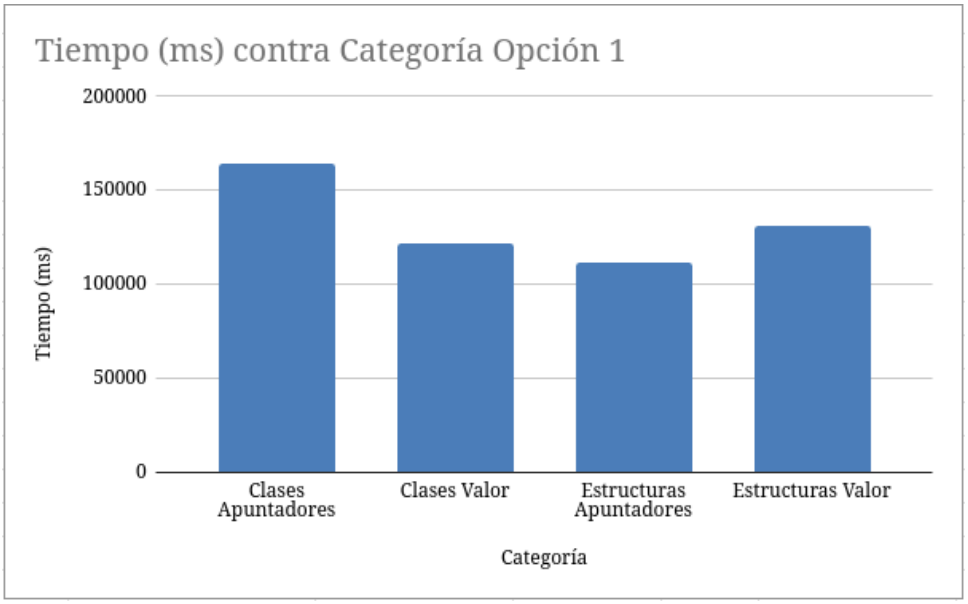
3. Generación de datos





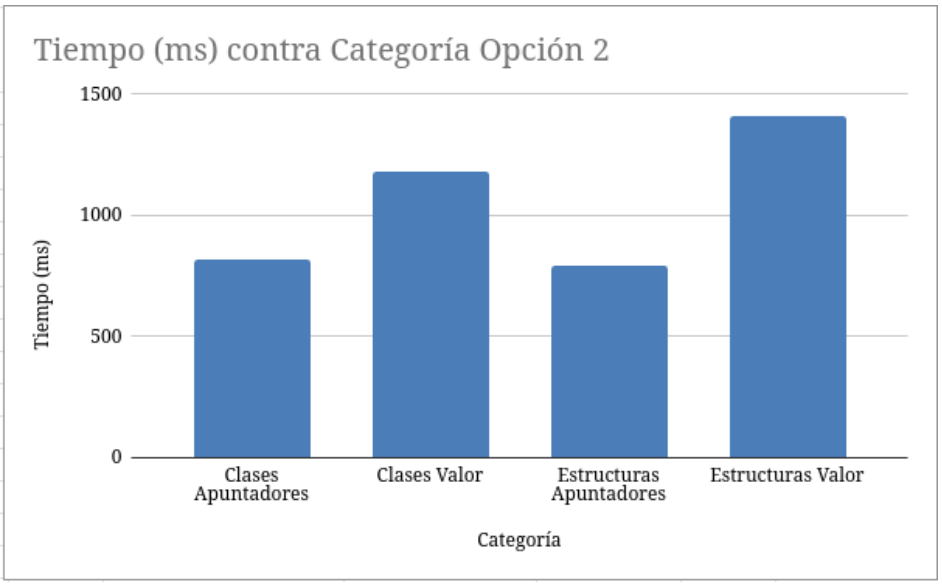
En la opción 0: creación de los 10 millones de registros se evidenció que el consumo de memoria fue prácticamente equivalente tanto en clases como en estructuras (≈2 GB), independientemente de si se usaron punteros o valores. Esto ocurre porque en ese momento ya están inicializados todos los objetos en memoria. Sin embargo, el tiempo de generación mostró una diferencia significativa: las clases tardaron 3 segundos, mientras que las estructuras tardaron más de 7 segundos, lo que sugiere diferencias en la forma en que se gestionaron las inserciones y optimizaciones del compilador. Es importante resaltar que, aunque la memoria base es parecida, la diferencia conceptual entre ambos enfoques es crítica: Mientras que los objetos creados por punteros se almacenan en el heap, su asignación masiva puede generar overhead, especialmente en el caso de clases, donde cada new individual añade costes de gestión de memoria. En contraste, cuando se usan valores, los 10 millones de registros se reservan como un bloque contiguo, lo que aprovecha mejor la localidad de memoria y reduce la sobrecarga inicial..

4. Visualización de datos

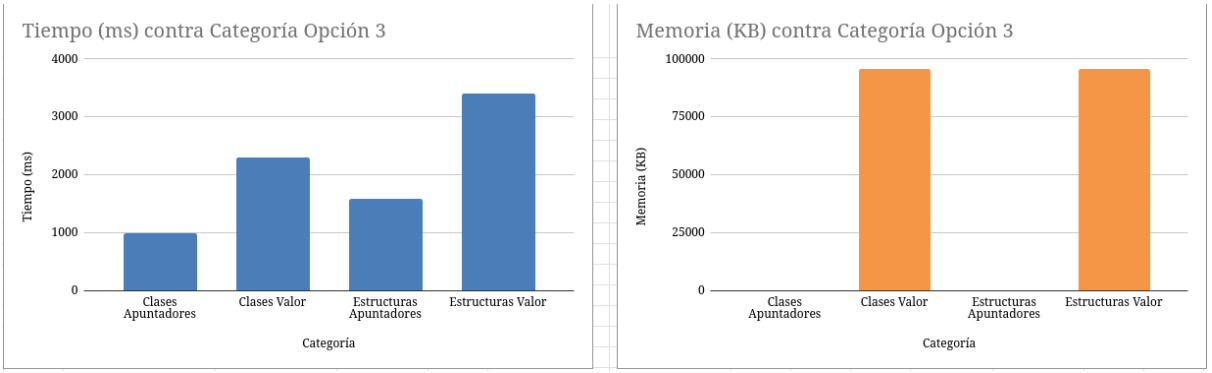


En la visualización completa de los 10 millones de registros (Opción 1) se observó que el consumo de memoria adicional reportado fue prácticamente nulo en todos los casos, ya que los datos ya estaban inicializados desde la opción anterior. La diferencia principal estuvo en

el tiempo de recorrido, donde las implementaciones por valor se beneficiaron de la localidad de memoria al tener los objetos almacenados en bloques contiguos, lo que permite a la CPU aprovechar la caché y recorrerlos con mayor eficiencia. En contraste, en las implementaciones por punteros el acceso es indirecto y, al estar los objetos distribuidos en distintas posiciones del heap, se generan más saltos de memoria y mayor probabilidad de fallos de caché, penalizando el rendimiento. Esto confirma que, aunque los punteros resultan más ventajosos en términos de evitar copias y reducir memoria en operaciones de búsqueda, en recorridos secuenciales largos el acceso por valor puede ofrecer mejor desempeño debido a la organización contigua de los datos.



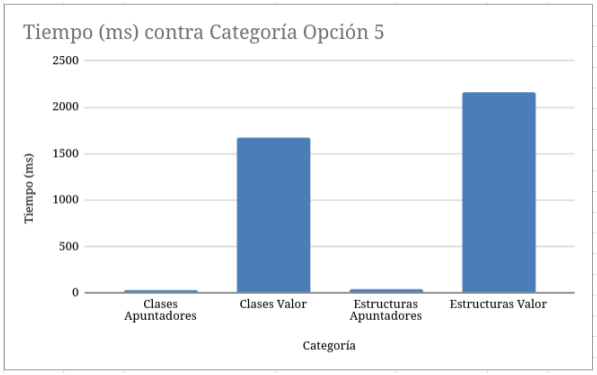
En la búsqueda por índice (Opción 2) se evidencia solo el tiempo empleado en la operación, el cual es el menor resultado de todas las operación debido a que no recorre todos los datos, sino que toma en el valor que se encuentre en esa posición en el array, esto también logra evidenciar como al realizar la operación por apuntadores se disminuye tiempo al acceder directamente por la memoria. Adicionalmente no se evidencia gasto adicional de moria porque no se hace un proceso de copia, se trabaja sobre el vector de datos directamente.

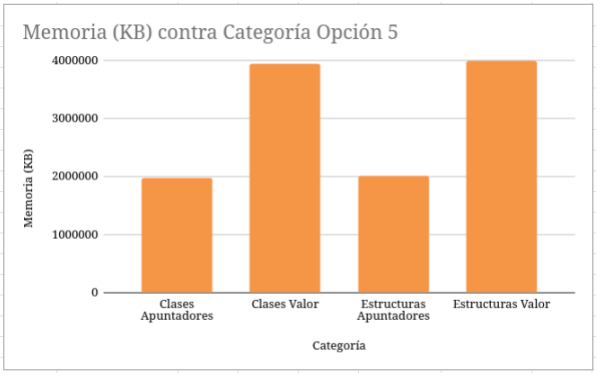


En buscar persona por ID (Opción 3) si se evidencia cambios en el tiempo empleado más variados incluso si es la misma forma de representación de los datos, esto por la manera en la que cada uno accede a la información a pesar de que implementan todos una búsqueda

binaria (aprovechando el hecho de que los ID's se encontraban organizados). Se sigue evidenciando que los apuntadores son más rápidos y además no generan un gasto de memoria puesto que no copian los datos en la función para trabajar con ellos.

5. Operaciones de búsqueda y consultas específicas (Opciones 5–10)





En las operaciones de búsqueda y consultas específicas (Opciones 5–10) se evidenció que trabajar con apuntadores, tanto en clases como en estructuras, mantiene la memoria más estable y reduce de forma considerable el tiempo de ejecución, ya que no se generan copias completas de los registros sino que se manipulan únicamente las direcciones de memoria. Esto permite evitar duplicados y hace que las búsquedas puntuales se ejecuten en milisegundos en lugar de segundos, como ocurre al trabajar por valor. En contraste, cuando se usan valores, cada operación implica la creación de copias temporales de los objetos, lo que en la práctica duplica el consumo de memoria (de ~2 GB a ~4 GB) y aumenta notablemente los tiempos. Aquí se observa que la diferencia entre clases y estructuras es mínima, ya que lo que realmente determina el desempeño es si los datos se manejan por punteros o por valor. La brecha se hace más marcada en consultas selectivas, como encontrar a la persona más longeva, mientras que en operaciones más globales o de escaneo (listar, contar, validar grupos) la ventaja se reduce, pues el costo dominante es recorrer todos los registros sin importar su representación.

6. Conclusión

El análisis mostró que la mayor diferencia de rendimiento no depende tanto de usar clases o estructuras, sino de la forma de acceso: punteros o valores. Los punteros resultan más eficientes en consultas puntuales al evitar copias completas y reducir memoria, mientras que los valores aprovechan mejor la localidad en recorridos largos. En conjunto, las clases ofrecen un comportamiento más consistente gracias a las optimizaciones del compilador, pero en promedio los tiempos tienden a converger. Así, la elección del enfoque debe responder al tipo de operación predominante y al equilibrio buscado entre memoria y velocidad.