Influencia del costo de operaciones en memoria principal en la duración de algoritmos de búsqueda en memoria secundaria

David Escobar Rupalí López Alfonso Valderrama

14 de octubre de 2020

Resumen

El objetivo principal de la investigación realizada consiste en estudiar la validez del modelo de memoria secundaria, mediante experimentación aplicada a implementaciones de algoritmos que intersectan listas almacenadas en disco. Usualmente, en el modelo de memoria secundaria, el costo de las operaciones en CPU es ignorado. En este estudio se evalúan tres algoritmos de búsqueda, los cuales difieren en cantidad de consultas I/O, y se muestra que, contrario a lo esperable, la duración de cada uno no es proporcional a dicha cantidad. Mediante la prueba de variantes al algoritmo que toma más tiempo, se evidencia que el número de operaciones más influyente en ese caso es el número de operaciones en memoria principal. Los resultados obtenidos revelan que los problemas que involucran consultas a memoria secundaria no deben pasar por alto el costo de las operaciones en CPU si este último es mayor al de las primeras. Se concluye además, que en dichos casos es necesario integrar ambos problemas para el diseño de algoritmos de esta categoría.

1. Introducción

Cuando el volumen de datos a manejar supera la capacidad de la memoria principal, éstos se almacenan en memoria secundaria. En teoría, cualquier algoritmo clásico puede usarse sin modificaciones sobre datos en memoria externa. Sin embargo, las operaciones en ese caso siempre son notoriamente más costosas que en la RAM. Usualmente y debido a ello, cuando se calculan complejidades algorítmicas se desprecian las operaciones en CPU y RAM, y simplemente se cuenta la cantidad de I/Os.

El estudio a realizar pretende determinar la validez del modelo descrito, mediante la comparación de tres implementaciones de algoritmos de búsqueda: una búsqueda binaria sin mayores diferencias respecto del algoritmo estándar; una búsqueda que recorre linealmente el arreglo almacenado en disco; y una búsqueda en base a la indexación de dicho arreglo. Para cada estrategia, se realiza una misma cantidad de experimentos variando las dimensiones de los arreglos involucrados; además, se extraen estadísticas respecto de la cantidad de operaciones I/O hechas y el tiempo tomado para la búsqueda.

2. Algoritmos a estudiar

2.1. Algoritmo de búsqueda binaria

La primera estrategia a probar es un algoritmo de búsqueda binaria en memoria secundaria. Dicho algoritmo consiste en realizar una búsqueda binaria, equivalente a la que se haría en memoria principal, de manera que cada vez que se pregunta por el valor de un elemento, se efectúa una consulta a memoria secundaria.

A continuación se presenta el pseudo-código para una implementación de búsqueda binaria en memoria principal.

Algoritmo 1 Búsqueda binaria

```
1:  procedure BINARYSEARCH
2:      P ← first input array
3:      T ← second input array
4:      O ← output array
5:      for p ∈ P do
6:          l ← 0
7:          h ← length of T
8:          while l < h do
9:              m ← 1 + (h - 1)/2
10:             if p ≤ T[m] then
11:                 h ← m
12:             else
13:                 l ← m + 1
14:         if p == P[l] then
15:             O.append(p)
        return O
```

Para el peor caso, el algoritmo de búsqueda binaria tiene complejidad $O(|P| \log |T|)$. Esto se fundamenta en el hecho de que la complejidad de aplicar búsqueda binaria sobre T es $O(\log |T|)$, y dado que se aplica por cada $p \in P$, se agrega el término $O(|P|)$. Entonces, la complejidad total es $O(|P|) * O(\log |T|)$ o, de manera equivalente, $O(|P| \log |T|)$. Luego, en el peor caso, si para cada búsqueda se escribe un elemento, entonces el costo sigue siendo el mismo.

2.2. Algoritmo de búsqueda lineal

La segunda estrategia a probar es un algoritmo de búsqueda lineal para memoria secundaria. Este algoritmo extrae secuencialmente bloques de T desde el disco, y para cada elemento de dicho bloque verifica si está en P, en cuyo caso lo agrega al resultado.

Algoritmo 2 Búsqueda lineal

```
1: procedure LinearSearch
2:     P ← first input array
3:     T ← second input array
4:     O ← output array
5:     Tchunks ← arrayChunks(T, B)
6:     for chunk ∈ Tchunks do
7:         for element ∈ chunk do
8:             if element ∈ P then
9:                 O.append(element)
       return O
```

La complejidad algorítmica de peor caso en términos de la cantidad de operaciones de I/0 correspondiente a la implementación mostrada es $O(|T|/B)$, lo que se fundamenta en el hecho de que el input de tamaño $|T|$ se divide en bloques de tamaño B, es decir, se tienen $|T|/B$ bloques, lo que equivale a realizar $|T|/B$ accesos a disco. Dado que se busca el valor de la intersección entre los dos archivos, será obligatorio acceder a todos los bloques, para así encontrar todos los posibles elementos que se compartan.

2.3. Algoritmo de búsqueda indexada

La tercera estrategia corresponde a un algoritmo de búsqueda indexada. Para implementarla, se divide T en bloques, y para cada uno de ellos se extrae el menor elemento y se guarda en memoria principal en un arreglo (un índice). Por cada elemento de P se realiza una búsqueda binaria en S, y luego se extrae el bloque donde podría estar el elemento y se busca en él.

Algoritmo 3 Búsqueda indexada

```
1: procedure IndexedSearch
2:     P ← first input array
3:     T ← second input array
4:     PinterT ← []
5:     list S[|T|/B]
6:     for i ∈ {0, 1..., |T|/B − 1} do
7:         S[i] ← T[i · B]
8:     for element ∈ P do
9:         where[B] ← binarySearchInterval(element, S)
10:        if element ∈ where then
11:            PinterT.append(element)
       return PinterT
```

La cantidad de consultas de I/O, se deduce a continuación:

- Para construir S, lo cual se hace una vez, se necesita acceder $|T|/B$ veces a la memoria secundaria. ($O(|T|/B)$)
- Buscar el representante del intervalo correspondiente de T en S no requiere llamadas a la memoria secundaria. ($O(0)$)
- Tras determinar el intervalo, para poder leerlo se debe cargar en la memoria principal una vez por cada búsqueda (es decir, una vez por cada elemento de P, lo cual es $O(1 * |P|)$). La búsqueda del elemento dentro del intervalo se hace dentro de la memoria principal.

- El total de llamadas a memoria secundaria no depende de cada caso, luego el peor caso también tendrá esta complejidad.
- El orden resultante es $O(|P| + |T|/B)$.

El número de operaciones en memoria, en el peor caso, corresponde al total de operaciones para la búsqueda binaria ($O(|P| \log_2 |T|/B)$) más el total de operaciones requeridas para encontrar el elemento (por simplicidad, se usará la búsqueda lineal) $O(|P|B)$.

3. Hipótesis y diseño experimental

A continuación, se describe la hipótesis formulada para la duración de los algoritmos en términos de la cantidad de I/O que realizan. Además, se procede a definir variables para realizar los experimentos.

Se consideran como variables dependientes la cantidad de operaciones I/Os y el tiempo de ejecución, en función del rango de los parámetros |P| y |T|. Se utilizará $|P| = 10^4$ fijo, ya que así será posible contener el archivo P completo en memoria principal, y deja suficiente espacio en ella para probar los casos borde de los algoritmos.

Para obtener el valor esperado de operaciones I/O, se usará la complejidad de cada algoritmo en cuanto a accesos a memoria secundaria, calculada en la sección 2. Así, se obtienen las siguientes ecuaciones:

$$|P| \cdot log|T| = |P| + \frac{|T|}{B} \tag{1}$$

$$|P| \cdot log|T| = \frac{|T|}{B} \tag{2}$$

Las cuáles corresponden a la intersección entre la complejidad de la búsqueda binaria y la búsqueda indexada, y la intersección entre la complejidad de la búsqueda binaria y la búsqueda lineal, respectivamente (se puede notar que las funciones de complejidad de las búsquedas lineal e indexada no intersectan a menos que $|P| = 0$, con lo cual no hay búsqueda, luego no forma parte de los cálculos).

Considerando $\frac{|T|}{|P|} = \rho$, y sabiendo que $B = 500$, es posible resolver dichas ecuaciones, con lo que se llega a que $\rho = 1{,}12 \cdot 10^3$ para la primera, y $\rho = 1{,}18 \cdot 10^3$ para la segunda.

La relación entre las tres implementaciones diferentes se evidencia en el gráfico de la Figura 5, que se encuentra en el Anexo A. A partir de éste, se infiere que:

- Cuando $\rho \in [-\infty, 1{,}12 \cdot 10^3]$, la búsqueda binaria hará la mayor cantidad de accesos a memoria, seguida de la búsqueda indexada, con la búsqueda lineal al final.
- Para $\rho \in (1{,}12 \cdot 10^3, 1{,}18 \cdot 10^3]$, la situación cambia de tal forma que la búsqueda indexada pasa a la delantera, luego la búsqueda binaria y finalmente la búsqueda lineal.
- Finalmente, para $\rho \in (1{,}18 \cdot 10^3, \infty]$, la búsqueda indexada se mantiene como la que realiza una mayor cantidad de accesos a disco, pero en este caso la sigue la búsqueda lineal, mientras que la búsqueda binaria se encuentra en el último lugar.

Dichas hipótesis se resumen en el cuadro 1.

En cuánto al tiempo de ejecución, es conocido que las operaciones de acceso a memoria secundaria toman una cantidad de tiempo notoriamente mayor a lo que demora cualquier operación en memoria principal. Por ello, se estima que las segundas son despreciables en comparación a las primeras.

Dado lo anterior, se considera pertinente realizar la suposición de que el tiempo de ejecución de las implementaciones de los algoritmos será directamente proporcional a cuántas operaciones de I/Os realicen. Así, las implementaciones que efectúen más accesos a disco demorarían más.

	# de operaciones de I/O (de mayor a menor)	Tiempo de ejecución (de mayor a menor)
$\rho \in [-\infty, 1{,}12 \cdot 10^3]$	1. Búsqueda binaria 2. Búsqueda indexada 3. Búsqueda lineal	1. Búsqueda binaria 2. Búsqueda indexada 3. Búsqueda lineal
$\rho \in (1.12 \cdot 10^3, 1{,}18 \cdot 10^3]$	1. Búsqueda indexada 2. Búsqueda binaria 3. Búsqueda lineal	1. Búsqueda indexada 2. Búsqueda binaria 3. Búsqueda lineal
$\rho \in (1.18 \cdot 10^3, \infty]$	1. Búsqueda indexada 2. Búsqueda lineal 3. Búsqueda binaria	1. Búsqueda indexada 2. Búsqueda lineal 3. Búsqueda binaria

Cuadro 1: Hipótesis sobre la relación entre distintas implementaciones de algoritmos de búsqueda

4. Ejecución de los algoritmos

4.1. Generación aleatoria de instancias

Se produjo una serie de instancias aleatorias para poder efectuar experimentos que permitieran verificar o refutar las hipótesis planteadas.

Se escribió una función *generator* que recibe como parámetros el tamaño deseado de P y T, y genera dos archivos que contienen elementos aleatorios. Para generar P, la función elige enteros al azar entre 1 y 10^9, y luego escribe cada uno directamente en el archivo. Para generar T, la función se comporta de manera similar, excepto que una vez que tiene el archivo completo ordena sus elementos.

4.2. Obtención de la cantidad k de instancias que genera menor desviación estándar sobre el costo en I/Os

Para obtener la cantidad k de instancias que parezca generar la menor desviación estándar, se utilizó la implementación del algoritmo de búsqueda binaria, para luego ejecutar k repeticiones del experimento, con $k \in [2, 25]$. Para cada instancia $i \in [0...k]$ se generaron nuevos P y T, de los tamaños antes descritos.

Para cada k del experimento se guardó la cantidad de operaciones I/Os efectuada, además del promedio y la desviación estándar correspondientes. También, se guardó el tiempo que demoraron las k ejecuciones, el promedio de éstas y su desviación.

Luego, se generó la visualización para la cantidad de operaciones I/O en promedio y la desviación estándar correspondiente en función del valor k. Dicha visualización se encuentra en la sección C del Anexo.

Se observa que el k que genera menos desviación estándar es $k = 2$, sin embargo, se estima que dicho valor no resultaría en un experimento lo suficientemente representativo, por lo que se optó por escoger el que genera la segunda menor desviación estándar, $k = 14$.

4.3. Parámetros relevantes para los experimentos

Se definió que P sería un archivo de tamaño fijo 10^4, dado que se empleó el mismo valor para formular la hipótesis. Se consideró dicho tamaño corresponde a cantidad de elementos.

Cada algoritmo fue probado con archivos T de 6 tamaños diferentes:

- $|T_1| = 10^6$
- $|T_2| = 10^7$
- $|T_3| = 1{,}12 \cdot 10^7$
- $|T_4| = 1{,}15 \cdot 10^7$
- $|T_5| = 1{,}18 \cdot 10^7$
- $|T_6| = 1{,}5 \cdot 10^7$

La razón detrás de dicha decisión es que $|T_1|$, $|T_2|$ y $|T_3|$ simulan el primer caso presentado en la hipótesis, $|T_4|$ simula el segundo, y $|T_5|$ y $|T_6|$ simulan el tercero.

Utilizando Python 3.8.5, se creó un programa que ejecuta cada uno de los algoritmos estudiados.

Los bloques de extracción de los algoritmos de búsqueda indexada y lineal poseen tamaño $B = 50$ líneas, debido a que como máximo se pueden leer 500 bytes en un bloque, y cada línea pesa 10 bytes.

Para cada algoritmo y para cada tamaño de P se ejecutó cada algoritmo 14 veces, tras lo cual se extrajeron promedios de operaciones I/O y de duraciones total de la ejecución.

Los detalles técnicos del computador en el que serán llevados a cabo los experimentos se pueden ver en la sección B del Anexo.

5. Resultados

A continuación se exponen los resultados de los experimentos descritos para los tres algoritmos de búsqueda: búsqueda binaria, indexada y lineal.

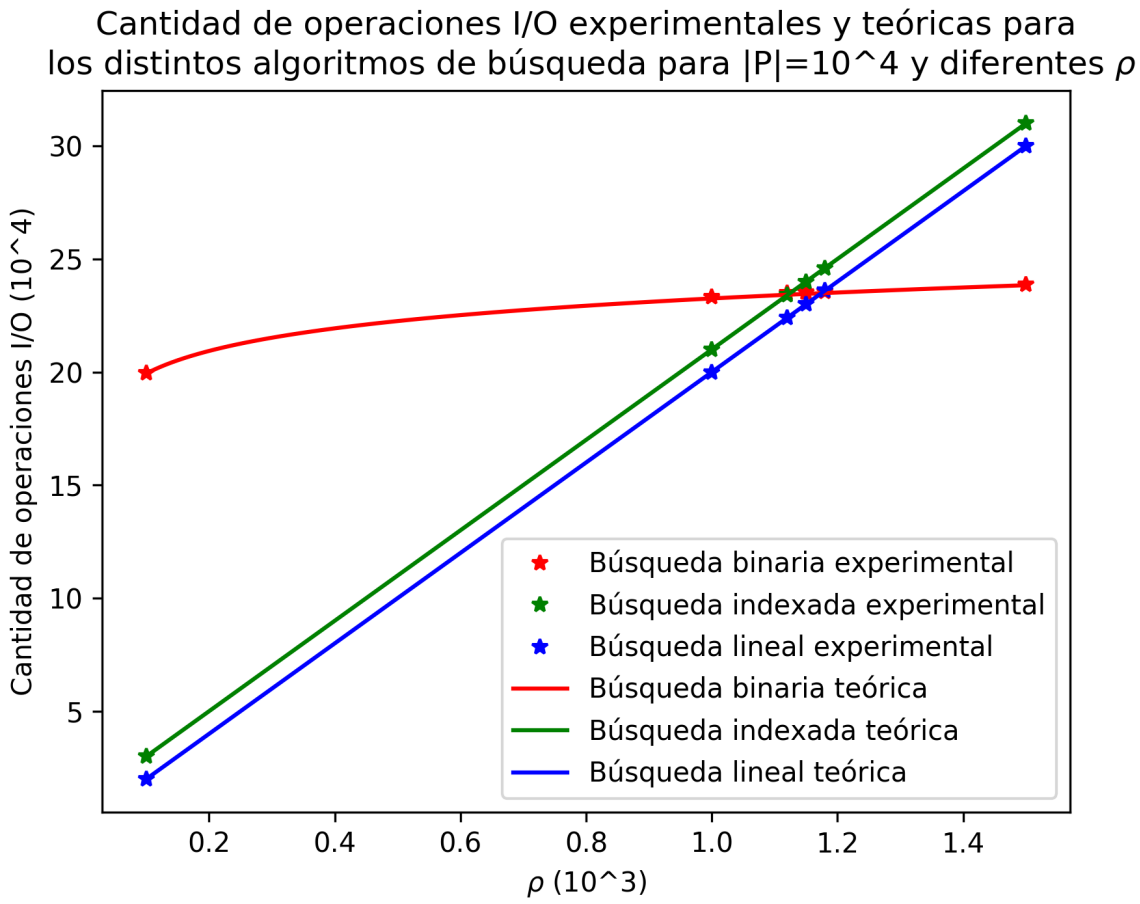

Figura 1: Gráfico que exhibe una comparación de la duración experimental promedio de la ejecución de los algoritmos y la esperada de forma teórica en función del valor ρ = |T|/|P|.

La figura 1 muestra alta coherencia entre lo esperado y lo obtenido en cuanto a cantidad de operaciones I/O. En particular, respecto de los datos utilizados, es coherente con lo mencionado en el cuadro 1.

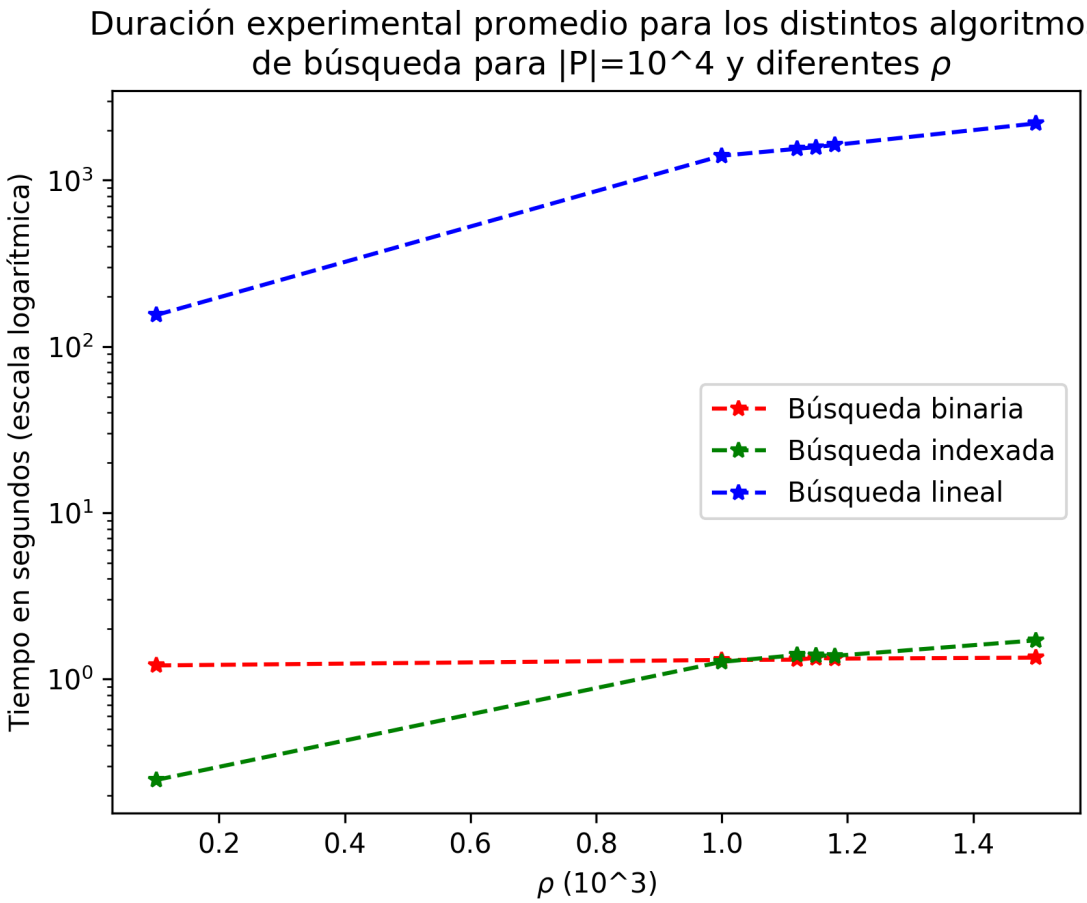

Figura 2: Gráfico que exhibe una comparación de la duración experimental promedio de la ejecución de los algoritmos de búsqueda binaria, lineal e indexada, en función del valor ρ = |T|/|P|. El eje vertical está en escala logarítmica dadas las diferencias entre las duraciones. No hay comparación teórica dado que no se dispone de un dato sobre el tiempo que tarda una consulta a disco.

La figura 2, la cual muestra la duración de cada algoritmo según el tamaño de T, es cualitativamente muy distinta a la figura 1. Principalmente, se ve que la búsqueda lineal demora, en cada caso, órdenes de magnitud más que las otras dos búsquedas. Además, si bien las curvas de tiempo de búsqueda binaria y la indexada se intersectan, lo hacen en un valor de ρ menor que en la figura 2, denostando que tiempo de ejecución y cantidad de I/O no son proporcionales. Adicionalmente, se puede apreciar que la curva de búsqueda lineal es similar a la de búsqueda indexada.

6. Discusión

Como se ve en los resultados, la cantidad de operaciones I/O coincide con la esperada, pero ésta no es directamente proporcional a la duración de cada algoritmo: la búsqueda binaria es la que menos tarda con valores de ρ más grandes, a pesar que la búsqueda lineal realiza menos operaciones I/O que ella. Es más, como se pudo observar en la figura 2, la búsqueda lineal demora considerablemente más que las otras dos. Sin embargo, se identifica que existe una relación entre duración y operaciones I/O, la cual persiste a lo largo de los experimentos (los resultados no son aleatorios).

Dado lo anterior, se plantea que las diferencias de duración entre las búsquedas no solo dependen del total de operaciones I/Os realizadas, si no que también de la cantidad de operaciones que se hacen en memoria principal. Por esto, es conveniente analizar el costo que tiene cada operación en memoria principal.

6.1. Costos en memoria principal

- **Búsqueda binaria:** La búsqueda binaria, cada vez que extrae un elemento del disco, solo realiza una consulta en CPU, de manera que hace tantas consultas en memoria principal como en secundaria, esto es, $(O(|P| \log_2 |T|))$.
- **Búsqueda lineal:** La búsqueda lineal, en CPU, por cada elemento de T debe recorrer P linealmente (P no está ordenado, luego el elemento buscado puede estar en cualquier parte de este, o bien no estar, para lo cual deben realizarse a lo más $|P|$ consultas), lo cual, en el peor caso, significa realizar $|T||P|$ operaciones.
- **Búsqueda indexada:** La búsqueda indexada, por cada elemento de P realiza una búsqueda binaria en CPU, después de lo cual, si el elemento de S no coincide con el buscado, se extrae un bloque de T y se realiza una búsqueda lineal sobre éste, lo que significa que este algoritmo realiza $O(|P| \log_2 \left(\frac{|T|}{B}\right) + |P|B)$ operaciones en CPU.

Se concluye entonces, que la búsqueda lineal es quien es más costosa en memoria principal, justificando así que también sea quien tome más tiempo en ejecutarse.

6.2. Duraciones considerando operaciones en memoria principal

Considerando lo encontrado en las secciones anteriores, es posible plantear ecuaciones de tiempo de ejecución de los algoritmos (en el peor caso) en base a operaciones en memoria principal y búsquedas en memoria secundaria de la siguiente manera:

$$t_{binary} = t_{I/O} \cdot |P| \log |T| + t_{mem} \cdot |P| \log |T| = (t_{I/O} + t_{mem}) \cdot |P| \log |T|$$

$$t_{linear} = t_{I/O} \cdot \frac{|T|}{B} + t_{mem} \cdot |P||T|$$

$$t_{indexed} = t_{I/O} \cdot \left(|P| + \frac{|T|}{B}\right) + t_{mem} \cdot \left(\log_2 \frac{|T|}{B} + B\right) \cdot |P|$$

Donde $t_{I/O}$ es el tiempo que tarda una consulta I/O y t_{mem} es la duración de una operación en memoria principal. Puede notarse que el algoritmo de búsqueda lineal realiza una cantidad considerablemente mayor de operaciones en memoria principal respecto de los otros dos ($|T| \gg B > \log_2 |T|$). Además, si t_{mem} y $t_{I/O}$ difieren en pocos órdenes de magnitud, el número de operaciones en memoria principal prima por sobre las búsquedas en disco en el caso de búsqueda lineal.

6.3. Variantes de algoritmo de búsqueda lineal

Para comprobar que efectivamente las operaciones en memoria principal contribuyen al tiempo de ejecución, especialmente en el caso de la búsqueda lineal, se realizarán dos variantes de la forma de operar de este algoritmo. Una de las variantes realizará una búsqueda binaria sobre cada bloque de T extraído; la otra variante realizará un ordenamiento de P, después de lo cual hará una búsqueda de tipo merge entre éste y cada bloque de T.

6.3.1. Búsqueda lineal con búsqueda binaria

Esta variante, una vez que toma un bloque de T, realiza una búsqueda binaria sobre dicho bloque, de tamaño B, por cada elemento de P. Dado que debe extraer $|T|/B$ bloques, el algoritmo realiza en CPU, en el peor caso, $\frac{|T|}{B}|P|\log_2 B$.

Para ver el pseudo-código de este algoritmo, por favor dirigirse al anexo D.

6.3.2. Búsqueda lineal con merge

La búsqueda lineal usando merge primero ordena P en memoria, lo cual se hace una vez y toma a lo sumo $|P|\log_2|P|$ operaciones (utilizando quicksort). En la etapa de merge, cada vez que se extrae un bloque de T, se realiza a lo sumo B operaciones para mover el cursor del bloque y $|P|$ operaciones para mover el cursor de P. Como en total se extraen $|T|/B$ bloques, el total de operaciones es el producto entre este término y la suma de B y $|P|$, es decir, $(B+|P|)\cdot|T|/B = |T|+|P||T|/B$ operaciones. Sumando esto al ordenamiento, se llega a que el total de operaciones en CPU de este algoritmo es $O(|P|\log_2|P| + |T| + |P||T|/B)$.

Para ver el pseudo-código de este algoritmo, por favor dirigirse al anexo E.

6.4. Resultados de experimentos sobre las variantes del algoritmo de búsqueda lineal

A continuación se exhiben los resultados correspondientes a los experimentos que se llevaron a cabo sobre las variantes del algoritmo de búsqueda lineal, es decir, sobre la búsqueda lineal con búsqueda binaria y la búsqueda lineal con merge.

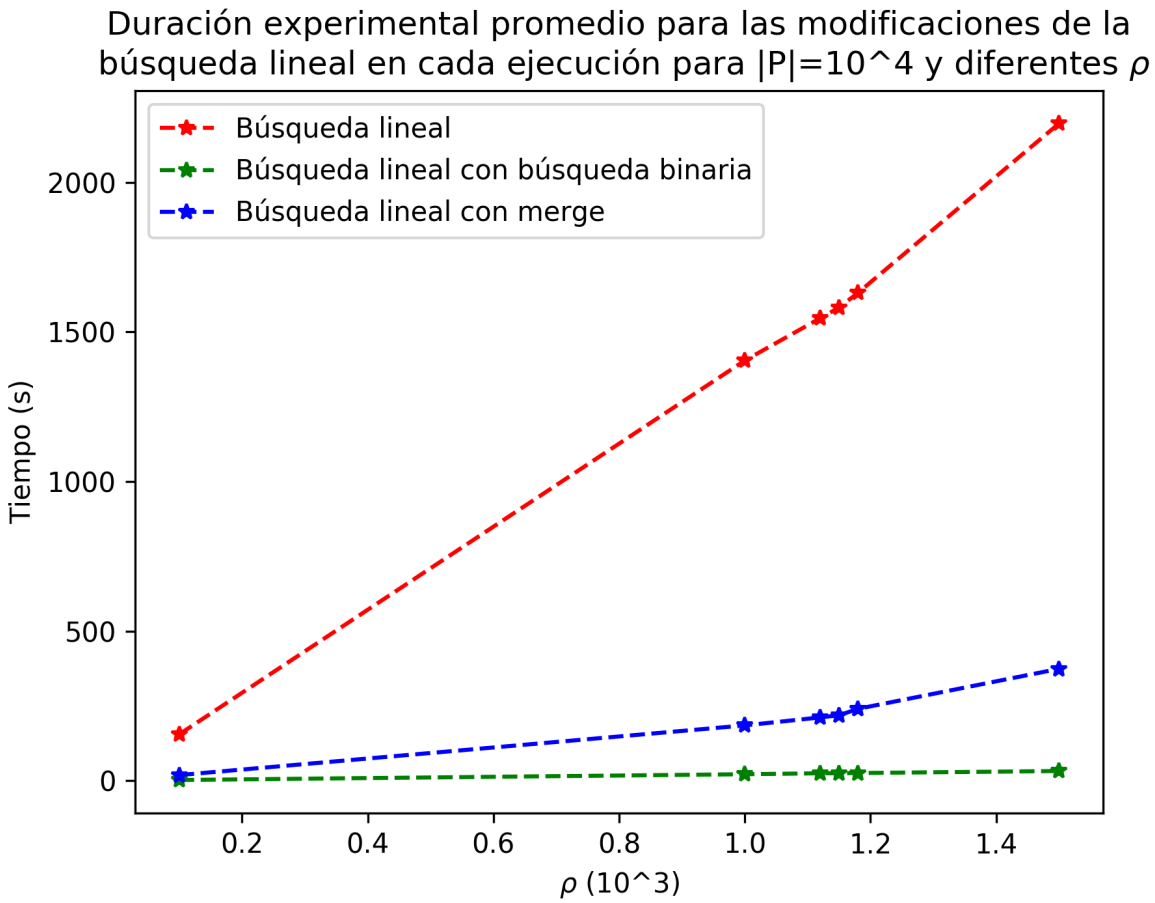

Figura 3: Gráfico que exhibe una comparación de la duración promedio de cada ejecución de las variantes de la búsqueda lineal en función del valor de $\rho = |T|/|P|$.

Con respecto a la Figura 3, en el eje X se aprecia el valor de ρ dado un P fijo $P = 10^4$ y a una escala de 10^3. En el eje Y, se puede ver el tiempo medido en segundos.

Puede verse que la búsqueda lineal original es la que toma más tiempo, creciendo esta (casi) linealmente a medida que ρ aumenta. Por otra parte, es pertinente notar que la siguen la búsqueda lineal con merge, y luego la búsqueda lineal con búsqueda binaria, siendo esta última la que menos demora. Además, se percibe que las variantes mencionadas crecen más lentamente que el algoritmo de búsqueda lineal.

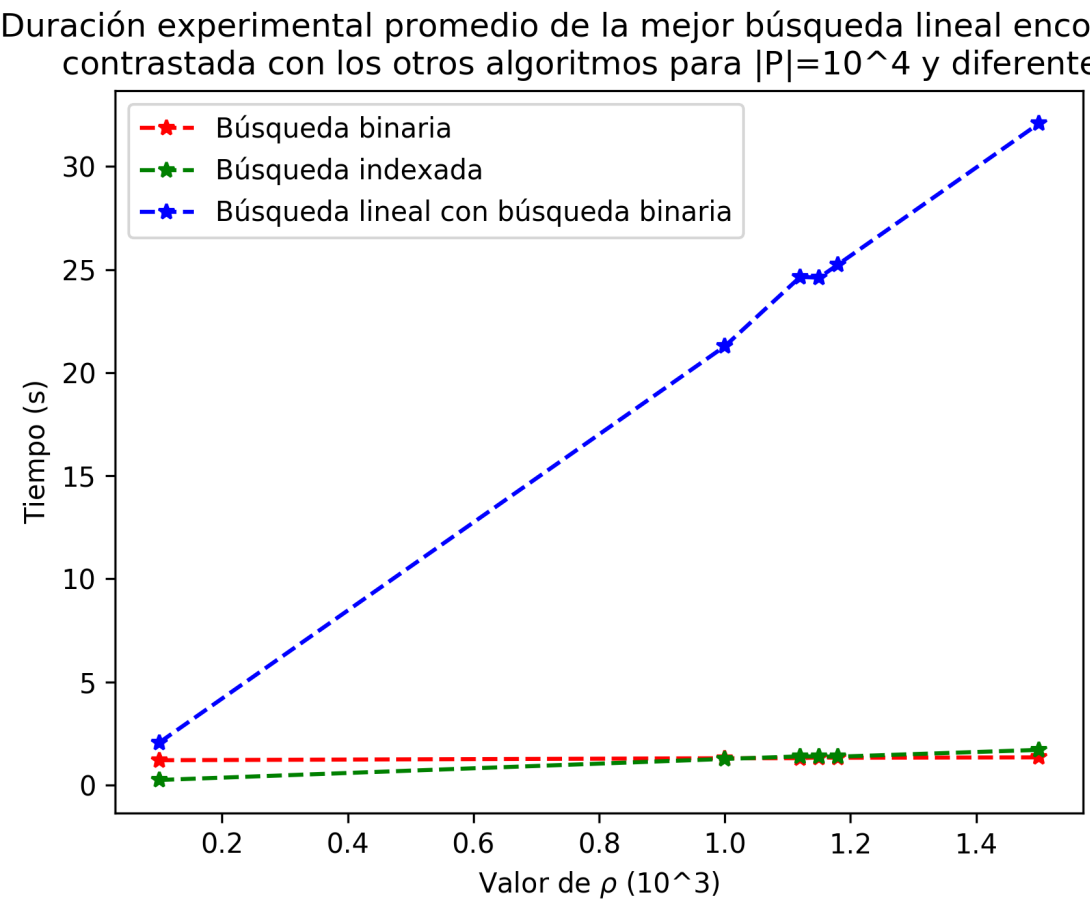

Figura 4: Gráfico que exhibe una comparación de los tiempos de ejecución entre el algoritmo de búsqueda lineal con búsqueda binaria, el algoritmo de búsqueda binaria solo y el de búsqueda indexada en función del valor $\rho = |T|/|P|$.

Luego, se tiene la Figura 4, cuyo eje X presenta el valor de ρ dado un P fijo $P = 10^4$ y a una escala de 10^3, y cuyo eje Y muestra el tiempo de ejecución medido en segundos. El gráfico compara la mejor optimización de la búsqueda lineal analizada con las otras dos estrategias.

A primera vista, se observa que la búsqueda lineal con búsqueda binaria es la que toma más tiempo, y crece linealmente a medida que ρ aumenta. Por otra parte, es pertinente notar que la siguen la búsqueda binaria y posteriormente, la búsqueda indexada, las cuales se intersectan aproxidamadamente en $\rho = 1 \cdot 10^3$.

6.5. Discusión sobre las variantes del algoritmo de búsqueda lineal

La diferencia entre duraciones de las distintas variantes muestra claramente la influencia del tiempo de operaciones en memoria principal en la duración total de la ejecución del algoritmo. La necesidad de considerar las operaciones en memoria principal resulta, entonces, imperativa.

Ha de notarse que pese a que las variaciones hechas a este algoritmo disminuyen notablemente la duración de la búsqueda, no superan en rapidez a la búsqueda indexada ni tampco a la búsqueda binaria, sugiriendo que una minimización de operaciones I/O no necesariamente conlleva a una optimización del tiempo de ejecución.

Además, pareciera que, en cualquier caso, la búsqueda indexada es la estrategia más versátil, ya que, en

caso de tener muchas listas P para las cuales se desea encontrar su intersección con T, la complejidad del algoritmo crece linealmente con la cantidad de listas P, pero basta con indexar T una sola vez. Esto significa que la búsqueda indexada tiene asociada un costo fijo para una operación, mientras las otras estrategias solo parecieran tener costo variable. Un experimento futuro podría comprobar que efectivamente esto es así.

7. Conclusiones

El estudio muestra que no es suficiente diseñar un algoritmo poco costoso en operaciones de memoria secundaria para optimizar su tiempo de ejecución. Cualquier algoritmo, ya sea que requiera o no búsquedas en disco, debe optimizar todas sus operaciones para maximizar su rendimiento.

Como muestra la relación entre los algoritmos estudiados, no basta con realizar una optimización de ambos aspectos por separado para obtener una cota inferior de duración. Esto se hace evidente por el hecho de que la implementación de búsqueda indexada no es la búsqueda que efectúa menos operaciones I/O entre las estudiadas, sin embargo, es una de las que menor tiempo de ejecución ha mostrado entre los casos estudiados y es la que pareciera tener la mayor versatilidad. Como consecuencia, un algoritmo óptimo que involucra acceder a memoria secundaria debe ser diseñado de manera integral.

Conforme avanza la tecnología, es esperable que los tiempos de consultas a disco sigan disminuyendo. Hemos mostrado aquí que ello significa un aumento en la complejidad de los problemas a resolver, pues significa considerar al mismo tiempo, e irreversiblemente, dos problemas previamente separables.

8. Anexo

A. Gráficos adicionales

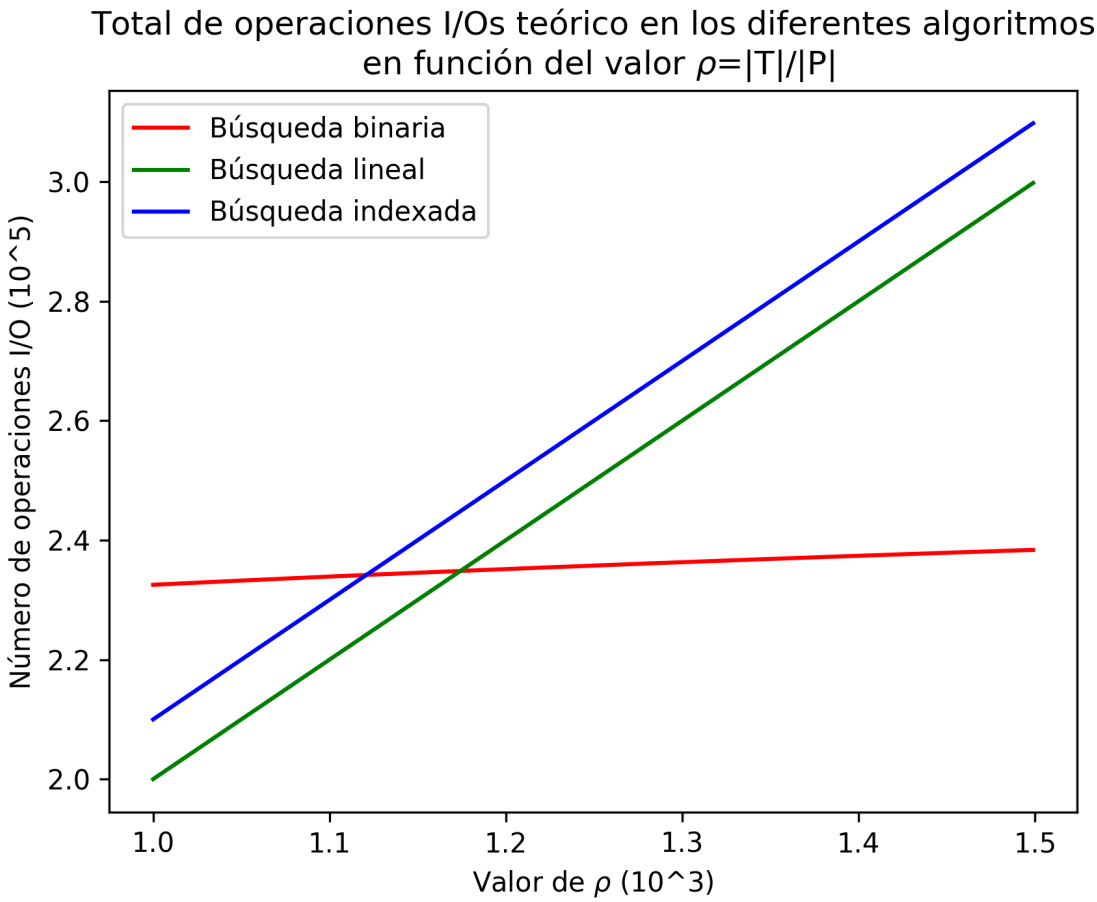

Figura 5: Gráfico que exhibe una comparación de la cantidad total de operaciones I/Os para los algoritmos de búsqueda binaria, lineal e indexada, en función del valor ρ

B. Detalles técnicos del equipo usado para los experimentos

El computador utilizado para llevar a cabo los experimentos corresponde a un Lenovo 330S 14-IKB con las siguientes especificaciones técnicas:

- Sistema Operativo: Manjaro Gnome x64 20.1
- CPU: Intel Core i7 8550U @1.7 - 4 GHz.
- RAM: 12GB SODIMM-DDR4 2400MHz
- Almacenamiento: Crucial P1 500GB - 1900 MB/s Lectura, 950 MB/s Escritura.

C. Visualización para la cantidad de operaciones I/O en promedio y la desviación estándar correspondiente en función del valor k

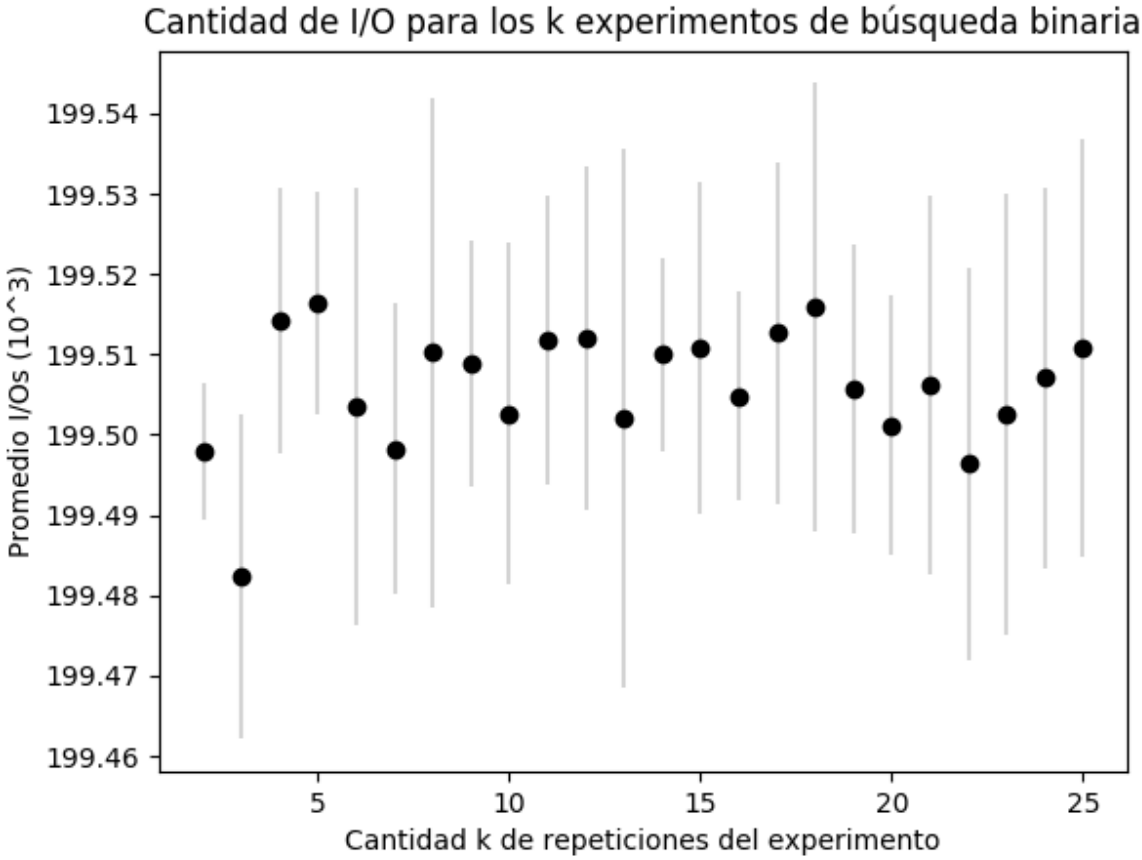

Figura 6: Gráfico que exhibe una comparación de la cantidad de operaciones I/O en promedio y la desviación estándar correspondiente en función del valor k

D. Pseudo-código búsqueda lineal con búsqueda binaria

Se describe el pseudo-código para la estrategia de búsqueda lineal con búsqueda binaria.

Algoritmo 4 Búsqueda lineal + búsqueda binaria

```
1:  procedure LINEARSEARCHPLUSBINARY
2:      toWrite ← []
3:      P.sort()
4:      output ← []
5:      for iterationIndex ∈ {0, blockSize, 2 · blockSize..., lenT − 1} do
6:          if iterationIndex + blockSize ≥ lenT then
7:              numberOfLines ← lenT − iterationIndex
8:          else
9:              numberOfLines ← blockSize
10:         lines ← readLines(start, numberOfLines, fileT)
11:         for element ∈ lines do
12:             if binarySearch(fileP, element, lenP − 1)! = −1 then
13:                 toWrite.append(foundElement)
14:                 if len(toWrite) · lineSize == blockSize then
15:                     output.write(toWrite)
16:                     toWrite ← []
17:     if len(toWrite)! = 0 then
18:         output.write(toWrite)
        return O
```

E. Pseudo-código búsqueda lineal con merge

Se describe el pseudo-código para la estrategia de búsqueda lineal con merge.

Algoritmo 5 Búsqueda lineal + merge

```
1: procedure LINEARSEARCHPLUSMERGE
2:     toWrite ← []
3:     P.sort()
4:     output ← []
5:     for iterationIndex ∈ {0, blockSize, 2 · blockSize..., lenT − 1} do
6:         if iterationIndex + blockSize ≥ lenT then
7:             numberOfLines ← lenT − iterationIndex
8:         else
9:             numberOfLines ← blockSize
10:        lines ← readLines(start, numberOfLines, fileT)
11:        indexP ← 0
12:        indexT ← 0
13:        while indexP ≤ lenP&&indexT ≤ len(lines) do
14:            if elementP ≥ elementT then
15:                indexT ← indexT + 1
16:            else if elementP ≤ elementT then
17:                indexP ← indexP + 1
18:            else
19:                toWrite.append(foundElement)
20:                indexP ← indexP + 1
21:                indexT ← indexT + 1
22:                if len(toWrite) · lineSize == blockSize then
23:                    output.write(toWrite)
24:                    toWrite ← []
25:    if len(toWrite)! = 0 then
26:        output.write(toWrite)
       return O
```