Tarea 1: Ordenamiento en Memoria Secundaria

Jorge Bahamonde, Ricardo Jacas y Víctor Rojas

December 21, 2011

1 Descripción del Problema

Se analizarán dos implementaciones de algoritmos de ordenamiento en memoria secundaria, determinándose sus parámetros óptimos para luego evaluar su rendimiento. Los algoritmos implementados se presentan a continuación:

1.1 Mergesort Multiario (adaptado a un modelo refinado para el almacenamiento secundario)

Este algoritmo se basa en un mergesort de k elementos, con el objetivo de realizar el mayor número posible de merges para reducir la cantidad de niveles necesarios. En un merge k-ario, se hace merge a k secuencias dentro de una sola secuencia de salida. Para el ordenamiento por paso, se utiliza una cola de prioridad, que entrega en cada paso el elemento a ser agregado a la secuencia de salida. Se necesita suficiente memoria interna para almacenar k secuencias de entrada y una de salida; cuando una secuencia de entrada se vacia, se rellena y se repite el proceso. Si la cola de salida se llena, se escribe en memoria externa. Para aprovechar el hecho de que las lecturas secuenciales se realizan con mayor velocidad que las aleatorias, se tienen secuencias de entrada de tamaño $l \times B$ elementos, con B el número de elementos que pueden ser leídos de una vez en el dispositivo de almacenamiento secundario.

1.2 Samplesort

Samplesort es un algoritmo de ordenamiento parecido a Quicksort. Sin embargo, en vez de usar un solo pivote., este algoritmo usa k - 1 elementos para separar la entrada en buckets. Los separadores son escogidos de forma que aseguren un tamaño máximo de n/k para cada bucket. Para la selección de los separadores se utiliza un entero a, escogiéndose aleatoriamente $a(1+k)-1$ muestras de la entrada. Asi, se definen los separadores como $s_i = S[(a+1)i]$ donde S es el conjunto de muestras, ordenado. Los buckets son ordenados recursivamente, hasta que uno quepa por completo en memoria, en cuyo caso se ordenan internamente. Existen análisis que recomiendan un $k \in \Theta(min(n/M, M/B))$ buckets y una muestra de tamaño $a = O(log\ k)$, para obtener buckets razonablemente llenos con buena probabilidad.

2 Hipótesis

Se hipotetiza que la implementación de mergesort escalará de una forma estable, ya que la optimalidad teórica de sus parámetros no se basa en el número de elementos a ordenar, sino que en parámetros de la unidad de almacenamiento secundario y del tamaño de la memoria principal.

Se espera ver una mayor variabilidad para samplesort, ya que sus parámetros son más bien heurísticos. Sin embargo, también debería escalar de buena forma, ya que el parámetro que se variará es en realidad una constante de proporcionalidad; el verdadero parámetro interno (número de pivotes) se calcula dinámicamente por el algoritmo para cada número de elementos a ordenar.

Se espera también observar un overhead en samplesort, debido a la creación y borrado continuo de archivos por parte del algoritmo: esta interacción con el sistema de archivos debería traer un costo asociado.

3 Diseño Experimental

El lenguaje utilizado fue C. Para desarrollar estos algoritmos, se dividió el proyecto en varias partes. Están los archivos que implementan los algoritmos, los archivos con funciones útiles, archivos para hacer pruebas, y los scripts para correr los algoritmos.

3.1 Generación de Instancias y Metodología

Para probar los algoritmos, se crearon archivos con datos aleatorios. En este punto fue necesario poner atención a la calidad de la aleatoriedad requerida y el tiempo requerido para obtener las instancias. Un primer acercamiento se realizó a través de los archivos especiales de Unix, */dev/random* y */dev/urandom*. Sin embargo, se determinó que el primero era insuficiente por volverse bloqueante ante falta de entropía (por lo que la generación de instancias suficientemente grandes se volvería impráctica); el segundo se descartó por disminuir la calidad de la aleatoriedad ante una falta de entropía. Finalmente, se escogió utilizar el generador de números aleatorios random() implementado en la biblioteca estándar de C, por ser una alternativa más rápida al ser un generador lineal congruente; además, el alto periodo de este generador es aceptable para los tamaños de archivo requeridos.

Para generar instancias de forma más rápida, se generó primero un archivo aleatorio de 100 GB; de este primer archivo se generaron las instancias más pequeñas, copiando datos del archivo mayor a partir de offsets aleatorizados.

En lo que sigue de este informe, se considerará a N como el número de elementos a ordenar; M, el número de estos elementos que caben en memoria principal; y B, el número de elementos que caben en una unidad de transferencia entre memoria principal y secundaria. Para todos los experimentos se consideró un valor de M igual a 26214400 (es decir, 100MB de enteros de 32 bits).

3.2 Determinación de Parámetros

Se comenzó por determinar los parámetros adecuados para cada algoritmo. En el caso de mergesort, se determinó la mejor aridad posible; para samplesort, se determinó el mejor número de pre-pivotes a escoger. Para esto, se realizaron ejecuciones con una cantidad fija de elementos, utilizándose $N = 256 \times M$. En ambos casos se utilizaron como referencia resultados teóricos para estos parámetros. Para cada ejecución se registró el tiempo real, tiempo de sistema, tiempo de usuario y número de accesos a disco totales y aleatorios.

3.2.1 Mergesort Multiario

Para el mergesort multiario, se tiene como resultado teórico (en un modelo refinado para el almacenamiento externo) que debieran utilizarse k colas de largo l bloques, de modo de que $k \times l \leq m$, el número de bloques que caben en memoria principal. Se utilizó $k \times l = m$, con el objetivo de aprovechar al máximo la memoria disponible y disminuir los accesos a disco. En este caso, el costo teórico del proceso se vuelve proporcional a la siguiente expresión:

$$\frac{t_{acceso}}{log(k)} + \frac{kB}{Mlog(k)}(t_{seek} + t_{latencia})$$

Para el disco utilizado, la minimización de esta cantidad con respecto a k entregó un valor teóricamente óptimo entre 10 y 11. Por esto, se realizaron pruebas para mergesort con valores para k de 10, 11 y 12. No se realizaron pruebas para un rango más amplio debido a restricciones de tiempo.

3.2.2 Samplesort

Para samplesort, se partió del resultado conocido de que un buen k (el número de pivotes con los cuales dividir los datos a ordenar) debe pertenecer a $\Theta(min(n/M, M/B))$. Por esto, se utilizó un valor para k igual a $t \times min(n/M, M/B)$, con t un parámetro por determinar. Se utilizó $a = t * logk$ para mantener esta proporcionalidad. Se experimentó con valores para t de 1, 2 y 4, por considerarse estos valores que mantendrían a k en el orden de magnitud requerido.

3.3 Experimentación con Parámetros Fijos

Una vez determinados los parámetros ideales para cada algoritmo, se procedió a realizar ejecuciones de éstos utilizando diferentes números de elementos a ordenar. Se utilizaron archivos con $N = 2^i M$ elementos, para $i \in \{2...8\}$, utilizándose siempre los parámetros escogidos al final de la etapa anterior. De la misma forma, se registraron los tiempos reales, de sistema y de usuario, así como el número de accesos a disco aleatorios y totales.

3.4 Dificultades y Problemas Encontrados

- Como ya se mencionó, el método de generación de datos aleatorios requirió una reevaluación para determinar la mejor alternativa que cumpliera con los requisitos necesarios.

- Una implementación *naïve* de los algoritmos utilizó los mecanismos usuales de manejo de archivo de la librería estandar de C sin mayor adaptación. No se notaron anomalías hasta el momento de utilizar archivos de tamaño mayor a 2 GB; en ese momento, las funciones de manejo de archivos comenzaron a presentar comportamientos inesperados, debido a que los offsets en los archivos ya no podían ser representados como enteros de 32 bits. De esta forma, se requirió investigar cómo manejar offsets de 64 bits, utilizando las funciones y definiciones correctas y consistentes tanto al momento de escribir el código como al momento de realizar la compilación de los programas.
- En la implementación de samplesort se volvió un desafío el realizar correctamente la división de los archivos, debido a que se hace necesario mantener una buena nomenclatura de estos archivos. De otro modo, se pierde el orden y la correspondencia entre los niveles de recursión y las diferentes llamadas a la función de ordenamiento. El resolver correctamente este problema permitió eliminar al "archivo padre" luego de generar la partición de éste, evitando generar un uso de disco insatisfacible por la máquina utilizada.

4 Descripción del Hardware Utilizado

Para todos los experimentos se utilizó una máquina virtual en el programa VirtualBox, con las siguientes características:

- Procesador Intel Core i5 2410M CPU @ 2.30 GHz (correspondiente a un procesador del host)
- Memoria RAM de la VM: 2061736k
- Memoria swap: 6036472k
- Sistema Operativo: Ubuntu Server versión 11.10

Datos del Host:

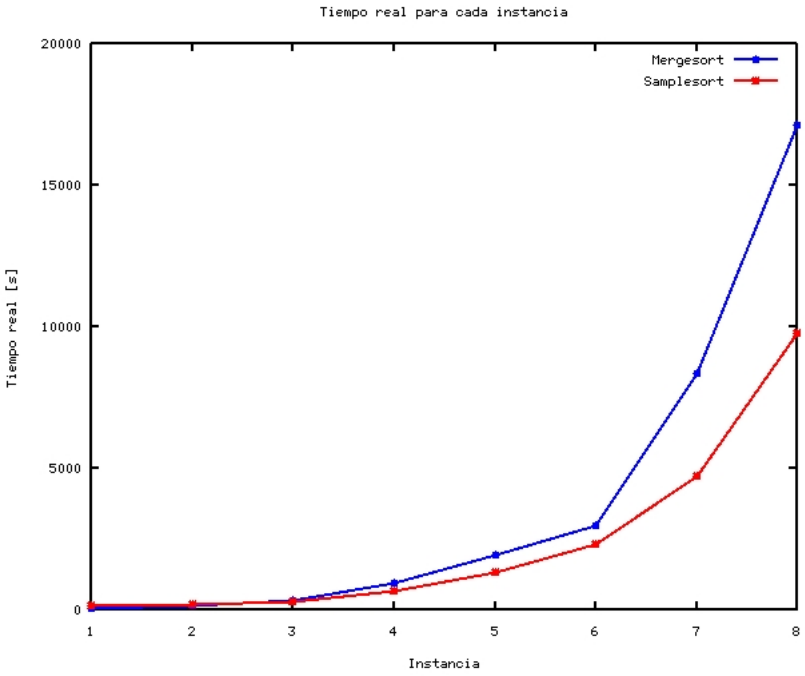
- Procesador Intel Core i5 2410M CPU @ 2.30 GHz (4 procesadores)
- Memoria RAM: 6043752k
- Memoria Swap: 12408828k
- Sistema Operativo: Debian testing (wheezy)

Disco Duro:

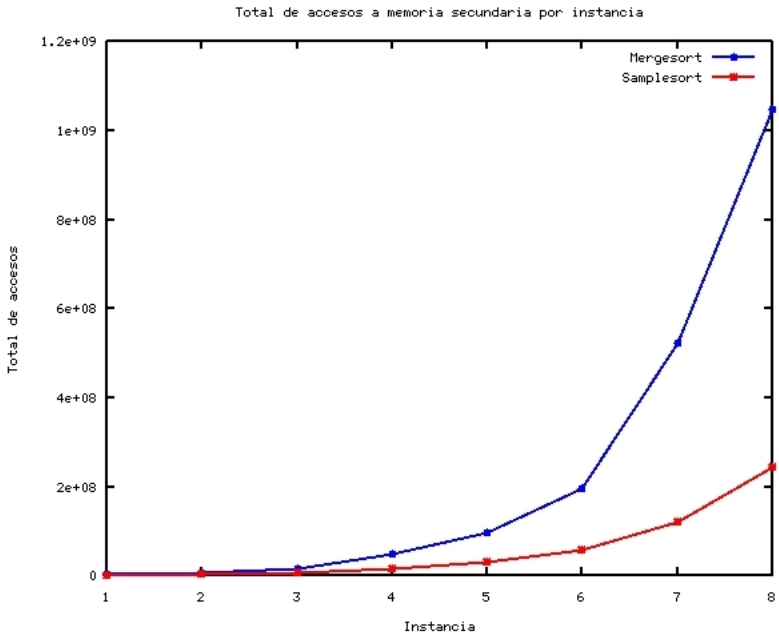
- Modelo: WD7500PVT
- Tasa de transferencia 3Gbits/s
- Tiempo de latencia: 5,5 ms
- Tiempo de seek: 12 ms
- Tamaño del sector: 512 bytes

5 Resultados

En la etapa de determinación de parámetros, se obtuvieron valores empíricamente óptimos para los parámetros de cada algoritmo. En el caso de mergesort, se determinó usar 10 secuencias de entrada; para samplesort, el parámetro de proporcionalidad t se escogió como 2. Se obtuvieron los siguientes resultados para la experimentación con parámetros fijos:



Tiempo real de ejecución en cada instancia para ambos algoritmos.



Total de accesos a memoria secundaria por cada instancia para cada algoritmo.

6 Interpretación y Conclusiones

Se aprecia, a primera vista, que los algoritmos escalan de la forma esperada. Se aprecia que si bien los algoritmos exhiben tiempos similares, samplesort realiza muchos más accesos aleatorios, mientras que mergesort realiza muchos más accesos totales (de los cuales una porción pequeña consiste en accesos aleatorios). Esto termina por balancear los algoritmos, obteniéndose rendimientos similares en tiempo. Es posible que mergesort pudiera ser aun más eficiente si se considera que, para que los accesos secuenciales de mergesort sean tales, se requiere que no existan otros procesos realizando peticiones al disco (de forma que el brazo no se mueva del lugar de lectura anterior). Las pruebas se realizaron en un sistema operativo en el que corren procesos en *background*, posiblemente interactuando

con el almacenamiento secundario (por ejemplo, procesos que realizan *logging*).

Por otro lado, samplesort termina por utilizar mucho más tiempo de sistema que mergesort, que pasa más tiempo en modo usuario. Esto proviene del hecho de que la implementación samplesort utilizada interactúa mucho con el sistema de archivos, no sólo escribiendo y leyendo en éstos, sino que creando y borrando archivos. Esto involucra muchas más llamadas a sistema.

Se aprecia que el costo para samplesort, relativo al costo teórico esperado, baja al crecer las instancias. Esto puede deberse a que el overhead de crear archivos es más notorio para instancias pequeñas. Sin embargo, se considera que la principal razón es que el valor óptimo para el parámetro t sea dependiente del tamaño de la instancia; en este caso, la curva indicaría que el valor de t utilizado es mejor para instancias mayores, y que posiblemente existe un mejor t para instancias de menor tamaño. De todos modos, se hace necesario investigar más a fondo la relación entre el tamaño de la instancia y la optimalidad de este parámetro para confirmar esta nueva hipótesis.

El hecho de que el escalamiento de mergesort sea irregular pudiera deberse a que el k utilizado no es necesariamente el óptimo. Una posible razón para esto es que para la determinación del óptimo teórico (en torno al cual se buscó un óptimo empírico) se realizó en base a parámetros nominales de la unidad de almacenamiento secundario, y que no se exploró con la suficiente profundidad el espacio de parámetros.

En conclusión, se puede decir que parámetros que pudieran ser optimales para un tamaño de instancia no necesariamente lo serán para otro tamaño, especialmente en el caso de parámetros netamente heurísticos: aquellos parámetros con una mejor base teórica para su optimalidad pueden verse como más "estables" ante cambios de parámetros que no afectaran al óptimo teórico. En este sentido, se confirma la hipótesis sobre la variabilidad en el comportamiento de samplesort al cambiarse el tamaño de instancia, notándose sin embargo que mergesort presenta una variabilidad que debiera ser estudiada más a fondo.

7 Anexos

7.1 Determinación de parámetros

7.1.1 Mergesort

k	T. real [s]	T. de usuario [s]	T. de sistema [s]	#Total accesos	#Accesos aleatorios
10	9673,744	7472,399	1112,330	1048583200	21880
11	12771,129	10090,803	1337,380	1048604592	26258
12	12946,579	10230,659	1241,386	1048601504	25945

Con estos valores, se determinó utilizar un valor de k igual a 10 para la experimentación posterior.

7.1.2 Samplesort

t	T. real [s]	T. de usuario [s]	T. de sistema [s]	#Total accesos	#Accesos aleatorios
1	8684,824	3969,824	3777,764	237591842	92585536
2	7339,479	3218,137	3199,584	239552829	93566091
4	13288,239	6201,232	6151,504	238281095	92930203

Con estos resultados, se determinó utilizar un valor de t igual a 2 para la experimentación posterior.

7.2 Experimentación con Parámetros Fijos

7.2.1 Mergesort con k=10

N	T. real [s]	T. de usuario [s]	T. de sistema [s]	#Total accesos	#Accesos aleatorios
2^1M	72,888	42,303	23,425	4096000	80
2^2M	140,956	93,342	29,714	8192000	160
2^3M	344,587	223,566	72,273	16384000	320
2^4M	936,570	509,636	251,176	49152000	1080
2^5M	1941,412	1082,092	499,096	98304000	2160
2^6M	2959,969	1644,907	737,294	196608000	4020
2^7M	8355,270	4348,711	2281,735	524293600	12440
2^8M	17128,339	4699,026	2466,718	1048583200	21880

7.2.2 Samplesort con t=2

N	T. real [s]	T. de usuario [s]	T. de sistema [s]	#Total accesos	#Accesos aleatorios
2^1M	130,346	42,431	73,317	1861507	725979
2^2M	184,171	63,056	96,950	3737728	1459323
2^3M	309,811	107,295	159,118	6428201	2394994
2^4M	662,263	193,196	382,756	15779464	6251573
2^5M	1331,645	408,750	763,344	32624769	13036114
2^6M	2333,341	728,118	1300,781	59630034	23262394
2^7M	4724,473	1428,245	2671,563	120982464	47386028
2^8M	9740,495	2884,808	5498,868	244008674	95794141
2^9M	19412,682	7546,560	8635,800	476543013	185851036