Algoritmos para Memoria Externa

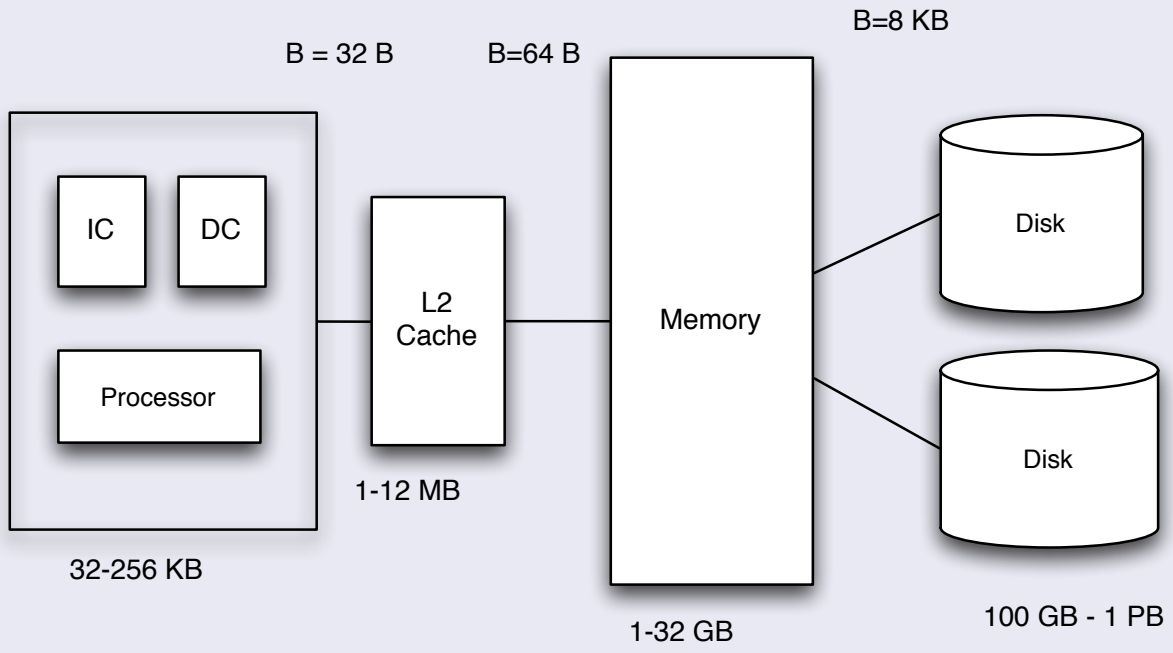
M. Andrea Rodríguez-Tastets
Ayudante: Erick Elejalde

Universidad de Concepción,Chile
www.inf.udec.cl\ ∼andrea
andrea@udec.cl

I Semestre - 2014

Introducción

- Por razones de economía, los sistemas de computadores de propósito general usualmente tienen una jerarquía de niveles de memoria, cada una de las cuales tiene su propio costo y características de rendimiento.



Introducción

- La mayoría de los lenguajes de programación se basan en un modelo de programación el cual consiste de un espacio de direcciones uniforme.
- Los programadores tiene la tendencia a asumir que todas las referencias a memoria requieren el mismo tiempo de acceso, lo cual puede ser razonable para muchos casos, en particular para los casos donde el volumen de datos es pequeño. Sin embargo, no todas las referencias son creadas de igual forma y en muchas aplicaciones de proceso masivo de datos, la comunicación de entrada y salida es el cuello de botella.
- Localidad es un término que se refiere el hecho que ciertos datos son referenciados repetidamente por un tiempo por un programa, el cual luego cambia su atención a otro conjunto de datos. Esto es una característica que puede ser ocupada en el diseño de un algoritmo para mejorar el rendimiento de un programa.
- Heurísticas de caching y prefetching reducen la ocurrencia de un “fault”, en el cual el dato referenciado no se encuentra en cache y debe ser recuperado desde una memoria de mayor nivel.
- Se les dicen a los algoritmos que hacen uso de movimientos entre memorias de distintos nivel, en especial entre disco y memoria principal, como algoritmos de *external memory algorithms, I/O algorithms, o out-of-core algorithms*.

Modelo de Disco Paralelo

- Cuando los datos son muchos para calzar en la memora principal, entonces se usa memoria externa en uno o más discos magnéticos.
- La lectura en disco tiene asociado dos tiempo: *seek* y *latency*. *Seek* es el tiempo necesario para que la cabeza ("head") del disco se posicione en el cilindro correcto y *rotational latency* es el tiempo de espera de la cabeza para posicionarse en la pista ("track"). Luego *latency = seek + rotational latency* (en el orden de milisegundos aprox.).
- Para minimizar el tiempo inicial de posicionamiento (latency), la idea es transferir un bloque de datos cada vez.
- Los algoritmos pueden correr considerablemente más rápido si el patrón de acceso a memoria exhibe localidad de referencias, permitiendo que los bloques transfieran datos que tienen la atención de un programa en conjunto. Aún así existe una diferencia de rendimiento en el acceso entre memoria internal y externa. Esta diferencia crece ya que los chips de memoria se hacen más rápidos y se usan procesadores en paralelo (multicores). Como resultado, sistemas de almacenamiento tales como RAID organizan discos multiples que pueden ser accesados en paralelo de manera de tener ancho de banda adicional.

Modelo de Disco Paralelo (PDM) (cont.)

- PDM es un buen modelo de programación genérico que facilita el diseño de algoritmos eficientes de I/O. Sin embargo existen otros modelos que abordan simplificaciones que hace el PDM y que quedan fuera del alcance de este curso.
- Los dos mecanismos claves para el diseño eficiente de algoritmos en PDM son (i) localidad de referencia que toma ventaja de la transferencia en bloques y (ii) acceso de disco en paralelo que toma ventaja de múltiple discos.
- En una I/O, cada uno de los discos D puede simultáneamente transferir B ítemes de datos contiguos.
- Los principales parámetros de PDM son:

N	Tamaño del problema (en unidades de ítemes de datos)
M	Tamaño de la memoria interna (en unidades de ítemes de datos)
B	Tamaño de transferencia de bloques (en unidades de ítemes de datos)
D	Número de drivers independientes de disco
P	Número de CPUs

donde $M < N$ y $1 \leq DB \leq M/2$, los N se consideran de tamaño fijo y el $i-$ésimo bloque en cada disco, para $i \geq 0$ consiste de localizaciones $iB, iB+1, \ldots, (i+1)B-1$.

Modelo de Disco Paralelo (PDM) (cont.)

- Si $P \leq D$, cada uno de los P procesadores puede manejar cerca de D/P discos. Si $D < P$, cada disco es compartido por cerca de P/D procesadores.
- La memoria interna es de tamaño M/P por procesador y los P procesadores están conectados por una red de interconexión o memoria compartida o una combinación de los dos.
- Para consideraciones de ruteo, una propiedad deseable de la red es que la capacidad para ordenar los M ítemes en la memorias internas colectivas de los procesadores en paralelo en tiempo óptimo sea $O((M/P)logM)$.
- Además se definen
  - Q Número de consultas (para problemas que se ejecutan en batch)
  - Z Tamaño de la respuesta
- Los parámetros se pueden definir en términos de bloques dividiéndolos por B.

Modelo de Disco Paralelo (PDM) (cont.)

- Los datos para un problema son inicialmente separados en los D discos en unidades de bloques y se requiere que los datos finales estén similarmente repartidos. Un formato como este permite que un archivo de N ítemes de datos sea entrado o sacado en $O(N/DB)$ I/Os, lo cual es óptimo.
- Las medidas de rendimiento en PDM son:
  (1) El número de operaciones I/O realizadas
  (2) La cantidad de espacio en disco utilizada
  (3) El tiempo computacional interno (secuencial o en paralelo)
- Idealmente los algoritmos y estructuras de datos deberían usar espacio lineal, lo cual significa $O(N/B)$ bloques de disco de almacenamiento.

Operaciones fundamentales de I/O y cotas

- Las siguientes operaciones son fundamentales en el rendimiento de I/O para muchos algoritmos y estructuras de datos:

  (1) *Scanning* un archivo de N datos, lo que implica la lectura o escritura secuencial de los ítemes de un archivo.
  (2) *Ordenamiento* de un archivo de N datos.
  (3) *Búsqueda* en línea a través de los N datos ordenamos.
  (4) *Salida* de los Z ítemes de una respuesta a una consulta.

Operaciones fundamentales de I/O y cotas (cont.)

Operación	**Cota I/O para** $D = 1$	**Cota I/O para** $D \geq 1$
Scan(n)	$\Theta\left(\frac{N}{B}\right)$	$\Theta\left(\frac{N}{DB}\right)$
Sort(n)	$\Theta\left(\frac{N}{B} log_{M/B} \frac{N}{B}\right)$	$\Theta\left(\frac{N}{DB} log_{M/B} \frac{N}{B}\right)$
Search(n)	$\Theta\left(log_B N\right)$	$\Theta\left(log_{DB} N\right)$
Output(Z)	$\Theta\left(max\left\{1, \frac{Z}{B}\right\}\right)$	$\Theta\left(max\left\{1, \frac{Z}{DB}\right\}\right)$

Localidad y balance de carga: disco único

- En forma simplificada, el objetivo de un algoritmo EM para problemas batch es derivar algoritmos eficientes tal que los N y Z términos en una cota de I/O para un algoritmo “naive” sean reemplazados por N/B y Z/B,
- Para algoritmo en línea se espera que la base logarítmica de la cota sea B y que se reemplace Z por B.

Disk striping: múltiples discos

- *Disk striping* es un paradigma práctico que puede facilitar la programación con múltiples discos. Cuando se usa, I/O son permitidos solo sobre tiras (stripes) completos, un stripe a la vez. En la figura que sigue se consideran bloques de tamaño 2.

	D_0		D_1		D_2		D_3		D_4	
stripe 0	0	1	2	3	4	5	6	7	8	9
stripe 1	10	11	12	13	14	15	16	17	18	19
stripe 2	20	21	22	23	24	25	26	27	28	29
stripe 3	30	31	32	33	34	35	36	37	38	39

- Este paradigma convierte un algoritmos que usa un disco con tamaño de bloque DB en un algoritmo que usa D discos, cada uno con tamaño de bloque B.

Ordenamiento externo y problemas relacionados

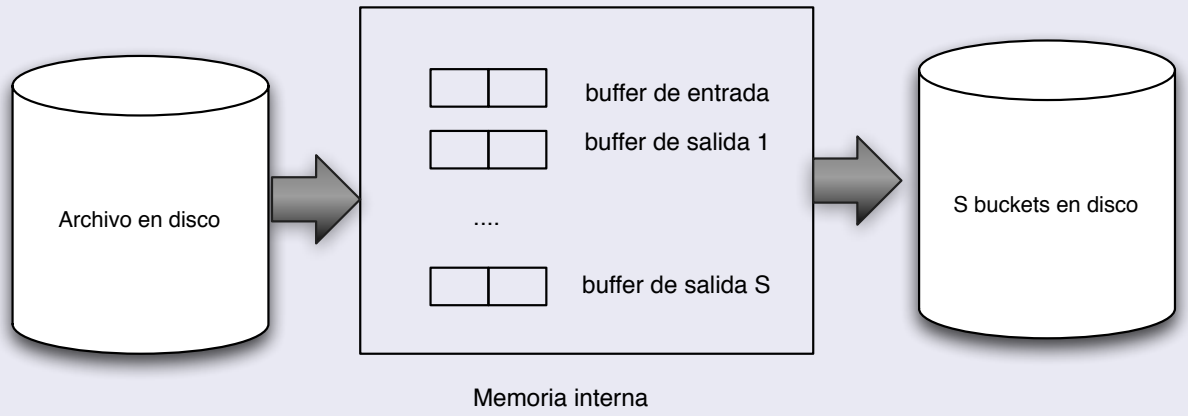
Teorema: El número de I/O en el caso promedio y el peor caso requerido para ordenar $N = nB$ ítemes de datos usando D discos es:

$$Sort(N) = \Theta\left(\frac{n}{D} log_m n\right).$$

donde m = M/B.

Ordenamiento por *Distribución*

- *Distribution sort* es un proceso recursivo en el cual usamos un conjunto de $S-1$ elementos de partición $e_1, \ldots, e_{S-1}$ que dividen los datos en S sub archivos disjuntos (o *buckets*).



- El $i-$ésimo bucket, con $1 \leq i \leq S$, consiste de todos los ítemes con valor de clave en el intervalo $[e_{i-1}, e_i)$, donde por convención $e_0 = -\infty$ y $e_S = +\infty$.
- Se puede ordenar recursivamente los buckets individuales y concatenarlos juntos para formas una única lista ordenada completa.

Ordenamiento por *Distribución* (cont.)

- La idea básica es que se trabaja en cada subconjunto definido por los elementos de partición e_i, con $0 \leq i \leq S$ en forma recursiva hasta que el subconjunto sea tan pequeño que quepa en la memoria, en cuyo momento simplemente se ordena el subconjunto. Luego el ordenamiento de la lista completa se puede obtener en $O(N/B)$ I/O que combinan todas los subconjuntos pequeños.
- Lo fundamental es escoger las particiones en forma eficiente. Si S no es muy grande, esto es más simple que ordenar los datos porque no nos importanta el ordenamiento en cada subconjunto.

Ordenamiento por *Distribución*: elementos de partición

- Un requerimiento es escoger las $S-1$ particiones de manera que los buckets sean relativamente del mismo tamaño. Cuando eso sucede, el tamaño del bucket decrece desde un nivel a otro de la recursividad en un factor de $\Theta(S)$, y entonces hay $O(log_S n)$ niveles de recursividad.
- En cada recursión, se revisan los datos. A medida que los datos entran a memoria principal, ellos se dividen en S buckets en una forma online. Cuando un buffer de tamaño B se llena por uno de los buckets, su bloque puede ser enviado a disco, y otro buffer es usado para almacenar el próximo conjunto entrante de datos de ese bucket.
- Entonces el máximo número de buckets es $\Theta(M/B)$, con un número de niveles de recursión de $\Theta(log_{M/B} N/B)$. Por otro lado en el último nivel, no tiene sentido tener un bucket de menos de $\Theta(M)$ ítemes, de manera que se limita S a $O(N/M)$. Luego, el número deseado de S es $\Theta(min\{M/B, N/M\})$.
- Parece difícil encontrar $S = \Theta(min\{M/B, N/M\})$ elementos de partición usando $\Theta(N/DB)$ y que se tengan bucket dentro de un factor constante entre ellos. Parece más fácil usar modelos probabilísticos para encontrar los elementos de partición basado en muestreos aleatorios.

Ordenamiento por *Distribución*: elementos de partición (cont.)

Una estrategia para obtener particiones en forma probabilística es el siguiente.

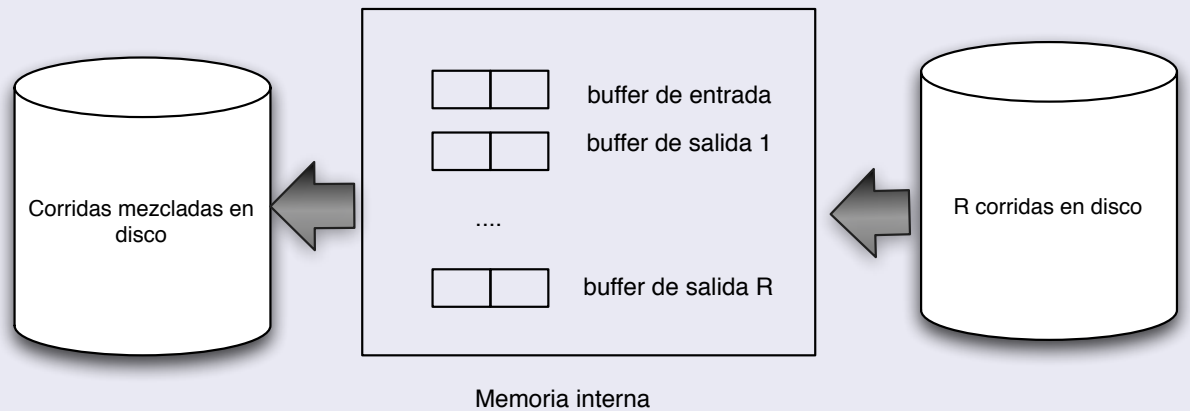
- Sea $d = O(logS)$.
- Se toma una muestra aleatoria de dS ítemes, se ordenan la muestra y se escojen de ella cada $d-$ésimo de la lista ordenada para ser el pivote o elemento de partición. Cada bucket tiene el tamaño deseado de $O(N/S)$ ítemes.
- El número de I/O necesarios para escoger los elementos de partición son $O(dS + Sort(dS))$.

Ordenamiento por *Distribución*: Balance de carga entre discos

- Para alcanzar el rendimiento definido en el teorema se deben forman buckets en cada nivel de la recursión usando $O(n/D)$ I/Os, lo cual es fácil para un $D = 1$.
- En el caso general, cada paso de entrada o salida debe involucrar en promedio $O(D)$ discos.
- Cada archivo que se particiona es en realidad uno de los buckets del nivel de recursión anterior y se debe distribuir uniformemente entre los discos para no producir cuellos de botella.
- Existen diferentes trabajos que han propuesto formas de distribuir homogeneamente los buckets en los D discos, varios de estos trabajos usan técnicas de alearidad.

Ordenamiento por *Merge*

- *Merge sort* es de alguna manera ortogonal al método por distribución. En cada paso de la fase de formación, se recorren lo n bloques de datos, cada carga de memoria a la vez, la cual se ordena y envía en series de stripes al disco. Al final de la fase de formación, existen N/M corridas ordenadas, cada una dividida (striped) entre los discos. En la fase de merge, cada paso agrupa R corridas. Se recorren las R corridas y se mezclan los ítemes en una manera online. A lo más $R = \Theta(M/B)$ corridas pueden ser mezcladas a la vez y el número de pasos resultante es $O(log_{M/B} N/B)$.



algoritmo Sort-Merge

Sea $i \leftarrow 1$; $j \leftarrow b$ (tamaño del fichero en bloques);
  $k \leftarrow n_b$ (tamaño del búfer en bloques); $m \leftarrow \lceil (j/k) \rceil$;
(Fase de clasificación)
mientras ($i \leq m$)
  hacer {
    leer siguientes k bloques del fichero en el búfer o si quedan menos de k bloques,
    entonces leer los restantes bloques;
    clasificar los registros en el búfer y escribirlos como un subfichero temporal;
    $i \leftarrow i + 1$;
  }
(Fase de fusión: fusionar archivos hasta que sólo quede 1)
Sea $i \leftarrow 1$; $p \leftarrow \lceil log_{k-1} m \rceil$ (p es el número de pasadas de la fase);
  $j \leftarrow m$;
mientras ($i \leq p$)
  hacer {
    $n \leftarrow 1$; $q \leftarrow \lceil (j/(k-1)) \rceil$; (número de archivos a escribir en este paso)
    mientras ($n \leq q$)
      hacer {
        leer los siguientes $k - 1$ subarchivos o los subarchivos restantes un bloque cada vez;
        fusionar y escribir un nuevo subarchivo;
        $n \leftarrow n + 1$;
      }
    $j \leftarrow q$; $i \leftarrow i + 1$;
  }

Ordenamiento por *Merge* (cont.)

- Para el caso con $D \geq 1$, cada operación de entrada paralela durante la fase de mezcla en promedio trae los próximos $\Theta(D)$ bloques que se necesitan. El desafío es entonces asegurar que estos bloques residan en diferentes discos de manera que ellos pueden ser entrados en una operación paralela de I/O. La dificultad es que las corridas a ser mezcladas fueron formadas durante el paso de mezcla anterior, sin saber cómo ellos interactúan con otros en mezclas posteriores.

Lower Bound para el ordenamiento externo

Teorema: Ordenamiento externo requiere $\Omega(n log_m n)$ I/O en un modelo de comparación.

La idea básica es que dado una entrada de N, hay $N!$ permutaciones posibles para el ordenamiento correcto que son consistentes con la entrada. Entonces, se debe ver cuánto se puedo reducir este número usando una operación de entrada y cualquier número de operaciones de comparación, asumiendo un adversario que escoge la peor posible salida de la comparación. Las operaciones de salida no ayudan a reducir las posibilidades.

Lower Bound para el ordenamiento externo (cont.)

- Sea P el conjunto de todas las $N!$ permutaciones posibles. Después de ordenar un bloque, se reduce el tamaño de P por un factor de $B!$. Entonces al final de este paso, existen aún $N!/(B!)^n$ permutaciones en P para escoger.
- Cada vez que un bloque es leído en memoria, se considera que el algoritmo ya lo ordenó con los otros $M - B$ elementos en memoria, descartando las permutaciones en P inconsistentes con el ordenamiento de los M ítemes.
- Si se asume el orden de los elementos en memoria antes de entrar un nuevo bloque, entonces hay a lo más $\binom{M}{B}(B!)$ ordenamientos posibles de los registros en memoria interna.
- Si S es el número de ordenamientos antes de la última entrada, entonces existe al menos 1 de los $\binom{M}{B}(B!)$ de los ordenamientos en memoria interna, tal que el número total de ordenamientos consistentes con este orden es al menos $\frac{S}{\binom{M}{B}(B!)}$
- Se desprende que después de t operaciones de entrada, el número de posible ordenamientos es al menos $\frac{N!}{\left(\binom{M}{B}(B!)\right)^t}$.

Lower Bound para el ordenamiento externo (cont.)

- Lo anterior fue asumiendo que no se sabe el orden de los B elementos leídos en memoria, pero este puede no ser el caso ya que los B elementos pudieron ya estar en memoria juntos.
- El número de veces que leemos B y que no hayan sido leídos previamente son N/B. Luego, después de t entradas que hay al menos $\frac{N!}{\left(\binom{M}{B}^t (B!)^{n/B}\right)}$ ordenes consistentes con la información obtenida por el abversario.
- Lo que se quiere es reducir la expresión hasta 1, y el número de I/O deba ser al menos t, tal que $\frac{N!}{\left(\binom{M}{B}^t (B!)^{n/B}\right)} \leq 1$.

Lower Bound para el ordenamiento externo (cont.)

- Usando que $logx! = xlogx$ y $log(\binom{M}{B}) = BlogM/B$, entonces:

$$\frac{N!}{(\binom{M}{B})^t (B!)^{N/B}} \leq 1$$

$$(\binom{M}{B})^t (B!)^{N/B} \geq N!$$

$$tlog(\binom{M}{B}) + N/Blog(B!) \geq log(N!)$$

$$tBlog(M/B) + (N/B)BlogB \geq NlogN$$

$$tBlog(M/B) \geq Nlog(N/B)$$

$$t \geq N/B \frac{log(N/B)}{log(M/B)}$$

$$t \geq nlog_m n$$

Prefetching

- Dada una secuencia de bloques $\Sigma = (b_1, b_2, \ldots, b_N)$.
- La posición inicial de los bloques en los discos D es pre especificada por una función arbitraria $disk : \Sigma \rightarrow \{1, 2, \ldots D\}$. Entonces un bloque b_i es posicionado en un disco $disk(b_i)$.
- El objetivo del *prefetching* es lograr el mínimo número posible de operaciones de entrada I/O desde los discos D de manera que los bloques sean leídos por un programa en el orden dado por Σ.
- Se usan los bloques en memoria interna como *prefetch buffers*.

Prefetching (cont.)

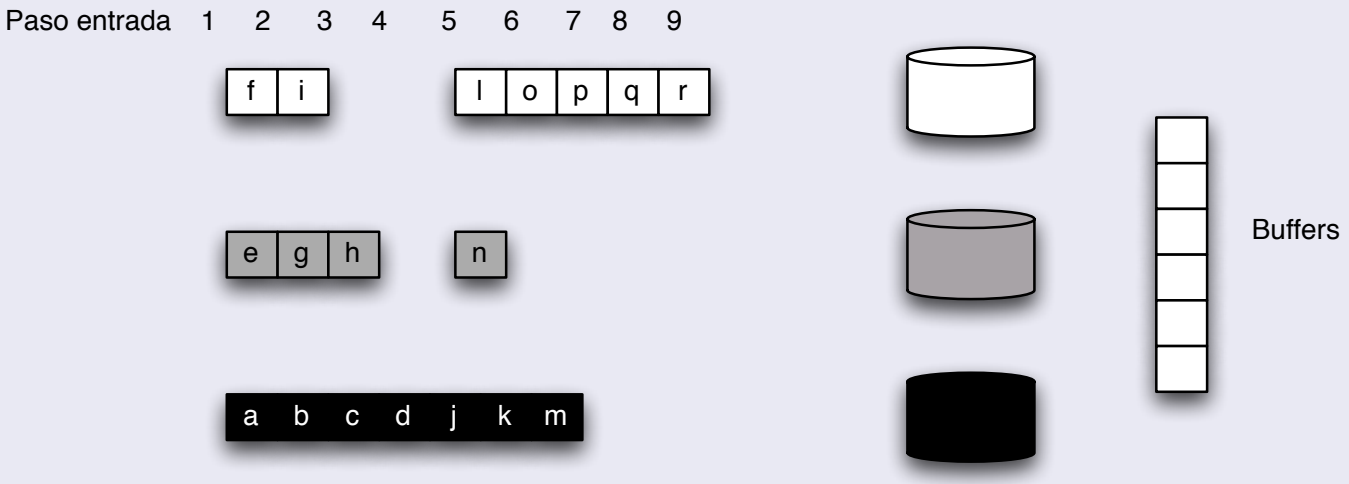
- Cuando los bloques en Σ son distintos, el problema de prefetching se denomina *read-once scheduling*.
- Cuando existen bloques repetido en Σ, entonces es deseable dejar en la cache bloques en los prefetch buffers de manera de evitar re entradas de ellos más tarde, lo cual se llama el problema de *read-many scheduling*.
- Una forma de tomar decisiones de prefetching más informadas es usar conocimiento o predicción de requerimientos futuros de lectura, lo que se dice informalmente *lookahead*.
- Existe el problema de *output scheduling* el cual se basa en escritura de una aplicación y salidas a disco. Similarmente al basado en lecturas, se definen los problemas *write-once scheduling* y *write-many scheduling*.

Greedy read-once scheduling

- Asuma que los bloques en $b_1, b_2, \ldots, b_i$ de Σ han sido leidos en un paso anterior y son removidos de los prefetch buffers.
- El paso siguiente consiste en leer el próximo bloque en Σ que ya está en los prefetch buffers. Suponga entonces bloques $b_{i+1}, b_{i+2}, \ldots, b_j$ están en los prefetch buffers, pero que b_{j+1} está en disco. Entonces bloques $b_{i+1}, b_{i+2}, \ldots, b_j$ son leídos y luego removidos desde los prefetch buffers.
- La segunda etapa consiste en la entrada desde disco. En esta etapa considere por cada uno de los D discos, el bloque de mayor prioridad aún no entrado, de acuerdo al orden especificado por Σ. Denotemos P a este conjunto de bloques.
- Sea Res los bloques que están residentes en el prefetch buffer, y sea Res' los bloques con las m mayores prioridades en $P \cup Res$. Se entran los bloques $Res' \cap P$. Al término del paso de I/O, el prefetch buffer contiene los bloques en Res'. Si un bloque en Res no permanece en Res', entonces el algoritmo lo ha sacado de los prefetch buffers y deberá ser re entrado en un paso de I/O posterior.

Greedy read-once scheduling (ejemplo)

Sea $\Sigma = (a, b, c, d, e, \ldots, r)$, $D = 3$ y $m = 6$ prefetch buffers.

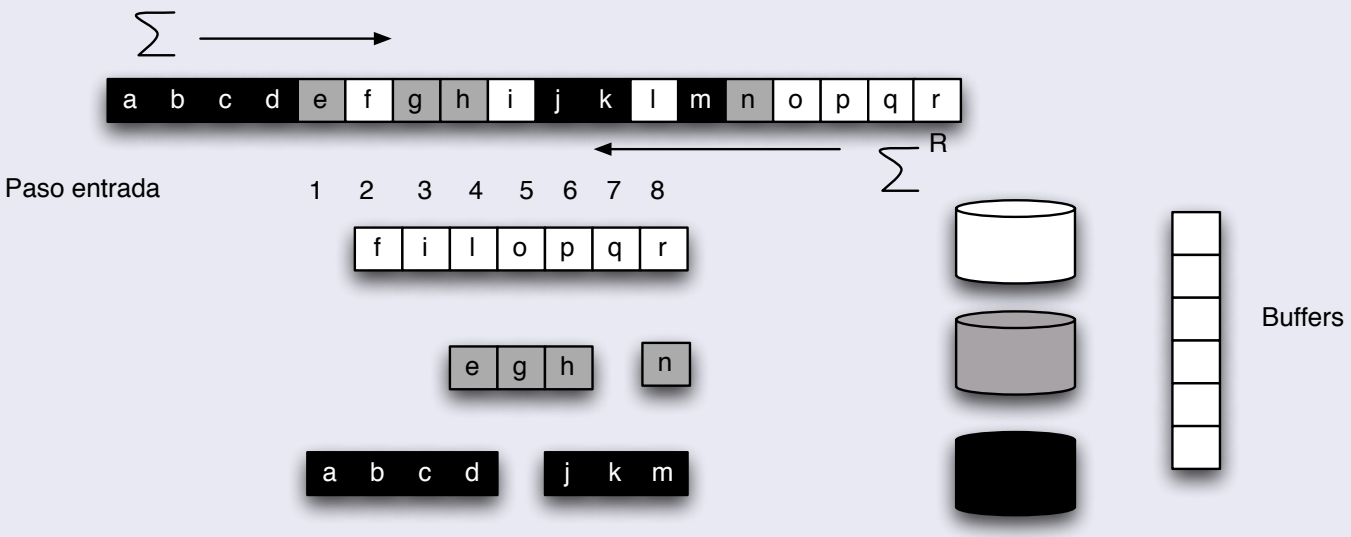


Duality: read-once scheduling

- Asuma secuencia read-once Σ y una secuencia write-once Σ^R, donde Σ^R denota la secuencia Σ en order reverso.
- Para el problema de write-once problem, un algoritmo greedy es el siguiente: Cuando los bloques de Σ^R son escritos, se pone cada bloque en un buffer de salida para el disco designado. Hay m buffers de salida y cada escritura puede proceder cuando se libera el buffer de salida. En cada paso de I/O paralelo, se libera espacio sacando un bloque en la fila a cada disco que tiene al menos un bloque en un buffer de salida.

Duality: read-once scheduling(ejemplo)

Sea $\Sigma = (a, b, c, d, e, \ldots, r)$, $D = 3$ y $m = 6$ prefetch buffers.

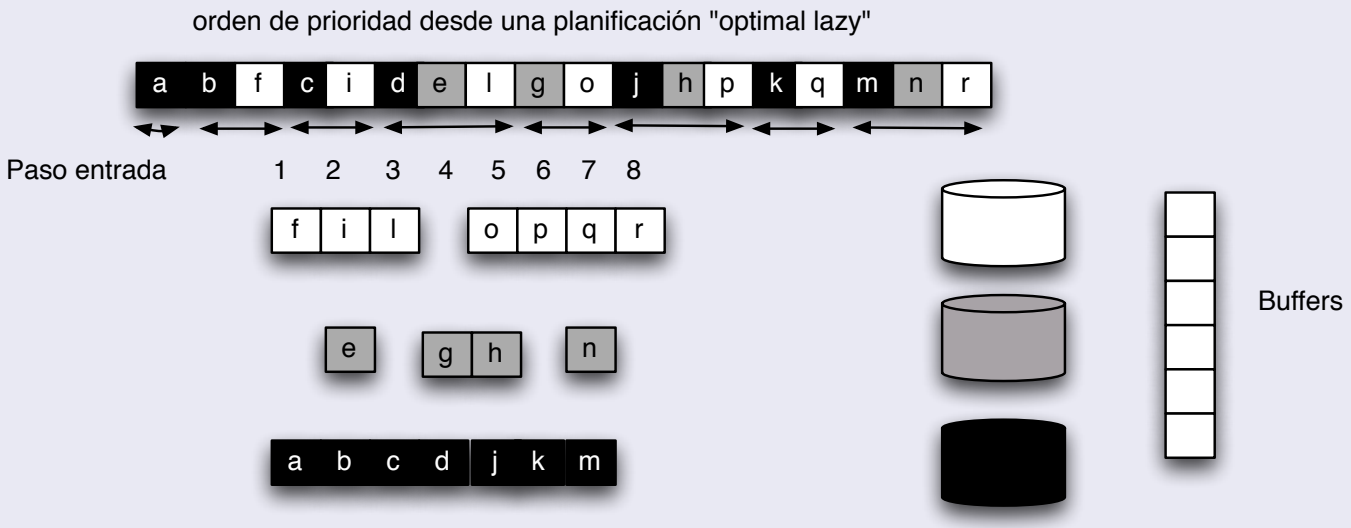


Duality: prudent prefetching

- Una forma que garantiza una solución óptima y que los I/O son los menos posibles es *prudent prefetching*.
- Se redefine la prioridad de un bloque para ser el orden en que aparece en el paso de entrada de la planificación lazy (i.e., a,b,f,c,i,...). Prefetch buffers son reservados para bloques en el mismo orden de prioridad. Luego se usa el mismo algoritmo greedy para read-once scheduling.

Duality: prudent prefetching (ejemplo)

Sea $\Sigma = (a, b, c, d, e, \ldots, r)$, $D = 3$ y $m = 6$ prefetch buffers.



Duality: teorema

Si cada una de las S subsecuencias que forman Σ es almacenada en un disco en forma aleatoria (randomized cycling layout), el número esperado de I/O para el método de prefetching óptimo (prudent o lazy) es:

$$\left(1 + O\left(\frac{D}{m}\right)\right)\frac{n}{D} + min\left\{S + \frac{n}{D}, O\left(\frac{m}{D}logm\right)\right\}$$

Entonces para m suficientemente grande, el número de I/O es:

$$\lfloor\frac{n}{D}\rfloor + S \tag{1}$$

Duality: read-many scheduling

- Para bloques que aparecen más de una vez en Σ, es útil dejar los prefetch buffers a estos bloques para evitar entrarlos más de una vez para lecturas posteriores.
- Un idea es que a cada paso de salida y por cada disco, se escoja el bloque a sacar a disco que aparezca más tarde en el Σ^R.
- Intuitivamente, el bloque que viene más lejos o tarde será menos usado para mantenerlo en memoria para una escritura posterior.