Evaluación de Consultas

Implementación de los Operadores de los lenguajes de manipulación de datos (LMD).

Una de las ventajas de los lenguajes de consulta de los SGBD relacionales es que están compuestos por pocos operadores y que las consultas deben ser optimizadas para lograr un mejor rendimiento en el manejo de la base de datos.

Existen varias alternativas para implementar cada operador y ninguna es la solución, universalmente, ideal.

El mejor algoritmo en un momento dado depende de varios factores:

- Tamaño de las relaciones involucradas en la consulta.
- Si existen índices.
- Si las relaciones están ordenadas.
- Tamaño del *buffer.*
- Políticas de re-emplazo en el *buffer.*

Técnicas usadas para desarrollar algoritmos para operadores relacionales:

Iteración: examina todas las tuplas de una relación de entrada iterativamente. En ocasiones se utilizan índices con entradas de datos que son más pequeños y contienen los datos necesarios.

Indexación: si la consulta especifica una condición de selección o una condición de un *join*, se usan índices para examinar sólo las tuplas que satisfacen la condición.

Particionamiento: se crean particiones de tuplas según una clave de ordenamiento, con ello se puede descomponer un operador en operaciones menos costosas que trabajan sobre las particiones. Dos técnicas usadas para hacer *particionamiento* son el *Sorting* y el *Hashing*.

Caminos de Acceso (*Access Paths*) y Selectividad:

Existen dos formas de acceder a las tuplas de una o varias relaciones:

1.- File Scan: consiste en recorrer toda la relación tupla por tupla.

2.- Índice más una condición de selección:

Si se tiene un índice cuya clave de búsqueda es *atr* y la condición de selección es del tipo ***atr op valor***, donde ***op*** $\in \{=, \neq, \geq, >, <, \leq\}$.

Caminos de Acceso (*Access Paths*) y Selectividad:

- La *selectividad* de un camino de acceso se refiere al número de páginas recuperadas (las páginas de índices más las páginas de datos, dado el caso) para obtener las tuplas deseadas.

- El camino más selectivo es el que requiere la menor cantidad de páginas para obtener las tuplas.

- Usando el camino más selectivo se minimiza el costo de consultar los datos.

Preliminares: Caso de estudio

Estudiante(*ci-e* **int**, *nombre-e* **string**, *promedio* **int**, *edad* **int**)
Preparador(*ci-p* **int**, *asig* **int**, *fecha* **date**, *nombre-p* **string**)

- Cada tupla de la relación Preparador mide 40 bytes. Una página puede contener 100 tuplas de Preparador y en la relación hay 1.000 páginas (cardinalidad 100*1.000, tamaño 40*100.000 bytes).

- Cada tupla de la relación Estudiante mide 50 bytes. Una página puede contener 80 tuplas de Estudiante y en la relación hay 500 páginas (cardinalidad 80*500, tamaño 50*80*500 bytes).

Preliminares: Caso de estudio

Estudiante(*ci-e* **int**, *nombre-e* **string**, *promedio* **int**, *edad* **int**)
Preparador(*ci-p* **int**, *asig* **int**, *fecha* **date**, *nombre-p* **string**)

- Los costos se miden en operaciones de I/O. Én una operación I/O se lee o se escribe una página.

- La complejidad computacional del algoritmo se expresa en términos de O-grande de acuerdo al parámetro de entrada. Por ejemplo, el costo de una operación de *scan* sobre una relación es O(M), donde M es el número de páginas de la relación.

OPERADORES RELACIONALES

Selección:

σ *attr op value* (R)

- Sin Indices, ni Ordenamiento
- Sin Indices, Con Ordenamiento.
- Con un índice *B+ tree* y una condición de selección.
- Con un índice *Hash* si la condición de selección es una igualdad.

OPERADORES RELACIONALES

Selección: σ *attr op value* (R)

Estudiante(*ci-e* **int**, *nombre-e* **string**, *promedio* **int**, *edad* **int**)
Preparador(*ci-p* **int**, *asig* **int**, *fecha* **date**, *nombre-p* **string**)

```
SELECT *
FROM Preparador P
WHERE P.nombre-p = 'Lucia';
```

1. SIN INDICE NI ORDEN:

Si no hay índice en Preparador sobre el atributo nombre-p y Preparador no está ordenada por el atributo nombre-p, el camino de acceso más selectivo a los datos deseados, es un *scan* o recorrido completo de la relación. Para cada tupla se chequea la condición P.nombre-p = 'Lucia' y si es verdadera la tupla se añade al resultado.

Costo: 1000 I/Os porque Preparador tiene 1000 páginas.

Selección: σ *attr op value* (R)

Estudiante(*ci-e* **int**, *nombre-e* **string**, *promedio* **int**, *edad* **int**)
Preparador(*ci-p* **int**, *asig* **int**, *fecha* **date**, *nombre-p* **string**)

```
SELECT *
FROM Preparador P
WHERE P.nombre-p = 'Lucia';
```

2. SIN INDICE CON ORDEN:

Si no hay índice en Preparador sobre el atributo nombre-p pero Preparador está ordenada por el atributo nombre-p, se puede hacer una búsqueda binaria para localizar la primera tupla que satisfaga la condición y se hace un *scan ordenado* hasta que deje de cumplirse la condición.

Costo: costo busquedaBinaria + X, con X en [0,1000].
O(log 1000) + X = 3 + X I/Os.

Estudiante(*ci-e* **int**, *nombre-e* **string**, *promedio* **int**, *edad* **int**)
Preparador(*ci-p* **int**, *asig* **int**, *fecha* **date**, *nombre-p* **string**)

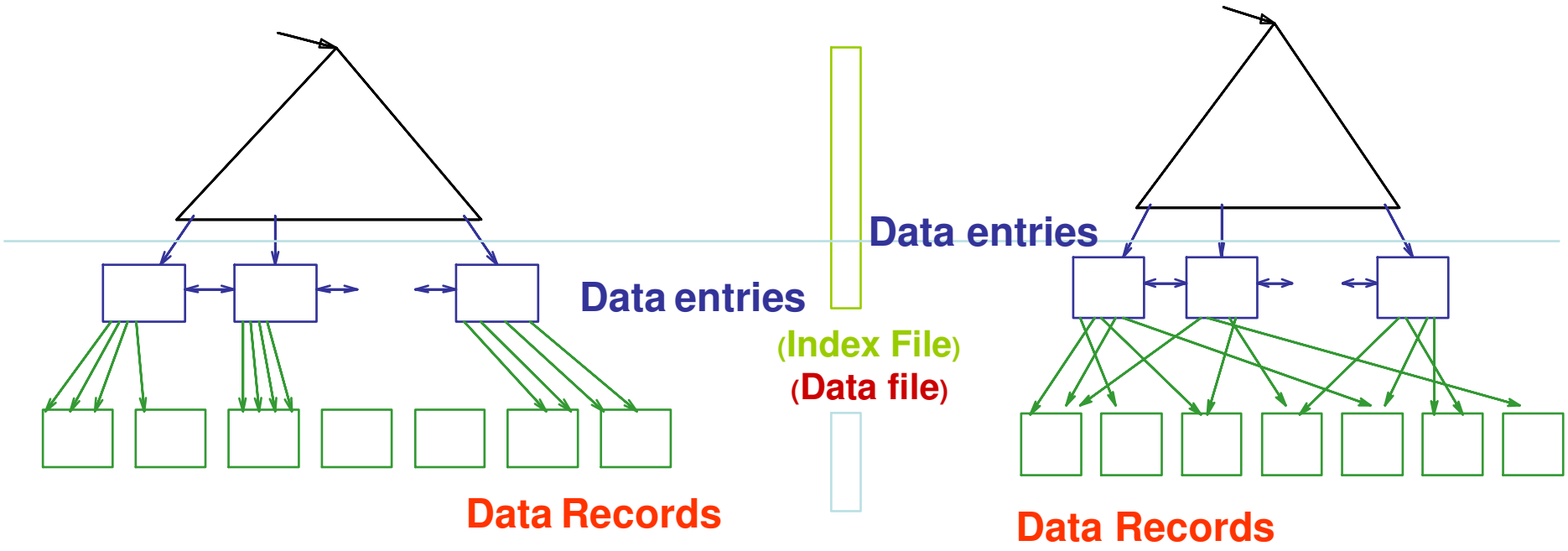
```
SELECT *
FROM Preparador P
WHERE P.nombre-p = 'Lucia';
```

3. CON INDICE de ARBOL B+:

Si hay un índice B+ en Preparador sobre el atributo nombre-p se busca en el árbol hasta encontrar el nodo hoja que apunta a la primera tupla que cumple con la condición, se recuperan las tuplas correspondientes de Preparador. El costo de llegar al nodo hoja suele ser de 2 o 3 I/Os. El costo de recuperar las tuplas depende del número de tuplas y de si el índice está *clusterizado*.
Supongamos que un 10% de Preparador está en el resultado, un total 10.000 tuplas en 100 páginas.
Si el índice es *clusterizado* el costo de recuperar las tuplas es de 100 I/Os. Si no está *clusterizado* puede, en el peor de los casos, tener un costo de 10.000 I/Os pues cada tupla puede estar en una página diferente.

Costo: 100 I/Os + 3 I/Os (clustered).
Costo: 10.000 I/Os + 3 I/Os (unclustered).

Propiedades de los Indices



CLUSTERED

UNCLUSTERED

Tomado de [1]

OPERADORES RELACIONALES

Proyección:

$$\prod_{<attr1,\ attr2,\ldots,attrn>} (R)$$

Estudiante(*ci-e* **int**, *nombre-e* **string**, *promedio* **int**, *edad* **int**)
Preparador(*ci-p* **int**, *asig* **int**, *fecha* **date**, *nombre-p* **string**)

```
SELECT P.ci-p, P.asig
FROM Preparador P;
```

Proyección: Π <attr1, attr2,...,attrn> (R)

Estudiante(*ci-e* **int**, *nombre-e* **string**, *promedio* **int**, *edad* **int**)
Preparador(*ci-p* **int**, *asig* **int**, *fecha* **date**, *nombre-p* **string**)

```
SELECT distinct P.ci-p, P.asig
FROM Preparador P;
```

El algoritmo basado en ordenamiento sigue los siguientes pasos:

1. Se escanea Preparador y produce un conjunto de tuplas que contiene sólo los atributos deseados (ci-p,asig). Suponga que cada nueva tupla ocupa 10 bytes, una cuarta parte de la tupla original, entonces T es una cuarta parte de M. Escribir T toma 250 I/Os. El costo de este paso es:
   Costo: 1000 I/Os + 250 I/Os.
2. Se ordena el conjunto de tuplas usando la combinación (ci-p,asig) como clave de ordenamiento. Con Costo: O(TlogT) con QuickSort o Merge-Sort.
3. Se escanea el resultado ordenado comparando tuplas adyacentes y eliminando duplicados. Con Costo: T.

Costo Total ≈ 1000 + 250 + 250 * 3 + 250 = 2.250 I/Os.

OPERADORES RELACIONALES

Join: $R \bowtie_{a=b} S$

Estudiante(*ci-e* **int**, *nombre-e* **string**, *promedio* **int**, *edad* **int**)
Preparador(*ci-p* **int**, *asig* **int**, *fecha* **date**, *nombre-p* **string**)

```
SELECT *
FROM Preparador P, Estudiante E
WHERE P.ci-p = E.ci-e
```

La implementación de la operación JOIN trata de evitar la enumeración completa del producto cartesiano debido a su costo. No materializarlo.

R se llama la relación externa y tiene pr tuplas por página, con M páginas.
S se llama relación interna y tiene ps tuplas por página, con N páginas.

- ***Nested Loop Join (por tupla, por página)***

Estudiante(*ci-e* **int**, *nombre-e* **string**, *promedio* **int**, *edad* **int**)
Preparador(*ci-p* **int**, *asig* **int**, *fecha* **date**, *nombre-p* **string**)

```
SELECT *
FROM Preparador P, Estudiante E
WHERE P.ci-p = E.ci-e
```

¿Es indiferente escoger cualquier relación como la externa?

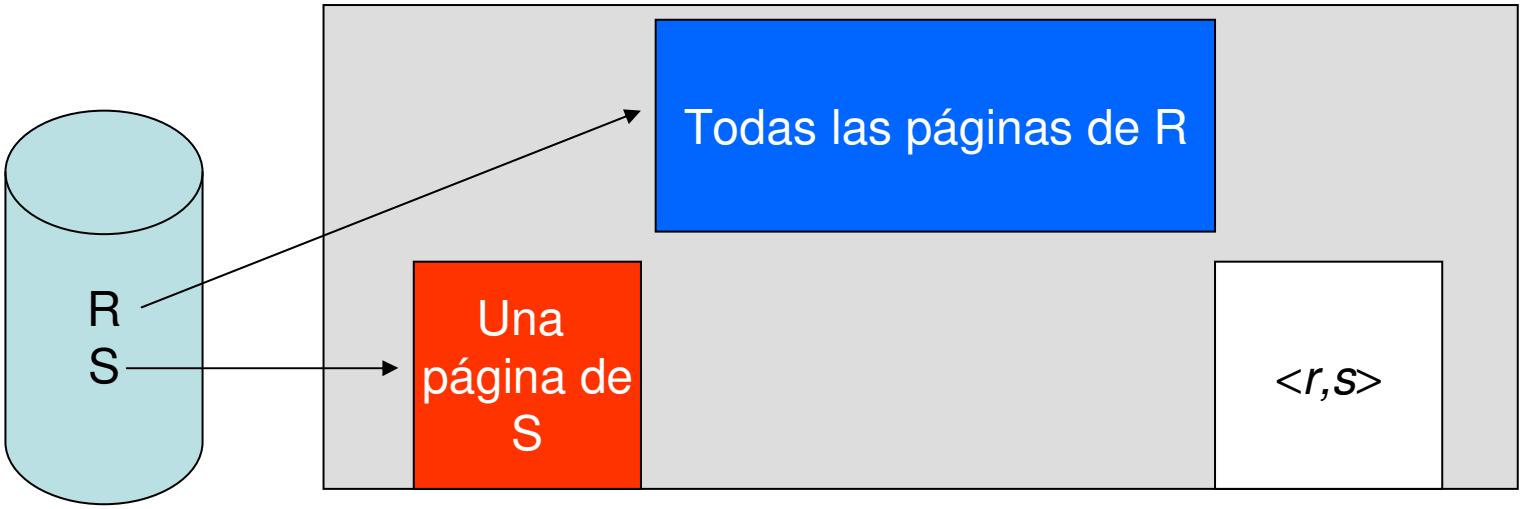
```
foreach tuple r ∈ R  do
  foreach tuple s ∈ S do
   if ri = = sj then add <r,s> to result
```

Costo: M + pr*M*N ***(por tupla: lectura de R y por cada tupla de R escanear S).***

Costo: M + M*N ***(por página: lectura de R y por cada página de R escanear S).***

1. Block Nested Loop Join

Si R cabe en memoria principal y hay dos buffers adicionales:

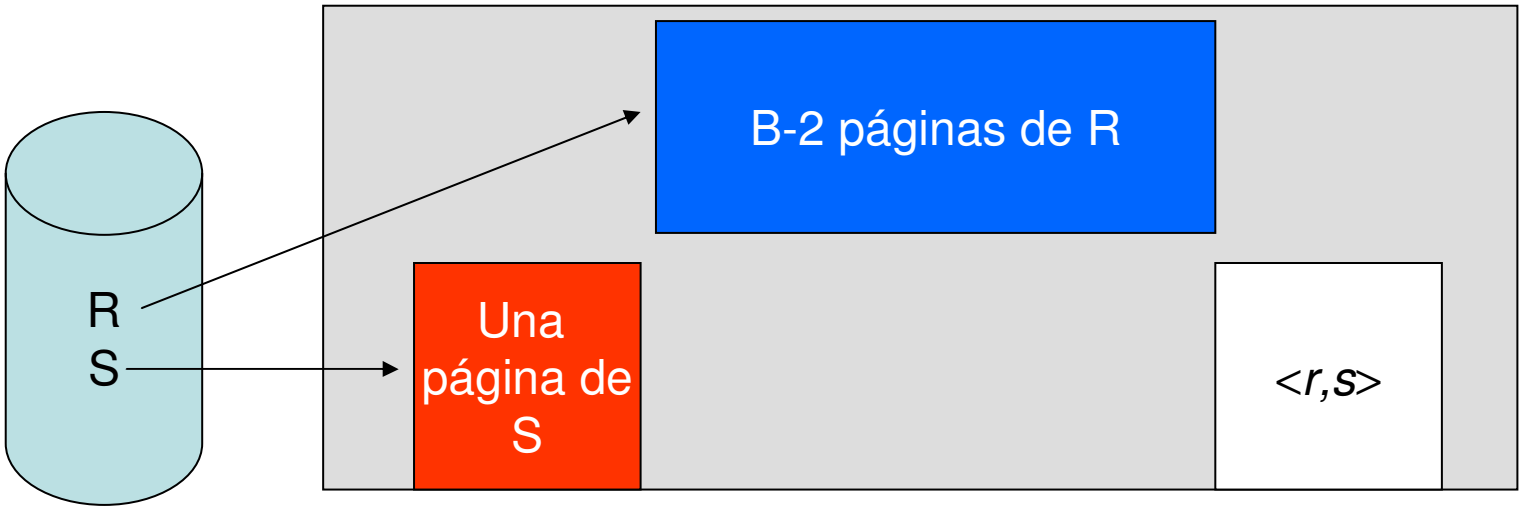


Costo: M + N. ← Costo optimal

1. Block Nested Loop Join

Si R no cabe en memoria y hay B buffers adicionales:

```
foreach block of B-2 pages of R  do
   foreach page of S do {
      for all matching in-memory tuples r ∈ R-block and
          s ∈  S-page add <r,s> to result }
```

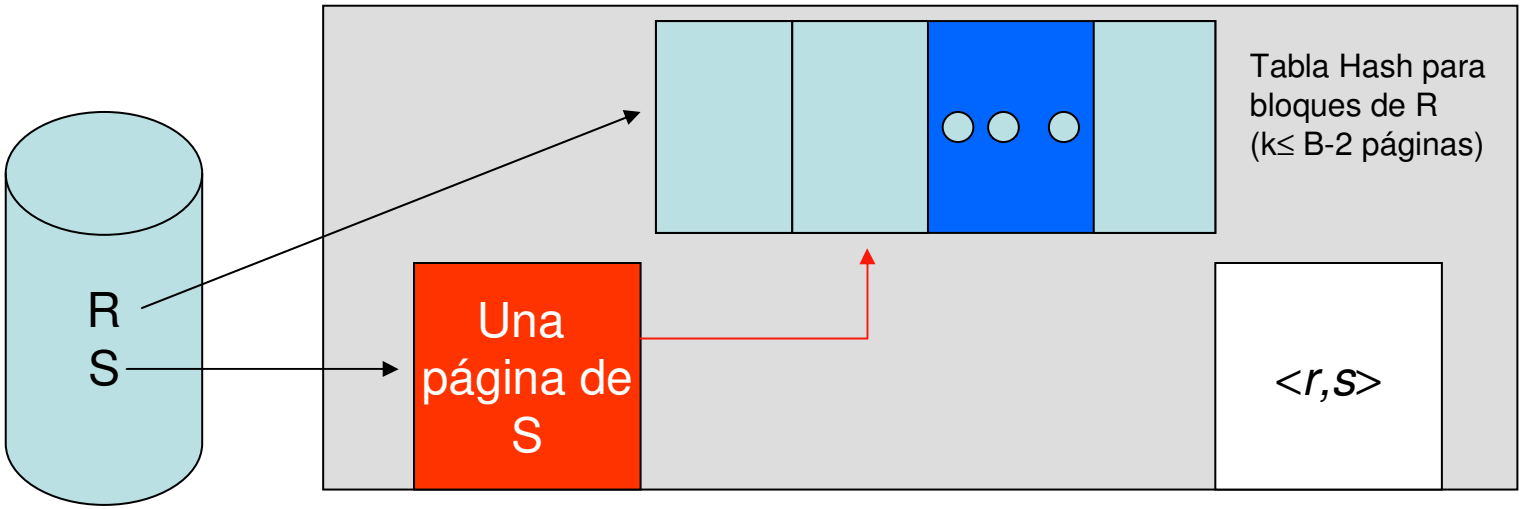


Costo: M + (M/(B-2)) * N

1. Block Nested Loop Join

Si R no cabe en memoria y hay B buffers adicionales:

```
foreach block of B-2 pages of R  do
   foreach page of S do {
      for all matching in-memory tuples r ∈ R-block and
          s ∈  S-page add <r,s> to result }
```



Costo: M + (M/(B-2)) * N

2. Index Nested Loop Join

```
foreach tuple r ∈ R  do
foreach tuple s ∈ S where ri = = sj add <r,s> to result
```

Costo: M + 3 + M*pr. (en índice clusterizados, con 3 para alcanzar la hoja y una I/O en S por cada tupla de R).

Costo: M + 3 + M*pr*N. (en índices no clusterizados).

- ***Sort-Merge Join***

Ordena las dos relaciones del join (R y S) por el atributo del *join* haciendo ordenamiento y mezcla de las dos relaciones:

El paso de ordenamiento: se hace usando cualquier algoritmo *sort* . Con ello, se agrupan las tuplas que tienen el mismo valor en el atributo del join en una misma partición.

Costo de ordenar R es O(MlogM) y de ordenar S es O(NlogN).

El paso de *merge*: se hace aplicando un *scan* a las relaciones R y S buscando las tuplas adecuadas (tuplas Tr en R y Ts en S para las cuales Tri = Tsj).

Costo de mezclar es M + N.

- ***Hash Join***

Fase 1: Particiona *R* en *k* partes:

```
foreach tuple r ∈ R  do
  read r and add it to buffer page h(ri);
```

Fase 2: Particiona S en *k* partes:

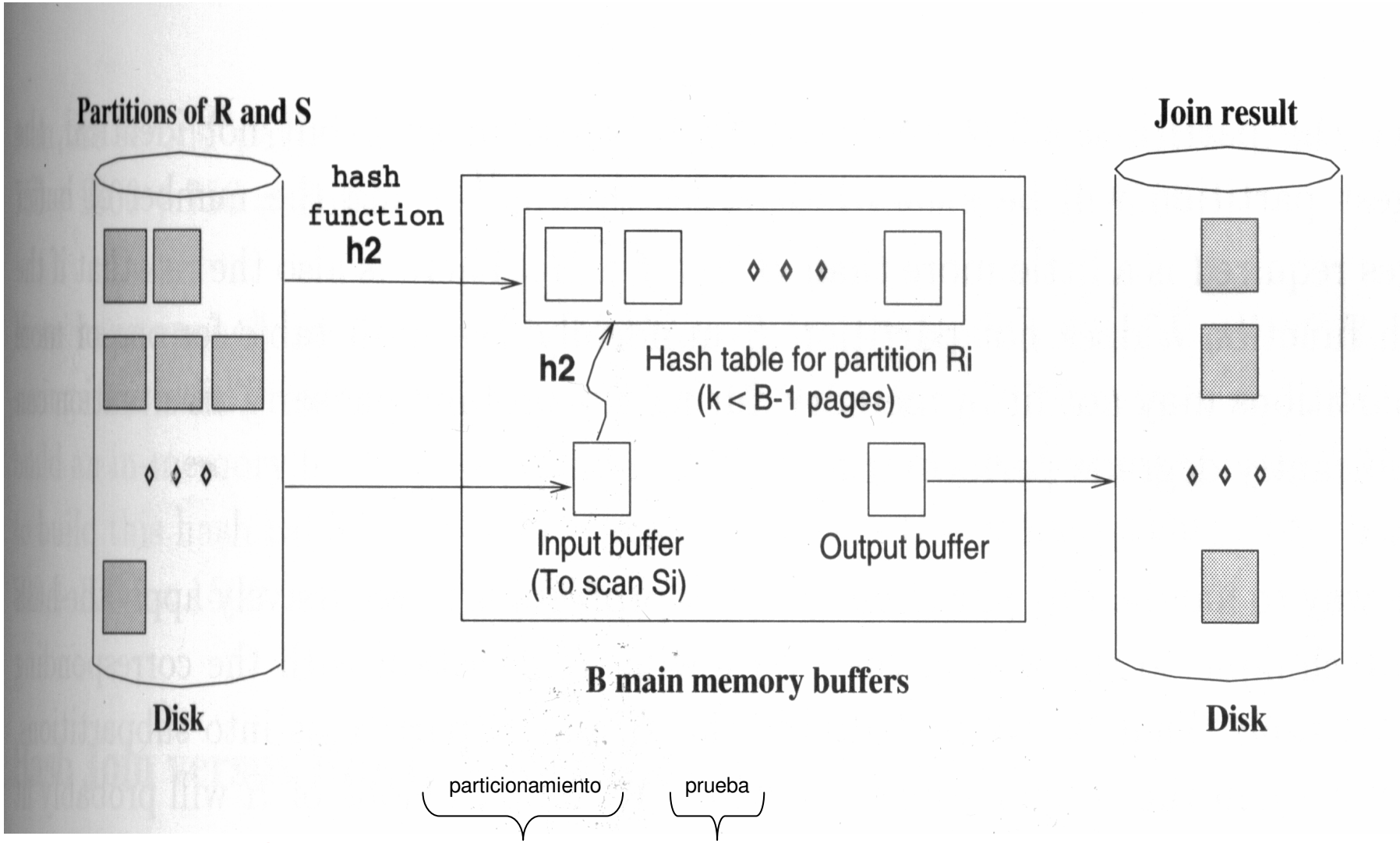
```
foreach tuple s ∈ S  do
  read s and add it to buffer page h(sj);
```

Fase 3: Exploración

```
for l = 1,…, K {
  // Build in-memory hash table for Rl using h2
foreach tuple r ∈ partition Rl  do
        read r and insert into hash table using  h2(ri);

  // Scan Sl and probe for matching Rl tuples
foreach tuple s ∈ partition Sl  do {
       read s and probe table using h2(sj):
       for matching R tuples r, output <r,s> };
clear hash table to prepare for next partition; }
```

Hash Join

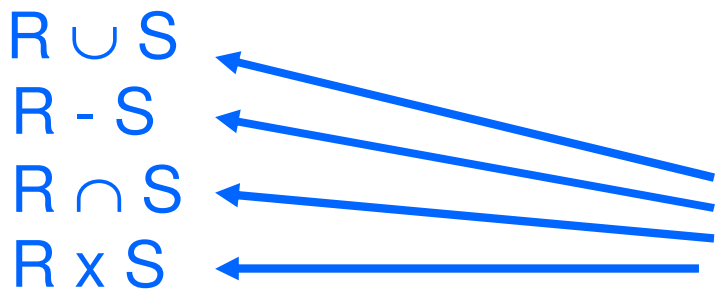


particionamiento prueba

Costo: 2 (M+N) + (M+N) = 3 (M+N)

Tomada de [2]

OPERADORES DE CONJUNTOS (SET)



Se pueden implementar usando *sorting* y *hashing.*

- Con *sorting*, R y S son primeros ordenados y la operación *set* se realiza en el paso de *merge*.
- Con *hashing*, R y S son primero particionados utilizando una función *hash* y la operación *set* se ejecuta cuando son procesadas las distintas particiones.

AGREGADOS (Min, Max, Sum, Avg, Count)

- *Información acerca de las tuplas* de cada relación a la que se le aplica el agregado, se escanea toda la relación. Se llevan variables globales.
- Aplicar un algoritmo de *sort* o de *hashing* (identificar particiones).
- La cláusula GROUP BY requiere dos fases: particionamiento y agregación y a cada grupo se le aplica la función agregado.

Fuentes consultadas:

[1] Prof. Elsa Liliana Tovar.
Notas de clase compiladas entre 1997-2004.

[2] Ramakrishnan Raghu. ,
“Database Management Systems".