# JOINS

Diana Benavides

José Abásolo

# CONCEPTOS PREVIOS

- Variables de costo:
  - M = Número de buffers disponibles en memoria
  - B(R) = Número de bloques de la relación R
  - T(R) = Número de tuplas de la relación R
  - V(a, R) = Número de distintos valores de a en R
  - **Se ignora el costo de escribir resultados finales en disco**

# CONCEPTOS PREVIOS

- “Clustered/unclustered relation”:
  - Clustered relation: Los datos de una misma tabla están en bloques contiguos (menor costo → B(R))
  - Unclustered relation: Los datos de una misma tabla están en bloques dispersos (mayor costo → T(R))

- Índice:
  - Estructura de datos “resumen”

# CONCEPTOS PREVIOS

- Tipos de operaciones:
  - Unarias, una tupla a la vez
  - Unarias, todo el contenido de la tabla
  - **Binarias, todo el contenido de la tabla**

# OPERACIÓN DE “JOIN”

$R \bowtie S$

- Trae las tuplas que hacen “**match**” en las dos relaciones.
- Debe **comparar cada tupla** de una relación con cada tupla de la otra, con el fin de determinar cuáles tuplas estarán en el resultado.
- El resultado dependerá del tamaño de las relaciones (**número de tuplas**).

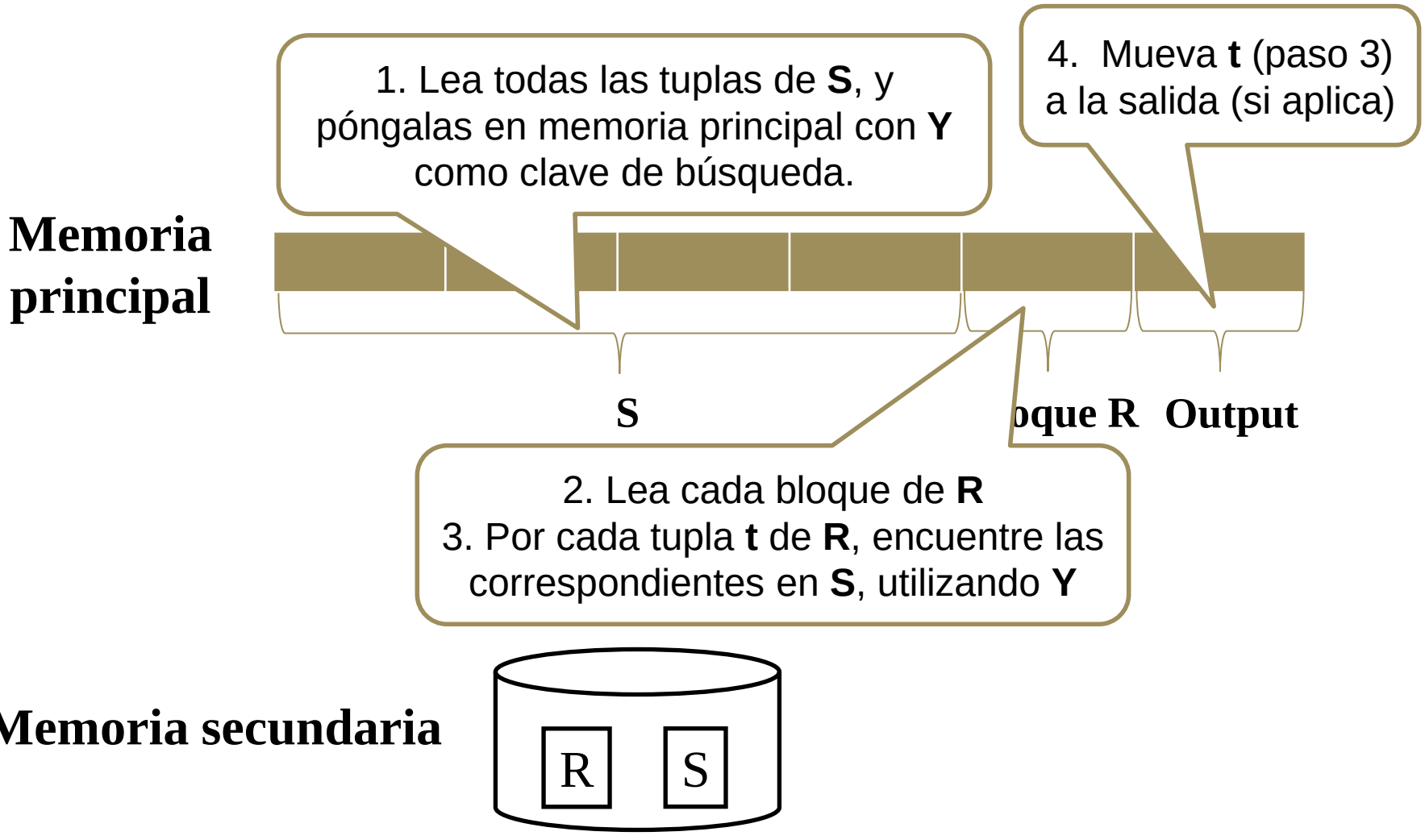
# ALGORITMOS PARA JOIN

- Tipos de algoritmos:
  - Con una sola lectura de datos de disco (**one-pass**)
  - **One-and-a-half-pass**
  - Con dos lecturas de datos de disco (**two-pass**)
    - Basados en ordenamiento (sorting-based)
    - Basados en hash (Hash-based)
    - Basados en índices (Index-based)
  - Con múltiples lecturas de datos de disco (**multiple-pass**)
    - Basados en ordenamiento (sorting-based)
    - Basados en hash (Hash-based)
    - Basados en índices (Index-based)

# ALGORITMOS PARA JOIN: ONE-PASS


- **Requisitos**:
  - min(B(R), B(S)) <= M
  - Aproximadamente M buffers se usan para almacenar la relación más pequeña
  - 1 buffer se usa para leer los bloques de la relación más grande

# ALGORITMOS PARA JOIN: ONE-PASS



# ALGORITMOS PARA JOIN: ONE-PASS

- ¿Cuál es el costo?

B(R) + B(S) operaciones de I/O en disco

# ALGORITMOS PARA JOIN: ONE-PASS

- ¿R ⋈ S, asumiendo que R cabe en memoria?

1. Lea todas las tuplas de R a memoria principal.
2. Por cada tupla t de S, encuentre las tuplas de memoria que hacen match, según Y. Escriba esta tupla a la salida.
3. Marque como “usada” aquellas tuplas de R que hacen match con t.
4. Al terminar S, examine las tuplas de memoria no marcadas como “usadas” y agregue valores nulos. Escriba a la salida.

# ALGORITMOS PARA JOIN:
# ONE (AND A HALF)-PASS JOIN: NESTED LOOP JOINS

- Puede ser utilizado para relaciones de cualquier tamaño → no es necesario que una de las relaciones quepa en memoria principal.

- Dos variantes:
  - Tuple-based nested-loop join
  - Block-based nested-loop join

# ALGORITMOS PARA JOIN:
# ONE (AND A HALF)-PASS JOIN: NESTED LOOP JOINS

## Tuple-based nested loop join

¿Cómo se lo imaginan?

T(R) T(S)

```
FOR each tuple s in S DO
    FOR each tuple r in R DO
        IF r and s join to make a tuple t THEN
            output t;
```

¿Costo asociado?

# ALGORITMOS PARA JOIN:
# ONE (AND A HALF)-PASS JOIN: NESTED LOOP JOINS

**Tuple-based nested loop join**

Ejemplo: B(R) = 1000, T(R) = 10000, B(S) = 500, T(S) = 5000, M = 101

En el peor de los casos:

10000 * 5000 = 50000000

# ALGORITMOS PARA JOIN:
# ONE (AND A HALF)-PASS JOIN: NESTED LOOP JOINS

**Block-based nested loop join**

- Organizar el acceso por bloques, no por tuplas
- Usar el máximo de memoria principal disponible para mantener tantas tuplas de S como sea posible

¿Cómo se lo imaginan?

# ALGORITMOS PARA JOIN:
# ONE (AND A HALF)-PASS JOIN: NESTED LOOP JOINS

## Block-based nested loop join

1. Lea un **conjunto de bloques** de **S** a memoria principal
2. Organícelos en una estructura con **Y** como llave de búsqueda

```
FOR each chunk of M-1 blocks of S DO BEGIN
    read these blocks into main-memory buffers;
    organize their tuples into a search structure whose
        search key is the common attributes of R and S;
    FOR each block b of R DO BEGIN
        read b into main memory;
        FOR each tuple t of b DO    IN
            find the tuples of S        in memory that
                join with t;
            output the join
```

3. Lea un bloque de **R** a memoria principal
4. Por cada tupla **t** del bloque, encuentre las tuplas de **S** que hacen match

5. Mueva **t** (paso 4) a la salida (si aplica)

# ALGORITMOS PARA JOIN:
# ONE (AND A HALF)-PASS JOIN: NESTED LOOP JOINS

## Block-based nested loop join

¿Cuál es el costo?

B(S) / M-1

B(S) (B(S) B(R)) / (M-1)

```
FOR each chunk of M-1 blocks of S DO BEGIN
    read these blocks into main-memo
    organize their tuples into a sea
        search key is the common attrib          of R and S;
    FOR each block b of R DO BEGIN
        read b into main memory;
        FOR each tuple t of b DO BEGIN
            find the tuples of S in main memory that
                join with t;
    END;
END;
```

Aproximadamente, B(S) B(R) / M

# ALGORITMOS PARA JOIN:
# ONE (AND A HALF)-PASS JOIN: NESTED LOOP JOINS

**Block-based nested loop join**

Ejemplo: B(R) = 1000, T(R) = 10000, B(S) = 500, T(S) = 5000, M = 101

$$B(S)\ B(R) / M$$

$$500 * 1000 / 101 = 5500$$

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTING

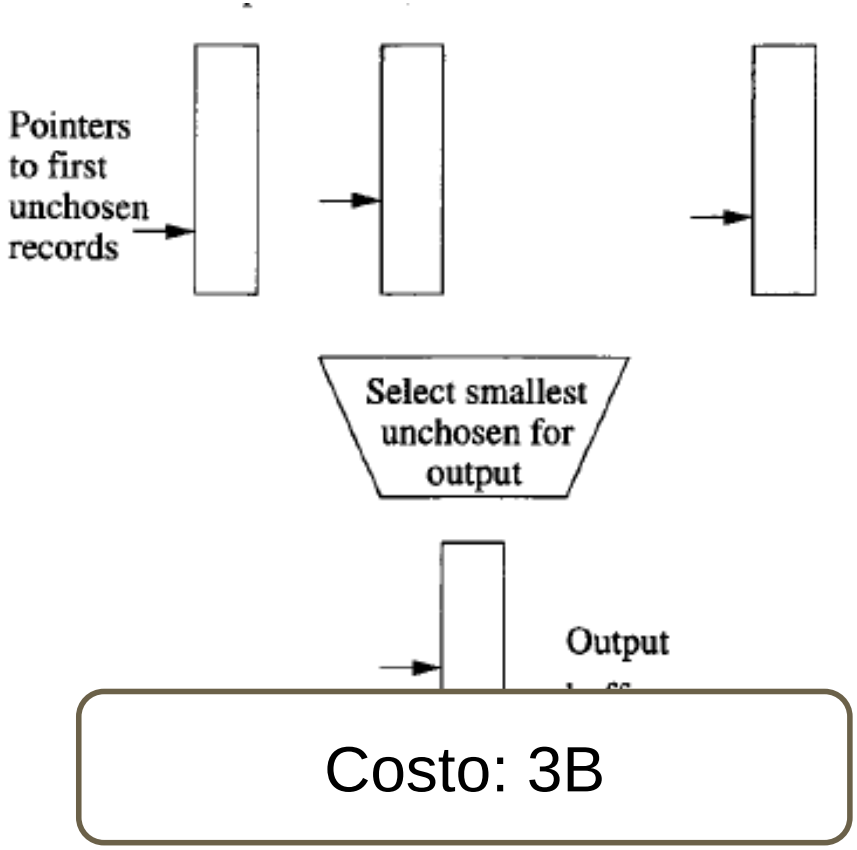
**Two pass multi-way merge sort (2PMMS)**

- Fase 1: Lea M bloques de R en los buffers disponibles de memoria y ordénelos. Escriba cada lista ordenada en memoria secundaria.

- Fase 2: Mezcle las listas ordenadas. **Pueden existir máximo M – 1 listas ordenadas**.

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTING

## Two pass multi-way merge sort (2PMMS)




Costo: 3B

Figure 15.10: Main-memory organization for multiway merging

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTING

**Join con two pass multi-way merge sort (2PMMS)**

1. Ordene R usando 2PPMS.
2. Ordene S usando 2PPMS.
3. Mezcle R y S ordenados.

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTING

## Join con two pass multi-way merge sort (2PMMS)

3. Mezcle R y S ordenados.

4. Agregue a la salida la tupla **t** que hizo match.

**Memoria principal**


**Memoria secundaria**

1. Encuentre el menor valor **y** de **Y** al inicio de cada bloque actual.
2. Si **y** no aparece al inicio del otro bloque, descarte la tupla **t**.
3. Si aparece, encuentre todas las tuplas de las dos relaciones que coinciden en **y**.

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTING

**Join con two pass multi-way merge sort (2PMMS)**

¿Costo asociado?

Paso 1 = 4B(R)
Paso 2 = 4B(S)
Paso 3 = B(R) + (B(S)

**Costo = 5(B(R)) + 5 (B(S)**

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTING

**Join con two pass multi-way merge sort (2PMMS)**

Ejemplo: B(R) = 1000, T(R) = 10000, B(S) = 500, T(S) = 5000, M = 101

Paso 1 = 4(1000) = 1000

Paso 2 = 4(500) = 2000

Paso 3 = 1500

**Costo = 7500**

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTING

**Join con two pass multi-way merge sort (2PMMS)**

¿Cómo mejorarlo?

Combinar la fase de ordenamiento de 2PPMS en B(R) y B(S) con el join:

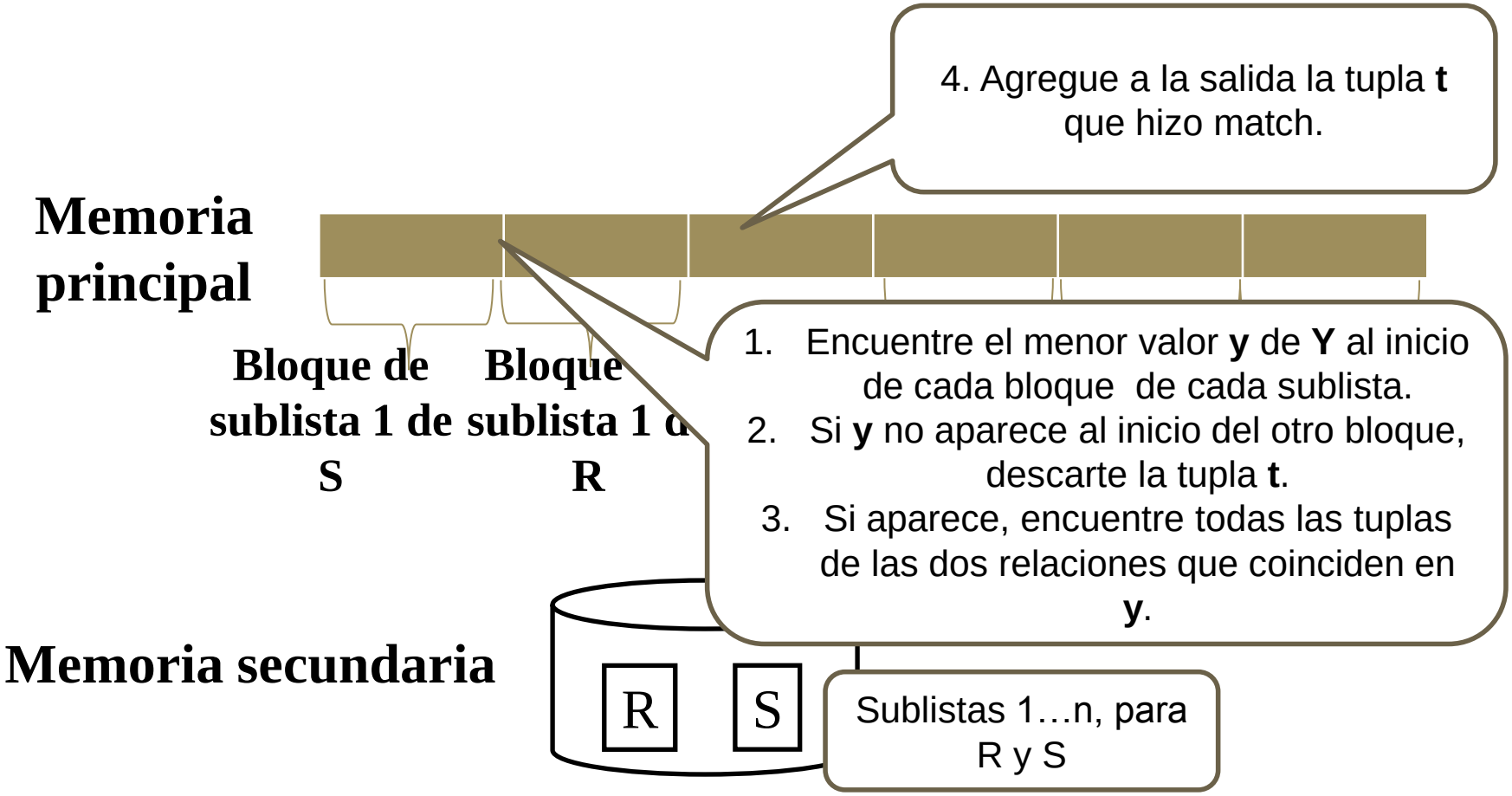
Paso 1 = Traer R a memoria principal, ordenar y crear sublistas ordenadas de R en memoria secundaria

Paso 3 = Traer S a memoria principal, ordenar y crear sublistas ordenadas de S en memoria secundaria

Paso 3 = Hacer join de sublistas ordenadas de R y S

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTING

## Join con two pass multi-way merge sort (2PMMS)



# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTING

**Join con two pass multi-way merge sort (2PMMS)**

Ejemplo: B(R) = 1000, T(R) = 10000, B(S) = 500, T(S) = 5000, M = 101

Paso 1 = 2B(R) = 2000

Paso 2 = 2B(S) = 1000

Paso 3 = B(R) + B(S) = 1500

**Costo total = 3B(R) + 3B(S) = 4500**

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON HASHING

**Hash-Join**

Fase 1: Asignar las tuplas de R y S a “buckets”.

1. Cada buffer de memoria principal corresponde a un “bucket” (esto es, cada “bucket” tendrá el mismo tamaño y habrán M-1 “buckets”)
2. El último buffer de memoria principal contiene el bloque de R (ó S)

Fase 2: Hacer el join utilizando los “buckets” Ri, Si hasta Rn, Sn.,

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON HASHING

## Hash-Join: Fase 1

```
initialize M-1 buckets using M-1 empty
FOR each block b of relation R DO
    read block b into the Mth buffer;
    FOR each tuple t in b DO BEGIN
        IF the buffer for bucket h(t) has no room for t THEN
            BEGIN
                copy the buffer to disk;
                initialize a new empty block in that buffer;
            END;
        copy t to the buffer for bucket h(t);
    END;
END;
FOR each bucket DO
    IF the buffer for this bucket is
        write the buffer to disk;
```

Figure 15.12: Partitioning a relation $R$ into $M-1$ buckets

Se aplica **h(t)** sobre cada tupla de R (ó S), y se asigna a un “bucket” - un buffer de memoria principal.

**h(t)** debería establecerse sobre el atributo de join.

Si el buffer está lleno, se escribe a memoria secundaria

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON HASHING

## Hash-Join: Fase 2

Inicia con los buckets Ri,…,Rn y Si,…,Sn en memoria secundaria, que incluyen los atributos de join como “hash key”:

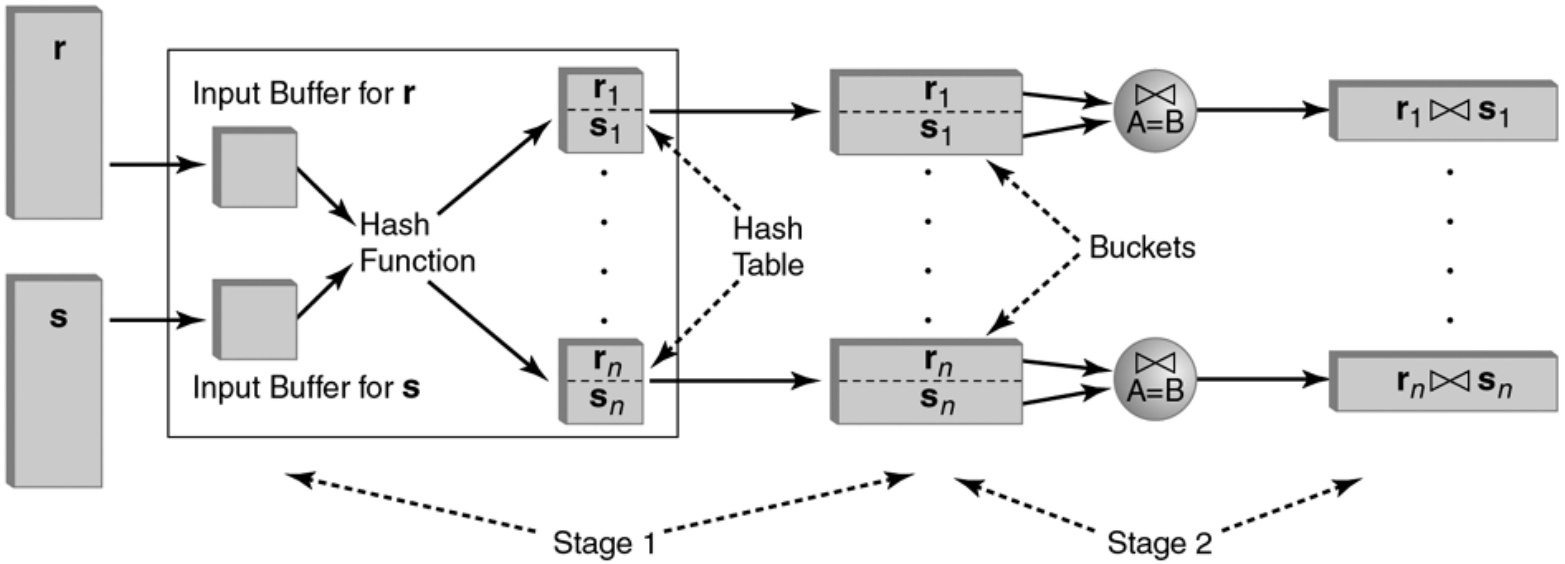
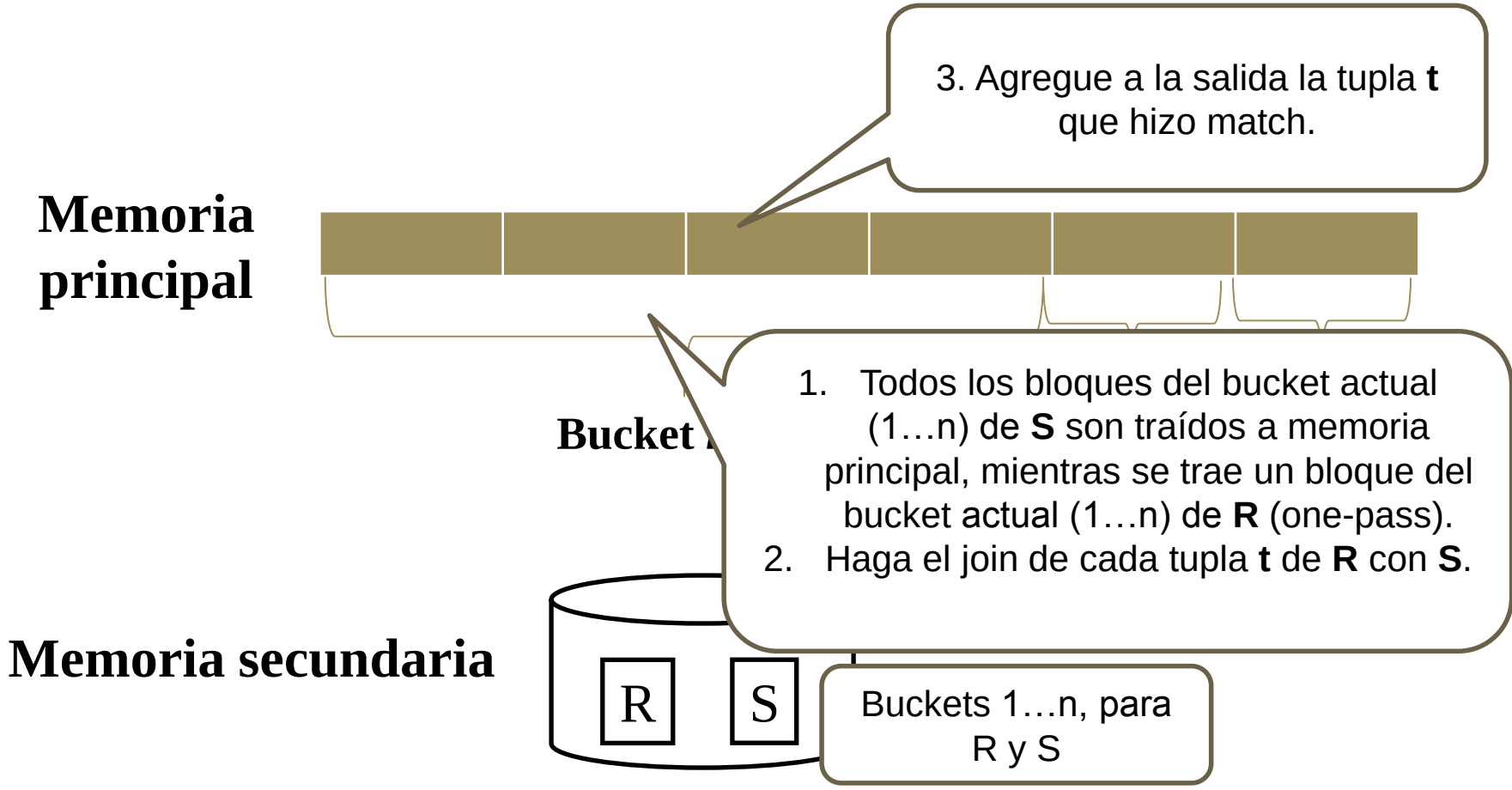



**FIGURE 10.8** Hash join.

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON HASHING

## Hash-Join: Fase 2



# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON HASHING

**Hash-Join: Fase 2**

Ejemplo: B(R) = 1000, T(R) = 10000, B(S) = 500, T(S) = 5000, M = 101

Fase 1 = 2B(R) + 2B(S) = 3000

Fase 2 = B(R) + B(S) = 1500

**Costo total = 3B(R) + 3B(S) = 4500**

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON INDEX

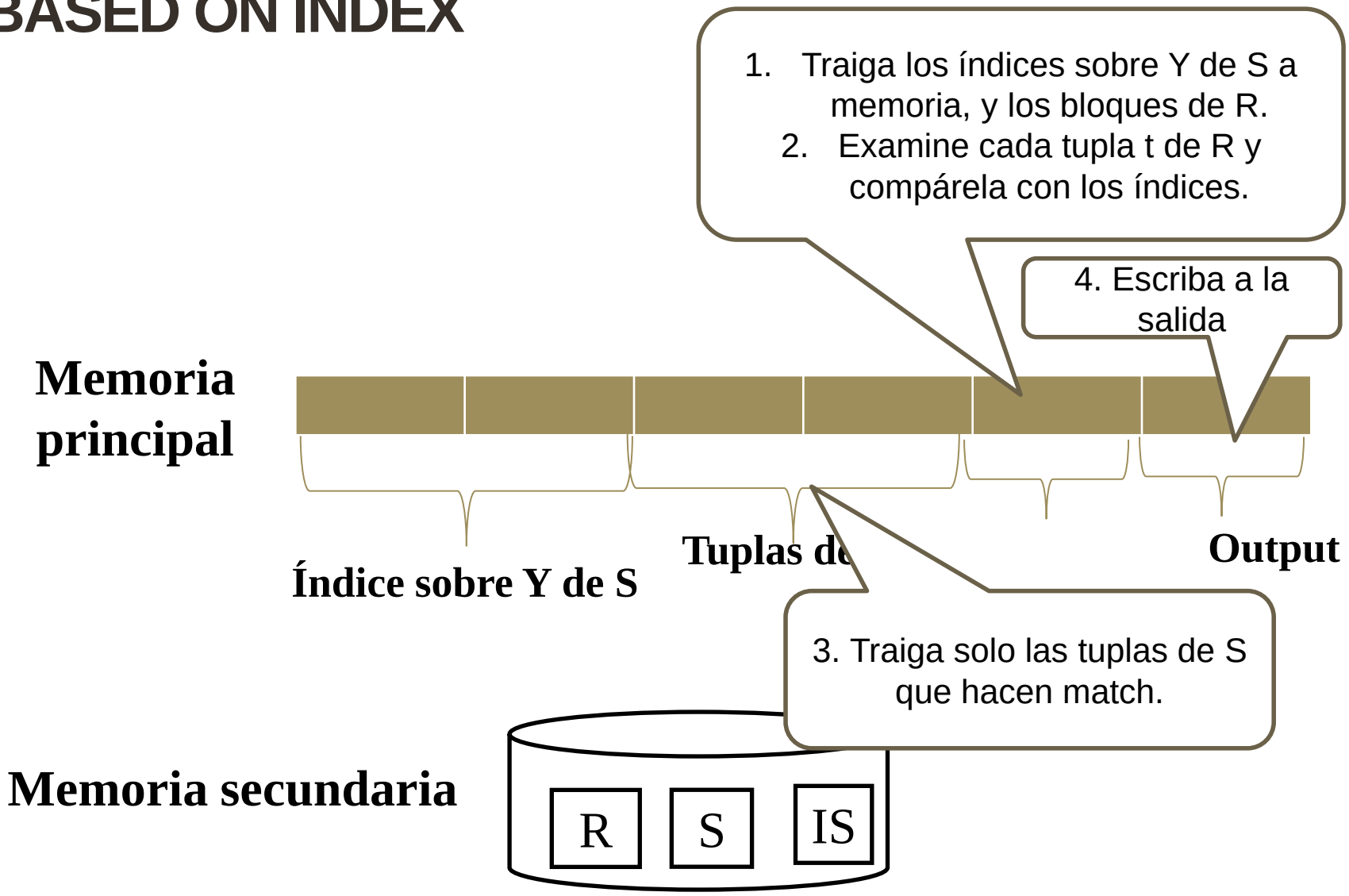
Dada la existencia de un índice sobre el atributo **Y** de **S**:

1. Traer a memoria las tuplas de cada bloque B de R.
2. Traer a memoria el índice sobre el atributo Y de S.
3. Comparar cada tupla t de R con el índice; traiga solo aquellas tuplas que hacen match.

¿Costo asociado?

B(R) + T(R) * T(S) / V(S, Y)
ó B(R) + T(R) * B(S) / V(S, Y)

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON INDEX



# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON INDEX

Ejemplo: B(R) = 1000, T(R) = 10000, B(S) = 500, T(S) = 5000, M = 101, V(S, Y) = 100

1 = B(R) = 1000

2 = Se ignora el costo de traer el índice a memoria principal

3 = T(R) **B(S)** / V(S, Y) = 10000 * 500 / 100 = 50000

**Costo total (índice clustered) = 51000**

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTED INDEX

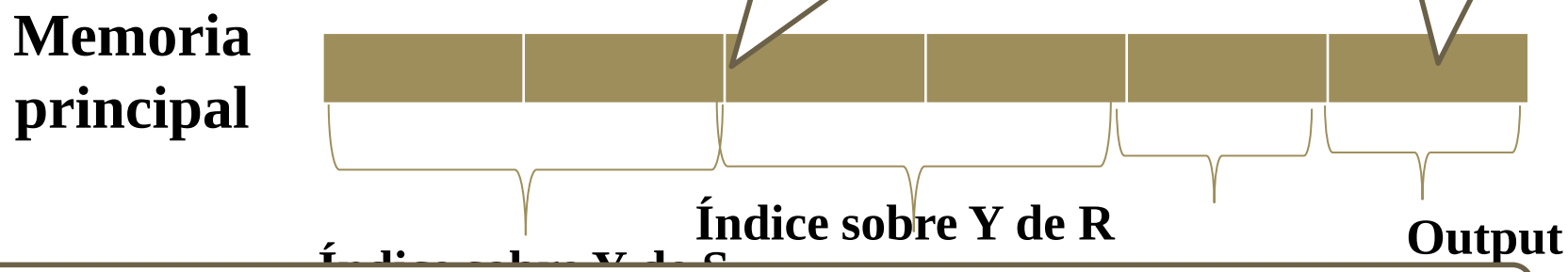
Dada la existencia de un índice para el atributo **Y** en las relaciones **R** y **S**:

1. Traer a memoria los índices sobre **Y** de **R** y de **S**.
2. Comparar uno a uno los valores de los índices sobre Y.
3. Si los índices hacen match, traer las tuplas correspondientes de memoria secundaria.

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTED INDEX

1. Compare los índices sobre cada relación.
2. Si hacen match, traiga las tuplas correspondientes de R y S. haga el join y forme la tupla t.

3. Escriba t a la salida




¿Costo asociado?

B(R) + B(S)

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTED INDEX

Ejemplo: B(R) = 1000, T(R) = 10000, B(S) = 500, T(S) = 5000, M = 101

1 = Se ignora el costo de traer el índice a memoria principal

2 = Se ignora el costo de traer el índice a memoria principal

3 = B(R) + B(S) = 1500

# ALGORITMOS PARA JOIN: TWO-PASS JOIN BASED ON SORTED INDEX

Importante tener en cuenta el **costo del índice** mismo:

- Si se tiene en forma de árbol B+, el costo de accederlo se traduce a:

**n** operaciones de I/O

Donde **n** es el número de niveles del árbol, generalmente no mayor a 3

(ej. con 255 apuntadores en la raíz se llega a 65.025 nodos en segundo nivel y 16.6 nodos en tercer nivel).

# COMPARATIVO ALGORITMOS

- **One-Pass:** No crece linealmente con el número de tuplas. Sin embargo, exige que una de las relaciones quepa totalmente en memoria.
- **Tuple-based nested-loop join:** Crece linealmente con el número de tuplas.
- **Block-based nested-loop join:** Apropiado si hay una tabla considerablemente más grande que la otra.
- **Sort-based:**
- **Hash-based:**
- **Index-based:** Si existe índice sobre Y de una de las relaciones no es tan efectivo. Sin embargo, si existe índice sobre Y en las dos es el más efectivo de todos.

# EJERCICIO EN CLASE

**Considere el siguiente par de tablas:**

```
StarsIn(movieTitle, movieYear, name)
MovieStar(name, address, gender, birthdate)
```

En donde la tabla MovieStar, que nombraremos **S**, contiene datos de estrellas de cine, mientras que la tabla StarsIn, que nombraremos **R**, contiene las películas en las que ha aparecido una estrella de cine.

Se requiere hacer el join entre estas dos tablas. Suponga que:

- B(R) = 5.000
- B(S) = 800
- T(R) = 100.000
- T(S) = 20.000
- M = 50

Suponga además, que las tablas **R** y **S** contienen índices sobre el atributo **"name"**.

**Determine y compare el costo de hacer el join utilizando:**

a) El algoritmo de join en una y media pasada, basado en bloques (nested-loop-join).
b) El algoritmo de join en dos pasadas, basado en ordenamiento (sort-based).
c) El algoritmo de join en dos pasadas, basado en hash (hash-based).
d) El algoritmo de join en dos pasadas, basado en índice (index-based).