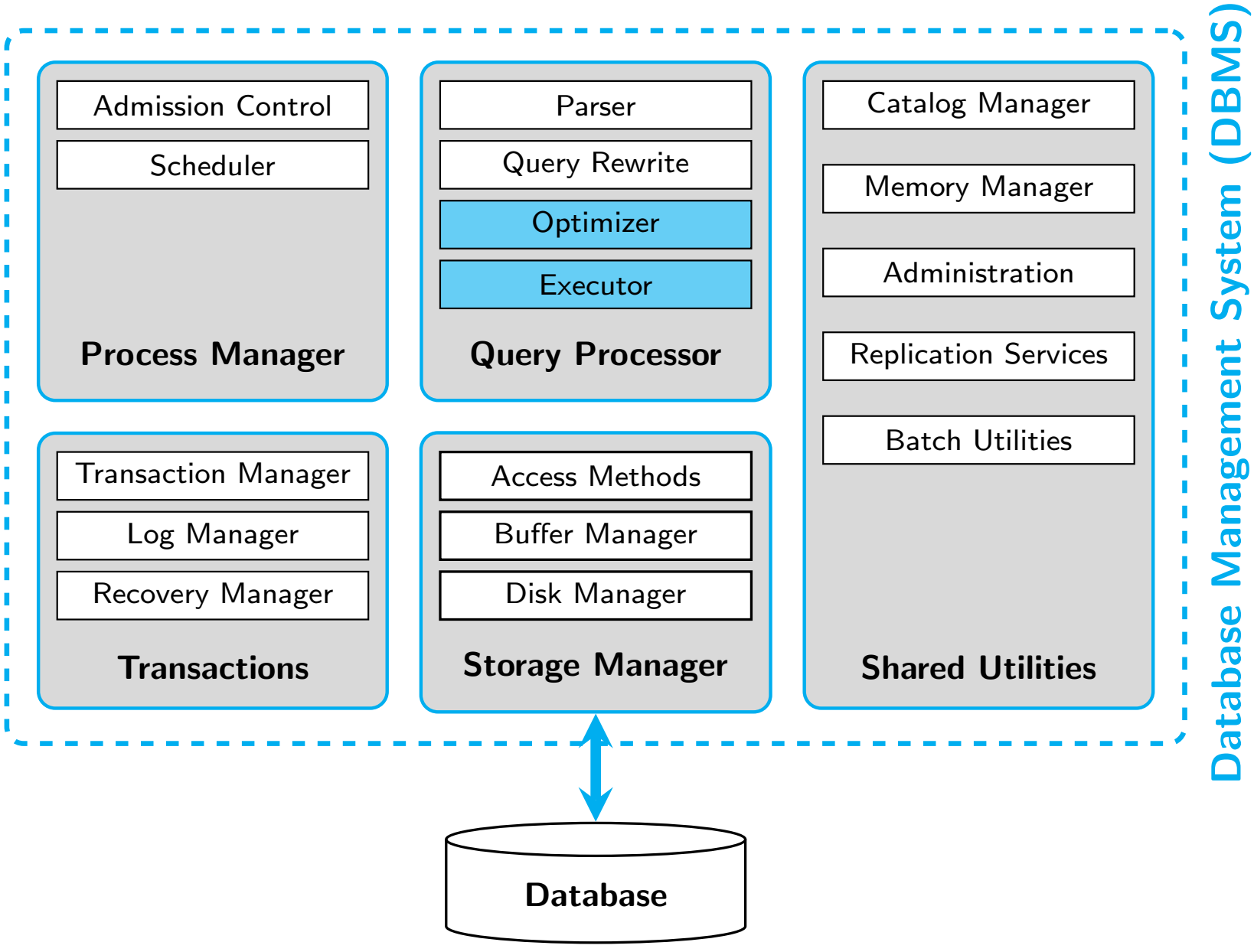
Implementación de Joins

Clase 13

IIC 3413

Prof. Cristian Riveros

Implementación de operadores relacionales



Implementación de operadores y sus variantes

Operador físico:

- Cada operador físico implementa un operador relacional (lógico).
- Implementado para desempeñarse bien en una tarea específica.

Cada **variante** aprovecha propiedades físicas de los datos:

- Presencia o ausencia de índices.
- Orden del input.
- Tamaño del input.
- Cantidad de elementos distintos.
- Espacio disponible en memoria.

Recordatorio: parámetros para medir costo

Definición

Durante esta clase denotamos con R o S una "consulta" relacional:

- una relación, o
- el resultado de una consulta.

Ejemplo

R := relación `Players`.

S := consulta $\pi_{Id}(\texttt{Matches})$.

Recordatorio: parámetros para medir costo

Parámetros de interés:

$\text{cost}(R)$:	costo (en I/O) para computar R.
$\text{pages}(R)$:	cantidad de páginas necesarias para almacenar R.
$\lvert R\rvert$:	cantidad de tuplas/records en R.
$\text{rsize}(R)$:	tamaño de una tupla/record (promedio) en R.
$\text{sel}_p(R)$:	fracción de tuplas/records en R que satisfacen p.
$\text{distinct}(R)$:	cantidad de elementos distintos en R.
$\text{distinct}_a(R)$:	cantidad de elementos distintos en el campo $R.a$.
$\lvert\text{page}\rvert$:	tamaño/espacio de una página*.

$$0 \leq \text{sel}_p(R) = \frac{|\sigma_p(R)|}{|R|} \leq 1$$

Recordatorio: durante esta clase, estudiaremos los siguientes parámetros...

cost(R):	costo (en I/O) para computar R.
pages(R):	número de páginas necesarias para almacenar R.
$\|R\|$:	número de tuplas/records en R.
rsize(R):	tamaño de una tupla/record (promedio) en R.

Los siguientes parámetros:

distinct(R):	cantidad de elementos distintos en R.
distinct$_a$(R):	cantidad de elementos distintos en el campo $R.a$.
sel$_p$(R):	fracción de tuplas/records en R que satisfacen p.

los **estimaremos** en otra clase (por ahora los supondremos dados).

Recordatorio: operadores y tamaño del input

Suposición para todos los operadores físicos siguientes:

> *Se asume que para todos los operadores unarios* $*(R)$ *o binarios* $R * S$*, ambas relaciones* R *y* S *son de tamaño* **mayor** *a la disponible en el* **buffer**.

Si R o S pueden ser almacenadas en el **buffer** (memoria), entonces:

$$\begin{aligned} \text{cost}(*(R)) &= \text{cost}(R) \\ \text{cost}(R * S) &= \text{cost}(R) + \text{cost}(S) \end{aligned}$$

Operadores físicos relacionales

Selección (σ)

Proyección (π)

Join ($\bowtie$)

Intersección ($\cap$)

Unión ($\cup$)

Elim. duplicados (δ)

Sorting (τ)

GroupBy (γ)

Operadores físicos relacionales

Selección (σ)

Unión ($\cup$)

Proyección (π)

Elim. duplicados (δ)

Join ($\bowtie$)

Sorting (τ)

Intersección ($\cap$)

GroupBy (γ)

Recoratorio de tipos de Joins ($\bowtie$)

- Producto cruz: $R_1 \times R_2$
- p-join: $R_1 \bowtie_p R_2 = \sigma_p(R_1 \times R_2)$
  - p es una combinación booleana $(\wedge, \vee)$ de terminos:

$$\begin{array}{rcl} \mathit{attributo}_1 & \text{op} & \mathit{attributo}_2 \\ \mathit{attributo} & \text{op} & \mathit{constante} \end{array}$$

  con op $\in \{=, \leq, \geq, <, >\}$.
- Equi-join: $R_1 \bowtie_\phi R_2 = \sigma_\phi(R_1 \times R_2)$
  - ϕ solo contine igualdades.
- Natural-join: $R_1 \bowtie R_2 = \sigma_\phi(R_1 \times R_2)$
  - $\phi = \bigwedge_{a \in \text{att}(R_1) \cap \text{att}(R_2)} R_1.a = R_2.a$.

Nos concentraremos de ahora en adelante en p-joins ($\bowtie_p$)

Joins físicos para $R_1 \bowtie_p R_2$

1. Nested loops join.
2. Block nested loops join.
3. Index nested loops join.
4. Sort-merge join.
5. Hash join.

Joins ($\bowtie_p$): Nested loops join

Algoritmo

input: Operadores R y S, y un predicado p.

```
R.open()
foreach r = R.next() do
    S.open()
    foreach s = S.next() do
        if (r, s) satisfacen p then
            Incluimos (r, s) en el resultado
    S.close()
R.close()
```

Implementación directa de join basada en "for".

Joins ($\bowtie_p$): Nested loops join

Algoritmo

input: Operadores R y S, y un predicado p.

```
open()
    R.open()
    S.open()
    r := R.next()

close()
    R.close()
    S.close()
```

```
next()
    while r ≠ NULL do
        s := S.next()
        if s = NULL then
            S.close()
            r := R.next()
            S.open()
        else if (r, s) satisfacen p then
            return (r, s)
    return NULL
```

Joins ($\bowtie_p$): Nested loops join

Costo y parámetros de **nested loops join**:

$$
\begin{aligned}
\text{cost}(R \bowtie_p S) &= \text{cost}(R) + |R| \cdot \text{cost}(S) \\
\text{pages}(R \bowtie_p S) &= \text{sel}_p(R \times S) \cdot \text{pages}(R) \cdot \text{pages}(S) \\
|R \bowtie_p S| &= \text{sel}_p(R \times S) \cdot |R| \cdot |S| \\
\text{rsize}(R \bowtie_p S) &= \text{rsize}(R) + \text{rsize}(S)
\end{aligned}
$$

¿qué tan eficiente es este algoritmo?

Joins ($\bowtie_p$): Nested loops join

Ejemplo

Considere:

- R y S son tablas de $16MB$.
- Cada página tiene 8KB.
- Tuplas son de 300 Bytes.

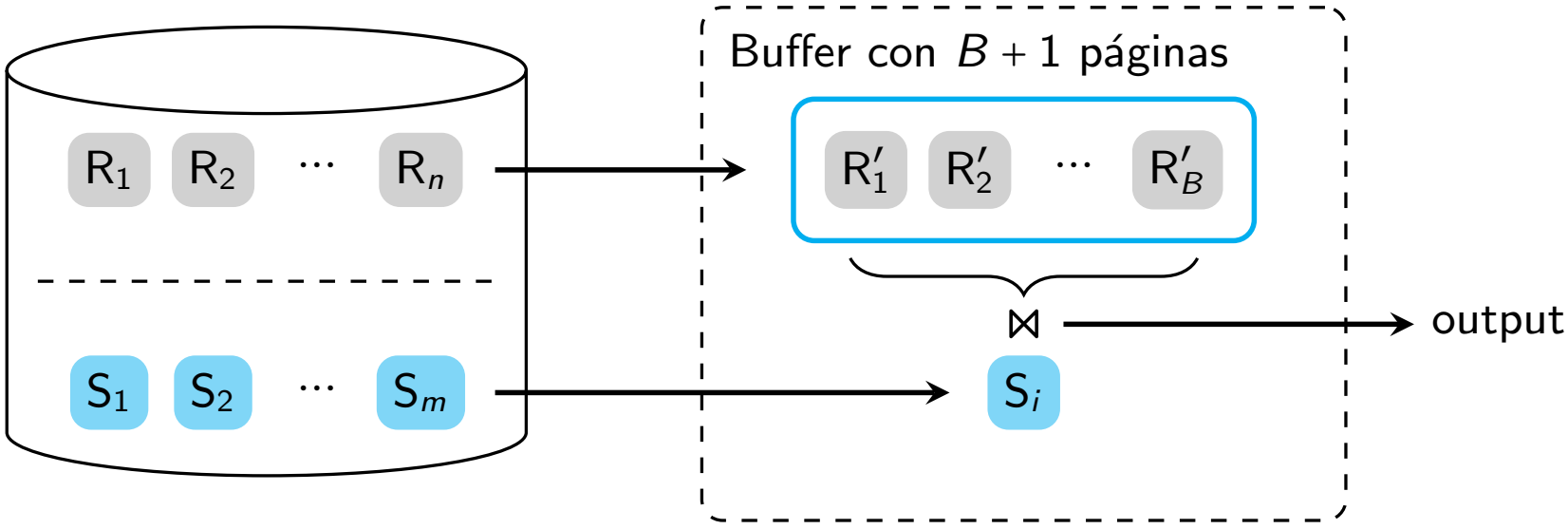
Esto significa que:

- Cada relación tiene 2048 páginas.
- Cada relación tiene $\approx$ 55.000 tuplas.

Si consideramos que cada I/O toma $\approx$ 0.1ms (mínimo):

3.1 horas!!!

Joins ($\bowtie_p$): Block nested loops join



Joins ($\bowtie_p$): Block nested loops join

Algoritmo

input: Operadores R y S, un predicado p, y un **Buffer** con $B+1$ páginas.

```
open()
    R.open()
    fillBuffer()

close()
    R.close()
    S.close()
```

```
fillBuffer()
    Buff := ∅
    r := R.next()
    while r ≠ NULL do
        Buff := Buff ∪ {r}
        if Buff.isFull() then
            break
        r := R.next()
    S.open()
    s := S.next()
```

Joins ($\bowtie_p$): Block nested loops join

Algoritmo

input: Operadores R y S, un predicado p, y un **Buffer** con $B+1$ páginas.

```
next()
    while Buff ≠ ∅ do
        while s ≠ NULL do
            r := Buffer.next()
            if r = NULL then
                Buff.reset()
                s := S.next()
            else if (r, s) ⊨ p then
                return (r, s)
        fillBuffer()
    return NULL
```

Joins ($\bowtie_p$): Block nested loops join

Costo de **Block nested loops join**:

$$\begin{aligned} \text{cost}(R \bowtie_p S) &= \text{cost}(R) + \frac{|R|}{|\text{page}| \cdot B} \cdot \text{cost}(S) \\ &\Downarrow \\ \text{cost}(R \bowtie_p S) &= \text{cost}(R) + \frac{\text{pages}(R)}{B} \cdot \text{cost}(S) \end{aligned}$$

¿es este algoritmo de **join** más eficiente?

Joins ($\bowtie_p$): Block nested loops join

Ejemplo

Considere:

- R y S son tablas de $16MB$.
- Cada página tiene 8KB.
- Un **buffer** de $1MB$.

Esto significa que:

- Cada relación tiene 2048 páginas.
- Tenemos 128 páginas de buffer.

Si consideramos que cada I/O toma $\approx$ 0.1ms (mínimo):

3.4 segundos!!!

... ¿una buena optimización?.

Joins ($\bowtie_p$): Block nested loops join

Ejemplo

Considere un caso mas extremo (o actual):

- R y S son tablas de $250GB$.
- Cada página tiene 8KB.
- Un **buffer** de $32GB$.

Esto significa que:

- Cada relación tiene 32.768.000 páginas.
- Tenemos 4.194.304 páginas de buffer.

Si consideramos que cada I/O toma $\approx$ 0.1ms (mínimo):

8 horas!!!

Joins ($\bowtie_p$): Block nested loops join

$$\text{cost}(R \bowtie_p S) \quad = \quad \text{cost}(R) + \frac{\text{pages}(R) \cdot \text{cost}(S)}{B}$$

¿qué tiene de extraño esta formula?

- $\text{cost}(R \bowtie_p S) \neq \text{cost}(S \bowtie_p R)$
- R y S como relaciones (no operadores):

$$\text{pages}(R) + \frac{\text{pages}(R) \cdot \text{pages}(S)}{B} \quad \neq \quad \text{pages}(S) + \frac{\text{pages}(R) \cdot \text{pages}(S)}{B}$$

Regla: la relación más pequeña tiene que ir la exterior.

Evitar hacer el producto cruz

¿existe un mejor algoritmo alternativo
a enumerar todas las tuplas de un producto cruz de R y S?

1. Usar índices.
2. Hacer **cluster** de tuplas.
   - con sorting.
   - con hashing.

Joins ($\bowtie_p$): Index nested loops join

Para dos operaciones R y S:

- Suponga que contamos con un índice I sobre S.
- I hace match con el predicado p.

Entonces, podemos usar el índice I para calcular $R \bowtie_p S$:

1. Iteramos por cada tupla/record t en R.
2. Usamos I para buscar las tuplas en S que hacen join con t.

Joins ($\bowtie_p$): Index nested loops join

Algoritmo

input: Operadores R y S, y un predicado p.

```
open()
    R.open()
    Index(S).open()
    r := R.next()
    l := S.index(r, p)

close()
    R.close()
    Index(S).close()
```

```
next()
    while r ≠ NULL do
        s := l.next()
        if s = NULL then
            r := R.next()
            l := S.index(r, p)
        else
            return (r, s)
    return NULL
```

Joins ($\bowtie_p$): Index nested loops join

Costo de **Index nested loops join**:

Clustered Index

$$\text{cost}(R \bowtie_p S) = \text{cost}(R) + |R| \cdot \left\lceil \text{pages}(S) \cdot \text{sel}_{R \bowtie_p}(S) \right\rceil$$

Unclustered Index

$$\text{cost}(R \bowtie_p S) = \text{cost}(R) + |R| \cdot \left\lceil |S| \cdot \text{sel}_{R \bowtie_p}(S) \right\rceil$$

En este costo, **NO** estamos considerando el **costo del índice**.

Joins ($\bowtie_p$): Index nested loops join

Ejemplo

Considere:

- R y S son tablas de $16MB$.
- Cada página tiene 8KB.
- Tuplas de 300 Bytes (27 tuplas por página).
- S tiene un clustered index y la selectividad es $0,0001$.

Si consideramos que cada I/O toma $\approx$ 0.1ms (mínimo):

5.7 segundos!!!

...peor que BNLJ!!

¿por qué el costo de este algoritmo pareciera tan malo?

Reflexiones sobre **index** nested loops join

Clustered Index

$$\text{cost}(R \bowtie_p S) \quad = \quad \text{cost}(R) + |R| \cdot \left\lceil \text{pages}(S) \cdot \text{sel}_{R \bowtie_p}(S) \right\rceil$$

Unclustered Index

$$\text{cost}(R \bowtie_p S) \quad = \quad \text{cost}(R) + |R| \cdot \left\lceil |S| \cdot \text{sel}_{R \bowtie_p}(S) \right\rceil$$

Utilizar INLJ, cuando:

- Selectividad es muy alta ($\text{sel}_{R \bowtie_p}(S) \approx 0$).
- Relación (consulta) R tiene pocas tuplas.

Joins físicos para calcular $R_1 \bowtie_p R_2$

1. Nested loops join. ✓
2. Block nested loops join. ✓
3. Index nested loops join. ✓
4. Merge join.
5. Hash join.

Joins ($\bowtie$): Sort-merge join

Join de dos tablas es mucho mas fácil cuando están ordenadas.

A	B
Juan	1
Pedro	2
Diego	2
Andrés	2
Bastián	4

$\bowtie_{B=C}$

C	D
1	BD
2	Lógica
2	Autómatas
3	Ing. Software

(esto es cierto para **equi-joins**)

¿cuál es la diferencia entre
un merge y un join de dos runs?

Joins ($\bowtie_{A=B}$): Sort-merge join

Algoritmo

input: Operadores R y S, un predicado $A = B$.

```
open()
    R' := merge-sort(R)
    S' := merge-sort(S)
    r := R'.next()
    s := S'.next()
    fillBuffer()

close()
    R'.close()
    S'.close()
```

```
fillBuffer()
    while r(A) ≠ s(B) do
        if r < s then
            r := R'.next()
        else
            s := S'.next()
    Buff := ∅
    while r(A) = s(B) do
        Buff := Buff ∪ {s}
        s := S'.next()
```

Joins ($\bowtie_{A=B}$): Sort-merge join

Algoritmo

input: Operadores R y S, un predicado $A = B$.

```
next()
    while r ≠ NULL do
        t := Buffer.next()
        if t = NULL then
            r' := r
            r := R'.next()
            if r(A) = r'(A) then
                Buff.reset()
            else
                fillBuffer()
        else
            return (r, s)
    return NULL
```

Joins ($\bowtie_{A=B}$): Sort-merge join

Costo de **Sort-merge join**:

Formula general

$$\text{cost}(R \bowtie_{A=B} S) = \text{cost}(R) + \text{cost}(S) + 2 \cdot (\text{pages}(R) + \text{pages}(S))$$

Relaciones

$$\text{cost}(R \bowtie_{A=B} S) = 3 \cdot (\text{pages}(R) + \text{pages}(S))$$

¿es este algoritmo de **join** más eficiente?

Joins ($\bowtie_{A=B}$): Sort-merge join

Ejemplo

Considere:

- R y S son tablas de $16MB$.
- Cada página tiene 8KB.
- Cada relación tiene 2048 páginas.

Si consideramos que cada I/O toma $\approx$ 0.1ms (mínimo):

$$1,2 \text{ segundos!!!}$$

$\ldots\frac{1}{3}$ menos que BNLJ!!

Joins ($\bowtie_{A=B}$): Sort-merge join

Ejemplo

Considere:

- R y S son tablas de $250GB$.
- Cada página tiene 8KB.
- Cada relación tiene 32.768.000 páginas.

Si consideramos que cada I/O toma $\approx$ 0.1ms (mínimo):

$5,4$ horas!!!

...mejor que BNLJ, pero todavía "lento" !!

¿es posible hacer join de estas big tablas en segundos?

Reflexiones sobre Sort-merge join

Formula general

$$\text{cost}(R \bowtie_{A=B} S) \quad = \quad \text{cost}(R) + \text{cost}(S) + 2 \cdot (\text{pages}(R) + \text{pages}(S))$$

Relaciones

$$\text{cost}(R \bowtie_{A=B} S) \quad = \quad 3 \cdot (\text{pages}(R) + \text{pages}(S))$$

- Operador físico preferido para $\bowtie_{A=B}$ (sin suposiciones).
- Más rápido aún si ambas relaciones ya están ordenadas.

¿cuándo podría suceder que ambas relaciones estén ordenadas?

Joins ($\bowtie_{A=B}$): Sort-merge join + B+-tree

Costo:

$$\text{cost}(R \bowtie_{A=B} S) \quad = \quad \text{pages}(R) + \text{pages}(S)$$

Ejemplo

Considere:

- R y S son tablas de $250GB$.
- Cada página tiene 8KB.
- R y S tienen un clustered B+-tree en A y B.

Si consideramos que cada I/O toma $\approx$ 0.1ms (mínimo):

$$1,8 \text{ horas!!!}$$

Lo menor posible!!!

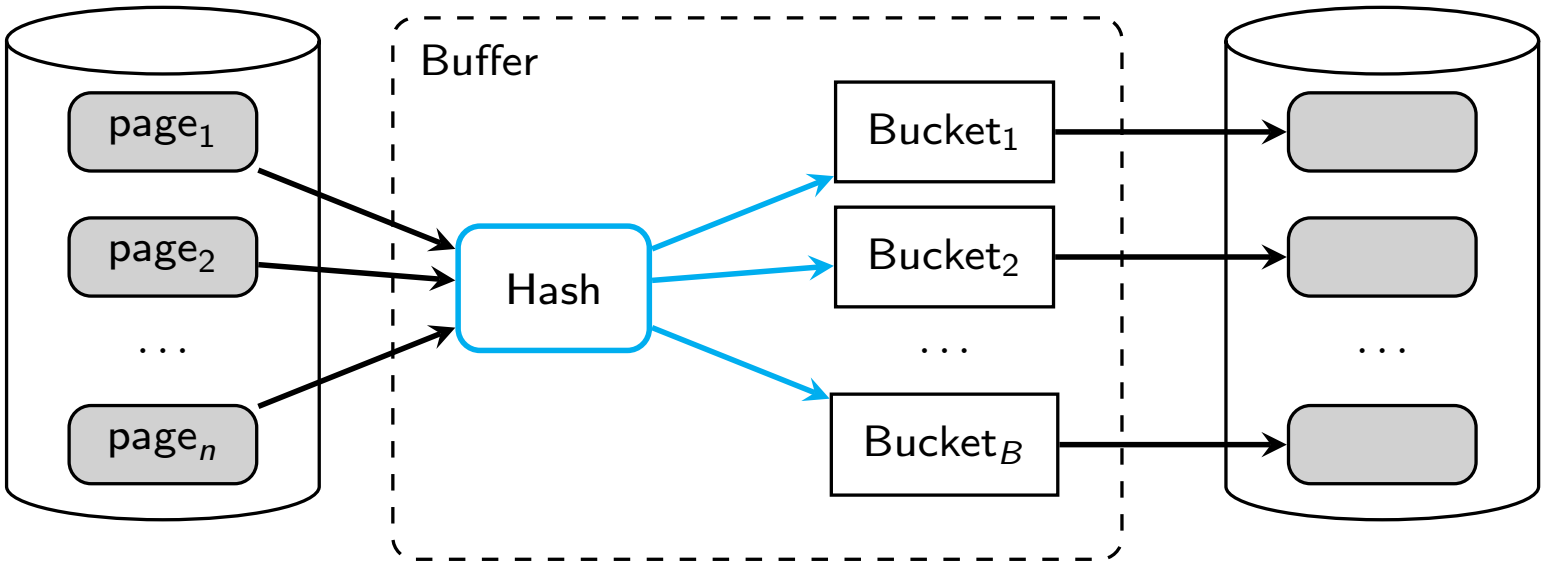
Joins físicos para calcular $R_1 \bowtie_p R_2$

1. Nested loops join. ✓
2. Block nested loops join. ✓
3. Index nested loops join. ✓
4. Merge join. ✓
5. Hash join.

Joins ($\bowtie_p$): Hash join

Misma idea que eliminación de duplicados:

1. Fase de partición.



Joins ($\bowtie_p$): Hash join

Misma idea que eliminación de duplicados:

1. Fase de partición.
   - Cada página p del operador R se lee a memoria.
   - Por cada $t \in p$ se computa $h(t)$ y se envía al bucket $R_{h(t)}$.
   - Cada página p del operador S se lee a memoria.
   - Por cada $t \in p$ se computa $h(t)$ y se envía al bucket $S_{h(t)}$.

   (si un bucket esta completo, se vacía y materializa en disco)

Joins ($\bowtie_p$): Hash join

Misma idea que eliminación de duplicados:

1. Fase de partición.

2. Fase de join de inter-particiones.

   - Join $R_i \bowtie_p S_i$ entre los bucket R_i y S_i, recursivamente.
   - (Idealmente) R_i o S_i sea almacenado completamente en el buffer.

¿cuántos buckets son suficientes
para tener solo una **fase de partición**?

Joins ($\bowtie_{A=B}$): Hash join

Costo de **Hash join**:

Formula general

$$\text{cost}(R \bowtie_{A=B} S) = \text{cost}(R) + \text{cost}(S) + 2 \cdot (\text{pages}(R) + \text{pages}(S))$$

Relaciones

$$\text{cost}(R \bowtie_{A=B} S) = 3 \cdot (\text{pages}(R) + \text{pages}(S))$$

Mismo costo que **Sort merge join**!

Sort-merge join vs. Hash join

Si el costo de ambos algoritmos es:

$$\text{cost}(R \bowtie_{A=B} S) \quad = \quad \text{cost}(R) + \text{cost}(S) + 2 \cdot (\text{pages}(R) + \text{pages}(S))$$

¿cuál es la diferencia?

- Sort-merge join funciona con cualquier **distribución** de los datos.
- El resultado de sort-merge join es **ordenado**.
- Hash join permite sacarle mayor provecho al buffer.

Resumen de costos para algoritmos de Join

- Block nested loops join

$$\text{pages}(R) + \frac{\text{pages}(R) \cdot \text{pages}(S)}{B}$$

- Index nested loops join

$$\text{pages}(R) + |R| \cdot \left\lceil \text{pages}(S) \cdot \text{sel}_{R \bowtie_p}(S) \right\rceil$$

- Sort-merge join

$$3 \cdot (\text{pages}(R) + \text{pages}(S))$$

- Hash join

$$3 \cdot (\text{pages}(R) + \text{pages}(S))$$

¿cuál preferirían?