IIC2413

AYUDANTÍA 10

PARTE 1

EVALUACIÓN DE CONSULTAS

RECORDATORIO

Costo de un algoritmo

- **Pregunta clave:** ¿Cuántas veces tengo que leer una página desde el disco, o escribir una página al disco?
- Operaciones en buffer (**RAM**) son órdenes de magnitud **más rápidas** que leer/escribir al **disco** (se consideran **costo 0**)
- Costo se mide solo en operaciones I/O a disco

RECORDATORIO

Páginas, disco y buffer

- **Datos** de una relación están guardado en **páginas en disco**
- Para operar con una tupla, el DBMS primero debe **cargar** la **página** que contiene esa tupla **al buffer** (RAM)
- El buffer es un **espacio reservado** en RAM
- Si la página **ya está** en el buffer, **no se hace un nuevo I/O** (ya la tenemos)

ITERADOR LINEAL

- **Todos** los operadores de álgebra relacional implementan la **misma interfaz:**
  - open() → se posiciona antes de la primera tupla
  - next() → devuelve la siguiente tupla o NULL
  - close() → cierra el iterador
- Para una **relación R:**
  - R.open() carga la primera página
  - R.next() devuelve la siguiente tupla; cuando se acaba una página carga la siguiente
  - R.close() libera el buffer

OPERADORES LINEALES

Selección

- Algoritmo **cambia** dependiendo si es una consulta de **igualdad** (=) **o de rango** (<, >)
- Depende si el **atributo** a seleccionar está **indexado**
- Implementa la interfaz de **iterador lineal**

OPERADORES LINEALES

Selección sin índice

- Necesariamente tenemos que recorrer todo R → "FULL SCAN"

Funcionamiento operador

$\sigma_{cond}(R)$

```
open()
    R.open()

next() // retorna el siguiente seleccionado
    t:= R.next()
    while t != null do
        if t satisface condición then
            return t
        t:= R.next()
    return null
close()
    R.close()
```

Uso operador

Sel = $\sigma_{cond}(R)$

```
Sel.open()

t := Sel.next()
while t != null do
        output t
        t:= Sel.next()
Sel.close()
```

OPERADORES LINEALES

Selección con índice y consulta de igualdad

- Sólo **leer páginas que satisfacen** la condición (más I/O si muchas tuplas satisfacen)
- Cambia si el índice es Clustered o Unclustered
  - **Clustered:** tuplas que cumplen están **cerca** físicamente → **pocos accesos** a páginas
  - **Unclustered:** tuplas pueden estar **dispersas** → **muchos** accesos a páginas cuando el resultado es grande.
- Si el atributo es **llave primaria** entonces la operación prácticamente tiene **I/O~1**

Selección sobre una **tabla R** a un atributo **indexado** con un **índice I**

```
open()
    I.open()
    I.search(Atributo = valor)

next()
    t:= I.next()
    while t != null do
        return t
    return null

close()
    I.close()
```

OPERADORES LINEALES

Proyección

$\pi_{\text{atributos}}(R)$

- Necesariamente tenemos que **recorrer todo R** (no se puede evita full scan)
  - **No sabemos** de antemano **en qué páginas están** los valores de los atributos que queremos

```
open()
    R.open()

next()
    t:= R.next()
    while t != null do
        return project(t, atributos)
    return null

close()
    R.close()
```

NESTED LOOP JOIN

Joins

- Operación muy **costosa**
- ¿**Por qué** el join es más costoso que la selección o la proyección?
    - Para cada tupla de una relación hay que buscar en la otra relación, lo que **multiplica los accesos** a disco.

NESTED LOOP JOIN

- **Join** entre R y S, cuando se satisface un **predicado p**
- Implementación directa basada en un **loop**
- Para **cada tupla de R debemos leer S entera,** aparte de leer R entera una vez
- **Costo en I/O = Costo(R) + Tuplas(R)·Costo(S)**

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

Join = $R \bowtie_p S$

```
Join.open()

 t:= Join.next()
while t != null do
            output t
    t:= Join.next()

Join.close()
```

BLOCK NESTED LOOP JOIN

- **Join** entre R y S, cuando se satisface un **predicado p**
- Ahora **cargamos** muchas páginas de R **a buffer**
- Por cada vez que llenamos el buffer **recorremos a S entera una vez**
- Costo en I/O = Costo(R) + (Páginas(R)/(Buffer−2))·Costo(S)

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

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

```
fillBuffer()
    Buff = empty
    r:= R.next()
    while r != null do
        Buff = Buff union r
        if Buff.isFull() then
            break
        r:= R.next()
    S.open()
    S.next()
```

```
next()
    while Buff != empty do
        while s != null do
            r:= Buffer.next()
            if r == null then
                Buffer.reset()
                s:= S.next()
            else if (r,s) satisfacen p then
                return (r,s)
        fillBuffer()
    return null
```

EXTERNAL MERGE SORT

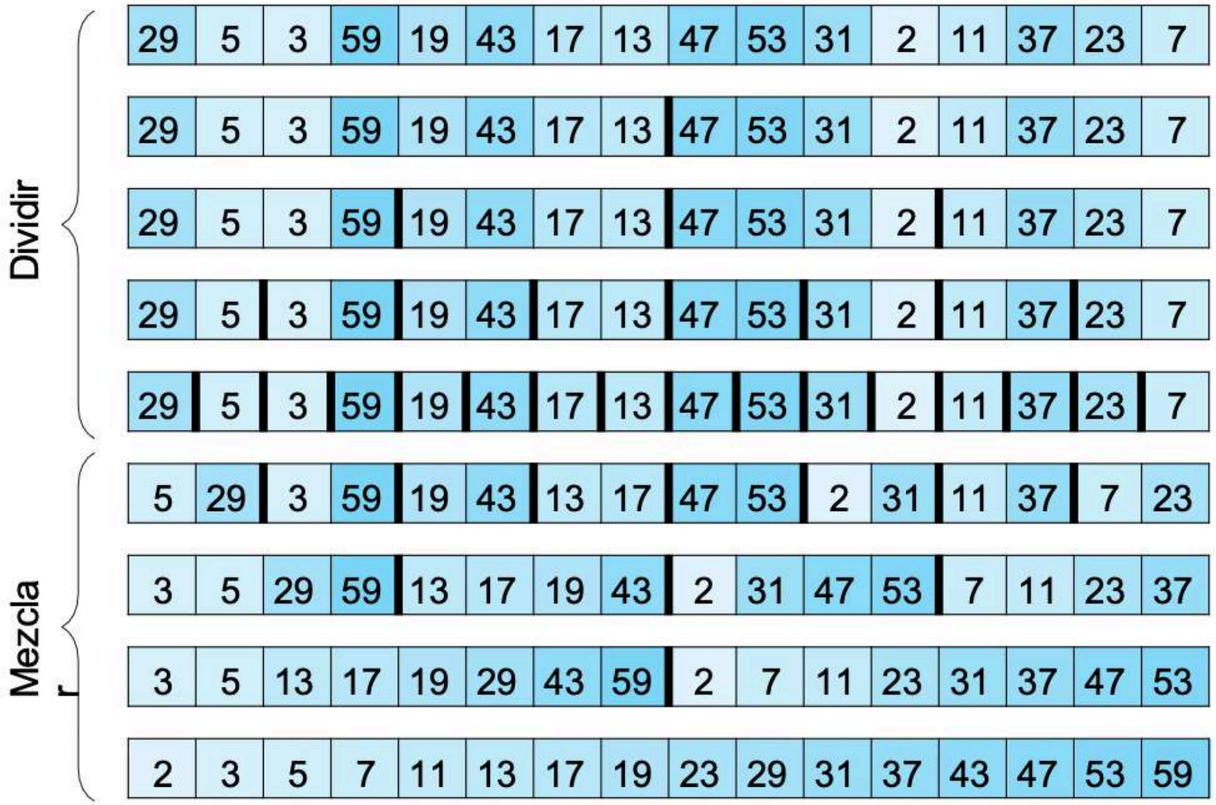
Sort y MergeSort

Merge(A, B):

1. Iniciamos *C* vacía
2. Sean *a* y *b* los primeros elementos de *A* y *B*
3. Extraer de su secuencia respectiva el menor entre *a* y *b*
4. Insertar el elemento extraído al final de *C*
5. Si quedan elementos en *A* **y** *B*, volver a la línea 2
6. Concatenar *C* con la secuencia que aún tenga elementos
7. **return** *C*

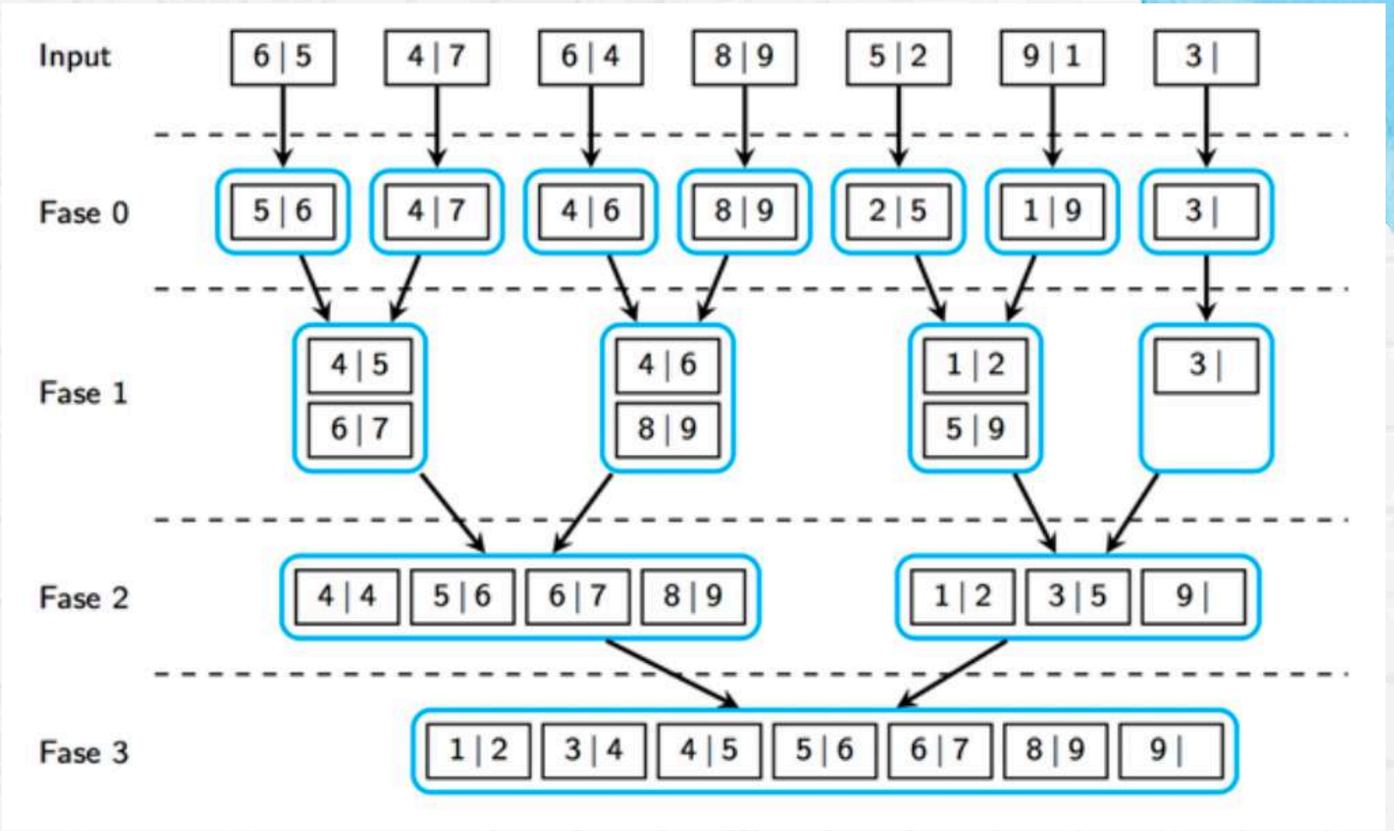
MergeSort *(A)*:

1. **if** $|A| = 1$ **: return** *A*
2. Dividir *A* en mitades A_1 y A_2
3. $B_1 \leftarrow \text{MergeSort}(A_1)$
4. $B_2 \leftarrow \text{MergeSort}(A_2)$
5. $B \leftarrow \text{Merge}(B_1, B_2)$
6. **return** *B*



EXTERNAL MERGE SORT

- **Fase 0:** creamos los **runs iniciales**
- **Fase i:**
  - Traemos los **runs a memoria**
  - Hacemos el **merge de cada par** de runs
  - **Almacenamos el nuevo run** a disco (i.e. materializamos resultados intermedios)



PARTE 2

NOSQL

NOSQL

Cuándo usar SQL vs NoSQL

SQL

- Datos muy estructurados
- Muchos cruces (joins) frecuentes
- Necesito ACID garantizado
- Datos cambian constantemente

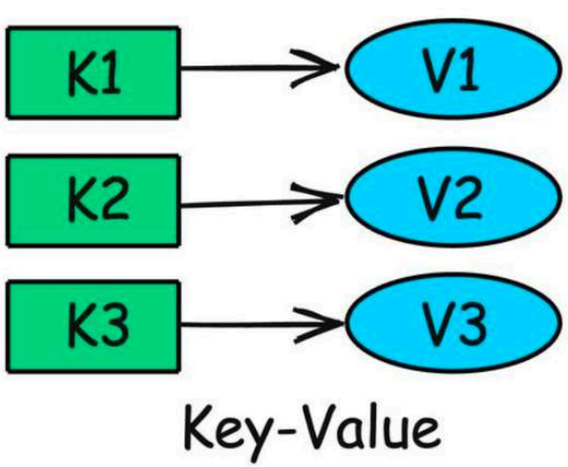
NoSQL

- Datos semi/no estructurados
- Alta escala y distribución
- Despliegue de información estática
- Consistencia eventual es aceptable

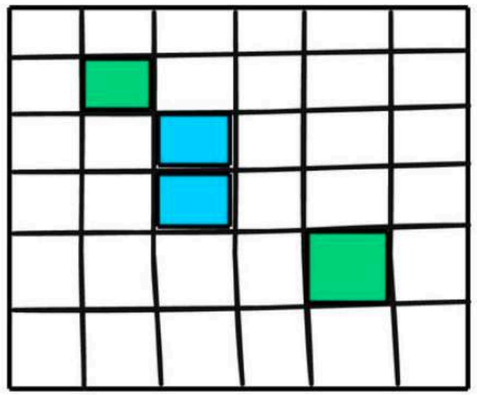
NOSQL

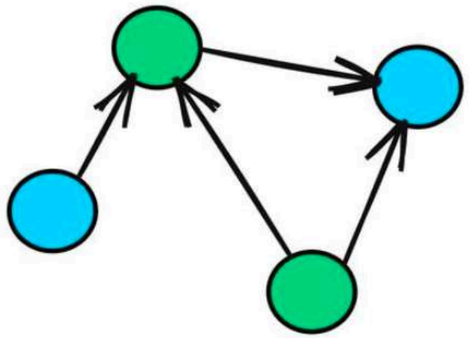
Tipos

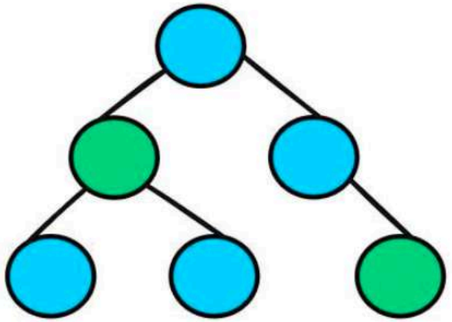
- **Key-Value:** operaciones put/get/delete, gran volumen, simple. Ejemplos: Redis, DynamoDB
- **Columnar:** alto volumen, complejidad media. Ejemplo: Cassandra
- **Documentos:** JSON, colecciones, flexibilidad. Ejemplos: MongoDB, CouchDB
- **Grafos:** relaciones complejas, menor volumen. Ejemplos: Neo4j, Virtuoso



Key-Value


Column Store


Graph


Document

MONGODB

Conceptos clave

- **Documento:** unidad básica, es un **JSON con un _id ObjectId** generado automáticamente
- **Colección:** agrupación de documentos similares (≈ **tabla**)
- **Base de datos:** contiene colecciones relacionadas
- Un servidor puede tener varias bases de datos
- **Esquema flexible:** dos documentos en la misma colección pueden tener campos distintos

Colección

Usuarios	
Key	JSON
60bfd90e002ce228636e506b	
60bfd90e002ce228636e5215	...

Base de datos

Usuarios	
Key	JSON
...	...

Mensajes	
Key	JSON
...	...

Likes	
Key	JSON
...	...

MONGODB

Sintáxis

- **find()** permite consultar documentos de una colección
- Filtros y proyecciones son **análogos a WHERE y SELECT**
- **Operadores de comparación:** $gte, $gt, $lt, $lte, $ne
- **createIndex()** crea índices para optimizar consultas
- Sintaxis se basa en **documentos JSON**

```
// Ver todo
db.usuarios.find()

// WHERE
db.estudiantes.find({"carrera":"Ingeniería"})

// WHERE edad >= 22
db.estudiantes.find({"edad": {$gte:22}})

// SELECT nombre, edad
db.estudiantes.find({}, {"nombre":1, "edad":1, "_id":0})

// Crear índice
db.estudiantes.createIndex({"carrera":1})

// Búsqueda de texto
db.ramos.find({$text: {$search:"Bases de datos"}})
```

PARTE 3

EJERCICIOS

EJERCICIOS

Se dispone de dos relaciones para realizar el join **Compras** $\bowtie$ **Entregas** por el atributo `CID` (clave foránea en **Entregas**). No hay índices salvo que se mencione lo contrario. El buffer disponible es de $M = 16$ páginas. Los tamaños lógicos y físicos son:

- **Compras**(`CID`, `fecha`, `total`, `estado`) con 60.000 tuplas y 30 tuplas/página.
- **Entregas**(`EID`, `CID`, `fecha`) con 150.000 tuplas y 75 tuplas/página.

(a) Estime el costo de un **Nested Loop Join (NLJ)** usando **Compras** como relación externa.

(b) Estime el costo de un **Block Nested Loop Join (BNLJ)** usando **Compras** como externa.

(c) Estime el costo de un **Merge Join** si **ambas** relaciones ya están ordenadas por `CID`.

(d) Asuma que ambas relaciones **no** están ordenadas, estime el **costo total** de un **Sort-Merge Join** (ordenar **ambas** relaciones y luego *merge*).

(e) ¿Qué estrategia elegirias y por qué?

EJERCICIOS

DATOS

- Compras: 60.000 tuplas, 30 tuplas/página
- Entregas: 150.000 tuplas, 75 tuplas/página
- M = 16 páginas

CÁLCULO PÁGINAS

- P_Compras = 60.000 / 30 = 2.000 páginas
- P_Entregas = 150.000 / 75 = 2.000 páginas

EJERCICIOS

(a) Estime el costo de un **Nested Loop Join (NLJ)** usando **Compras** como relación externa.

- Costo = P_ext + T_ext · P_int
- Costo = 2.000 + 60.000 · 2.000
  = 2.000 + 120.000.000
  = 120.002.000 I/O

120.002.000
lecturas de página

EJERCICIOS

(b) Estime el costo de un **Block Nested Loop Join (BNLJ)** usando **Compras** como externa.

- Costo = P_ext + ceil(P_ext / (M – 2)) · P_int

Como M = 16

- M – 2 = 14
- Costo = 2.000 + ceil(2.000 / 14) · 2.000
  = 2.000 + 143 · 2.000
  = 288.000 I/O

288.000 lecturas de página

EJERCICIOS

(c) Estime el costo de un **Merge Join** si **ambas** relaciones ya están ordenadas por `CID`.

- Costo = P_Compras + P_Entregas
- Costo = 2.000 + 2.000 = 4.000 I/O

4.000 lecturas de página

EJERCICIOS

(d) Asuma que ambas relaciones **no** están ordenadas, estime el **costo total** de un **Sort-Merge Join** (ordenar **ambas** relaciones y luego *merge*).

Para cada relación

- B = 2.000
- M = 16
- M – 1 = 15

Runs iniciales

- ceil(2.000 / 16) = 125 runs

Pasadas de merge

- 125 → ceil(125/15) = 9
- 9 → ceil(9/15) = 1

EJERCICIOS

1 pasada de generación de runs + 2 pasadas de merge = 3 pasadas totales

Costo de una relación:
$2 \cdot 2.000 \cdot 3 = 12.000$ I/O

Como son 2 relaciones:
Ordenar Compras + ordenar Entregas = 12.000 + 12.000 = 24.000

Luego merge join:
2.000 + 2.000 = 4.000

Costo total
24.000 + 4.000 = 28.000 I/O

EJERCICIOS

(e) ¿Qué estrategia elegirias y por qué?

Para cada relación

- NLJ : 120.002.000 I/O
- BNLJ : 288.000 I/O
- Merge Join: 4.000 I/O si ya están ordenadas
- Sort-Merge Join: 28.000 I/O sin no están ordenadas

Si están ordenadas, conviene **Merge Join.** Si no, conviene **Sort-Merge Join.**

EJERCICIOS

Se tiene la relación $R(a, b, c)$ con 480.000 tuplas y 80 tuplas por página. El buffer es de $M = 15$ páginas. Se desea ejecutar:

```
SELECT DISTINCT b FROM R;
```

Para la relación proyectada con solo el atributo b, cada página almacena 200 valores de b. Cálculos base:

$$B_R = \frac{480.000}{80} = 6.000 \text{ páginas}, \qquad B_{\text{proj}(b)} = \frac{480.000}{200} = 2.400 \text{ páginas}.$$

(a) Proponga un algoritmo basado en **sort** para `SELECT DISTINCT b FROM R` e indique el costo de la *proyección*.

(b) ¿Cuántas **pasadas** requiere ordenar T_b con $M = 15$?

(c) Calcule el **costo total** (proyectar + ordenar + deduplicar).

(d) Si el buffer creciera a $M = 60$, ¿cuántas pasadas y qué costo tendría *solo* el ordenamiento de T_b?

(e) ¿Cuál es el **mínimo** M para ordenar T_b en **dos pasadas totales** (run-gen + un solo merge)?

EJERCICIOS

DATOS

- R(a,b,c): 480.000 tuplas
- 80 tuplas/página
- M = 15
- SELECT DISTINCT b FROM R

- B_R = 480.000 / 80 = 6.000 páginas
- B_proj(b) = 480.000 / 200 = 2.400 páginas

EJERCICIOS

Se tiene la relación $R(a, b, c)$ con 480.000 tuplas y 80 tuplas por página. El buffer es de $M = 15$ páginas. Se desea ejecutar:

```
SELECT DISTINCT b FROM R;
```

Para la relación proyectada con solo el atributo b, cada página almacena 200 valores de b. Cálculos base:

$$B_R = \frac{480.000}{80} = 6.000 \text{ páginas}, \qquad B_{\text{proj}(b)} = \frac{480.000}{200} = 2.400 \text{ páginas}.$$

(a) Proponga un algoritmo basado en **sort** para `SELECT DISTINCT b FROM R` e indique el costo de la *proyección*.

(b) ¿Cuántas **pasadas** requiere ordenar T_b con $M = 15$?

(c) Calcule el **costo total** (proyectar + ordenar + deduplicar).

(d) Si el buffer creciera a $M = 60$, ¿cuántas pasadas y qué costo tendría *solo* el ordenamiento de T_b?

(e) ¿Cuál es el **mínimo** M para ordenar T_b en **dos pasadas totales** (run-gen + un solo merge)?

EJERCICIOS

(a) Proponga un algoritmo basado en **sort** para `SELECT DISTINCT b FROM R` e indique el costo de la *proyección*.

1. Escanear R.
2. Proyectar solo el atributo b.
3. Escribir la relación temporal Tb.
4. Ordenar Tb por b.
5. Eliminar duplicados recorriendo Tb ordenada.

Costo de proyección = Leer R + escribir Tb

Costo = 6.000 + 2.400 = 8.400 I/O

EJERCICIOS

(b) ¿Cuántas **pasadas** requiere ordenar T_b con $M = 15$?

- B_Tb = 2.400 páginas
- M = 15
- M – 1 = 14

Runs iniciales

- ceil(2.400 / 15) = 160 runs

Pasadas de merge

- 160 → ceil(160/14) = 12
- 12 → ceil(12/14) = 1

1 generación de runs + 2 pasadas de merge = 3 pasadas totales

EJERCICIOS

(c) Calcule el **costo total** (proyectar + ordenar + deduplicar).

Costo de ordenar Tb:

2 · B_Tb · pasadas = 2 · 2.400 · 3 = 14.400 I/O

Costo total:

Proyección + ordenamiento = 8.400 + 14.400 = 22.800 I/O

La deduplicación se puede hacer durante la última pasada del sort o con un escaneo final de la salida ordenada, así que normalmente no se suma un costo extra relevante si se hace integrada.

EJERCICIOS

(d) Si el buffer creciera a $M = 60$, ¿cuántas pasadas y qué costo tendría *solo* el ordenamiento de T_b?

Datos

- B_Tb = 2.400
- M = 60
- M – 1 = 59

Runs iniciales

ceil(2.400 / 60) = 40 runs, Como 40 <= 59, se puede hacer en un solo merge

1 generación de runs + 1 merge = 2 pasadas totales

2 · 2.400 · 2 = 9.600 I/O

EJERCICIOS

(e) ¿Cuál es el **mínimo** M para ordenar T_b en **dos pasadas totales** (run-gen + un solo merge)?

Condición, con B = 2.400

- $B \leqslant M(M-1) \rightarrow 2.400 \leqslant M(M-1)$

Probamos

$49 \cdot 48 = 2.352$ no alcanza
$50 \cdot 49 = 2.450$ sí alcanza

M mínimo = 50 páginas

¡GRACIAS!