IIC2413

AYUDANTÍA 10

PARTE 1

TRANSACCIONES — UNDO Y REDO LOGGING

TRANSACCIONES

¿Por qué las necesitamos?

- **Tipos de fallas:** datos erróneos, fallas de hardware, catástrofes, fallas del sistema
- El Log y Recovery Manager garantiza **Atomicidad y Durabilidad** (A y D de ACID)
- **Objetivo clave:** si el sistema se cae, queremos volver a un estado consistente

UNDO LOGGING

Logs:

<START T>	Transacción comenzó
<T, X, t>	T modificó X, con t el valor antiguo de X
<COMMIT T>	T terminó con éxito
<ABORT T>	T fue abortada

UNDO LOGGING

Idea central

- Guarda el **valor antiguo** de cada dato modificado: <T, X, t_antiguo>
- **Regla 1:** antes de escribir el nuevo valor de X en disco, primero escribir el log <T, X, t>
  - Así si hay error justo después de escribir el dato nuevo se **conoce dato antiguo** para poder **deshacer**
- **Regla 2:** <COMMIT T> va después de que todos los datos que modificó T ya están en disco
  - COMMIT es **señal** de que la transacción terminó exitosamente. Si el sistema lo ve **asume** que los **datos ya están guardados** y no los deshace

UNDO LOGGING

Algoritmo UNDO Recovery

1. Leer log **desde el final hacia el principio**
2. Si aparece <COMMIT T> o <ABORT T>, marcar T como terminada
3. Si aparece <T, X, t> y T **no** está terminada, restituir X = t en disco
4. Si aparece <START T>, no se deshace nada, solo indica el comienzo

UNDO LOGGING

Checkpoints

Checkpoint completo

- Retrocedo hasta el <START CKPT (T1,...,Tn)> correspondiente
- Hago UNDO de las transacciones sin COMMIT que aparezcan desde ahí
- Todo lo anterior al <START CKPT> ya estaba OK → no necesito mirarlo

Checkpoint incompleto

- Falla ocurrió mientras se estaba haciendo el checkpoint
- Debo leer hasta inicio de transacción más antigua listada entre T1...Tn
- Hago UNDO desde ahí

REDO LOGGING

Logs:

<START T>	Transacción comenzó
<T, X, t>	T modificó X, con t el valor nuevo de X
<COMMIT T>	T terminó, log en disco
<ABORT T>	T fue abortada
<END T>	T terminó, datos en disco

REDO LOGGING

Idea central

- Guarda el **valor nuevo** de cada dato: <T, X, v_nuevo>
- **Regla:** todo el log (incluyendo <COMMIT T>) debe estar en disco antes de escribir los datos
    - Así si el sistema se cae antes de escribir los datos en disco, se **puede repetir el cambio** usando el valor nuevo que está en el log
- Aparece **<END T>** cuando los **datos y el commit están en disco**

REDO LOGGING

Algoritmo REDO Recovery

1. Identificar **transacciones con COMMIT**
2. Ignorar transacciones sin COMMIT
3. Si una transacción ya tiene **END**, normalmente **no** hay que **rehacerla**
4. Recorrer **hacia adelante** el tramo elegido del log
5. Para cada <T, X, t> de una transacción a rehacer, escribir **X = t**

Como los datos nunca se escriben en disco **antes del COMMIT, esos **datos nunca llegan a disco**, y no hay nada que deshacer o rehacer

REDO LOGGING

Checkpoints

Checkpoint completo

- Retrocedo hasta el <START CKPT (T1,...,Tn)> correspondiente
- Hago redo solo de las transacciones con COMMIT sin END que aparezcan desde la más antigua entre T1...Tn
- Todo lo anterior al <START CKPT> ya estaba OK → no necesito mirarlo

Checkpoint incompleto

- Falla ocurrió mientras se estaba haciendo el checkpoint
- Busco el <END CKPT> válido anterior más cercano.
- Hago REDO desde ahí

UNDO VS REDO

Tabla comparativa

	UNDO	REDO
Registro de escritura	Valor antiguo	Valor nuevo
Lectura principal	Hacia atrás	Hacia adelante
Sin COMMIT	Se deshace	Se ignora
Checkpoint incompleto	Leer desde la transacción activa más antigua	No usar ese checkpoint

PARTE 2

ÍNDICES

ÍNDICES

Costo I/O

- Ir a buscar datos al disco es **extremadamente lento** comparado con RAM
- **Métrica:** número de **páginas leídas del disco** (I/O)
- Ej: SELECT * FROM T WHERE a = 4 sin índice → leer **todas las páginas** de T

ÍNDICES

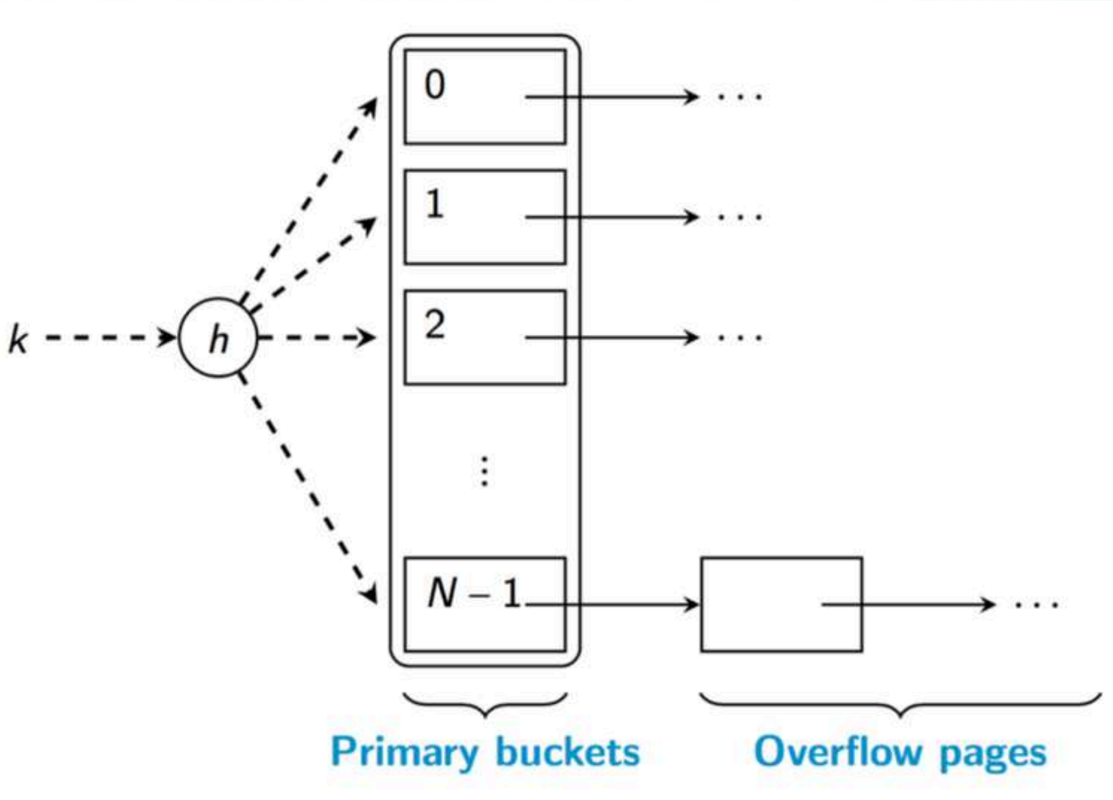
¿Qué es?

- **Estructura auxiliar** que permite encontrar tuplas con un valor dado **sin recorrer toda la tabla**
- Conceptos clave:
  - **Search Key:** el atributo por el que busco (ej: id)
  - **Data Entry:** lo que retorna el índice (la tupla misma o un puntero a ella)
- **SQL:** CREATE INDEX nombre ON tabla(atributo)
- Las primary keys están indexadas por defecto

ÍNDICES

Hash index

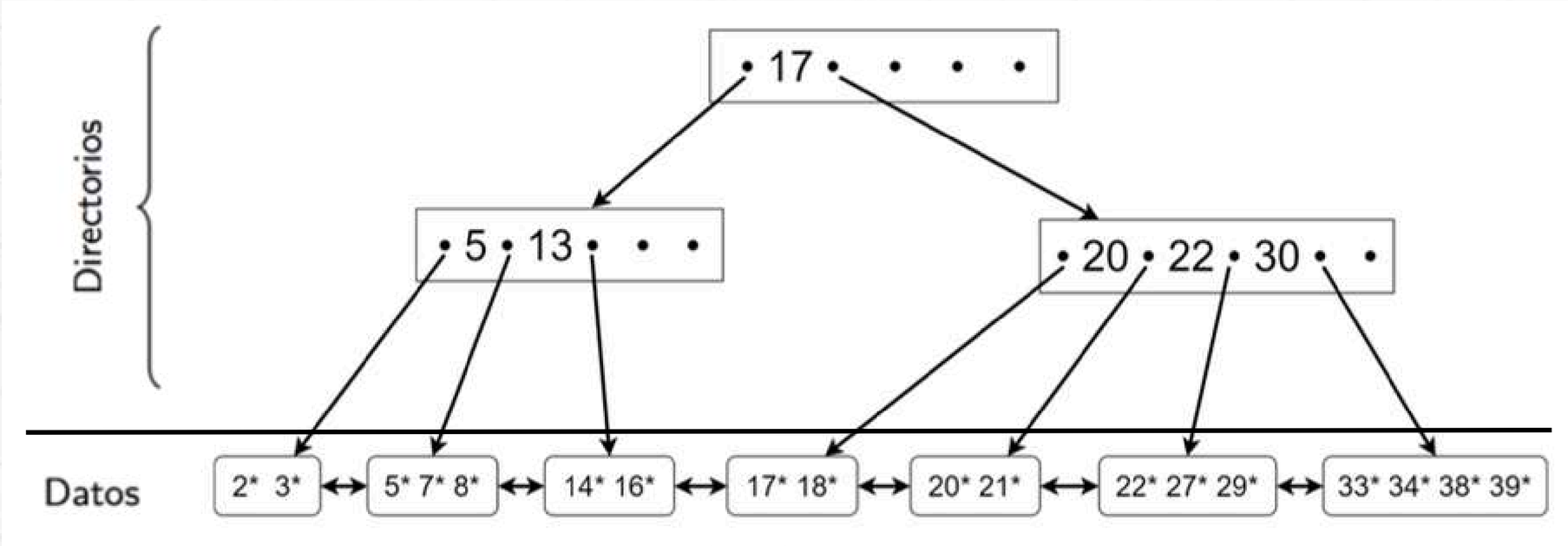
- Función **h(k) = k mod m** para **mapear** la search key a un bucket (página)
- Ideal para **consultas de igualdad,** ineficiente para rangos
- **Búsqueda exacta: O(1).** Solo calcular h(k) y acceder al bucket
- **Peor caso:** O(⌈|R|/B⌉), con |R| = número total de tuplas de la tabla, B = número de tuplas que caben en una página del disco
    - Es decir, en una consulta de rango se leen **todas** las páginas de la tabla



ÍNDICES

B+ Tree index

- Árbol balanceado de **altura h**
- Soporta **igualdad y rango** eficientemente
- Cada nodo y hoja ocupa una página
- **Costo de búsqueda:** O(log⌈|R|/B⌉) **log en base a B



ÍNDICES

Clustered VS Unclustered

Clustered

- Índice guarda la tupla directamente en el bucket
- Acceso directo a la página de datos

Unclustered

- Índice guarda un puntero a la página con la tupla
- Lee índice y luego página de datos

ÍNDICES

Tabla resumen costos I/O

Índice	Tipo	Igualdad	Rango
Sin índice	–	$\lceil \lvert R \rvert / B \rceil$	$\lceil \lvert R \rvert / B \rceil$
Hash	Clustered	1	$\lceil \lvert R \rvert / B \rceil$
Hash	Unclustered	2	$\lceil \lvert R \rvert / B \rceil$
B+ Tree	Clustered	h	$h + \lceil nR / B \rceil$
B+ Tree	Unclustered	h+1	$h + \lceil n / P \rceil + n$

Donde:

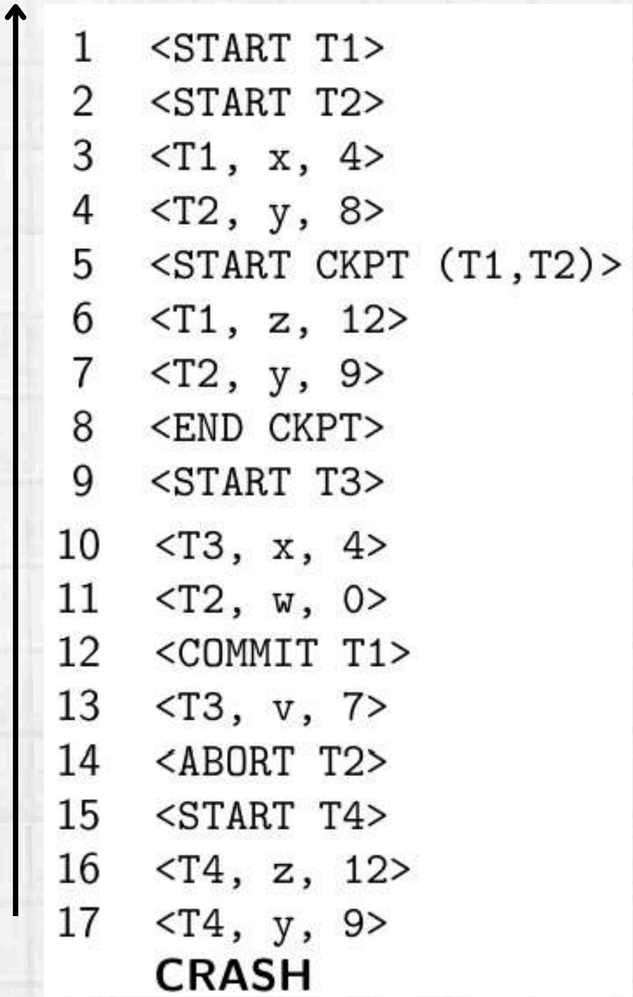
- $|R|$ → número total de tuplas
- B → tuplas por página
- h → altura del árbol B+
- n → tuplas que calzan con la búsqueda
- P → punteros por página ($P > B$)

PARTE 3

EJERCICIOS

EJERCICIOS: UNDO

(a) ¿Hasta qué parte del log debe leerse para ejecutar la fase de *undo*?



```
 1   <START T1>
 2   <START T2>
 3   <T1, x, 4>
 4   <T2, y, 8>
 5   <START CKPT (T1,T2)>
 6   <T1, z, 12>
 7   <T2, y, 9>
 8   <END CKPT>
 9   <START T3>
10   <T3, x, 4>
11   <T2, w, 0>
12   <COMMIT T1>
13   <T3, v, 7>
14   <ABORT T2>
15   <START T4>
16   <T4, z, 12>
17   <T4, y, 9>
     CRASH
```

EJERCICIOS: UNDO

(a) ¿Hasta qué parte del log debe leerse para ejecutar la fase de *undo*?

```
 1   <START T1>
 2   <START T2>
 3   <T1, x, 4>
 4   <T2, y, 8>
 5   <START CKPT (T1,T2)>
 6   <T1, z, 12>
 7   <T2, y, 9>
 8   <END CKPT>
 9   <START T3>
10   <T3, x, 4>
11   <T2, w, 0>
12   <COMMIT T1>
13   <T3, v, 7>
14   <ABORT T2>
15   <START T4>
16   <T4, z, 12>
17   <T4, y, 9>
     CRASH
```



Lectura

El checkpoint está completo porque aparecen `START CKPT` y `END CKPT`.

EJERCICIOS: UNDO

(a) ¿Hasta qué parte del log debe leerse para ejecutar la fase de *undo*?

```
 1  <START T1>
 2  <START T2>
 3  <T1, x, 4>
 4  <T2, y, 8>
 5  <START CKPT (T1,T2)>
 6  <T1, z, 12>
 7  <T2, y, 9>
 8  <END CKPT>
 9  <START T3>
10  <T3, x, 4>
11  <T2, w, 0>
12  <COMMIT T1>
13  <T3, v, 7>
14  <ABORT T2>
15  <START T4>
16  <T4, z, 12>
17  <T4, y, 9>
```

Respuesta

Se lee desde el final hasta <START CKPT (T1,T2)>.

EJERCICIOS: UNDO

(b) ¿Qué variables deben *deshacerse* y con qué valores quedan?

Paso 1: marcar transacciones terminadas

```
1   <START T1>
2   <START T2>
3   <T1, x, 4>
4   <T2, y, 8>
5   <START CKPT (T1,T2)>
6   <T1, z, 12>
7   <T2, y, 9>
8   <END CKPT>
9   <START T3>
10  <T3, x, 4>
11  <T2, w, 0>
12  <COMMIT T1>
13  <T3, v, 7>
14  <ABORT T2>
15  <START T4>
16  <T4, z, 12>
17  <T4, y, 9>
    CRASH
```

```
12  <COMMIT T1>
13  <T3, v, 7>
14  <ABORT T2>
15  <START T4>
16  <T4, z, 12>
17  <T4, y, 9>
```

Al leer hacia atrás

- $T1$: terminada por COMMIT.
- $T2$: terminada por ABORT.
- $T3$: no aparece COMMIT ni ABORT.
- $T4$: no aparece COMMIT ni ABORT.

Candidatas a UNDO

$T3$ y $T4$.

EJERCICIOS: UNDO

(b) ¿Qué variables deben *deshacerse* y con qué valores quedan?

Paso 2: deshacer T4

```
1   <START T1>
2   <START T2>
3   <T1, x, 4>
4   <T2, y, 8>
5   <START CKPT (T1,T2)>
6   <T1, z, 12>
7   <T2, y, 9>
8   <END CKPT>
9   <START T3>
10  <T3, x, 4>
11  <T2, w, 0>
12  <COMMIT T1>
13  <T3, v, 7>
14  <ABORT T2>
15  <START T4>
16  <T4, z, 12>
17  <T4, y, 9>
    CRASH
```

```
15  <START T4>
16  <T4, z, 12>
17  <T4, y, 9>
```

EJERCICIOS: UNDO

(b) ¿Qué variables deben *deshacerse* y con qué valores quedan?

Paso 2: deshacer T4

```
1   <START T1>
2   <START T2>
3   <T1, x, 4>
4   <T2, y, 8>
5   <START CKPT (T1,T2)>
6   <T1, z, 12>
7   <T2, y, 9>
8   <END CKPT>
9   <START T3>
10  <T3, x, 4>
11  <T2, w, 0>
12  <COMMIT T1>
13  <T3, v, 7>
14  <ABORT T2>
15  <START T4>
16  <T4, z, 12>
17  <T4, y, 9>
    CRASH
```

```
15  <START T4>
16  <T4, z, 12>
17  <T4, y, 9>
```

Orden de aplicación

UNDO en $T4$: aplicar <T4, y, 9> y luego <T4, z, 12>.

Acciones

y:=9
z:=12

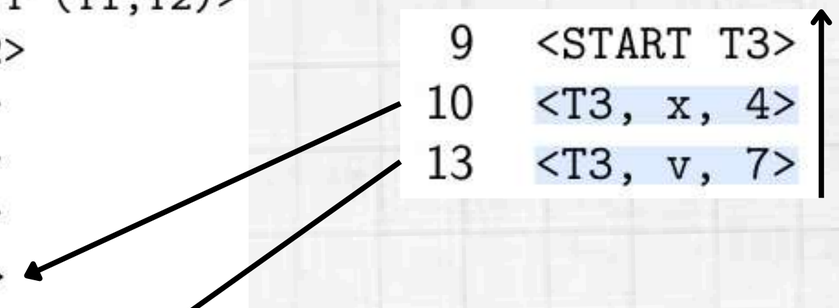
EJERCICIOS: UNDO

(b) ¿Qué variables deben *deshacerse* y con qué valores quedan?

Paso 3: deshacer T3

```
1   <START T1>
2   <START T2>
3   <T1, x, 4>
4   <T2, y, 8>
5   <START CKPT (T1,T2)>
6   <T1, z, 12>
7   <T2, y, 9>
8   <END CKPT>
9   <START T3>
10  <T3, x, 4>
11  <T2, w, 0>
12  <COMMIT T1>
13  <T3, v, 7>
14  <ABORT T2>
15  <START T4>
16  <T4, z, 12>
17  <T4, y, 9>
    CRASH
```

```
9   <START T3>
10  <T3, x, 4>
13  <T3, v, 7>
```



Orden de aplicación

UNDO en $T3$: aplicar <T3, v, 7> y luego <T3, x, 4>.

Acciones

v:=7

x:=4

EJERCICIOS: UNDO

(b) ¿Qué variables deben *deshacerse* y con qué valores quedan?

Transacción T4

- <T4, y, 9> $\Rightarrow y = 9$
- <T4, z, 12> $\Rightarrow z = 12$

Transacción T3

- <T3, v, 7> $\Rightarrow v = 7$
- <T3, x, 4> $\Rightarrow x = 4$

EJERCICIOS: UNDO

(c) ¿Qué variables **no** cambian durante el proceso de recuperación?

Regla de clase

En UNDO, solo se revierten escrituras de transacciones incompletas.

EJERCICIOS: UNDO

(c) ¿Qué variables **no** cambian durante el proceso de recuperación?

Regla de clase

En UNDO, solo se revierten escrituras de transacciones incompletas.

Sí cambian en recuperación

Variables tocadas por $T3$ y $T4$:

x, v, z, y según los valores antiguos del log.

No cambian

No se alteran los efectos de $T1$ ni de $T2$.

En particular, no se rehace ni se deshace nada por $T1$ o $T2$ durante esta recuperación.

EJERCICIOS: UNDO

(d) Si **no** existiera `<END CKPT>` correspondiente a ese `START CKPT`, ¿desde dónde se debería leer?

```
1  <START T1>
2  <START T2>
3  <T1, x, 4>
4  <T2, y, 8>
5  <START CKPT (T1,T2)>
6  <T1, z, 12>
7  <T2, y, 9>
   <END CKPT> faltante
8  <START T3>
9  ... hasta el crash
```

EJERCICIOS: UNDO

(d) Si **no** existiera `<END CKPT>` correspondiente a ese `START CKPT`, ¿desde dónde se debería leer?

```
1   <START T1>
2   <START T2>
3   <T1, x, 4>
4   <T2, y, 8>
5   <START CKPT (T1,T2)>
6   <T1, z, 12>
7   <T2, y, 9>
    <END CKPT> faltante
8   <START T3>
9   ... hasta el crash
```

Respuesta

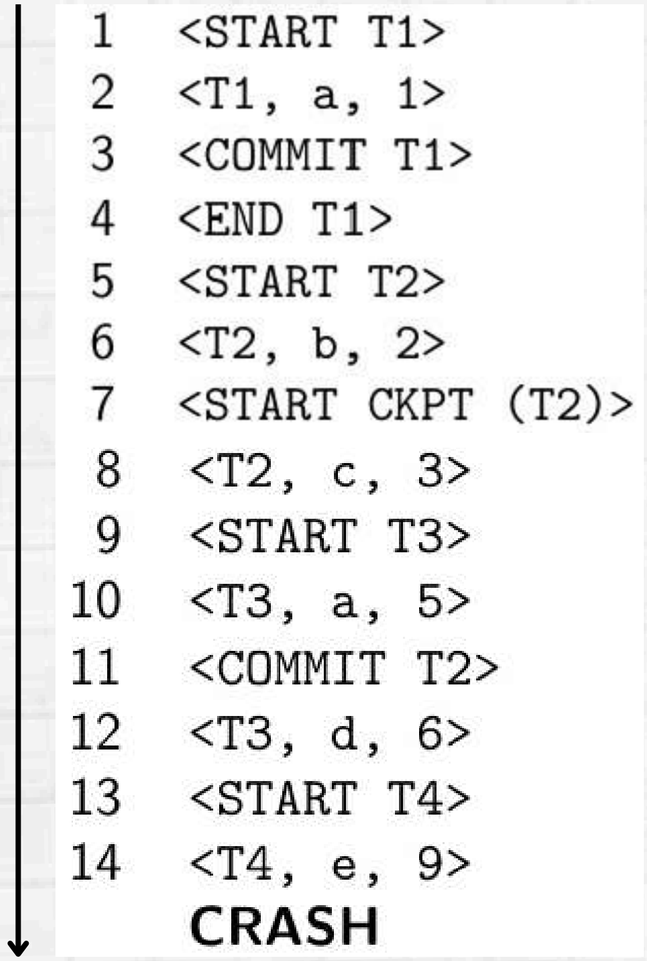
Si el checkpoint está incompleto, se lee desde el **inicio de la transacción más antigua** listada en `START CKPT`.

Aquí

El checkpoint lista $T1$ y $T2$. La más antigua es $T1$, por lo tanto se lee desde `<START T1>`.

EJERCICIOS: REDO

(a) ¿Desde qué parte del log se **comienza** el *redo*?



```
 1   <START T1>
 2   <T1, a, 1>
 3   <COMMIT T1>
 4   <END T1>
 5   <START T2>
 6   <T2, b, 2>
 7   <START CKPT (T2)>
 8   <T2, c, 3>
 9   <START T3>
10   <T3, a, 5>
11   <COMMIT T2>
12   <T3, d, 6>
13   <START T4>
14   <T4, e, 9>
     CRASH
```

EJERCICIOS: REDO

(a) ¿Desde qué parte del log se **comienza** el *redo*?

```
 1   <START T1>
 2   <T1, a, 1>
 3   <COMMIT T1>
 4   <END T1>
 5   <START T2>
 6   <T2, b, 2>
 7   <START CKPT (T2)>
 8   <T2, c, 3>
 9   <START T3>
10   <T3, a, 5>
11   <COMMIT T2>
12   <T3, d, 6>
13   <START T4>
14   <T4, e, 9>
```

Lectura

Hay `START CKPT (T2)`, pero no aparece `END CKPT`.

EJERCICIOS: REDO

(a) ¿Desde qué parte del log se **comienza** el *redo*?

Respuesta

Checkpoint incompleto: se descarta y se retrocede hasta el último `END CKPT` válido anterior. Si no aparece en el fragmento, se usa todo el tramo disponible.

```
1   <START T1>
2   <T1, a, 1>
3   <COMMIT T1>
4   <END T1>
5   <START T2>
6   <T2, b, 2>
7   <START CKPT (T2)>
8   <T2, c, 3>
9   <START T3>
10  <T3, a, 5>
11  <COMMIT T2>
12  <T3, d, 6>
13  <START T4>
14  <T4, e, 9>
```

EJERCICIOS: REDO

(b) ¿Qué variables deben *rehacerse* y a qué valores quedan?

```
 1   <START T1>
 2   <T1, a, 1>
 3   <COMMIT T1>
 4   <END T1>
 5   <START T2>
 6   <T2, b, 2>
 7   <START CKPT (T2)>
 8   <T2, c, 3>
 9   <START T3>
10   <T3, a, 5>
11   <COMMIT T2>
12   <T3, d, 6>
13   <START T4>
14   <T4, e, 9>
     CRASH
```

```
 3   <COMMIT T1>
 4   <END T1>
11   <COMMIT T2>
 9   <START T3>
13   <START T4>
```

EJERCICIOS: REDO

(b) ¿Qué variables deben *rehacerse* y a qué valores quedan?

```
 1   <START T1>
 2   <T1, a, 1>
 3   <COMMIT T1>
 4   <END T1>
 5   <START T2>
 6   <T2, b, 2>
 7   <START CKPT (T2)>
 8   <T2, c, 3>
 9   <START T3>
10   <T3, a, 5>
11   <COMMIT T2>
12   <T3, d, 6>
13   <START T4>
14   <T4, e, 9>
     CRASH
```

```
 3   <COMMIT T1>
 4   <END T1>
11   <COMMIT T2>
 9   <START T3>
13   <START T4>
```

Decisión

- $T1$: tiene `COMMIT` y `END`; ya quedó cerrada.
- $T2$: tiene `COMMIT`, pero no `END`; se rehace.
- $T3$, $T4$: no tienen `COMMIT`; se escribe `<ABORT T3>` y `<ABORT T4>`.

Candidata a REDO

Solo $T2$. Las transacciones $T3$ y $T4$ quedan cerradas con `ABORT`.

EJERCICIOS: REDO

(b) ¿Qué variables deben *rehacerse* y a qué valores quedan?

```
 1   <START T1>
 2   <T1, a, 1>
 3   <COMMIT T1>
 4   <END T1>
 5   <START T2>
 6   <T2, b, 2>
 7   <START CKPT (T2)>
 8   <T2, c, 3>
 9   <START T3>
10   <T3, a, 5>
11   <COMMIT T2>
12   <T3, d, 6>
13   <START T4>
14   <T4, e, 9>
     CRASH
```

```
 5   <START T2>
 6   <T2, b, 2>
 7   <START CKPT (T2)>
 8   <T2, c, 3>
11   <COMMIT T2>
```

Orden de aplicación

REDO en $T2$: aplicar <T2, b, 2> y luego <T2, c, 3>.

Acciones

b:=2

c:=3

EJERCICIOS: REDO

(b) ¿Qué variables deben *rehacerse* y a qué valores quedan?

Transacción T2

- <T2, b, 2> $\Rightarrow b = 2$
- <T2, c, 3> $\Rightarrow c = 3$

EJERCICIOS: REDO

(c) ¿Qué variables **no** se tocan?

No se rehacen

$T1$, porque ya tiene `END T1`; $T3$ y $T4$, porque no tienen `COMMIT`.

EJERCICIOS: REDO

(c) ¿Qué variables **no** se tocan?

No se rehacen

$T1$, porque ya tiene `END T1`; $T3$ y $T4$, porque no tienen `COMMIT`.

Sí se tocan

Solo las variables escritas por $T2$:

$b := 2$, $c := 3$.

No se tocan

- a de $T1$: ya está cerrada con `END T1`.
- a, d de $T3$: no hay `COMMIT T3`; se escribe `<ABORT T3>`.
- e de $T4$: no hay `COMMIT T4`; se escribe `<ABORT T4>`.

EJERCICIOS: REDO

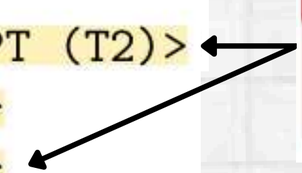
(d) Si existiera un `END CKPT` que cerrara ese `START CKPT (T2)`, ¿desde dónde comenzar el *redo*?

```
 1   <START T1>
 2   <T1, a, 1>
 3   <COMMIT T1>
 4   <END T1>
 5   <START T2>
 6   <T2, b, 2>
 7   <START CKPT (T2)>
 8   <T2, c, 3>
     <END CKPT>
 9   <START T3>
10   <T3, a, 5>
11   <COMMIT T2>
12   <T3, d, 6>
13   <START T4>
14   <T4, e, 9>
```

EJERCICIOS: REDO

(d) Si existiera un `END CKPT` que cerrara ese `START CKPT (T2)`, ¿desde dónde comenzar el *redo*?

```
 1   <START T1>
 2   <T1, a, 1>
 3   <COMMIT T1>
 4   <END T1>
 5   <START T2>
 6   <T2, b, 2>
 7   <START CKPT (T2)>
 8   <T2, c, 3>
     <END CKPT>
 9   <START T3>
10   <T3, a, 5>
11   <COMMIT T2>
12   <T3, d, 6>
13   <START T4>
14   <T4, e, 9>
```



Respuesta

Con checkpoint completo, el REDO comienza en `<START CKPT (T2)>`.

EJERCICIOS: REDO

(d) Si existiera un `END CKPT` que cerrara ese `START CKPT (T2)`, ¿desde dónde comenzar el *redo*?

```
1   <START T1>
2   <T1, a, 1>
3   <COMMIT T1>
4   <END T1>
5   <START T2>
6   <T2, b, 2>
7   <START CKPT (T2)>
8   <T2, c, 3>
    <END CKPT>
9   <START T3>
10  <T3, a, 5>
11  <COMMIT T2>
12  <T3, d, 6>
13  <START T4>
14  <T4, e, 9>
```

Luego

Se rehacen solo transacciones con `COMMIT` posteriores que no tengan `END`; aquí, $T2$.

Respuesta

Con checkpoint completo, el REDO comienza en `<START CKPT (T2)>`.

EJERCICIOS: CONSTRUIR UN LOG

Suponga valores iniciales $A = 20, \quad B = 50, \quad C = -2, \quad X = 100, \quad Y = 7.$

Ejecución de T1

```
T1 START
   R(A)
A = A + 10
   W(A)
   R(C)
X = A + C
   W(X)
   R(B)
B = B - 5
   W(B)
   R(A)
A = A * 2
   W(A)
   R(Y)
Y = B - Y
   W(Y)
T1 COMMIT
```

EJERCICIOS: CONSTRUIR UN LOG

(a) Escribe el log que se genera usando *Undo Logging*.

Ejecución de T1
`T1 START`
`R(A)`
`A = A + 10`
`W(A)`
`R(C)`
`X = A + C`
`W(X)`
`R(B)`
`B = B - 5`
`W(B)`
`R(A)`
`A = A * 2`
`W(A)`
`R(Y)`
`Y = B - Y`
`W(Y)`
`T1 COMMIT`

Paso	Escritura	Valor antiguo registrado
1	`W(A)` antes de dejar $A = 30$	$A = 20$
2	`W(X)` antes de dejar $X = 28$	$X = 100$
3	`W(B)` antes de dejar $B = 45$	$B = 50$
4	`W(A)` antes de dejar $A = 60$	$A = 30$
5	`W(Y)` antes de dejar $Y = 38$	$Y = 7$

EJERCICIOS: CONSTRUIR UN LOG

(a) Escribe el log que se genera usando *Undo Logging*.

Ejecución de T1
T1 START
R(A)
A = A + 10
W(A)
R(C)
X = A + C
W(X)
R(B)
B = B - 5
W(B)
R(A)
A = A * 2
W(A)
R(Y)
Y = B - Y
W(Y)
T1 COMMIT

Paso	Escritura	Valor antiguo registrado
1	W(A) antes de dejar $A = 30$	$A = 20$
2	W(X) antes de dejar $X = 28$	$X = 100$
3	W(B) antes de dejar $B = 45$	$B = 50$
4	W(A) antes de dejar $A = 60$	$A = 30$
5	W(Y) antes de dejar $Y = 38$	$Y = 7$

Log UNDO de *T1*

Log UNDO
<START T1>
<T1, A, 20>
<T1, X, 100>
<T1, B, 50>
<T1, A, 30>
<T1, Y, 7>
<COMMIT T1>

EJERCICIOS: CONSTRUIR UN LOG

(b) Escriba el log que se genera usando *Redo Logging*.

Ejecución de T1
`T1 START`
`R(A)`
`A = A + 10`
`W(A)`
`R(C)`
`X = A + C`
`W(X)`
`R(B)`
`B = B - 5`
`W(B)`
`R(A)`
`A = A * 2`
`W(A)`
`R(Y)`
`Y = B - Y`
`W(Y)`
`T1 COMMIT`

Paso	Escritura	Valor nuevo
1	`W(A)` después de $A = 20 + 10$	$A = 30$
2	`W(X)` después de $X = A + C = 30 - 2$	$X = 28$
3	`W(B)` después de $B = 50 - 5$	$B = 45$
4	`W(A)` después de $A = 30 * 2$	$A = 60$
5	`W(Y)` después de $Y = B - Y = 45 - 7$	$Y = 38$

EJERCICIOS: CONSTRUIR UN LOG

(b) Escriba el log que se genera usando *Redo Logging*.

Ejecución de T1
T1 START
R(A)
A = A + 10
W(A)
R(C)
X = A + C
W(X)
R(B)
B = B - 5
W(B)
R(A)
A = A * 2
W(A)
R(Y)
Y = B - Y
W(Y)
T1 COMMIT

Paso	Escritura	Valor nuevo
1	W(A) después de $A = 20 + 10$	$A = 30$
2	W(X) después de $X = A + C = 30 - 2$	$X = 28$
3	W(B) después de $B = 50 - 5$	$B = 45$
4	W(A) después de $A = 30 * 2$	$A = 60$
5	W(Y) después de $Y = B - Y = 45 - 7$	$Y = 38$

Log

Log REDO
<START T1>
<T1, A, 30>
<T1, X, 28>
<T1, B, 45>
<T1, A, 60>
<T1, Y, 38>
<COMMIT T1>
<END T1>

UNDO/REDO LOGGING

Idea

Cada escritura guarda ambos valores: <T, X, valor antiguo, valor nuevo>.

Log de ejemplo

<START T1>
<T1, A, 20, 30>
<T1, B, 50, 45>
<COMMIT T1>
<START T2>
<T2, C, 7, 9>
CRASH

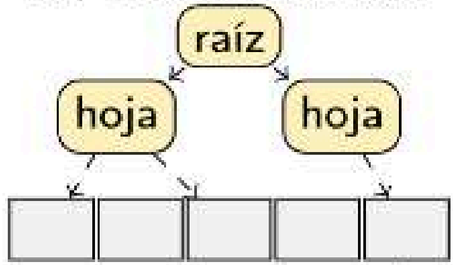
EJERCICIOS: ÍNDICES

Sin índice
heap



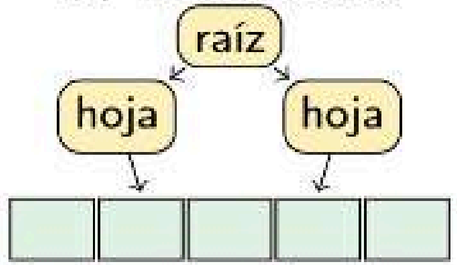
páginas de datos
sin orden por *a*

B+Tree unclustered



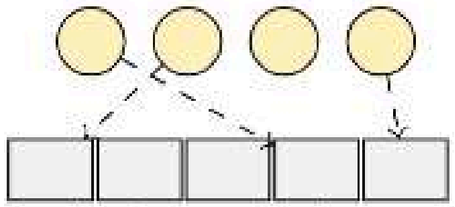
punteros pueden saltar
a muchas páginas

B+Tree clustered



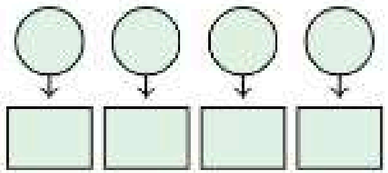
datos ordenados: ran-
gos quedan contiguos

Hash unclustered



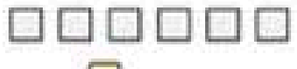
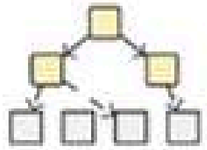
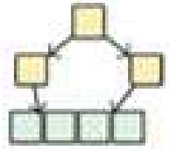
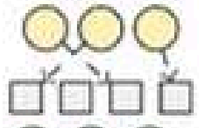
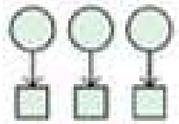
bucket + salto a
página de datos

Hash clustered



bucket coincide con el
dato: igualdad es directa

EJERCICIOS: ÍNDICES

Estructura de almacenamiento	Costo general para obtener k tuplas	Funcionamiento
1. Sin índice **Siempre revisa las N tuplas.**	$\left\lceil \frac{N}{P} \right\rceil$	
2. B+Tree Unclustered	$h + \left(\left\lceil \frac{k}{M} \right\rceil - 1 \right) + k$	
3. B+Tree Clustered	$h + \left(\left\lceil \frac{k}{P} \right\rceil - 1 \right)$	
4. Hash Unclustered	$2 \cdot k$	
5. Hash Clustered	k	

EJERCICIOS: ÍNDICES

Problema base

Relación $S(a, b, c, d)$ con $N = 500{,}000$ tuplas. Cada página de datos almacena P tuplas.

El atributo a es único y uniforme en $[0, 499{,}999]$, por lo que es candidato a llave.

Notación

- P: tuplas por página de datos.
- M: entradas por página de índice.
- h: altura del B+Tree.
- k: tuplas que cumplen la consulta.

EJERCICIOS: ÍNDICES

Problema base

Relación $S(a, b, c, d)$ con $N = 500{,}000$ tuplas. Cada página de datos almacena P tuplas.

El atributo a es único y uniforme en $[0, 499{,}999]$, por lo que es candidato a llave.

Notación

- P: tuplas por página de datos.
- M: entradas por página de índice.
- h: altura del B+Tree.
- k: tuplas que cumplen la consulta.

Consultas a resolver

- I) Obtener todas las tuplas.
- II) Obtener tuplas con $a \leq 120$.
- III) Obtener la tupla con $a = 120$.

EJERCICIOS: ÍNDICES

I)Todas las tuplas

Cálculo

Se deben obtener las 500,000 tuplas de S. Para una consulta sin selectividad, leer secuencialmente suele ser mejor que seguir punteros.

Escenario	Costo I/O	Explicación
Sin índice	$\left\lceil \frac{500{,}000}{P} \right\rceil$	Leer todas las páginas del heap
B+Tree unclustered	$h + \left(\left\lceil \frac{500{,}000}{M} \right\rceil - 1 \right) + 500{,}000$	Hojas del índice + una página por tupla
B+Tree clustered	$h + \left(\left\lceil \frac{500{,}000}{0{,}7P} \right\rceil - 1 \right)$	Datos ordenados con ocupación efectiva $0{,}7P$
Hash unclustered	$2 \cdot 500{,}000$	Bucket + página de datos por tupla
Hash clustered	500,000	Un bucket/dato por valor

EJERCICIOS: ÍNDICES

II) Rango a<=120

Cálculo

Como a es único y uniforme en $[0, 499{,}999]$, la consulta retorna $k = 121$ tuplas.

Escenario	Costo I/O	Explicación
Sin índice	$\left\lceil \frac{500{,}000}{P} \right\rceil$	Sin índice, se escanea todo
B+Tree unclustered	$h + \left(\left\lceil \frac{121}{M} \right\rceil - 1\right) + 121$	Llegar al rango + hojas + saltos a datos
B+Tree clustered	$h + \left(\left\lceil \frac{121}{0{,}7P} \right\rceil - 1\right)$	Rango contiguo en páginas de datos
Hash unclustered	$2 \cdot 121 = 242$	No sirve para rango ordenado; acceso por valor
Hash clustered	121	Un bucket/dato por cada valor del rango

EJERCICIOS: ÍNDICES

III) Búsqueda exacta a=120

Cálculo

La condición de igualdad retorna una única tupla porque *a* es candidata a llave.

Escenario	Costo I/O	Explicación
Sin índice	$\left\lceil \frac{500.000}{P} \right\rceil$	Hay que buscar en todo el heap
B+Tree unclustered	$h + \left(\left\lceil \frac{1}{M} \right\rceil - 1\right) + 1$	Recorrido del árbol + página de datos
B+Tree clustered	$h + \left(\left\lceil \frac{1}{0,7P} \right\rceil - 1\right)$	Recorrido hasta la página clustered
Hash unclustered	2	Bucket del índice + página de datos
Hash clustered	1	Acceso directo al bucket/dato

RESUMEN

Ideas clave

Recuperación

- **UNDO:** deshace transacciones **sin COMMIT,** usando valores antiguos.
- **REDO:** rehace transacciones **con COMMIT,** usando valores nuevos.
- **Checkpoints completos** reducen tramo del log que se debe revisar

Costos de I/O

- **Sin índice:** leer toda la relación
- **B+Tree clustered:** mejor para rangos
- **Hash clustered:** mejor para búsqueda exacta
- **Unclustered** debe realizar saltos de punteros a página.

¡GRACIAS!