Pontificia Universidad Católica de Chile
IIC2413 - Bases de datos
Profesor: Adrián Soto
Ayudantes: Daniela Concha, Tamara Cucumides

Ayudantía I3

31 de mayo, 2019

---

1. Repaso

1.1. Clustered Index

- Define el orden de los datos.
- Sólo se puede tener **un** índice *clustered*.
- Entrega la tupla buscada.
- Ejemplo: Libreta de teléfonos.

1.2. Unclustered Index

- Crea la tabla complementaria con los punteros a los datos reales.
- Se pueden tener múltiples índices *unclustered*.
- Entrega un puntero a la tupla buscada, y por tanto requerirá un acceso a memoria adicional para obtener la tupla.
- Ejemplo: Índice por palabras al final de un libro.

1.3. Hash Index

- Se deben manejar colisiones, por ejemplo con las listas ligadas.
- Bueno para las *queries* de igualdad, malo para las de rangos.

$$\textbf{Costo de I/O} \sim \mathcal{O}\left(\frac{|\text{Records}|}{R \cdot P}\right)$$

$$R = \frac{\text{Records}}{\text{Páginas}} \qquad P = \frac{\text{Páginas}}{\text{Buckets}}$$

Nota: Por lo general se asume que el costo es igual a uno.

1.4. B+ Tree Index

- Mantiene balanceado el árbol para garantizar profundidad logarítmica.
- Bueno para *queries* de igualdad (no tanto como hash index) y **rangos de datos**.

$$\textbf{Costo de I/O} \sim \mathcal{O}\left(\log_{R/2}\left(\frac{2 \cdot |\text{Records}|}{R}\right)\right)$$

$$R = \frac{\text{Records}}{\text{Páginas}}$$

1.5. Nested Loop Join

- Consiste en iterar sobre R y luego sobre S, teniendo que recorrer S tantas veces como tuplas tenga R.

$$\text{R} \bowtie \text{S} \rightarrow \textbf{Costo I/O} = \text{Costo(R)} + \text{Tuplas(R)} \cdot \text{Costo(S)}$$

1.5.1. Block Nested Loop Join

- Si S tiene muchas tuplas, lo anterior es ineficiente. Es mejor cargar un *buffer* con páginas de R e iterar S una sola vez para comparar con ese conjunto de tuplas. Luego volver a llenar el *buffer* y seguir hasta que no queden tuplas en R.

$$\text{R} \bowtie \text{S} \rightarrow \textbf{Costo I/O} = \text{Costo(R)} + \big(\text{Páginas(R)}/|\textit{Buffer}|\big) \cdot \text{Costo(S)}$$

1.6. External Merge Sort

- Gran cantidad de datos → No caben todos en la RAM.
- Cada fase requiere I/O de las N páginas a disco.
- *Run:* Colección de páginas ordenadas.
- Fase 0: Cada página es ordenada con algún algoritmo. (i.e. *quicksort*)
- Fase ≥ 1: Hacen *merge* de K *runs*.

$$\textbf{Costo I/O} = 2 \cdot \text{N} \cdot \#\text{Fases}$$

1.6.1. Algoritmo base

Buffer de $2 + 1$ páginas y *runs* iniciales de 1 página:

$$\textbf{\#Fases} = 1 + \lceil \log_2(N) \rceil$$

1.6.2. Algoritmo optimizado en I/O

Buffer de $B + 1$ páginas:

$$\textbf{\#Fases} = 1 + \left\lceil \log_B \left\lceil \frac{N}{B+1} \right\rceil \right\rceil$$

***Buffer* óptimo:**

$$\textbf{\#Fases} = 2 \rightarrow B \geq \sqrt{N}$$

Costo de I/O con *buffer* óptimo:

$$\textbf{Costo} = 4 \cdot N \cdot \textbf{\#Fases} = 4 \cdot N$$

Y si el último resultado no lo escribimos en disco:

$$\textbf{Costo} = 4 \cdot N - N = 3 \cdot N$$

Pregunta 1: Índices y algoritmos

Conceptual

a) ¿Puede haber más de un *clustered index* sobre una relación? ¿Qué ventaja/desventaja tendría esto?

b) ¿Puede haber más de un *unclustered index* sobre una relación? ¿Qué ventaja/desventaja tendría esto?

c) ¿Cómo podemos escoger sobre qué atributo hacer un *clustered index*? ¿Qué relevancia tiene esta elección?

d) ¿Qué diferencia hay entre un árbol binario corriente (B Tree) y un B+ Tree?

e) ¿Cómo se guardan los datos con un *hash index*? ¿Cuándo es conveniente usarlos?

f) Discuta si la siguiente afirmación es verdadera o falsa: *En comparación a tener los datos sin ordenar, un índice hace que* ***todas*** *las consultas sean más eficientes.*

Costos I/O

Considere el siguiente esquema:

```
Empleado(eid int PRIMARY KEY, nombre varchar(10), apellido varchar(10),
                salario int, departamento varchar(20))
```

Para cada una de las siguientes consultas, decida qué tipo de índice es el más adecuado (clustered, unclustered, B+ Tree, Hash Index, sin índice), indicando su costo en operaciones I/O. Explique su respuesta.

a) Obtener los datos de todos los empleados que trabajen en el departamento de `VENTAS`

b) Obtener los datos de todos los empleados cuyo salario esté entre 100.000 y 200.000

c) Obtener los datos de todos los empleados con `eid >100`

d) Obtener los datos del empleado con `eid = 1331`

Indicación: Considere que la relación empleado tiene 1.000.000 de tuplas y en cada página caben 100 tuplas. Puede considerar que un B+ Tree tiene altura h con páginas ocupadas a un 70 % y que un hash index no tiene overflow pages (solo 1 página por bucket). Si necesita más supuestos, explicítelos.

Pregunta 2: Transacciones (I3 2018-2)

Considere el Schedule del Cuadro 1. Diga si es o no *conflict serializable*. En caso de que no lo sea, explique por qué e indique cómo Strict-2PL puede resolver el problema si las transacciones llegan en ese orden.

T1	T2	T3
R(a)		
	R(b)	
		W(a)
		W(c)
R(c)		
	R(c)	

Cuadro 1: Schedule pregunta transacciones.

Pregunta 3: Logging (I3 2018-2)

a) Suponga que su sistema tuvo una falla. Al reiniciar el sistema, el sistema se encuentra con el *log file* que se muestra a continuación, en la tabla "Log Undo". Suponiendo que la polıtica de *recovery* es la de *Undo Logging*, indique hasta qué parte del *log* se debe leer, qué variables deben deshacer sus cambios (y cuáles no cambian) y con qué valor quedan.

b) Suponga que su sistema tuvo una falla. Al reiniciar el sistema, el sistema se encuentra con el *log file* que se muestra a continuación, en la tabla "Log-Redo". Suponiendo que la polıtica de *recovery* es la de *Redo Logging*, indique hasta qué parte del *log* se debe leer, qué variables deben deshacer sus cambios (y cuáles no cambian) y con qué valor quedan.

c) Para el último caso, indique cómo sería la recuperación si no estuviese la linea de `<END CKPT>`.

Log Undo
<START T1>
<START T2>
<T1, a, 4>
<T2, b, 5>
<T2, c, 10>
<COMMIT T1>
<START CKPT (T2)>
<START T3>
<START T4>
<T3, a, 10>
<T2, b, 7>
<T4, d, 5>
<COMMIT T2>
<END CKPT>
<START T5>
<COMMIT T3>
<T5, e, -3>

Log Redo
<START T1>
<T1, a, 1>
<COMMIT T1>
<START T2>
<T2, b, 2>
<T2, c, 3>
<COMMIT T2>
<START T3>
<T3, a, 10>
<START CKPT (T3)>
<T3, d, 23>
<START T4>
<END CKPT>
<COMMIT T3>
<T4, e, 11>

Pregunta 4: MongoDB (I3 2017-2)

Piense en una base de datos en MongoDB con dos colecciones, una de usuarios de una red social y otra de estados publicados en ella:

```
// Usuarios
{
  "uid": 1,
  "name": "Marcelo Saldías",
  "age": 21,
  "description": {
    "estudia_en": "PUC",
    "Animes favoritos": ["Haikyuu", "Love Live"]
  }
}
// Estados
{
  "eid": 1,
  "uid": 1,
  "content": "Grande FUEL #BurnBlue"
  "likes": [1, 4, 7]
}
```

En que los `usuarios` tienen anidado un documento `description` que indica dónde estudian y sus `animes favoritos` (con un JSON Array que contiene los labels de los animes). Además cada `estado` emitido por el usuario posee un arreglo con los id de los usuarios que le han dado `like` a ese estado.

Se pide que entregue la siguiente consulta en MongoDB (sin usar un lenguaje de programación externo):

a) Entregue el id y nombre de todos los usuarios que estudian en la `"PUC"` y tienen más de 20 años.

Ahora utilizando PyMongo se pide un procedimiento que entregue lo siguiente:

b) Entregue cada a usuario junto al número total de `likes` que tiene.

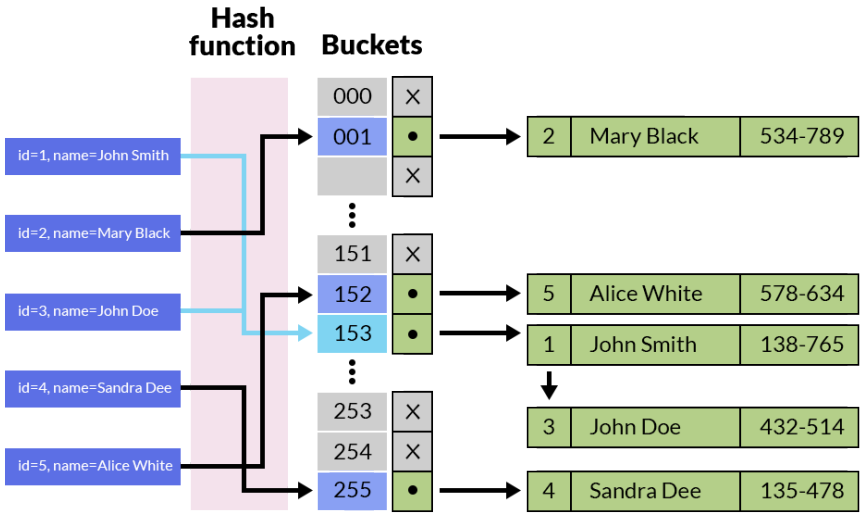
Ahora se pide que entregue una secuencia de pasos para crear el índice correspondiente, junto al procedimiento para responder la siguiente consulta:

c) Imprima el `id` de todos los estados que contienen el hashtag `"#BurnBlue"` pero no el hashtag `"#ShockTheWorld"`, junto al nombre de todos los usuarios que le dieron like al estado.

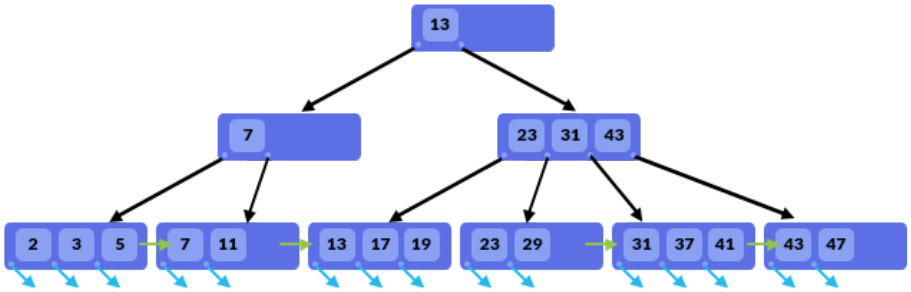
¡Vamos por el 7!

Ayuda Repaso

Hash Index



B+ Tree



Clustered y Unclustered Index