

Clase 8:

# SQL III y Optimización de Consultas

Sebastián Ferrada  
sferrada@dcc.uchile.cl

CC3201-Bases de Datos, 2016-2

Muy lindas tus consultas SQL y todo... ¿pero cómo agrego tuplas a mi base de datos?

*¿Cómo defino mis tablas y sus atributos?*

*¿Cómo borro o modifico mis tuplas?*

*¿Cómo creo un esquema?*

xqxqxQxqXq?

# SQL como Lenguaje de Definición de Datos

- Comandos para crear y modificar esquemas y tablas
- Instrucciones para insertar tuplas en la BD
- Edición y borrado de valores

# Crear esquemas

- `CREATE SCHEMA [nombre]`
  - crea un nuevo esquema con un nombre dado
- `SET search_path TO [nombres]`
  - indica el o los esquemas a utilizar

# Definir tablas

- `CREATE TABLE [nombre] (<atrib><tipo>)`

- Por ejemplo, la tabla de cervezas se define:

```
CREATE TABLE cerveza (  
    nombre varchar(20) PRIMARY KEY,  
    tipo varchar(2) NOT NULL,  
    grados float DEFAULT 0.0,  
    origen varchar(40) NOT NULL,  
);
```

# Definir tablas

- `CREATE TABLE [nombre] (<atrib><tipo>)`

- Por ejemplo: Definición de atributos y tipos (char(largo), float, boolean) define:

```
CREATE TABLE ejemplo (  
  nombre varchar(20) PRIMARY KEY,  
  tipo varchar(2) NOT NULL,  
  grados float DEFAULT 0.0,  
  origen varchar(40) NOT NULL,  
);
```

# Definir tablas

- `CREATE TABLE` [nombre]

Definir la llave primaria. Para llaves compuestas se usa `PRIMARY KEY(a,...)`

- Por ejemplo, la tabla de cerveza

```
CREATE TABLE cerveza (  
    nombre varchar(20) PRIMARY KEY,  
    tipo varchar(2) NOT NULL,  
    grados float DEFAULT 0.0,  
    origen varchar(40) NOT NULL,  
);
```

# Definir tablas

- `CREATE TABLE [nombre] (<atrib><tipo>)`

- Por ejemplo, la tabla de cervezas se define:

```
CREATE TABLE cerveza (  
  nombre varchar(25) NOT NULL,  
  tipo varchar(2) NOT NULL,  
  grados float DEFAULT 0.0,  
  origen varchar(25) NOT NULL,  
);
```

Prohibir valores nulos

Usar valores por defecto

# Modificar tablas

- ¿Eliminarla por completo?

```
DROP TABLE cerveza
```

- ¿Eliminar un atributo?

```
ALTER TABLE cerveza DROP grados
```

- ¿Agregar un atributo?

```
ALTER TABLE cerveza ADD COLUMN grados
```

# Definir tablas

- Y las llaves foráneas

```
CREATE TABLE en-stock
```

```
nombre varchar(255) PRIMARY KEY,
```

```
cantidad int CHECK (cantidad >= 0),
```

```
precio float CHECK (precio > 0.0));
```

```
ALTER TABLE en-stock ADD CONSTRAINT nfk
```

```
FOREIGN KEY (nombre) REFERENCES cerveza(nombre)
```

```
ON UPDATE xxx
```

```
ON DELETE xxx
```

**CHECK**(condición) verifica que la condición sea válida antes de insertar una tupla

valores?

# Definir tablas

- Y las llaves foráneas? cómo permitir solo ciertos valores?

```
CREATE TABLE en-stock(
```

```
    nombre varchar(50)
```

```
    cantidad int
```

```
    precio float
```

Agregar restricción: nombre  
referencia la PK de tabla cerveza

```
),
```

```
);
```

```
ALTER TABLE en-stock ADD CONSTRAINT nfk
```

```
FOREIGN KEY (nombre) REFERENCES cerveza(nombre)
```

```
ON UPDATE xxx
```

```
ON DELETE xxx
```

# Llaves foráneas

```
ALTER TABLE tabla  
ADD CONSTRAINT nombre_cond  
FOREIGN KEY (tabla.atrib)  
REFERENCES tabla_ref(atrib)  
ON UPDATE opción  
ON DELETE opción
```

opción ∈ {RESTRICT, CASCADE, SET NULL}

# Insertar valores

```
INSERT INTO tabla(atrib) VALUES(...)
```

Ahora con cervezas:

```
INSERT INTO cerveza VALUES  
( 'Austral Lager', 'Lager', 4.6, 'Punta Arenas' )
```

```
INSERT INTO cerveza(nombre, tipo, origen) VALUES  
( 'Szot Amber', 'Ale', 'Talagante' )
```

# Insertar valores

- También se pueden agregar valores obtenidos de una consulta

```
INSERT INTO profes_letales VALUES(  
    SELECT profesor  
    FROM curso  
    WHERE reprobación > 0.5);
```

# Editar tuplas

```
UPDATE tabla  
SET atrib = valor  
WHERE condición
```

# Editar tuplas

Nombre	Tipo	Grados	Origen
Austral Lager	Lager	4,6	Punta Arenas
Szot Amber	Ale	5,8	Talagante
Kunstmann Bock	Bock	5,3	Valdivia

```
UPDATE cerveza  
SET tipo='Lager'  
WHERE origen='Valdivia'
```

Nombre	Tipo	Grados	Origen
Austral Lager	Lager	4,6	Punta Arenas
Szot Amber	Ale	5,8	Talagante
Kunstmann Bock	Lager	5,3	Valdivia

# Borrar tuplas

**DELETE FROM** tabla **WHERE** condición

Nombre	Tipo	Grados	Origen
Austral Lager	Lager	4,6	Punta Arenas
Szot Amber	Ale	5,8	Talagante
Kunstmann Bock	Bock	5,3	Valdivia

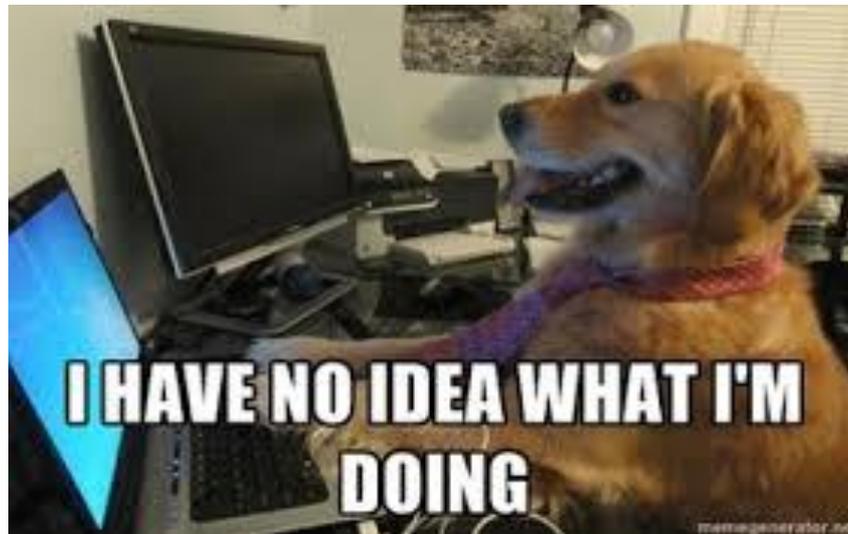
**DELETE FROM** cerveza **WHERE** grados<5

Nombre	Tipo	Grados	Origen
Szot Amber	Ale	5,8	Talagante
Kunstmann Bock	Bock	5,3	Valdivia

# Procesamiento de Consultas

# Lenguaje declarativo

- En SQL uno escribe lo que espera obtener de la consulta, pero no cómo obtener el resultado

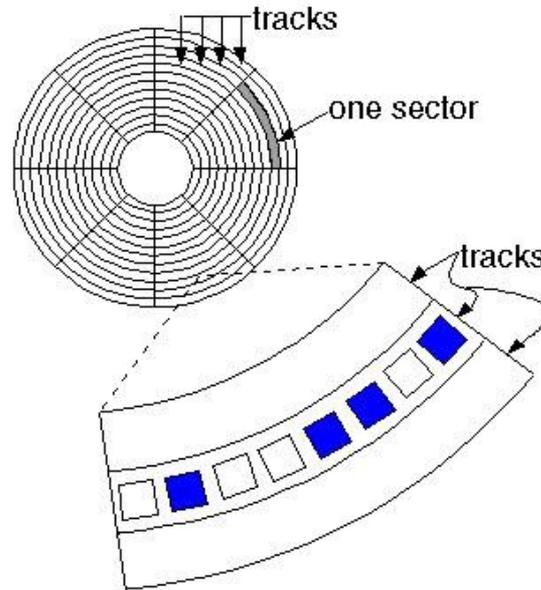


# Sistema de costos

- Las tuplas de una base de datos se almacenan en disco
- Las operaciones más caras son las que requieren lectura/escritura desde/hacia disco
- Las operaciones en RAM son despreciables, entonces un algoritmo es tan costoso como la cantidad de accesos a disco

# Lectura desde Memoria Secundaria

- Cuando se lee/escribe en disco se hace por bloques
- El tamaño del bloque depende de la máquina que consideremos y se denomina  $B$
- El tamaño de la memoria principal es  $M$



# Lectura desde memoria secundaria

- Entonces leer/escribir  $n$  tuplas desde disco con bloque  $B$  cuesta  $n/B$
- Llamaremos  $B(R)$  a la cantidad de bloques de memoria que cuesta leer la tabla  $R$ , es decir:

$$B(R) = \frac{|R|}{B}$$

- En memoria principal caben  $M/B$  bloques

# ¿Cómo procesar una consulta?

- Búsqueda secuencial
- Loops anidados
- Merge sort join
- Hash join
- Índices

# Búsqueda secuencial

- Se leen todas las tuplas de la relación  $R$
- Se seleccionan las que cumplen la condición
- Se proyectan las columnas necesarias
- Costo:  $B(R)$

# Búsqueda secuencial

```
cc3201=> EXPLAIN SELECT * FROM película WHERE calificación > 9;  
          QUERY PLAN
```

```
-----  
Seq Scan on "película" (cost=0.00..5.12 rows=2 width=27)  
  Filter: ("calificación" > 9::double precision)  
(2 rows)
```

```
cc3201=> EXPLAIN SELECT * FROM actor WHERE género = 'F';  
          QUERY PLAN
```

```
-----  
Seq Scan on actor (cost=0.00..274.45 rows=3748 width=18)  
  Filter: ("género" = 'F'::bpchar)  
(2 rows)
```

# Loop anidado

- $R \bowtie_c S$
- Por cada bloque de R y por cada tupla  $r$  en el bloque:
  - Por cada bloque de S y cada tupla  $s$  en el bloque:
    - Agregar  $(r, s)$  al resultado si  $r$  y  $s$  cumplen  $c$
- Costo:  $B(R) + |R| \cdot B(S)$
- Memoria:  $3B$

Alguna idea para minimizar esto?

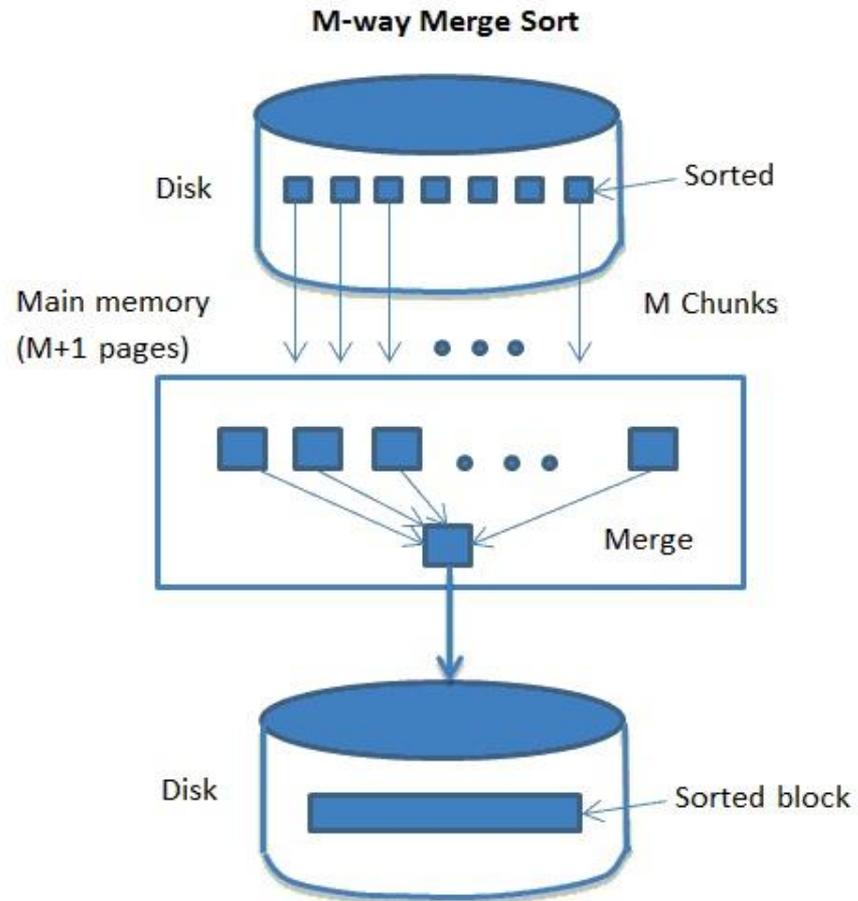
# Loop anidado

- Optimizaciones:
  - Terminar antes si se está haciendo equi-join con la llave de  $S$  (y si están ordenadas)
    - Reduce hasta la mitad de las comparaciones en el mejor caso
  - Guardar todos los bloques posibles de  $R$  en memoria
    - Costo:  $B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S)$
    - Memoria usada:  $M$  (lo más posible)

# Mergesort

- Para hacer mergesort en memoria secundaria, se leen bloques de memoria secundaria y se ordenan
- Luego se toman dos bloques ordenados y se mezclan
- Costo de hacer mergesort:  $B(R) \cdot \log_M B(R)$
- Costo en memoria: M

# Mergesort



# Merge join ordenado

- $R \bowtie_{R.A=S.B} S$
- Ordenar R y S según sus atributos de join
- Mezclar todas las tuplas  $r$  y  $s$  mientras tengan  $r.A = s.B$ 
  - Si  $r.A > s.B$  avanzar en  $s$
  - Si  $r.A < s.B$  avanzar en  $r$
  - Si no avanzar ambos
- Costo: ordenar +  $B(R) + B(S)$
- Peor caso:  $B(R) \cdot B(S)$  cuando todo calza



# Merge join ordenado - Ejemplo

$R:$	$S:$	$R \triangleright \triangleleft_{R.A=S.B} S:$
$\Rightarrow r_1.A = 1$	$\Rightarrow s_1.B = 1$	$r_1 s_1$
$\Rightarrow r_2.A = 3$	$\Rightarrow s_2.B = 2$	$r_2 s_3$
$r_3.A = 3$	$\Rightarrow s_3.B = 3$	$r_2 s_4$
$\Rightarrow r_4.A = 5$	$s_4.B = 3$	$r_3 s_3$
$\Rightarrow r_5.A = 7$	$\Rightarrow s_5.B = 8$	$r_3 s_4$
$\Rightarrow r_6.A = 7$		$r_7 s_5$
$\Rightarrow r_7.A = 8$		

# Merge join ordenado

```
cc3201=> EXPLAIN SELECT DISTINCT p1.a_nombre FROM personaje p1, personaje p2 WHERE p1.a_nombre = p2.a_nombre AND p1.p_nombre = p2.p_nombre AND p1.personaje <> p2.personaje;
```

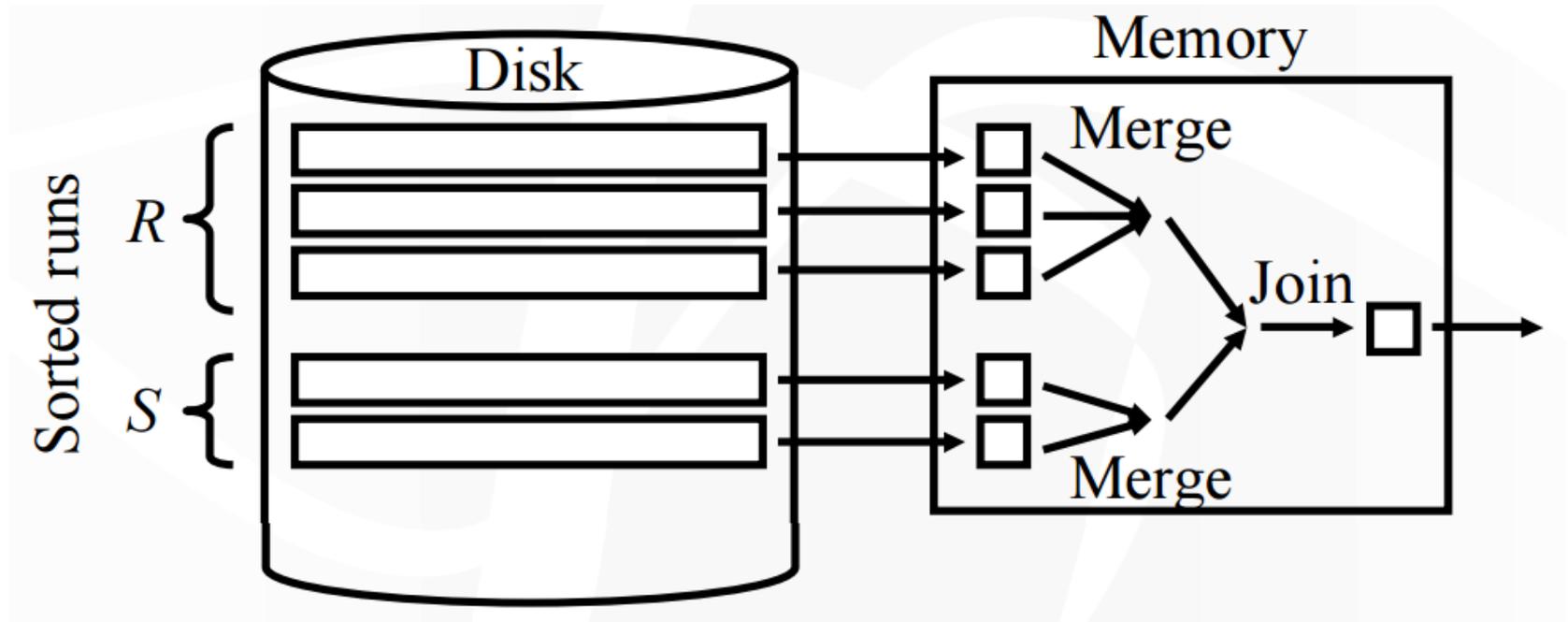
## QUERY PLAN

```
-----  
Unique (cost=0.00..3922.04 rows=141 width=16)  
-> Merge Join (cost=0.00..3921.69 rows=141 width=16)  
    Merge Cond: (((p1.a_nombre)::text = (p2.a_nombre)::text) AND ((p1.p_nombre)::text = (p2.p_nombre)::text))  
    Join Filter: ((p1.personaje)::text <> (p2.personaje)::text)  
    -> Index Scan using personaje_pkey on personaje p1 (cost=0.00..1873.82 rows=17229 width=47)  
    -> Index Scan using personaje_pkey on personaje p2 (cost=0.00..1873.82 rows=17229 width=47)  
(6 rows)
```

# Merge join - Optimización

- Podemos mezclar el join con la fase de merge del ordenamiento
- El ordenamiento produce bloques con *runs* ordenados de R y S
- Mezclar los *runs* de R, los de S y los del resultado al mismo tiempo

# Merge join - Optimización



# Merge join - Optimización

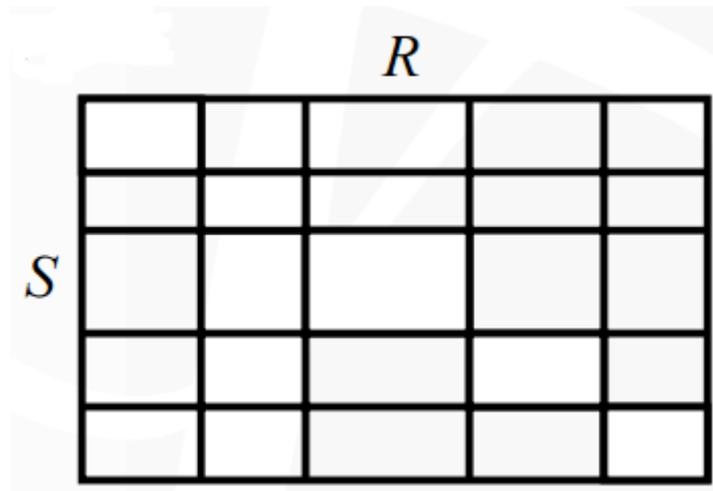
- Costo:  $3 \cdot (B(R) + B(S))$
- Memoria requerida: Para hacer el merge de una pasada necesitamos memoria para tener un bloque de cada *run* simultáneamente

$$M > \frac{B(R)}{M} + \frac{B(S)}{M}$$

$$M > \sqrt{B(R) + B(S)}$$

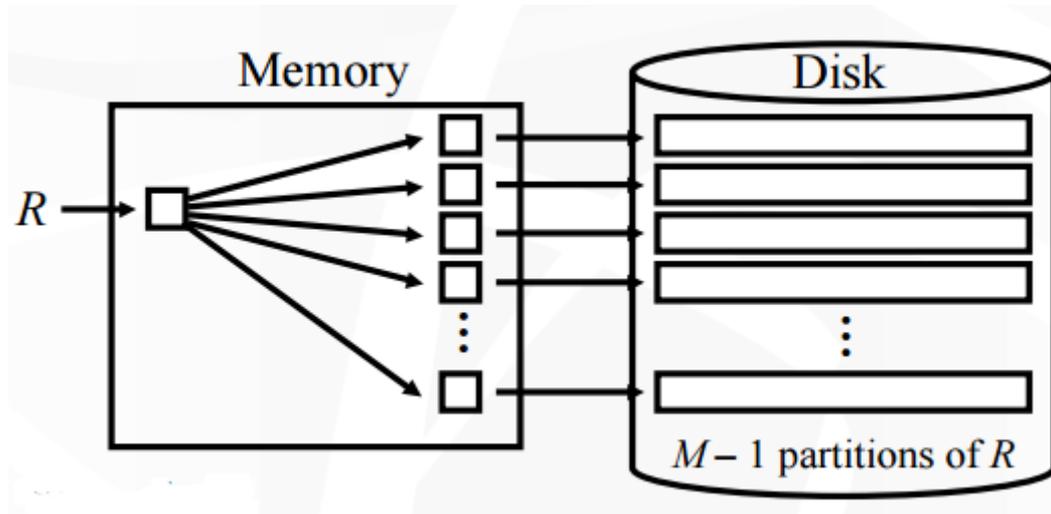
# Hash join

- $R \bowtie_{R.A=S.B} S$
- Idea:
  - Particionar R y S según el hash de los atributos de join (R.A y S.B)
  - Si dos tuplas  $r$  y  $s$  quedan en diferentes particiones, no son parte del resultado del join



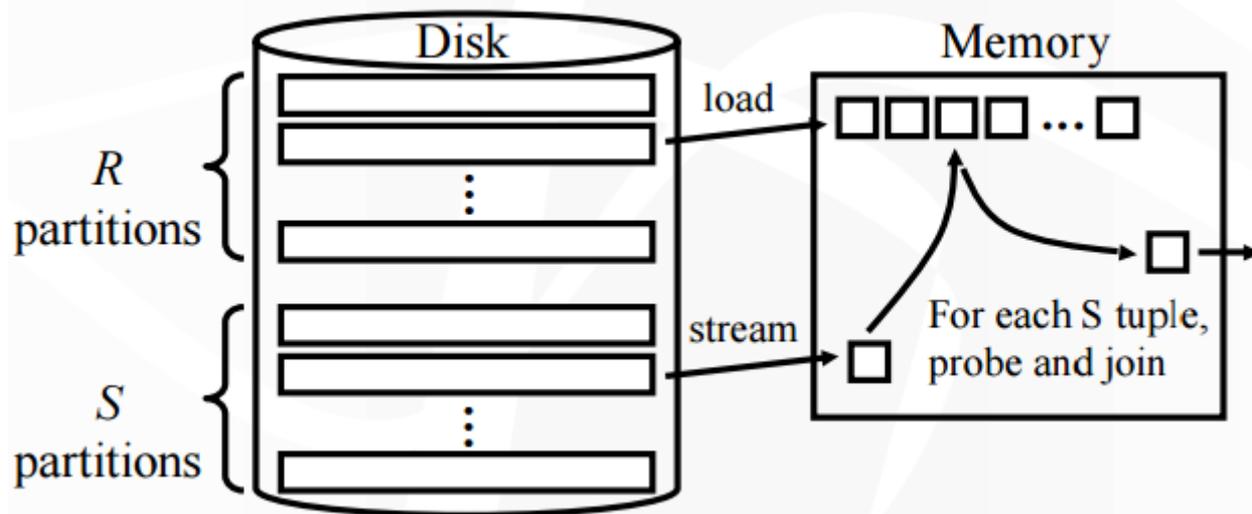
# Hash join - Particionamiento

- Particionar las tablas según la función de hash aplicada a cada uno de los atributos de join



# Hash join - Sondeo

- Para cada partición de R, probar con la partición respectiva de S y armar el join



# Hash join

- Costo:  $3 \cdot (B(R) + B(S))$
- Requerimiento de memoria: Necesitamos memoria suficiente para almacenar una partición completa

$$M - 1 \geq \frac{B(R)}{M - 1}$$

$$M > \sqrt{B(R)}$$

Siempre podemos elegir la tabla más pequeña para ahorrar memoria

¿¿¿Qué hacer si no cabe una partición entera en memoria!!??

# ¿Hash o Merge join?

- Ambos tienen el mismo costo de acceso a disco
- Hash requiere menor memoria y gana sobretodo cuando las tablas tienen tamaños muy distintos
- Hash join depende de la calidad de la función de hash usada: pueden obtenerse particiones desbalanceadas
- Merge join puede modificarse para no-equijoins
- Merge join gana si alguna de las tablas ya está ordenada
- Merge join gana si los resultados se necesitan en orden

# ¿Entonces nunca se usa loop anidado?

- Puede ser mejor si muchas tuplas se reúnen:
  - Ejemplo: no-equijoins no son muy selectivos

```
cc3201=> EXPLAIN SELECT P1.nombre, P2.nombre FROM película P1, película P2 WHERE
P1.calificación > P2.calificación
cc3201-> ;
```

## QUERY PLAN

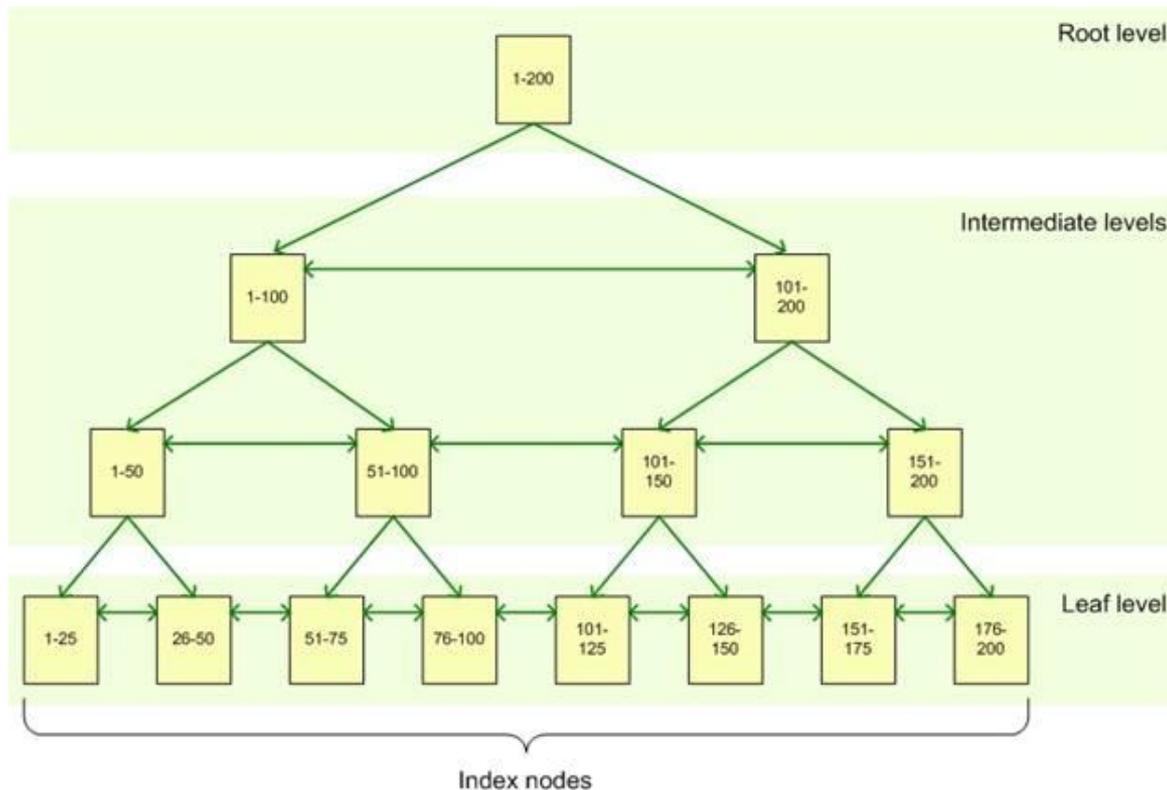
```
-----
Nested Loop (cost=0.00..947.12 rows=20833 width=34)
  Join Filter: (p1."calificación" > p2."calificación")
  -> Seq Scan on "película" p1 (cost=0.00..4.50 rows=250 width=25)
  -> Materialize (cost=0.00..5.75 rows=250 width=25)
      -> Seq Scan on "película" p2 (cost=0.00..4.50 rows=250 width=25)
(5 rows)
```

- Es necesario para predicados de caja negra:
  - Ejemplo: `WHERE user_function(R.A, S.B)`

Indexamiento

# ¿Qué es un índice?

- Una estructura para realizar búsquedas de forma más eficiente

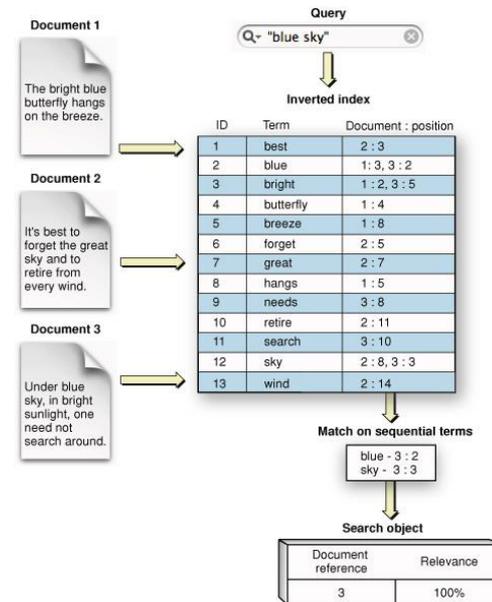
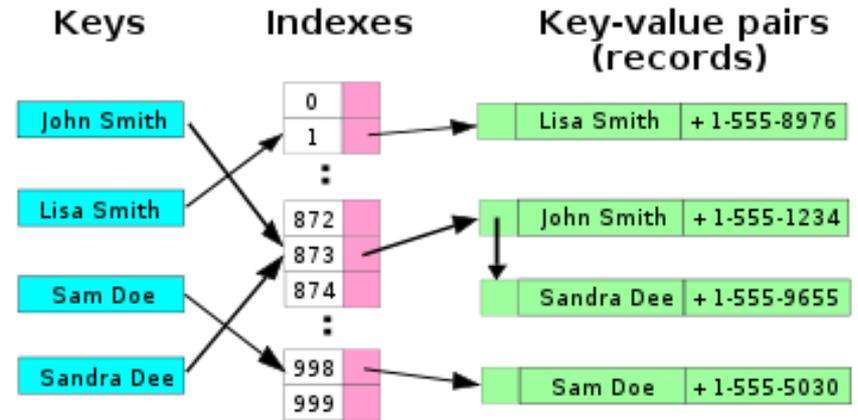
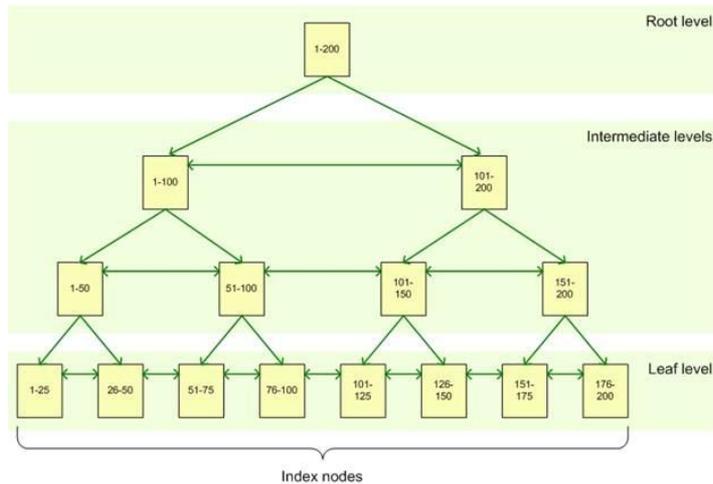


# Índices en Bases de Datos

- Son estructuras que toman el par (llave, atributo) de una tabla para agilizar la búsqueda
- Se busca el atributo en el índice y se retornan las llaves de los elementos encontrados
- Se buscan las llaves en la tabla
- OJO, el mal uso de los índices puede provocar una PEOR performance de la query

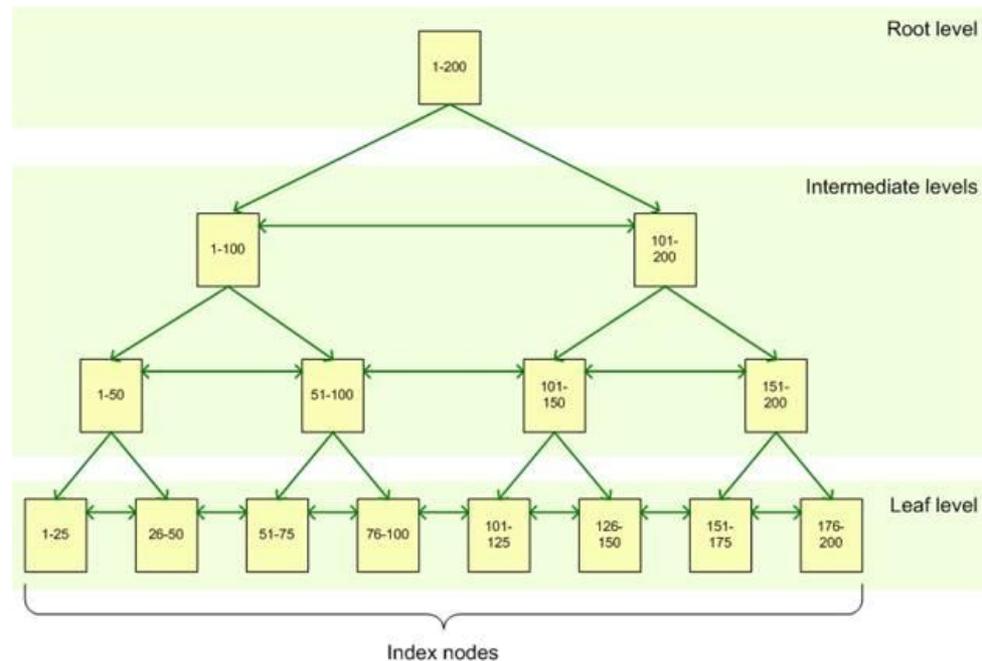
Entonces, hay que usar índices dependiendo de la query a optimizar

# Tipos de índices



# Índices B+

- Árbol balanceado: garantiza buena performance
- Basado en memoria secundaria: un nodo por bloque



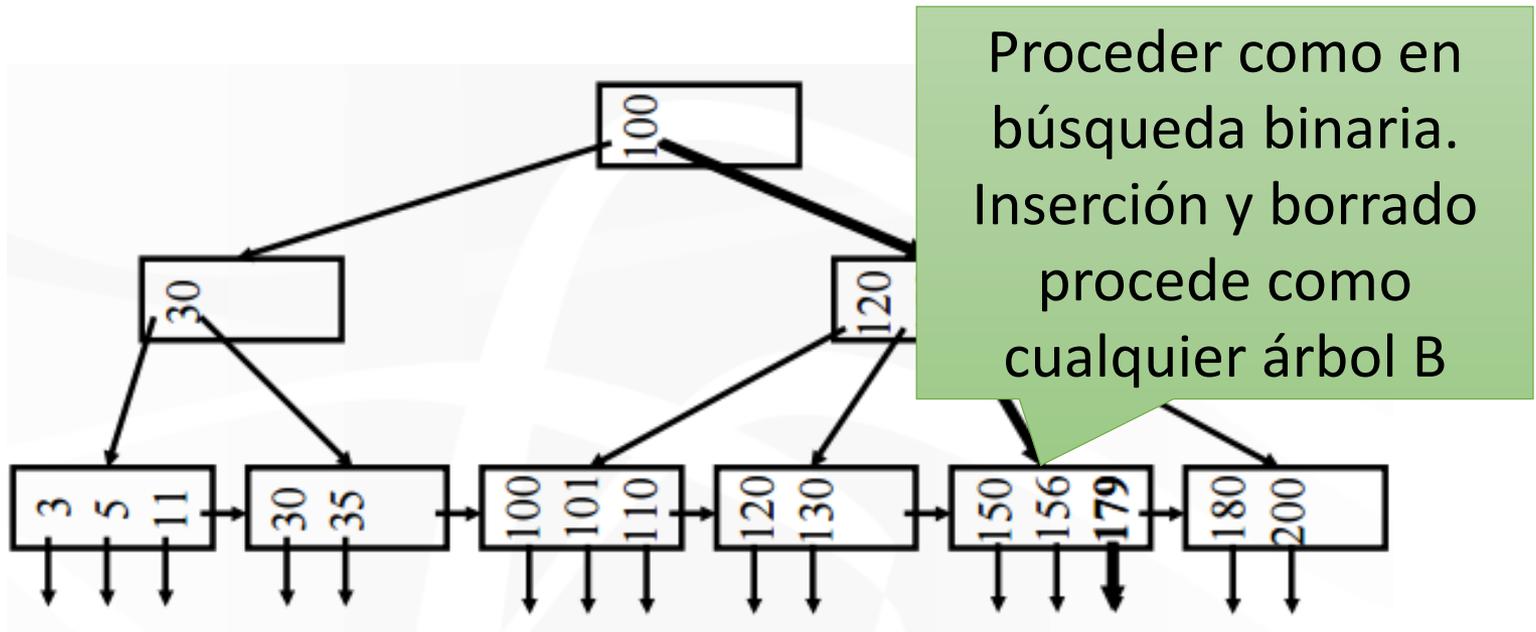
# Índices B+

- Todas las hojas al mismo nivel
- Cada nodo está lleno hasta la mitad al menos
- Para qué tipo de queries es mejor?

Queries por rango o  
queries para valores específicos

# Índices B+

- Ejemplo: `SELECT * FROM R WHERE k=179`



# Índices B+ - Performance

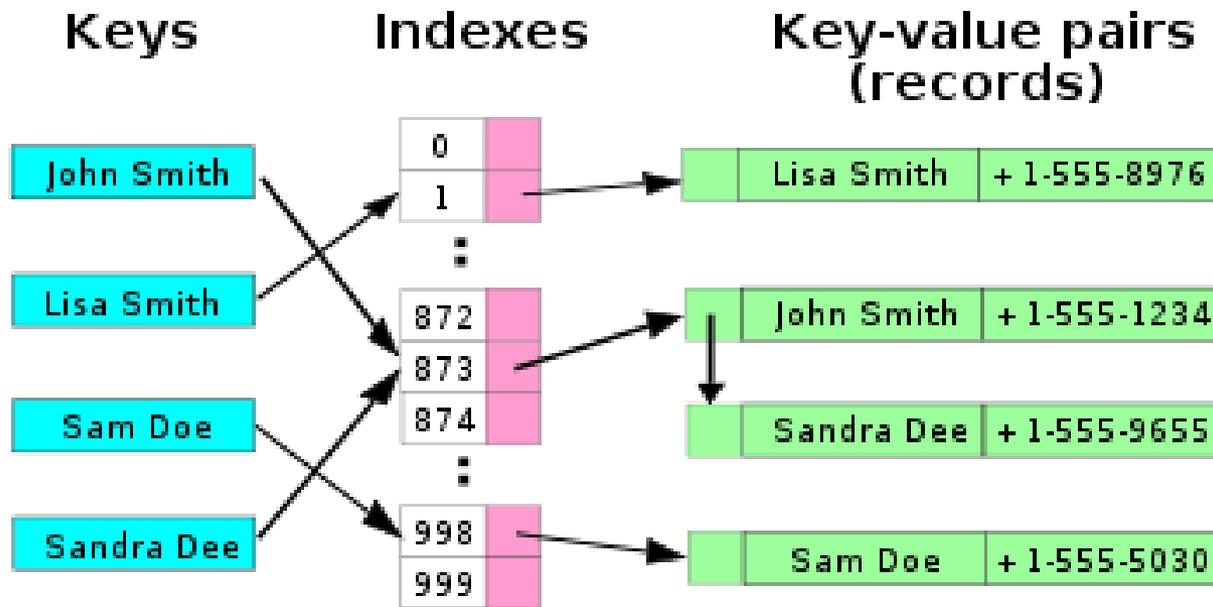
- Accesos a disco:  $O(\log_B |R|)$  (la altura del árbol)
  - $\log_B |R|$  para buscar los registros
  - 1 o 2 para manipular las tuplas
  - menos 1 si se cachea la raíz en memoria
- Construir desde cero o agregar  $n$  registros a la vez toma  $n \log(n)$

# Índices B+ en la práctica

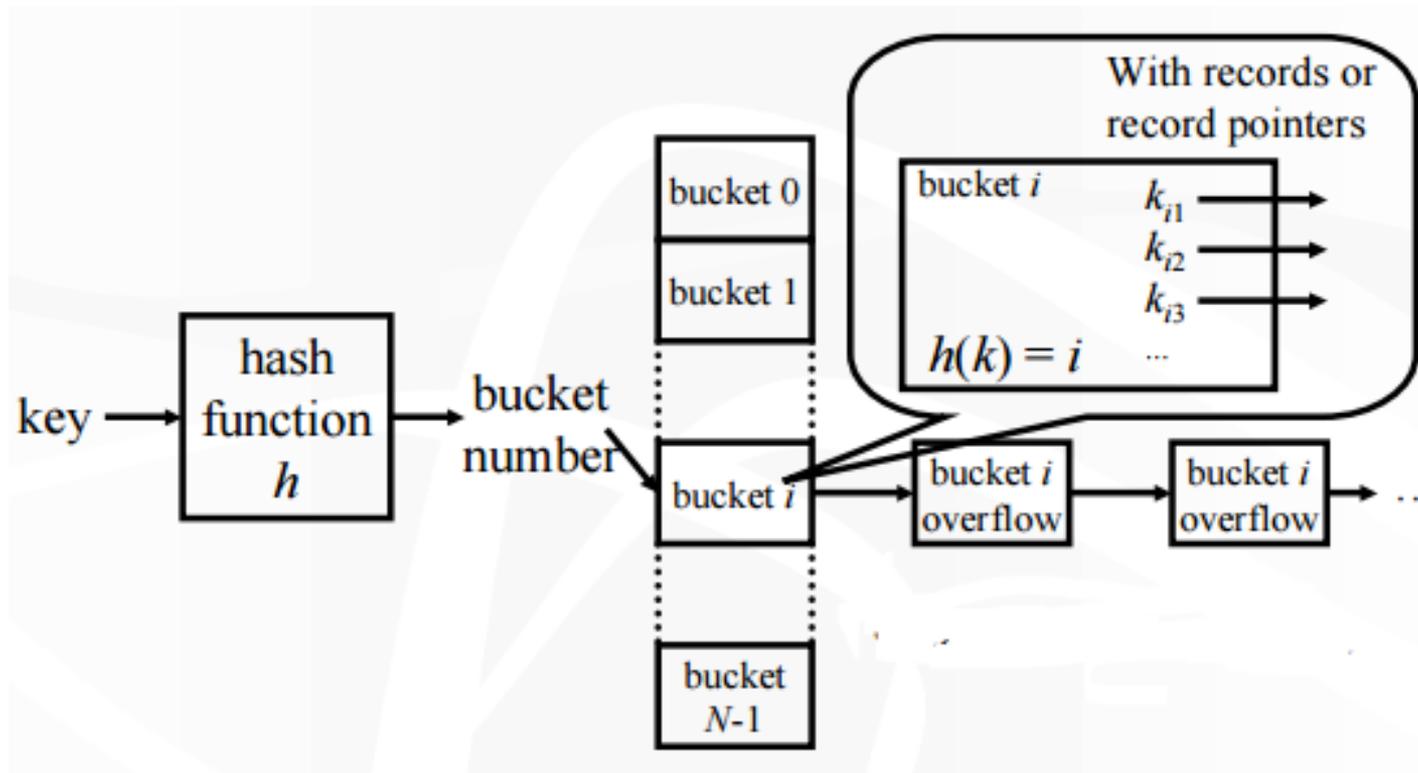
- La mayoría de los SADB crean índices B+ para las llaves primarias de las tablas
- Generalmente es el índice por defecto si no se especifica otro
- Por qué usar B+ y no B??

Tener tuplas en los nodos internos  
aumenta el tamaño del árbol!

# Índices de Hash



# Hash estático



# Hash estático - Performance

- Depende de la función de hash
  - Caso ideal:  $O(1)$
  - Peor caso: todas las llaves son enviadas al mismo índice
- Mantener 50% – 80% de ocupación
- ¿Cómo escalar el índice?
  - Hashing extensible
  - Hashing lineal

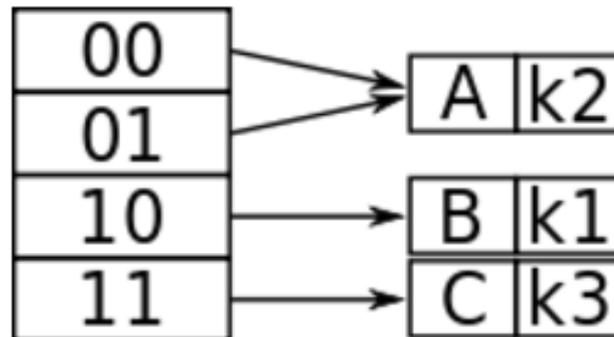
# Hashing extensible

- Usar solo los  $i$  primeros bits de la función de hash
- Si hay colisiones, usar un bit adicional

$$h(k_1) = 100100$$

$$h(k_2) = 010110$$

$$h(k_3) = 110110$$



# Hashing extensible

- Pros:
  - Maneja el crecimiento de los archivos
  - No requiere re-armar la tabla
- Cons:
  - Agrega un nivel de indirección
  - siempre se agrega el doble de “buckets” -> menor ocupación
  - Pese eso, a veces doblar no es suficiente!

# Hashing lineal

- Se inicia con una cantidad fija de buckets, cada bucket puede almacenar cierto número de entradas
- Si un bucket se rebalsa, se agrega uno nuevo y se recalculan las funciones de hash del  $i$ -ésimo bucket.
- El bucket que se divide se va turnando, para mantener la consistencia
- El nivel afectado gana un bucket de rebalse

# Hashing lineal

bucket#	primary pages	overflow pages
0 P →	4   8   12   16	
1	1   5	
2	6   10   22	
3	3   7   15   19	

---

bucket#	primary pages	overflow pages
0	8   16	
1 P →	1   5	
2	6   10   22	
3	3   7   15   19	→ <b>11</b>

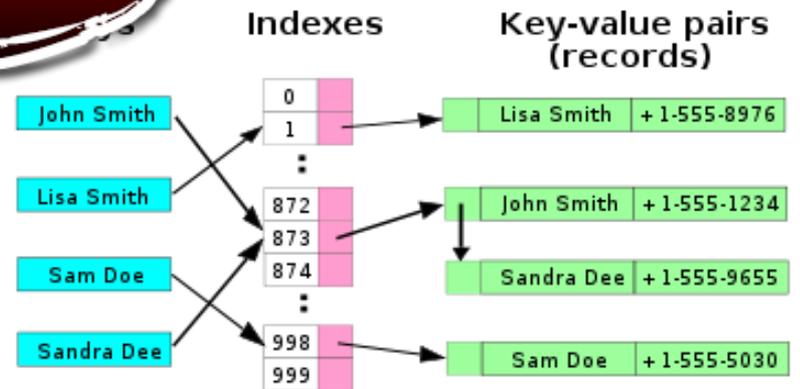
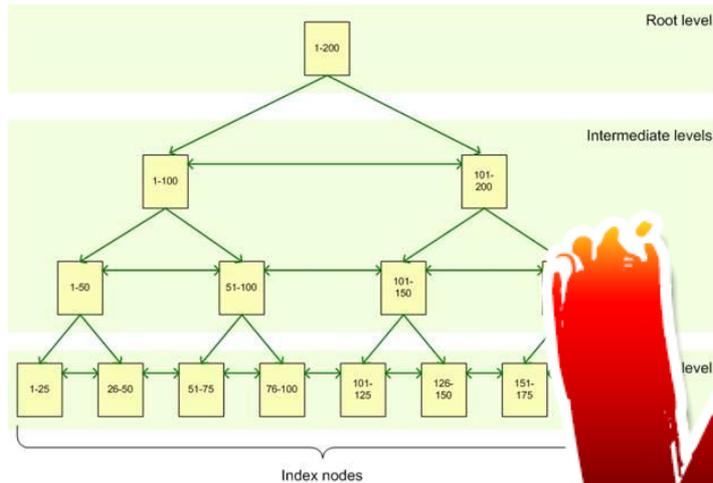
---

4	④   ⑫	
---	-------	--

# Hashing lineal - Performance

- Pros:
  - Maneja el problema de archivos en crecimiento
  - No tiene un nivel extra de indirección
  - No requiere re-hashing completo
- Cons:
  - Tiene cadenas de desborde, luego no es  $O(1)$
  - No siempre se divide el bucket que se desborda, entonces podrían requerirse varios bloques de disco

# Hashing vs B+



# Hashing vs B+

- Índices de hash son más rápidos en promedio, **pero el peor caso es muy malo!**
- B+ garantizan performance proporcional al tamaño del árbol, **generalmente es pequeño!**
- Hashing destruye el orden de los atributos
- B+ proveen orden y son útiles para consultas por rango
- Hashing puede ser bueno para consultas R.A=constante

# Índices invertidos

- Problema: encontrar los documentos en los que aparece una palabra en particular



Search Google or type URL



# Índices invertidos

- Idea ingenua: Matriz de aparición de términos

All documents

All keywords

	Document 1	Document 2	Document 3	...	Document <i>n</i>
"a"	1	1	1	...	1
"cat"	1	1	0	...	0
"database"	0	0	1	...	0
"dog"	0	1	0	...	1
"search"	0	0	1	...	0
...	...	...	...	...	...

# Índices invertidos

- Eventualmente la matriz puede ser demasiado grande
- Índices invertidos mantienen dos filas
- <keyword, document-id-list>
  - <'a'; 1, 5, 6>
  - <'hola'; 3, 5>
  - ...

# Índices invertidos

- Permiten buscar los documentos que contienen palabras claves
- Permiten conjunción, disyunción y negación
  - 'bases' AND 'datos'
  - 'jugo' OR 'bebida'
  - 'vegetales' AND NOT 'carne'

# Crear índices en SQL

```
CREATE INDEX nombre ON tabla(attr) USING  
method
```

- nombre: el nombre del índice
- tabla(attr): la tabla y atributos sobre los que se construirá el índice
- method: puede ser b-tree (por defecto), hash, GIN, etc

# Crear índices en SQL

- Para filtrar actores por género:

```
CREATE INDEX gen_idx ON actor USING hash(género)
```

```
cc3201=# EXPLAIN SELECT * FROM actor WHERE género = 'F';
          QUERY PLAN
-----
Seq Scan on actor  (cost=0.00..274.45 rows=3748 width=18)
  Filter: ("género" = 'F'::bpchar)
(2 rows)

Time: 0.452 ms
cc3201=# SELECT * FROM actor WHERE género = 'F';
Time: 5.828 ms
```

```
cc3201=# CREATE INDEX gen_idx ON actor USING hash(género);
CREATE INDEX
Time: 98.741 ms
cc3201=# EXPLAIN SELECT * FROM actor WHERE género = 'F';
          QUERY PLAN
-----
Bitmap Heap Scan on actor  (cost=133.30..271.15 rows=3748 width=18)
  Recheck Cond: ("género" = 'F'::bpchar)
  -> Bitmap Index Scan on gen_idx  (cost=0.00..132.36 rows=3748 width=0)
       Index Cond: ("género" = 'F'::bpchar)
(4 rows)

Time: 0.578 ms
cc3201=# SELECT * FROM actor WHERE género = 'F';
Time: 4.183 ms
```

# Crear índices en SQL

- Usar Primary Key index para encontrar una tupla en particular:

```
EXPLAIN SELECT * FROM personaje
WHERE p_año=1994 AND p_nombre='Pulp_Fiction'
AND a_nombre='Thurman, Uma' AND personaje='Mia Wallace';
```

```
Index Scan using personaje_pkey on personaje (cost=0.00..8.29 rows=1 width=49)
  Index Cond: (((a_nombre)::text = 'Thurman, Uma'::text) AND ((p_nombre)::text = 'Pulp Fiction'::text)
AND ("p_año" = 1994) AND ((personaje)::text = 'Mia Wallace'::text))
```

# OJO con los índices

- Por mucho que el índice exista, no siempre será usado, pues si se requieren muchas tuplas, el sobre costo será demasiado

```
cc3201=# EXPLAIN SELECT * FROM actor WHERE left(nombre, 1) = 'F';
          QUERY PLAN
-----
Seq Scan on actor  (cost=0.00..311.14 rows=73 width=18)
  Filter: ("left"(nombre)::text, 1) = 'F'::text)
(2 rows)
```

```
cc3201=# EXPLAIN SELECT * FROM actor WHERE nombre = 'Jackson, Samuel L.';
          QUERY PLAN
-----
Index Scan using actor_pkey on actor  (cost=0.00..8.28 rows=1 width=18)
  Index Cond: ((nombre)::text = 'Jackson, Samuel L.'::text)
(2 rows)
```

¿Preguntas?

# Siguientes actividades

- Jueves 20, laboratorio 7
- Lunes 24, **CONTROL 1**
  - Se permite una hoja de apuntes manuscritos
  - Dura 1:30! sean puntuales 😊
- Temario:
  - Modelar bases de datos (Modelo E/R y Relacional)
  - Álgebra relacional
  - SQL I, II y III