

Optimización SQL II

Alex Di Genova

25/05/2023

Contenidos

- Indices II
- Optimización SQL II
- Informaciones

Repaso

Vistas

Ejemplo Join Anidado

- Usos:
 - Para almacenar consultas frecuentes o complejas.
 - Para crear versiones de tablas mas amigables al usuario (tiempo y fechas).

- Crear una vista de un join anidado

```
create view join_album_artista_track as select * from
artist natural join album join track using (albumid)
```

- Como desplegamos los elementos de la lista?

```
select * from join_album_artista_track limit 10;
```

- Como eliminamos la vista?

```
drop view join_album_artista_track;
```

```
[7] 1 %%sql
2 create view join_album_artista_track as select * from artist natural join album join track using (albumid)

* sqlite:///content/chinook-database/ChinookDatabase/DataSources/Chinook_Sqlite.sqlite
Done.
[]
```

```
[8] 1 %%sql
2 select * from join_album_artista_track limit 10;
```

ArtistId	Name	AlbumId	Title	TrackId	Name:1	MediaTypeId	GenreId	Composer
1	AC/DC	1	For Those About To Rock We Salute You	1	For Those About To Rock (We Salute You)	1	1	Angus Young, Malcolm Young, I
2	Accept	2	Balls to the Wall	2	Balls to the Wall	2	1	None
2	Accept	3	Restless and Wild	3	Fast As a Shark	2	1	F. Baltes, S. Kaufman, U. Dirks Hoffman
2	Accept	3	Restless and Wild	4	Restless and Wild	2	1	F. Baltes, R.A. Smith-Diesel, S. Dirkschneider & W. Hoffman
2	Accept	3	Restless and Wild	5	Princess of the Dawn	2	1	Deaffy & R.A. Smith-Diesel
1	AC/DC	1	For Those About To Rock We Salute You	6	Put The Finger On You	1	1	Angus Young, Malcolm Young, I
1	AC/DC	1	For Those About To Rock We Salute You	7	Let's Get It Up	1	1	Angus Young, Malcolm Young, I
1	AC/DC	1	For Those About To Rock We Salute You	8	Inject The Venom	1	1	Angus Young, Malcolm Young, I
1	AC/DC	1	For Those About To Rock We Salute You	9	Snowballed	1	1	Angus Young, Malcolm Young, I
1	AC/DC	1	For Those About To Rock We Salute You	10	Evil Walks	1	1	Angus Young, Malcolm Young, I

```
1 #drop view
2 %%sql
3 drop view join_album_artista_track;
```

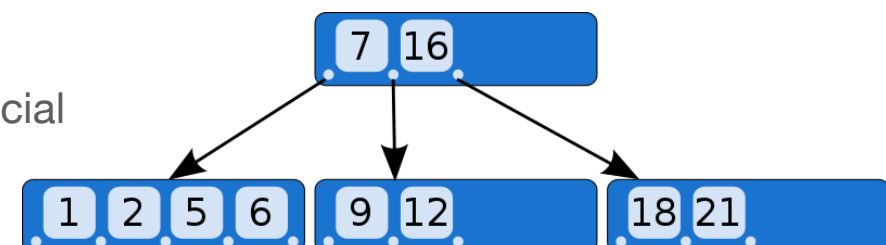
```
* sqlite:///content/chinook-database/ChinookDatabase/DataSources/Chinook_Sqlite.sqlite
Done.
[]
```

Indices

Optimización de consultas

- Sin índices, el motor de base de datos está obligado de revisar las tablas completas en cada consulta.
 - Los JOINS son costosos computacionalmente.
 - La idea es encontrar filas sin la necesidad de revisar toda la tabla.
- Cada índice agrega una carga adicional a INSERT, UPDATE y DELETE.
 - Debemos evaluar donde colocar los índices.
- Los índices no se pueden crear en Vistas.
- Como almacena los datos el motor SQL?
- Por defecto, cada tabla es almacenada utilizando una estructura indexada (B-Tree SQLite).
 - A medida que se insertan filas en el B-Tree, las filas se ordenan, organizan y optimizan, de modo que una fila con un ROWID específico y conocido se puede recuperar de manera relativamente directa y rápida.

Permite: búsquedas, inserciones, deletaciones y acceso secuencial
 $\text{Time}(n) = O(\log n)$



- Cuando creamos un índice, el sistema de base de datos crea otro B-Tree para almacenar los datos del índice.
- El nuevo B-Tree se ordena y organiza usando la columna o columnas que se especifican en la definición del índice (! ROWID).

Indices

Ejemplo

- Creamos una tabla con 4 x 1000 numeros.

```
create table tbl (a,b,c,d);
INSERT INTO tbl (a, b,c,d)
VALUES
(84,39,78,79),
(182,39,67,153),
(83,166,143,188),
(145,205,380,366),
(317,358,70,303), ...
```

```
1 %%time
2 %%sql
3 select * from tbl where a=29238;
```

```
* sqlite:///content/chinook-database/ChinookDatabase/DataSources/Chinook_Sqlite.sqlite
Done.
CPU times: user 5.2 ms, sys: 0 ns, total: 5.2 ms
Wall time: 7.23 ms
  a      b      c      d
29238 3171  23996 48794
29238 297097 765679 213680
```

```
[12] 1 %%time
      2 %%sql
      3 create index idx_tbl_a_b ON tbl(a,b)
```

```
* sqlite:///content/chinook-database/ChinookDatabase/DataSources/Chinook_Sqlite.sqlite
Done.
CPU times: user 9.09 ms, sys: 32 µs, total: 9.13 ms
Wall time: 22.4 ms
[]
```

```
1 %%time
2 %%sql
3 select * from tbl where a=29238;
```

```
[>] * sqlite:///content/chinook-database/ChinookDatabase/DataSources/Chinook_Sqlite.sqlite
Done.
CPU times: user 3.88 ms, sys: 0 ns, total: 3.88 ms
Wall time: 3.76 ms
  a      b      c      d
29238 3171  23996 48794
29238 297097 765679 213680
```

Indices II

Indices

SQLite3

- Busqueda
- Tablas sin indices

```
CREATE TABLE VentaFrutas(  
  Fruta TEXT,  
  Region TEXT,  
  Precio Double  
);
```

Rowid	Fruta	Ciudad	Precio
1	Naranja	ARI	1000
2	Manzana	IQQ	1500
4	Durazno	TAL	3000
5	Pera	SAN	800
8	Limon	ARI	500
19	Frutilla	IQQ	2500
20	Naranja	SAN	1800

`select precio from VentaFrutas Where fruta='Durazno'`

- **Revisión de toda la tabla (N)**

Indices

SQLite3

- Búsqueda por rowid
- Clave primaria

```
CREATE TABLE VentaFrutas(  
  Fruta TEXT,  
  Region TEXT,  
  Precio Double  
);
```

`select precio from VentaFrutas Where rowid=4`



Rowid	Fruta	Ciudad	Precio
1	Naranja	ARI	1000
2	Manzana	IQQ	1500
4	Durazno	TAL	3000
5	Pera	SAN	800
8	Limon	ARI	500
19	Frutilla	IQQ	2500
20	Naranja	SAN	1800

- **Busqueda binaria (logN)**
- **10 millones de registros ~ 1M + rápido**

Indices

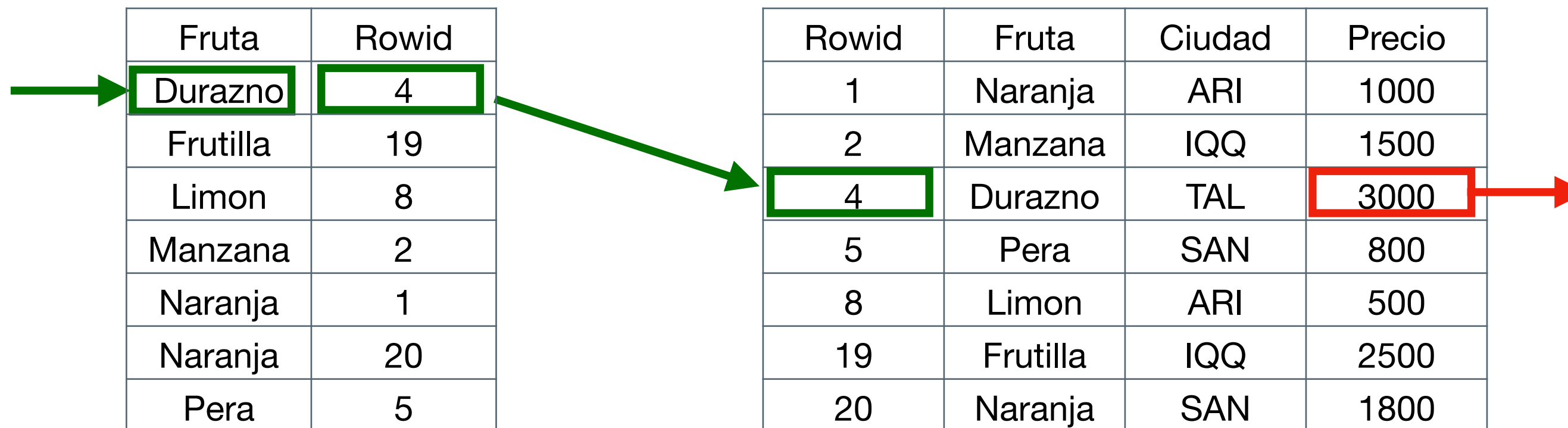
SQLite3

- Busqueda por indice
 - Indexamos Fruta

```
create index idx_fruta ON VentaFrutas(fruta)
```

```
select precio from VentaFrutas Where fruta='Durazno'
```

Fruta	Rowid
Durazno	4
Frutilla	19
Limon	8
Manzana	2
Naranja	1
Naranja	20
Pera	5



- 2 Búsquedas binarias (logN)

Indices

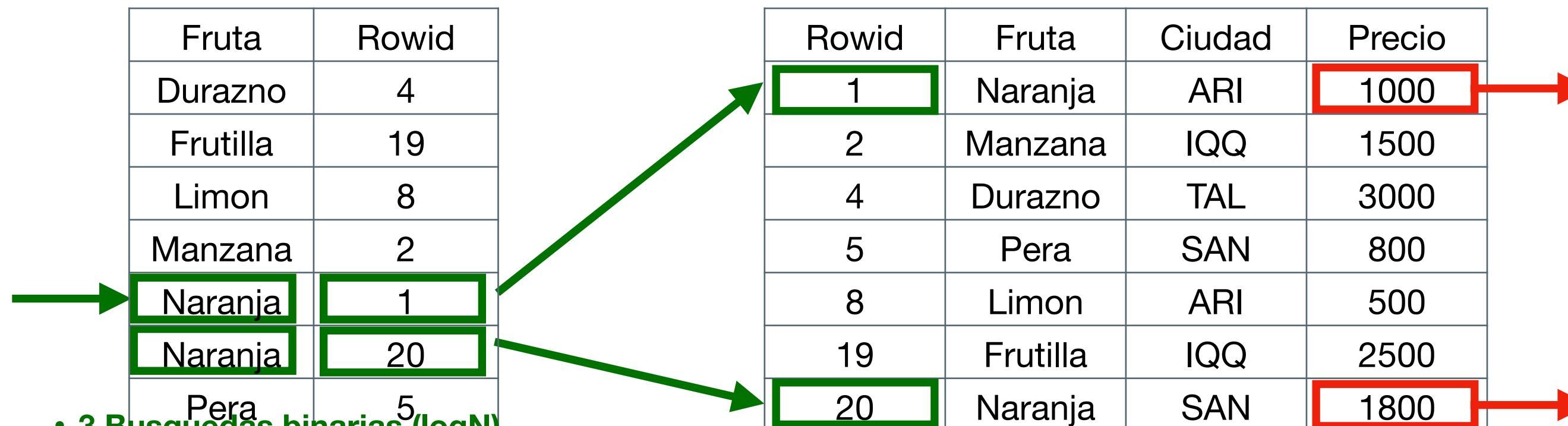
SQLite3

- Multiples resultados
 - Indexamos Fruta

```
create index idx_fruta ON VentaFrutas(fruta)
```

```
select precio from VentaFrutas Where fruta='Naranja'
```

Fruta	Rowid
Durazno	4
Frutilla	19
Limon	8
Manzana	2
Naranja	1
Naranja	20
Pera	5



- 3 Búsquedas binarias ($\log N$)
- $(K+1) \cdot \log N$


Indices

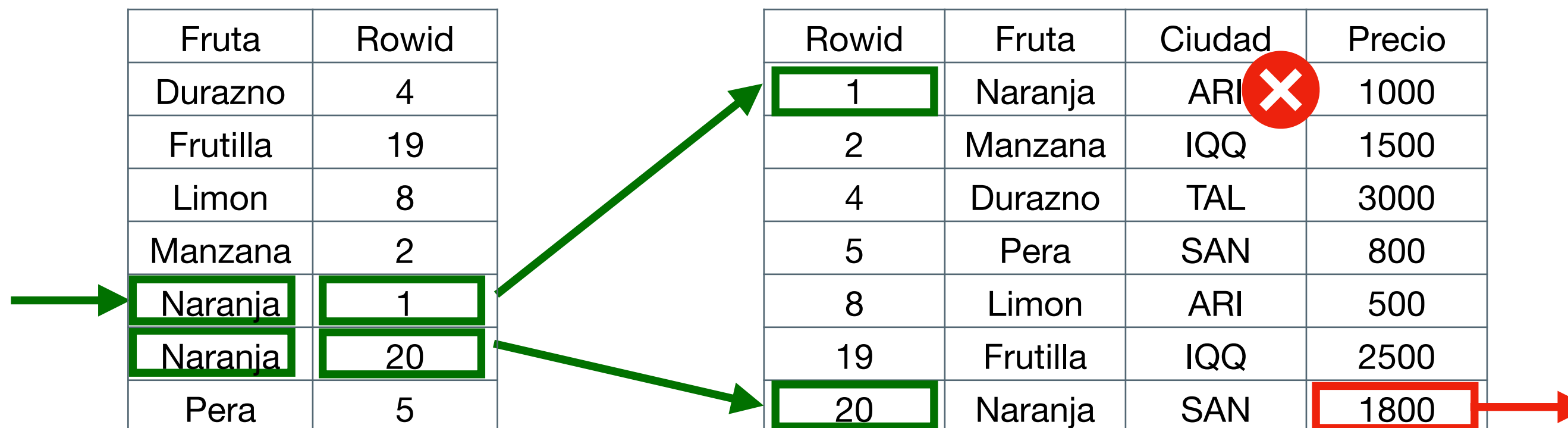
SQLite3

- Multiples ANDs en WHERE

SELECT precio **FROM** VentaFrutas
WHERE fruta='Naranja' **AND** Ciudad='SAN'

Fruta	Rowid
Durazno	4
Frutilla	19
Limon	8
Manzana	2
Naranja	1
Naranja	20
Pera	5

Fruta	Rowid	Rowid	Fruta	Ciudad	Precio
Durazno	4	1	Naranja	ARI 	1000
Frutilla	19	2	Manzana	IQQ	1500
Limon	8	4	Durazno	TAL	3000
Manzana	2	5	Pera	SAN	800
Naranja	1	8	Limon	ARI	500
Naranja	20	19	Frutilla	IQQ	2500
Pera	5	20	Naranja	SAN	1800



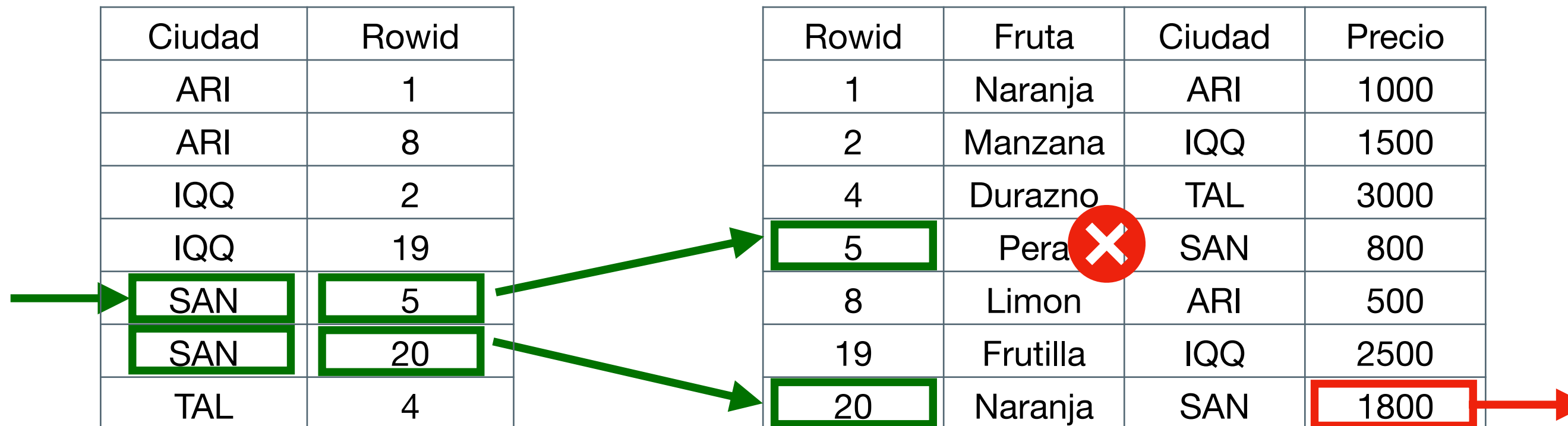
Indices SQLite3

- Multiples ANDs en WHERE
- Indexamos Ciudad

```
create index idx_ciudad ON VentaFrutas(ciudad)
```

```
SELECT precio FROM VentaFrutas  
WHERE fruta='Naranja' AND Ciudad='SAN'
```

Ciudad	Rowid
ARI	1
ARI	8
IQQ	2
IQQ	19
SAN	5
SAN	20
TAL	4



- El uso del indice Ciudad en lugar de fruta hace que SQLite examine un conjunto diferente de filas (mismo resultado)
- Cual indice SQLite selecciona?
 - Si el comando **ANALYZE** se ejecutó en la BD, de modo que SQLite tuvo la oportunidad de recopilar estadísticas sobre los índices disponibles, entonces SQLite sabrá que el índice Fruta generalmente limita la búsqueda **a un solo elemento**, mientras que el índice Ciudad normalmente solo reducirá la búsqueda **a dos filas**.

Indices

SQLite3

- Indices con multiples columnas
- Indexamos Ciudad y fruta

```
create index idx_fruta_ciudad ON VentaFrutas(fruta, ciudad)
```

```
SELECT precio FROM VentaFrutas  
WHERE fruta='Naranja' AND Ciudad='SAN'
```

Fruta	Ciudad	Rowid
Durazno	TAL	4
Frutilla	IQQ	19
Limon	ARI	8
Manzana	IQQ	2
Naranja	ARI	1
Naranja	SAN	20
Pera	SAN	5

Fruta	Ciudad	Rowid
Durazno	TAL	4
Frutilla	IQQ	19
Limon	ARI	8
Manzana	IQQ	2
Naranja	ARI	1
Naranja	SAN	20
Pera	SAN	5

Rowid	Fruta	Ciudad	Precio
1	Naranja	ARI	1000
2	Manzana	IQQ	1500
4	Durazno	TAL	3000
5	Pera	SAN	800
8	Limon	ARI	500
19	Frutilla	IQQ	2500
20	Naranja	SAN	1800

- 2 Busquedas binarias (logN)

Indices SQLite3

- Indices con multiples columnas
- Indexamos Ciudad y fruta

```
create index idx_fruta_ciudad ON VentaFrutas(fruta, ciudad)
create index idx_fruta ON VentaFrutas(fruta)
```

```
SELECT precio FROM VentaFrutas
WHERE fruta='Durazno'
```

Fruta	Ciudad	Rowid
Durazno	TAL	4
Frutilla	IQQ	19
Limon	ARI	8
Manzana	IQQ	2
Naranja	ARI	1
Naranja	SAN	20
Pera	SAN	5



Fruta	Ciudad	Rowid
Durazno	TAL	4
Frutilla	IQQ	19
Limon	ARI	8
Manzana	IQQ	2
Naranja	ARI	1
Naranja	SAN	20
Pera	SAN	5



Rowid	Fruta	Ciudad	Precio
1	Naranja	ARI	1000
2	Manzana	IQQ	1500
4	Durazno	TAL	3000
5	Pera	SAN	800
8	Limon	ARI	500
19	Frutilla	IQQ	2500
20	Naranja	SAN	1800

- **Idx_fruta** [prefijos] **idx_fruta_ciudad**
- **Regla:** Indices que son prefijos de otros no son necesarios => eliminar el indice con menor numero de columnas

Indices

SQLite3

- Indexando todas las columnas (cobertura)
- Indexamos fruta, ciudad, precio

```
create index idx_fruta_ciudad ON VentaFrutas(fruta, ciudad, precio)
```

```
SELECT precio FROM VentaFrutas  
WHERE fruta='Naranja' AND Ciudad='SAN'
```

Fruta	Ciuda	Precio	Rowid
Durazn	TAL	1500	4
Frutilla	IQQ	2500	19
Limon	ARI	500	8
Manza	IQQ	1500	2
Naranj	ARI	1000	1
Naranj	SAN	1800	20
Pera	SAN	800	5

Fruta	Ciudad	Precio	Rowid
Durazno	TAL	1500	4
Frutilla	IQQ	2500	19
Limon	ARI	500	8
Manzana	IQQ	1500	2
Naranja	ARI	1000	1
Naranja	SAN	1800	20
Pera	SAN	800	5



- 1 Búsqueda binaria (logN)
- 1/2 reducción en tiempo || primera indexación?
- 2 micro segundos a 1 micro segundo

Indices

SQLite3

- Términos relacionados con OR en la cláusula WHERE
- Indices con multiples columnas utiles con AND.

```
SELECT precio FROM VentaFrutas  
WHERE fruta='Naranja' OR Ciudad='SAN'
```

Fruta	Rowid
Durazn	4
Frutilla	19
Limon	8
Manza	2
Naranj	1
Naranj	20
Pera	5

Ciuda	Rowid
ARI	1
ARI	8
IQQ	2
IQQ	19
SAN	5
SAN	20
TAL	4

Union

Rowid	Fruta	Ciudad	Precio
1	Naranja	ARI	1000
2	Manzana	IQQ	1500
4	Durazno	TAL	3000
5	Pera	SAN	800
8	Limon	ARI	500
19	Frutilla	IQQ	2500
20	Naranja	SAN	1800

- 3 Busquedas binarias (logN)

Indices

SQLite3

- Ordenando
- Los indices pueden ser usados para acelerar busquedas y ordenamientos.
- Ordenando Tablas sin indices

`select * from VentaFrutas ORDER BY fruit`



Rowid	Fruta	Ciudad	Precio
1	Naranja	ARI	1000
2	Manzana	IQQ	1500
4	Durazno	TAL	3000
5	Pera	SAN	800
8	Limon	ARI	500
19	Frutilla	IQQ	2500
20	Naranja	SAN	1800

Ord
enar

- Tiempo: $N \log N$
- Almacenamiento adicional

Indices

SQLite3

- Ordenando

- Los indices pueden ser usados para acelerar busquedas y ordenamientos.

- Ordenando Tablas por rowid

- `select * from VentaFrutas ORDER BY rowid`
- `select * from VentaFrutas ORDER BY rowid DESC`



Rowid	Fruta	Ciudad	Precio
1	Naranja	ARI	1000
2	Manzana	IQQ	1500
4	Durazno	TAL	3000
5	Pera	SAN	800
8	Limon	ARI	500
19	Frutilla	IQQ	2500
20	Naranja	SAN	1800

- Se evita el ordenamiento.
- DESC: end->begin

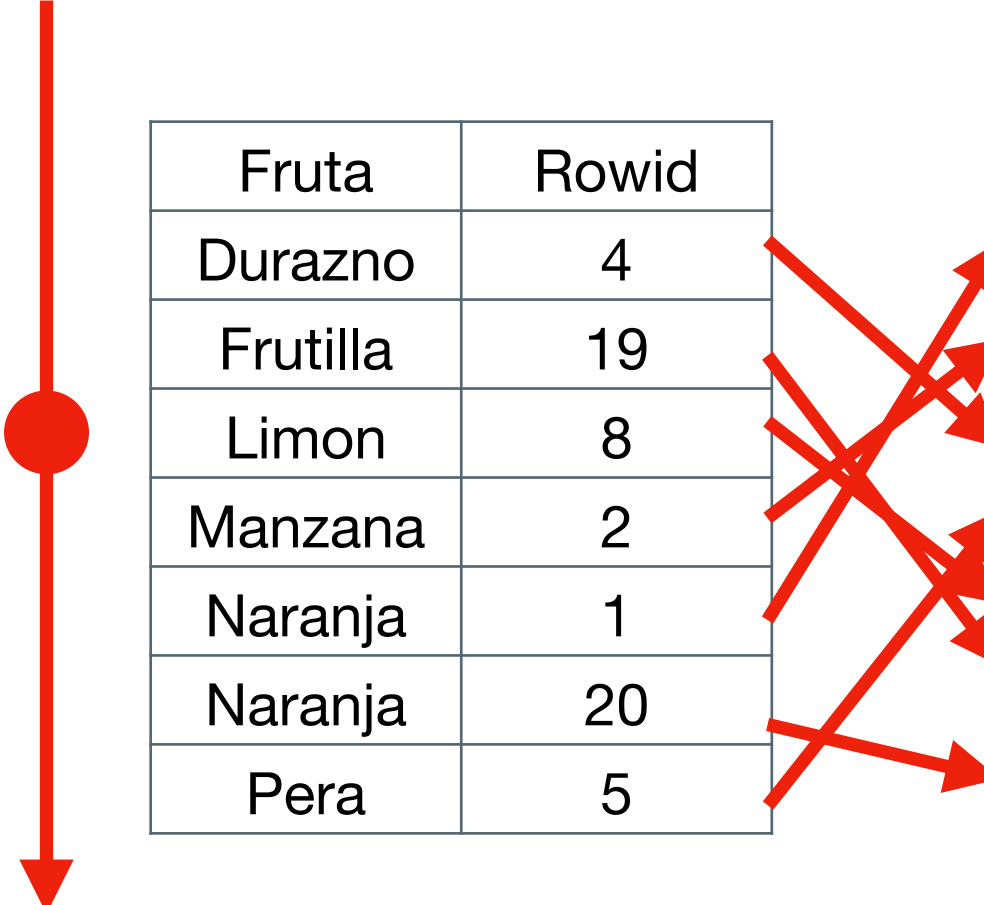
Indices

SQLite3

- Ordenando

- Los indices pueden ser usados para acelerar busquedas y ordenamientos.
- Ordenando Tablas por otra columna

- `select * from VentaFrutas ORDER BY fruit`



Fruta	Rowid
Durazno	4
Frutilla	19
Limon	8
Manzana	2
Naranja	1
Naranja	20
Pera	5

Rowid	Fruta	Ciudad	Precio
1	Naranja	ARI	1000
2	Manzana	IOO	1500
4	Durazno	TAI	3000
5	Pera	SAN	800
8	Limon	ARI	500
19	Frutilla	IOO	2500
20	Naranja	SAN	1800

- Tiempo NlogN
- Evita almacenamiento

Optimización SQL

Optimización SQL

Análisis de la cláusula WHERE

- La cláusula WHERE de una consulta se divide en "términos", donde cada término está separado de los demás por un operador AND.
- Si la cláusula WHERE se compone de restricciones separadas por el operador OR, se considera que toda la cláusula es un único "término" al que se aplica la optimización de la cláusula OR.
- Todos los términos de la cláusula WHERE se analizan para ver si se pueden responder utilizando índices.

```
column = expression  
column IS expression  
column > expression  
column >= expression  
column < expression  
column <= expression  
column IN (expression-list)  
column IN (subquery)  
column IS NULL  
expression = column  
expression > column  
expression >= column  
expression < column  
expression <= column
```

Optimización SQL

Análisis de la cláusula WHERE

- `CREATE INDEX idx_ex1 ON ex1(a,b,c,d,e...,x,y,z) # cobertura`
- El índice podría usarse si las columnas iniciales del índice (columnas a, b, etc.) aparecen en los términos de la cláusula WHERE.
- Las columnas iniciales del índice deben usarse con los operadores `=` o **IN** o **IS**.
- La columna más a la derecha que se utiliza puede emplear desigualdades (`>`, `<`, ...). Para la columna más a la derecha de un índice que se utiliza, puede haber hasta dos desigualdades que deben intercalar los valores permitidos de la columna entre dos extremos.
- No es necesario que todas las columnas de un índice aparezcan en un término de la cláusula WHERE para que se utilice ese índice.
 - **No pueden saltar columnas del índice que se utiliza.**
 - `WHERE a = "Alex" AND b = "Chile" AND d > 10` \Rightarrow solo a, b son usados.
- Las columnas de índice normalmente no se utilizarán si están **a la derecha de una columna que está restringida solo por desigualdades.**

Optimización SQL

Análisis de la cláusula WHERE (Ejemplos)

- CREATE INDEX idx_ex1 ON ex1(a,b,c,d,e...,x,y,z) # cobertura
- ... WHERE a=5 AND b IN (1,2,3) AND c IS NULL AND d='hello'
 - a, b, c y d del índice se podrían utilizar, ya que esas cuatro columnas forman un **prefijo del índice y todas están sujetas a restricciones de igualdad**.
- ... WHERE a=5 AND b IN (1,2,3) AND c>12 AND d='hello'
 - [a, b, c] ok d no se puede usar pues sigue a una desigualdad ($c > 12$).
- ... WHERE b IN (1,2,3) AND c NOT NULL AND d='hello'
 - El índice no se puede usar porque la columna más a la izquierda del índice (columna "a") no está restringida.
- ... WHERE a=5 OR b IN (1,2,3) OR c NOT NULL OR d='hello'
 - El índice no se puede utilizar porque los términos de la cláusula WHERE están conectados por OR en lugar de AND (full table scan).
 - si se agregan tres índices adicionales que contengan las columnas b, c y d como sus columnas más a la izquierda, entonces se podría aplicar la **optimización de la cláusula OR**.

Optimización SQL

Análisis de la cláusula WHERE

- Optimización BETWEEN
 - *expr1* BETWEEN *expr2* AND *expr3* (1)
 - *expr1* >= *expr2* **AND** *expr1* <= *expr3* (1t)
 - Solo **1t** se utiliza en los análisis, si sus terminos estan presentes en un indice.
- Optimizaciones OR
 - *column* = *expr1* OR *column* = *expr2* OR *column* = *expr3* OR ...
 - *column* **IN** (*expr1*,*expr2*,*expr3*,...)
 - *expr1* **OR** *expr2* **OR** *expr3* (a=5 or x>y ...)
 - Revisamos si el subtérmino es indexable por sí mismo.

```
rowid IN (SELECT rowid FROM table WHERE expr1
          UNION SELECT rowid FROM table WHERE expr2
          UNION SELECT rowid FROM table WHERE expr3)
```

Optimización SQL

Análisis de la cláusula WHERE

- Optimización BETWEEN
 - *expr1* BETWEEN *expr2* AND *expr3* (1)
 - *expr1* >= *expr2* **AND** *expr1* <= *expr3* (1t)
 - Solo **1t** se utiliza en los análisis, si sus terminos estan presentes en un indice.
- Optimizaciones OR
 - *column* = *expr1* OR *column* = *expr2* OR *column* = *expr3* OR ...
 - *column* **IN** (*expr1*,*expr2*,*expr3*,...)
 - *expr1* **OR** *expr2* **OR** *expr3* (a=5 or x>y ...)
 - Revisamos si el subtérmino es indexable por sí mismo.

```
rowid IN (SELECT rowid FROM table WHERE expr1
          UNION SELECT rowid FROM table WHERE expr2
          UNION SELECT rowid FROM table WHERE expr3)
```

Optimización SQL

Análisis de la cláusula WHERE

- La optimización Skip-Scan
- La regla general es que los índices son útiles si hay restricciones de cláusula WHERE en las columnas más a la izquierda del índice.

```
CREATE TABLE persona(  
    nombre TEXT PRIMARY KEY,  
    role TEXT NOT NULL,  
    estatura INT NOT NULL, -- in cm  
    CHECK( role IN ('estudiante','profesor') )  
);  
CREATE INDEX idx1_role_estatura ON people(role,estatura);
```

- `SELECT nombre FROM persona WHERE altura>=180;`

```
SELECT nombre FROM persona WHERE role='estudiante' AND estatura>=180  
UNION ALL  
SELECT nombre FROM persona WHERE role='profesor' AND estatura>=180;
```

Optimización SQL

Joins

- Los Joins se implementan como bucles anidados.
- La tabla del extremo izquierdo de la cláusula FROM forma el bucle externo y la tabla del extremo derecho forma el bucle interno.

```
CREATE TABLE nodo(  
    id INTEGER PRIMARY KEY,  
    nombre TEXT  
);  
CREATE INDEX idx_nodo ON nodo(nombre);  
CREATE TABLE arco(  
    orig INTEGER REFERENCES node,  
    dest INTEGER REFERENCES node,  
    PRIMARY KEY(orig, dest)  
);  
CREATE INDEX idx_arco ON arco(dest,orig);
```

```
foreach n1 where n1.nombre='alice' do:  
    foreach n2 where n2.nombre='bob' do:  
        foreach e where e.orig=n1.id and e.dest=n2.id  
            return n1.*, n2.*, e.*  
        end  
    end  
end
```

```
SELECT *  
FROM arco AS e,  
     nodo AS n1,  
     nodo AS n2  
WHERE n1.nombre = 'alice'  
      AND n2.nombre = 'bob'  
      AND e.orig = n1.id  
      AND e.dest = n2.id;
```

```
foreach n1 where n1.name='alice' do:  
    foreach e where e.orig=n1.id do:  
        foreach n2 where n2.id=e.dest and n2.name='bob' do:  
            return n1.*, n2.*, e.*  
        end  
    end  
end
```

Optimización SQL

Linearización de subconsulta

- Cuando se produce una subconsulta en la cláusula FROM de un SELECT, el comportamiento más simple es evaluar la subconsulta en una tabla temporal y luego ejecutar el SELECT externo contra la tabla temporal.
- Tal plan puede ser subóptimo ya que la tabla temporal no tendrá ningún índice y la consulta externa se verá obligada a realizar un revisión completa en la tabla transitoria.
- Trabajar en un ejemplo.

Informaciones

- Control 2
 - Jueves 01 Junio.
 - Sala por confirmar via ucampus.
 - Materia hasta clase del 25/05
 - SQL y Optimización SQL.

Consultas?

Consultas o comentarios?

Muchas gracias