¿Cómo funcionan los sistemas de bases de datos?

Parte II - Índices de una base de datos

En la primera lectura aprendimos cómo los sistemas de bases de datos guardan sus datos de manera persistente en el disco duro. Sin embargo, explicamos que ir a buscar una tupla arbitraria al disco duro es bastante lento, así que por eso necesitamos técnicas que nos permitan acelerar este proceso. Lo que vamos a hacer será **indexar** los datos almacenados en el disco duro. Esto es, vamos a contar con una estructura a la que le podremos preguntar dónde está la tupla que estamos buscando, y así la encontraremos casi al instante.

Sobre este material: El contenido que voy a enseñar en estas lecturas es una versión de alto nivel de lo que aprendí en el curso de Implementación de Bases de Datos que dicta el profesor Cristian Riveros, y es (más o menos) lo que enseñaba yo en el curso de Bases de Datos que dicté durante 5 años. Además está complementado con parte de la experiencia adquirida durante todos estos años trabajando en el área de Bases de Datos, tanto a nivel teórico como práctico.

Sobre el autor: Mi nombre es Adrián Soto Suárez y durante mi carrera he estudiado bastante este tema. Para más detalles sobre mí puedes ir a http://adriansoto.cl.

1. Hash Index en las bases de datos

Si has visto contenido de estructuras de datos, entonces conoces el concepto de **tabla de hash**. Una tabla de hash es una estructura que almacena pares *key* - *value*, en donde la operación de preguntar el valor asociado a una determinada *key* se hace súper rápido. Para entender mejor el funcionamiento de esta estructura, vamos a ver un ejemplo. Considera una tabla `Artistas(aid, nombre_artista)`, donde `aid` es el identificador del artista y `nombre_artista` es el nombre del artista, y donde las tuplas de la tabla son las indicadas en la Figura 1. Aquí las tuplas están guardadas en el disco duro sin ningún orden en particular, además cada tupla tiene un puntero hacia la tupla siguiente, la que puede estar en la misma página del disco duro[1], o bien puede estar en otra página (no necesariamente adyacente en el mismo *track*).

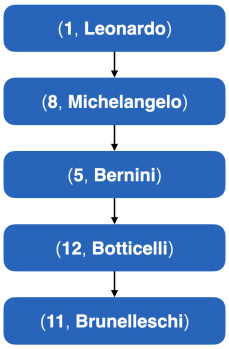

Figura 1: almacenamiento de una tabla en disco duro sin índices.

[1]Recordemos que en general, cada página del disco duro contiene muchas tuplas.

Como vimos en la lectura anterior, para encontrar una tupla de una tabla que cumpla una determinada condición debemos recorrer todas las páginas que corresponden a dicha tabla. En este caso, para encontrar la tupla en donde el identificador del artista (`aid`) sea 12, vamos a tener que pasar por las tuplas con identificador 1, 8 y 5 antes. Además, como no tenemos mayor información de la tabla, puede pasar que más adelante exista otra tupla con identificador 12 (no es este el caso). Ahora bien, para acceder más rápido a los datos, podemos instanciar una tabla de hash como la de la Figura 2.

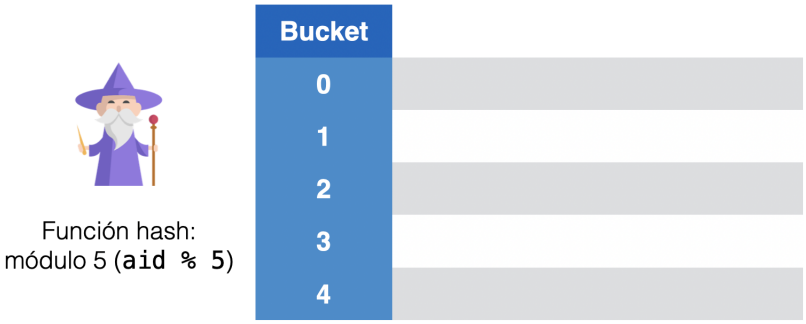

Figura 2: representación de una tabla de hash vacía.

Aquí estamos creando una tabla de hash con 5 casilleros (o *buckets* en inglés), donde cada uno de estos casilleros se representa por uno de los números entre 0 y 4. Además, aparte de estos casilleros generamos una función que va a recibir tuplas; esta función se conoce como **función de hash**. Esta función tomará el `aid` de la tupla y retornará el `aid` módulo (`%`) 5. Como recordarás, la función módulo retorna el resto de la división entera, por lo tanto, al obtener el módulo 5 de un número sabemos que el resultado estará entre 0 y 4, que son los valores de los casilleros. Así, al insertar en la tabla de hash las tuplas señaladas anteriormente, el resultado es el de la Figura 3.

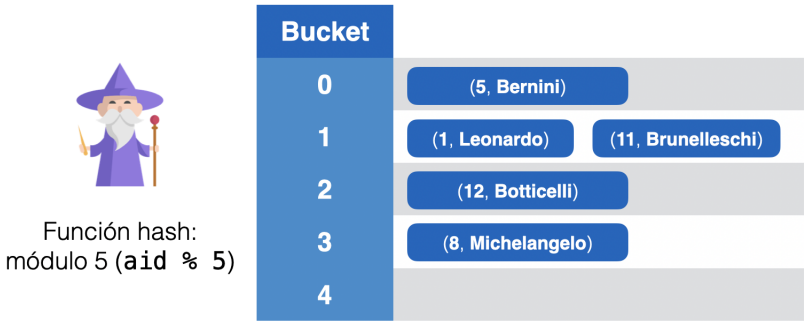

Figura 3: representación de nuestros datos en la tabla de hash.

En este caso, la tabla de hash está creada sobre el atributo `aid`, porque dependemos de este atributo para definir el casillero en el que la tupla queda. Así, podemos hacer busquedas por `aid`, es decir podemos preguntarle a la tabla de hash "dame el artista con `aid` 12", pero no podemos preguntar por un artista con un determinado nombre. Además, como se ve en la Figura 3, en una tabla de hash pueden haber colisiones. Esto es, dos valores pueden quedar en el mismo casillero.

Entonces, ¿qué pasa si queremos buscar un artista con un determinado `aid`?. Para esto podemos ver la Figura 4. En este ejemplo estamos buscando al artista con identificador 12. Lo que sucederá es que le preguntaremos a la función de hash el resultado de $12 \% 5$, que es 2; por lo tanto, iremos al casillero 2 a buscar el artista con identificador 12. Ahora, en vez de pasar por todas las tuplas, vamos a ir directamente a la que necesitamos. ¡Esto es un gran avance!. En el caso de que en el casillero haya más de una tupla,

deberemos recorrer todas las de ese casillero hasta encontrar la que queramos, pero discutiremos más de esto en unos minutos.

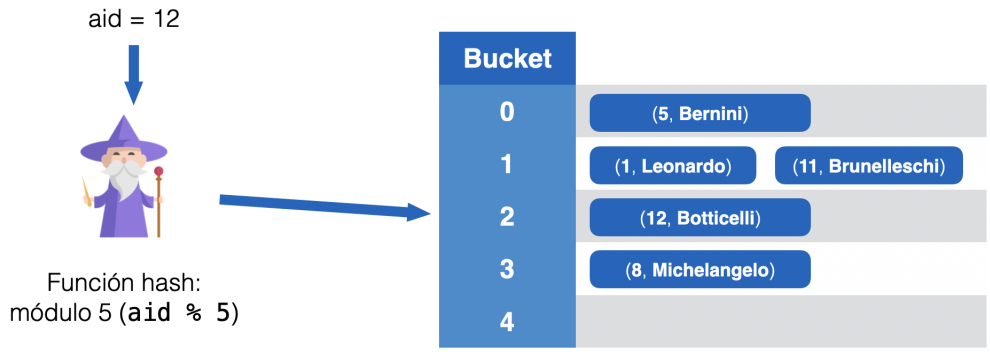

Figura 4: ejemplo de búsqueda en una tabla de hash.

Sobre las funciones de hash. En este caso, la función de hash era muy simple: sacar el módulo del identificador del artista. En general, las funciones de hash son mucho más complejas que esto. Además, se requiere que estas funciones tengan ciertas propiedades, como que por ejemplo, las posibles key distribuyan uniforme en los casilleros. En el ejemplo anterior, si todos los artistas tenían un `aid` divisible por 5, iban a parar todos al mismo casillero. Este tipo de comportamiento es algo que queremos evitar.

1.1. Detalles de implementación de un hash index

Debes recordar que en la lectura anterior mencionamos que lo que queríamos minimizar son las lecturas al disco duro, pero hasta ahora, no hemos hablado de cómo el hash index minimiza las llamadas al disco duro. Ahora, vamos a explicar a un alto nivel la implementación del hash index. Para esto, considera la Figura 5.

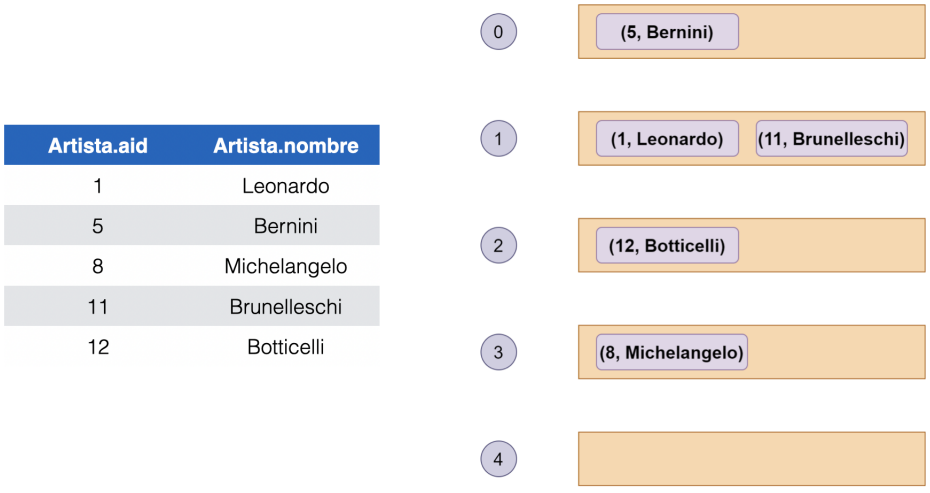

Figura 5: representación de un hash index.

Como bien explicamos antes, las tuplas se guardan en páginas de disco. Por lo tanto aquí, el hash index es una estructura que, generalmente, cabe en memoria, y a la que le podemos preguntar **la página del disco duro en la que está la tupla que estamos buscando**. En el ejemplo mostrado por la Figura 5, las páginas de disco son los rectángulos naranjos, donde estamos suponiendo que en cada página caben

dos tuplas (esto es claramente una exageración, pero ayuda a que entendamos la idea). Así, supongamos que estamos buscando al artista con identificador 11. Lo que pasará es que cargaremos en memoria el casillero 1, que contiene dos tuplas guardadas en una **única página del disco duro**. Así, recorreremos internamente esta página (i.e. pasaremos por Leonardo y luego llegaremos a Brunelleschi) para retornar la tupla que estamos buscando habiendo hecho una única lectura al disco duro. Así, diremos que el costo I/O de esta operación es 1, porque solo estamos leyendo una página de disco.

De esta forma, pasamos de leer la tabla entera para hacer esta consulta a solamente leer una página de disco duro. Ahora bien, ¿qué pasa si queremos insertar la tupla `(16, Giotto)`?. Como te habrás dado cuenta, esta tuple cae en el casillero 1, que ya está lleno. Lo que va a pasar, es que al final de la página vamos a agregar un puntero hacia una siguiente página en el disco duro, y esta página va a contener las dos siguientes tuplas que caigan en este casillero. La primera de estas tuplas será la que acabamos de insertar, y así quedará un espacio para una siguiente. Esto se ilustra en la Figura 6.

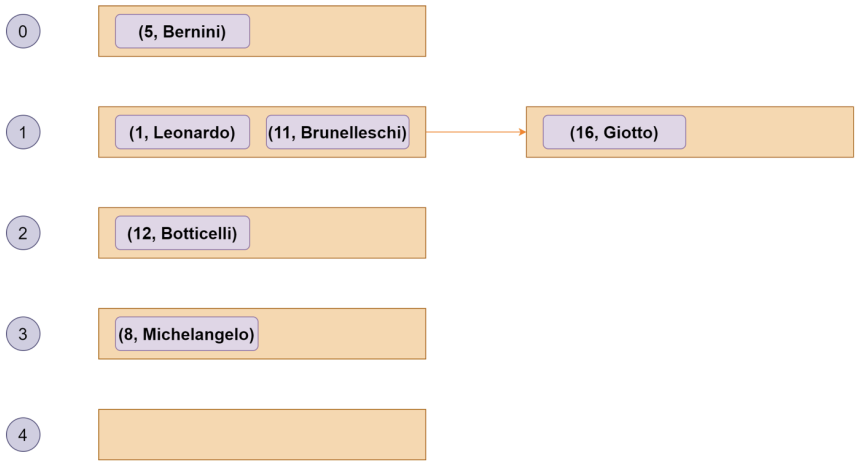

Figura 6: representación de un hash index con *overflow pages*.

Estas páginas adicionales para los casilleros se conocen como *overflow pages*. Como podrás suponer, si tenemos que ir a buscar una tupla al casillero 1, ahora en vez de leer una única página, vamos a tener que leer dos páginas. Esto nuevamente ilustra la necesidad de que las funciones de hash distribuyan uniforme, ya que por ejemplo en este caso tenemos un casillero vacío. Ahora bien, recuerda que en una página de disco entran muchas tuplas, por lo que hay que insertar varios datos para tener *overflow pages*. Sin embargo, hay una forma de hacer escalar estos índices para **hacer crecer el hash index a medida que sea necesario**. Las más famosas de estas estructuras dinámicas son los **extendable hash index** y los **linear hash index**. La intuición sobre estas estructuras, es que cuando hay una cantidad de *overflow pages* que supera cierto límite, vamos a generar nuevos casilleros y redistribuir los datos actuales sobre estos nuevos casilleros. Además, la diferencia entre estas dos estructuras es el criterio para hacer crecer el índice y la forma de crear nuevos casilleros.

Ahora, vamos a estudiar cómo funciona la creación de índices en la práctica. Además, vamos a discutir sobre la clasificación de estos índices entre primarios y secundarios.

2. Índices en la práctica

Ya hemos deslizado que los índices se aplican sobre ciertos atributos. Considera el ejemplo de los artistas: aquí el atributo que tenemos indexado es el atributo `aid`. Si nosotros queremos hacer la consulta "dame todos los artistas con nombre Giotto" no tendríamos más remedio que recorrer todas las tuplas de la tabla, porque el atributo `nombre_artista` **no está indexado**. Supongamos que sabemos que vamos a hacer la consulta de buscar artistas por nombre muy seguido; en ese caso, es una muy buena idea indexar

ese atributo. Y obviamente, esta operación no es automática, por lo que si queremos indexar un atributo tenemos que correr el comando `CREATE INDEX`:

```
CREATE INDEX nombre_índice
ON nombre_tabla(columna_a_indexar);
```

Así, este comando nos permite crear un índice[2], al que le tenemos que dar un nombre en particular, para luego indicar el nombre de la tabla y el atributo de aquella tabla a indexar. Por ejemplo, para indexar el nombre de los artistas, tendríamos que correr:

```
CREATE INDEX indice_nombre_artista
ON Artistas(nombre_artista);
```

Por lo mismo, ahora al realizar la consulta de artistas por nombre, nos vamos a demorar mucho menos. Entonces, si tenemos una tabla `R` e indexamos su atributo `a`, las consultas del estilo:

```
SELECT *
FROM R
WHERE a = <valor>
```

van a ser evaluadas de forma rápida. Ahora bien, hay una sutileza que no hemos discutido, y tiene que ver sobre la diferencia entre la indexación de la llave primaria y la indexación de otros atributos.

2.1. Índices primarios y secundarios

Para entender la diferencia entre los índices, primero vamos a recordar la tabla con la que estamos trabajando, pero también la vamos a modificar un poco. La tabla de artistas ahora se verá así: `Artistas(aid, nombre_artista, biografia)`, en la que estamos añadiendo un campo de texto arbitrario que contiene la biografía del artista. Si el modelo es como esperamos, nosotros haríamos que la *primary key* fuera el atributo `aid`; por lo mismo, al crear la tabla, el comando sería:

```
CREATE TABLE Artistas(
  aid INT PRIMARY KEY,
  nombre_artista VARCHAR(100),
  biografia TEXT
)
```

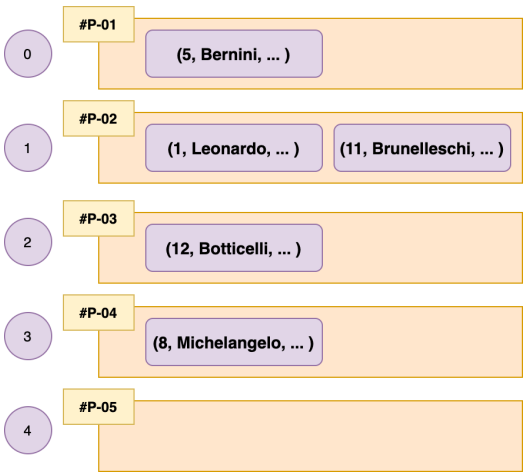

Figura 7: nueva representación del hash index.

[2]La sintaxis puede variar de sistema en sistema, pero la lógica es la misma.

Si creamos una tabla con una **llave primaria**, este atributo por defecto estará indexado y a este índice le vamos a llamar **índice primario**. Este índice se verá tal cual lo mostramos anteriormente (pero ahora con la biografía). Por claridad, vamos a ilustrar de nuevo cómo queda el índice, donde seguimos suponiendo que nos caben dos tuplas por página. Esto lo vemos en la Figura 7.

Notarás que ahora estamos indicando el nombre de la página en el disco duro con una etiqueta amarilla. Además, la elección del campo biografía, que es de tipo texto, no es arbitraria; es un campo que de por si va a usar mucho espacio más que el identificador y el nombre. Además, es importante que notes que dentro de este índice primario **tenemos guardada la relación completa**. Ahora, al crear un índice sobre el campo nombre, sucede lo que se muestra en la Figura 8.

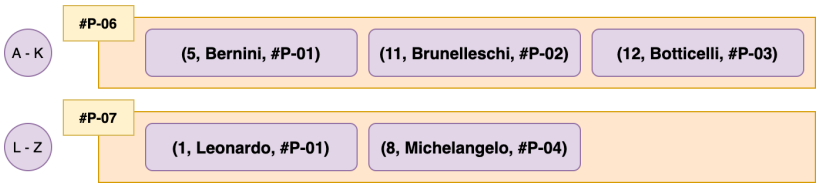

Figura 8: nueva representación del hash index.

Notamos que al crear este índice **no guardamos la tupla entera**, sino que guardamos el atributo indexado, que corresponde al nombre del artista, junto con la llave primaria de la tupla que representa ese nombre. Luego, viene un puntero hacia la página en el disco duro en la que está guardada la tupla que estamos buscando. Así, hemos construido un índice sobre un atributo que no es la llave primaria, por ende, este será un **índice secundario**. Y ahora, cuando hacemos una consulta de igualdad sobre el atributo nombre, por ejemplo:

```
SELECT *
FROM Artistas
WHERE nombre_artista = "Brunelleschi"
```

El sistema comprende que el índice que le conviene usar es el del atributo `nombre_artista`. Luego, se va al casillero que representa los nombres de la A a la K[3] y encuentra en la primera (y única) página de ese casillero el puntero a la única tupla con nombre `Brunelleschi`; este puntero es hacia la página `#P-02`. Luego, el sistema de bases de datos va a buscar esta página que está "guardada" como parte del índice primario, y encuentra la tupla que estamos buscando. En este caso, hicimos dos lecturas al disco duro: (1) la primera para encontrar el puntero de la página de disco que contiene a la tupla con nombre `Brunelleschi` y una (2) segunda lectura para cargar dicha página en memoria y rescatar el la tupla entera.

Como supondrás, el índice secundario es ligeramente más lento que el primario, porque ahora hay que hacer una lectura más. Ahora, ¿por qué no guardamos la tupla entera en el índice secundario?. La respuesta es que hacer esto implica replicar los datos, y no queremos usar tanto espacio para guardar una sola tabla. Es por eso, que los índices secundarios siempre van a apuntar a las tuplas guardadas en el índice primario, que finalmente corresponde a la forma en que se almacena físicamente la tabla en el disco duro.

Además, en general, en una página de disco duro van a poder ser almacenados muchos más punteros que tuplas. ¿Recuerdas que en el ejemplo nos cabían solamente dos tuplas por página de disco duro?. Bueno, dado que en el índice secundario no estamos almacenando los demás campos (en este caso, solo la biografía, que contiene valores extensos), en una página del índice secundario, alcanzo a guardar más registros. Además, obviamente, si en una página guardo muchos punteros y me quedo sin espacio, podemos agregar una *overflow page*.

[3]Esta probablemente no es la mejor función de hash, pero sirve para el ejemplo.

Índices primarios y secundarios en los ORM. Conocer esto es súper importante cuando estamos trabajando con algún ORM de algún framework web (como por ejemplo en Django, o Rails), ya que como vimos, lo que se indexa por defecto es la **llave primaria**, que en este caso es un identificador autogenerado. Así que, si las consultas que haremos serán por otro campo (como el nombre de un usuario), vamos a notar cierta lentitud al hacer consultas de igualdad si los atributos no están indexados[4]. Por otro lado, las operaciones CRUD por defecto se hacen sobre rutas que usan la *primary key* por defecto, por lo mismo, tienden a ser operaciones rápidas de hacer.

3. B + Trees

Hasta ahora hemos aprendido a indexar datos para hacer consultas de igualdad. Pero imagina que queremos hacer la siguiente consulta:

```
SELECT *
FROM Artistas
WHERE aid >= 5 AND aid <= 22
```

Esto es, queremos todos los artistas donde el `aid` esta entre 5 y 22 (inclusive). ¿Hay una forma de resolver esta consulta eficientemente usando un hash index? Lamentablemente no. Y por lo mismo, vamos a explorar otro tipo de índice, basado en árboles, que nos permite resolver este tipo de consultas, además de mantener los atributos indexados ordenados. Además, este índice sirve para responder rápidamente consultas de *join*.

El índice que vamos a estudiar ahora es el B+Tree, el que es por lejos el índice más usado en los sistemas de bases de datos, y más aún, en otros campos. Si conoces el funcionamiento de un árbol binario, este índice se te hará muy familiar. Para explicar como funciona este índice vamos a partir con un ejemplo. Considera que indexamos la llave primaria de la relación `Artistas` con un B+Tree y que luego de insertar tuplas tenemos el árbol de la Figura 9.

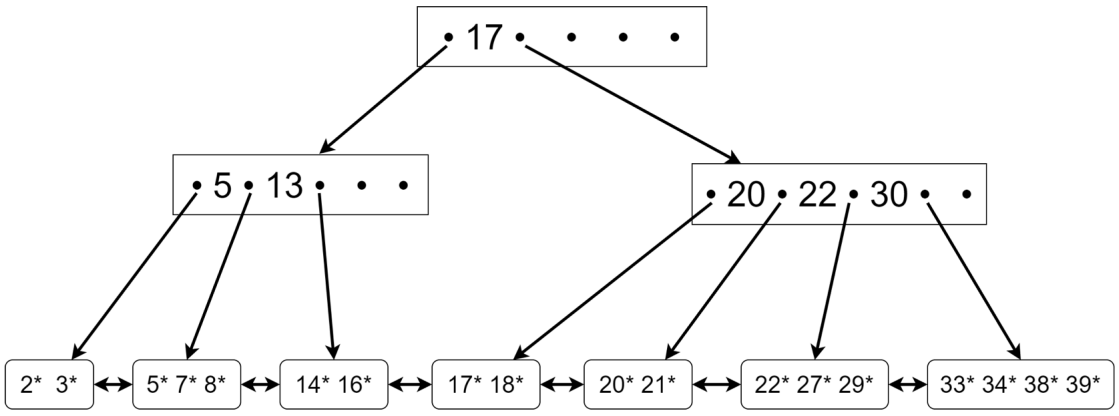

Figura 9: representación de un B+Tree.

El B+Tree se separa en los directorios, que son los nodos superiores y los nodos hoja, que son los del final. En los nodos directorios tenemos punteros hacia otros nodos (señalados por puntos en la figura). Mientras tanto en los nodos hojas tenemos los números 2^*, 3^*, 5^*, 7^*, ..., 38^*, 39^*, que vendrían a representar los identificadores de los artistas que hemos insertado (hemos cambiado la instancia de la base de datos para este ejemplo). Supongamos que cada uno de estos identificadores representa una tupla que

[4]Obviamente, esto cuando ta tenemos una cantidad razonable de datios.

está guardada en el índice. Ahora, para usar este índice, hacemos lo siguiente. Supongamos que queremos buscar la tupla donde el `aid` es 21. Lo que sucede se representa en la Figura 10.

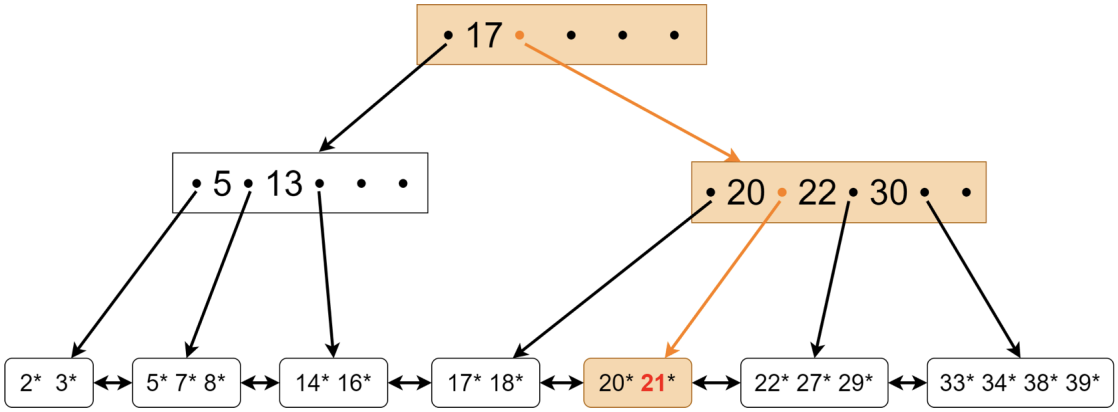

Figura 10: representación de búsqueda en un B+Tree.

En el primer nodo, seguimos el puntero a los elementos mayores o iguales a 17. Luego caemos en un nodo con varios punteros posibles. Tomamos el puntero de los elementos mayores o iguales a 20 y menores que 22. Finalmente, caemos a un nodo hoja en donde encontramos una tupla donde el `aid` es igual a 21. Aquí recibiremos (1) la tupla que estamos buscando si el índice es primario o bien (2) un puntero a la página del disco que contiene la tupla en caso de que sea un índice secundario. Vale la pena notar que cada uno de los nodos del árbol, sea directorio u hoja, son páginas distintas de disco, y en este caso, para traer la tupla vamos a hacer 3 lecturas a disco si el índice es primario y 4 si es un índice secundario. Además, en las páginas de directorio pueden haber varios punteros, y en general estos árboles tienen una altura muy baja.

Algo importante de los B+Tree, al igual que todas las estructuras de datos basadas en árboles, hay reglas que obligan a mantener balanceado el árbol. En este caso, todas las páginas (salvo el nodo raíz) deben estar ocupadas al menos a la mitad. Entonces, a la hora de insertar o eliminar un dato, hay que checkear si el árbol sigue balanceado, y en caso de que no, se debe rebalancear. Esto es importante para que el índice realice las búsquedas de forma rápida.

Ahora, ¿qué pasa con las consultas de rango?. Recordemos la consulta que planteamos anteriormente:

```
SELECT *
FROM Artistas
WHERE aid >= 5 AND aid <= 22
```

Para responder esta consulta vamos a buscar el elemento inferior del ra, que es el número 5 en el árbol. Y luego, nos vamos a aprovechar de que las **hojas de este árbol están encadenadas**, por lo que solamente es cosa de avanzar una vez que encontramos el primer elemento, hasta llegar al último elemento que cumpla con el rango solicitado. Esto lo podemos hacer porque sabemos que en las hojas el árbol mantiene **un orden ascendente** de los elementos. Esto se ilustra en la Figura 11.

De esta forma, al final en este caso particular estaríamos leyendo solo las páginas asociadas al rango, más las páginas de directorio, que tienden a ser súper pocas. Además, este índice nos permite ejecutar de forma rápida las consultas de `ORDER BY`. Por todo esto, el B+Tree es el índice más utilizado en la práctica. Ahora, tienes que considerar que si el B+Tree es utilizado como índice secundario, en el peor caso va a realizar una lectura adicional al disco por cada elemento que tenga que ir a buscar. En el ejemplo de la Figura 11, tendría que ir a buscar una página adicional por cada `aid` encontrado en el rango (5^*, 7^*, 8^*, ..., 21^*, 22^*).

B+Tree multicolumna. Quizás te estás preguntando que pasa cuando hay que indexar dos o más

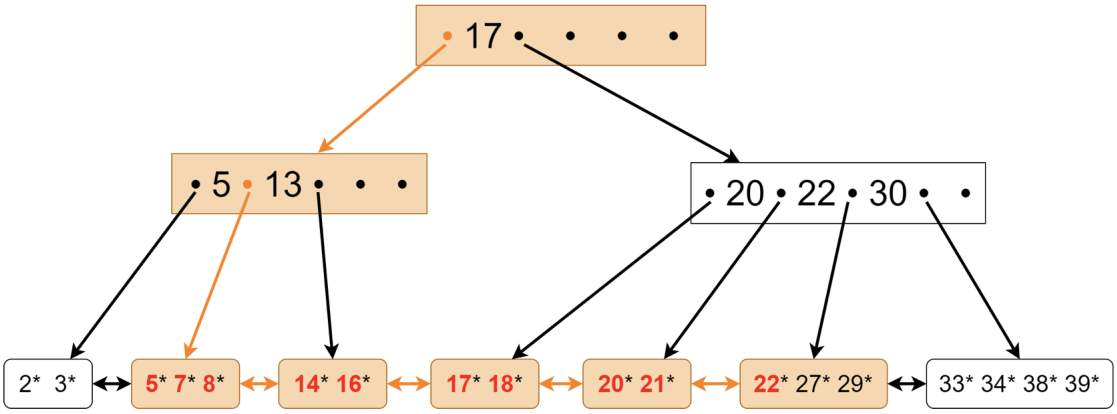

Figura 11: representación de búsqueda por rango en un B+Tree.

columnas. Por ejemplo, considera que tienes una tabla `Artistas_Obras(aid, oid)` que es una tabla intermedia entre Artistas y Obras, que dice que artista realizó que obra. Es posible correr el siguiente comando:

```
CREATE INDEX índice_multicolumna
ON Artistas_Obras(aid, oid);
```

Este comando va a crear un índice multicolumna. La idea a alto nivel de este índice es concatenar el `aid` y el `oid` de cada tupla de la relación, y guardar eso en un B+Tree, ordenado lexicográficamente. Ahora, este índice nos serviría para hacer una búsqueda del estilo:

```
SELECT *
FROM Artistas_Obras
WHERE aid = 1 AND oid = 10
```

Esto porque en el índice, está primero el atributo `aid` y luego el atributo `oid`. Así, como tenemos ordenadas las tuplas primero por `aid`, y para cada `aid` tendremos ordenados los `oid`, esta consulta puede ser hecha de forma sencilla. Sin embargo, si intentamos hacer una consulta del estilo:

```
SELECT *
FROM Artistas_Obras
WHERE oid = 10
```

el índice no nos va a ser útil, porque igual tendríamos que pasar por todos los `aid`. Obviamente una consulta por igualdad solo por `aid` se va a poder realizar sin problema. Ojo que estos índices no necesariamente están implementados en todas las bases de datos, pero en general vas a encontrar esto o soluciones similares. También existen otros índices multicolumnas más avanzados, como por ejemplo los basados en Bitmaps, que ofrecen otro tipos de garantías.

4. Palabras al cierre

En esta lectura aprendimos el funcionamiento más básico de los índices en los sistemas de bases de datos. Estas estructuras son clave para que las consultas puedan ser respondidas rápidamente. Primero estudiamos el **hash index**, un índice que permite contestar consultas de igualdad rápidamente. Luego estudiamos el **B+Tree**, el índice más usado por los sistemas, ya que permite mantener los datos ordenados, permite hacer consultas de rango y además responde las consultas de igualdad en una cantidad de lecturas a disco bastante pequeña. Por eso, hay algunos sistemas que incluso solamente implementan el índice de árbol.

Otro tema que no discutimos aquí fue el manejo de elementos duplicados. Sin embargo, las técnicas que discutimos aquí son fácilmente extendibles para soportar elementos duplicados. Finalmente, en la siguiente lectura discutiremos cómo usar índices basados en árbol para responder consultas de *join*.