Diseño físico de bases de datos

PID_00253051

Jordi Conesa Caralt
M. Elena Rodríguez González

Universitat Oberta de Catalunya

3. Índices

La gran mayoría de operaciones sobre una base de datos se realizan por valor. A su vez, el acceso por valor puede ser directo o secuencial. Más concretamente:

- El **acceso directo por valor** consiste en obtener todas las filas que contienen un determinado valor para una columna.

- El **acceso secuencial por valor** consiste en obtener diversas filas por el orden de los valores de una columna.

A continuación se muestran ejemplos sobre una tabla de clientes que contiene las siguientes columnas: código del cliente (clave primaria), nombre del cliente, población y edad.

a) Ejemplos de acceso directo por valor:

```
SELECT * FROM cliente WHERE poblacion='Arenys de Mar'

UPDATE cliente SET edad=50 WHERE codigo_cliente=20

DELETE FROM cliente WHERE codigo_cliente=100
```

b) Ejemplos de acceso secuencial por valor:

```
SELECT * FROM cliente ORDER BY edad

SELECT * FROM cliente WHERE edad>=40 AND edad<=50

DELETE FROM cliente WHERE poblacion IN ('Barcelona','Tarragona')
```

Los ejemplos previos muestran accesos por valor (secuenciales o directos) sobre una única columna. Pero también se puede acceder por el valor que muestran diversas columnas. Estos accesos se denominan **accesos por diversos valores**, y pueden ser directos, secuenciales y mixtos (en el caso de que una misma operación o sentencia combine a la vez accesos por valor directo y secuencial). A continuación se muestran algunos ejemplos sobre la tabla de clientes:

a) Acceso directo por diversos valores:

```
SELECT * FROM cliente WHERE poblacion='Arenys de Mar' AND edad=30

UPDATE cliente SET edad=edad+1 WHERE codigo_cliente='Barcelona' AND edad=40
```

b) Acceso secuencial por diversos valores:

```
SELECT * FROM cliente ORDER BY edad, poblacion

SELECT * FROM cliente WHERE edad>30 AND codigo_cliente<100
```

c) Acceso mixto por diversos valores:

```
SELECT * FROM cliente WHERE ciudad='Arenys de Mar' ORDER BY edad

DELETE FROM cliente WHERE edad<25 AND ciudad='Barcelona'
```

Por defecto, y a falta de estructuras de datos que den soporte a las operaciones mostradas en los ejemplos, cualquier operación sobre una tabla de la base de datos significará hacer un recorrido de toda la tabla y preguntar, para cada fila recuperada, si cumple o no las condiciones de búsqueda. Si este sistema ya puede resultar muy costoso por sí mismo, la situación se agrava en función del número de filas que contiene la tabla (por ejemplo, podrían ser decenas de miles), o si en la petición se incluyen combinaciones de tablas (operaciones de *join*) y/o agrupaciones (consultas con cláusula `GROUP BY`). En todos estos casos, la eficiencia puede caer en picado.

Para dar solución a esta problemática se han creado los **índices**, que son estructuras de datos que permiten mejorar el tiempo de respuesta de las peticiones que impliquen un acceso por valor.

El concepto de índice no es nuevo, sino que son estructuras centenarias que forman parte de nuestra vida cotidiana. Ejemplos de índices fuera del ámbito de las bases de datos serían el índice de capítulos de un libro, el índice de conceptos clave de un libro mencionado al inicio de este módulo y el mapa de sitio de una web.

Cabe destacar que los SGBD crean automáticamente algunos índices para gestionar las restricciones de integridad de la base de datos. Entre los índices creados por defecto por los SGBD encontramos los creados sobre la clave primaria y las claves alternativas.

En su versión más simple, un índice permite acceder a las filas de una tabla a partir de los valores de una de sus columnas. La columna sobre la cual se construye el índice se denomina columna indexada. Cada valor distinto *v* de la columna indexada se hace corresponder con el identificador de registro (RID[1]) que apunta a la fila de la tabla que tiene a *v* como valor de la columna indexada. Las diferentes parejas (*v*, *RID*) que se almacenan en el índice reciben el nombre de entradas del índice. Esta estructuración de los datos permite un acceso rápido a los mismos cuando se realizan accesos directos o secuenciales por valor, o se requiere la ordenación de las filas de una tabla de acuerdo a los valores de la columna indexada. El índice descrito en este párrafo responde a

(1)El RID (o *Record Identifier* en inglés) es un identificador que apunta al registro físico donde se almacena la fila de una tabla.

uno de los tipos de índices más simples: el índice único, donde la columna indexada es una clave candidata, es decir, su valor identifica unívocamente cada fila de la tabla con independencia de que esta sea clave primaria o alternativa de la tabla.

A partir de aquí los índices pueden ir sofisticándose según el tipo y número de columnas indexadas y el tipo de estructura de datos utilizada en su implementación. A continuación describimos los distintos tipos de índice en función del tipo y número de columnas indexadas y en las siguientes secciones veremos algunas de las estructuras de datos más utilizadas para crear índices en las bases de datos.

En algunos casos interesará indexar una tabla por una columna que no tome valores únicos, como, por ejemplo, la población de los clientes, que puede repetir valor en diferentes filas de la tabla. Estos casos implican modificar ligeramente la estructura del índice, por ejemplo, haciendo que la entrada del índice asociada a cada posible valor *v* de la columna indexada tenga asociados *N RID* ($\{RID_1, ..., RID_N\}$, $N>0$), donde cada RID_i apuntará hacia una fila de la tabla que tiene el valor *v* en la columna indexada.

También nos podremos encontrar con la necesidad de definir índices sobre múltiples columnas: por ejemplo, un índice sobre las columnas nombre y apellidos de nuestra tabla de clientes. Dichos índices podrán ser únicos o no únicos. En este caso, cada entrada del índice estará representada por una combinación de los valores indexados y apuntará al (los) RID de las filas de la tabla cuyas columnas contienen el valor indicado en la entrada del índice. A la hora de definir el índice, el orden de las columnas puede ser relevante, es decir, un índice definido sobre las columnas nombre y apellidos, puede ser diferente a un índice definido sobre las columnas apellidos y nombre, de tal manera que cada uno de estos índices ayudará a resolver de forma eficiente diferentes tipos de consulta.

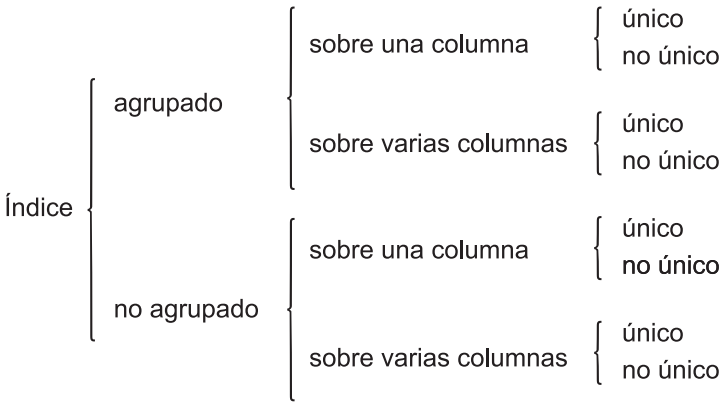
Los índices comentados hasta ahora se denominan índices ordenados porque las columnas indexadas (en general, alfanuméricas) permiten la ordenación de sus valores, hecho que facilita la estructuración del índice. No obstante los datos a los que apuntan, es decir las filas de la tabla, no tienen por qué estar ordenadas. Una alternativa a dichos índices sería almacenar las filas de la tabla de forma ordenada de acuerdo a las columnas indexadas. Esto es lo que se conoce con el nombre de índice agrupado (o *clustered index* en inglés). Ordenar las filas de una tabla puede mejorar significativamente el tiempo de respuesta en la resolución de peticiones que implican un acceso secuencial por valor, ya que pueden minimizar el número de operaciones de E/S. Notad que solo es posible tener un índice agrupado por tabla.

Número de operaciones de E/S

El número de operaciones de E/S es un factor que se debe reducir y permite mejorar significativamente el rendimiento del procesamiento de las consultas.

Tal y como se muestra en la siguiente figura, los índices explicados hasta este momento se pueden clasificar en función de si los datos se guardan de forma ordenada en el fichero que los contienen, del número de columnas que se indexan y de la unicidad de valores de las mismas.

Figura 2. Clasificación de índices



Aparte de los índices previamente presentados, tenemos otros tipos de índices que nos permiten indexar información no textual, como pueden ser los índices geográficos. Estos permiten consultar elementos geográficos dentro de mapas basándose en su posición, y facilitan el uso de funciones, como la distancia entre puntos o la intersección de figuras geométricas.

Existen distintas estructuras de datos que pueden utilizarse para crear índices en bases de datos, como por ejemplo los árboles B+, las funciones de dispersión (en inglés *hash*), los mapas de *bits* (en inglés *bitmap*) los índices parametrizables (GiST y GIN), los árboles R, los Quadtree, etc. En la siguiente sección veremos las estructuras de datos más utilizadas para almacenar y gestionar índices de bases de datos. Luego veremos cómo crear estos índices en Oracle y PostgreSQL. A continuación introduciremos brevemente los conceptos básicos asociados a optimización de consultas, y acabaremos con una serie de recomendaciones que ayuden al lector a saber qué índices se deben crear en cada caso concreto.

3.1. Tipos de índices más habituales

La estructura más utilizada en los índices de bases de datos son los árboles B+, porque se comportan bien en la mayoría de situaciones. No obstante, no hay un tipo de índice universal que funcione bien en todos los casos. Otros tipos de índices, como por ejemplo los basados en funciones de dispersión o en mapas de *bits*, pueden ser más adecuados en ciertas circunstancias. A continuación vamos a explicar qué son y cómo funcionan los índices basados en árboles B+, en funciones de dispersión y en mapas de *bits*. Asimismo, cuando hablemos de PostgreSQL, se introducirán un par de tipos de índices más particulares de dicho SGBD, los índices GiST y GIN.

3.1.1. Árboles B+

La estructura de árbol B+ es la más utilizada para implementar índices en bases de datos relacionales. De hecho, de todos los tipos de índices que hemos presentado, los árboles B+ (tal como los explicaremos o variantes similares) son los únicos que están disponibles en todos los SGBD relacionales.

Los índices basados en árboles B+ son índices ordenados. Por lo tanto, tal y como se indica en la clasificación de la figura 2, permiten indexar una o múltiples columnas, con independencia de si contienen valores únicos o no y de forma ordenada (árbol B+ agrupado) o no ordenada (árbol B+ no agrupado). Estos índices pueden ayudar a resolver eficientemente cualquier acceso por valor, ya sea directo o secuencial. También son especialmente útiles para devolver resultados de forma ordenada, como por ejemplo, en las operaciones SQL que contienen la cláusula `ORDER BY`.

Los árboles B+ son un tipo particular de árbol de búsqueda cuyo objetivo primordial es minimizar las operaciones de E/S en las búsquedas.

Un árbol B+ se compone de nodos que están interrelacionados entre ellos. Las relaciones entre nodos son dirigidas y relacionan un nodo padre (el origen de la relación) con un nodo hijo (el destino de la relación). De hecho, se trata de relaciones de orden entre los valores indexados, que se encuentran distribuidos entre los distintos nodos del árbol. En un árbol no se permiten ciclos, es decir, un nodo no puede estar interrelacionado consigo mismo, ni directa ni indirectamente. Cada nodo del árbol, excepto un nodo especial llamado raíz, tiene un nodo padre y diversos (cero o más) nodos hijo. El nodo raíz no tiene padre, los nodos que no tienen hijos se llaman nodos hoja y los nodos que no son hojas se llaman nodos internos.

Un árbol B+ tiene asociadas las siguientes características:

- **Los nodos hoja y los nodos no hoja** (nodo raíz y nodos internos) **tienen estructuras diferentes**. Los nodos no hoja contienen valores para dirigir la búsqueda hacia los nodos hoja y apuntadores hacia nodos hijo (que pueden ser nodos internos o nodos hoja). Por su parte, los nodos hoja contienen las entradas del índice (es decir, las parejas valor y RID). Además, cada nodo hoja puede contener hasta dos apuntadores más. Un apuntador al siguiente nodo hoja y un apuntador al nodo hoja anterior. Estos apuntadores tienen como objetivo facilitar la resolución de accesos secuenciales (parciales o completos) por valor.

- **Un árbol B+ tiene un número de orden *d* que indica la capacidad de sus nodos y**, en consecuencia, indica también **el número de nodos hijo que puede tener cada nodo interno**. Los nodos tienen como máximo *2d* valores o entradas, dependiendo de si se trata de un nodo interno u hoja, respectivamente. Adicionalmente, los árboles B+ también imponen una

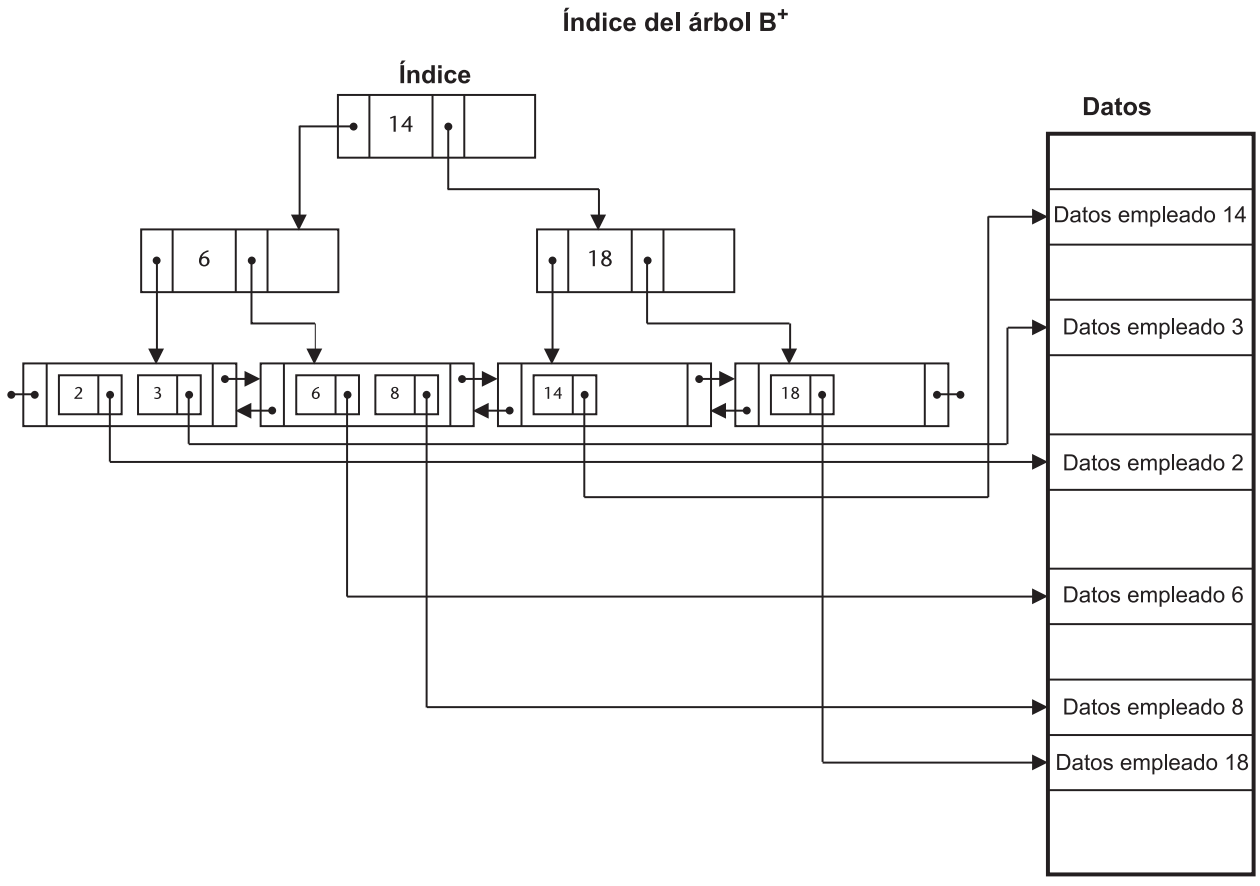
ocupación mínima para los nodos. Esta ocupación mínima es del 50%. Existe un nodo, en concreto el nodo raíz, que está exento de cumplir esta condición. Por lo tanto, en un árbol B+ de orden *d*, todos los nodos internos (a excepción de la raíz) tienen una ocupación mínima de *d* valores (y un mínimo de *d+1* apuntadores a nodos hijo) y una ocupación máxima de *2d* valores (y un número máximo de *2d+1* apuntadores a nodos hijo). En el caso de nodos hoja, tendrán entre *d* y *2d* entradas (y un máximo de 2 apuntadores). La condición de ocupación mínima causa que un árbol B+ sea equilibrado.

- Todos **los valores de un nodo interno deben estar presentes en algún nodo hoja y no pueden estar repetidos**.

- **Los nodos hoja deben contener todas las entradas**. Es decir, todos los valores que en un momento dado existen de la columna (o columnas) sobre la cual se construye el índice, tienen su correspondiente entrada en un nodo hoja.

La siguiente figura muestra un ejemplo de índice de árbol B+ definido sobre una columna de tipo numérico:

Figura 3. Ejemplo de árbol B+ para indexar una columna numérica



Estructura de un árbol B+

La figura muestra un índice árbol B+ de orden *d*=1 que sirve para acceder a datos de empleados según el valor del atributo `IDEmpleado`. Fijaos que los RID que apuntan a los datos solo se encuentran en los nodos hoja, que los nodos internos sirven para buscar los valores de las hojas y que el hecho de que las hojas estén conectadas entre sí facilita el acceso secuencial por valor.

Suponed que la figura anterior se corresponde con un índice único definido sobre la clave primaria (`IDEmpleado`) de una tabla que guarda datos sobre empleados. Suponed también que un usuario plantea la siguiente consulta:

```
SELECT *
   FROM EMPLEADO
   WHERE IDEmpleado > 16;
```

En este caso el SGBD detecta que se está filtrando los valores de la consulta en función de la columna `IDEmpleado` para la cual existe un índice en forma de árbol B+. Supongamos que en este punto el SGBD decide usar el índice. En tal caso, el SGBD cargaría en memoria el nodo raíz del árbol B+. A continuación consultaría el único valor que contiene (14) y lo compararía con el valor a buscar (16). Como 14 es menor que 16, se debe seguir el apuntador de la derecha del valor para acceder al siguiente nodo (recordad que el objetivo de los nodos intermedios es conducir la búsqueda hacia los nodos hoja). Después de cargarse en memoria, se consultaría el valor que contiene dicho nodo, que es 18. Como 18 es mayor que 16, se seguirá el apuntador de la izquierda del valor para acceder al nodo hoja. Este nodo hoja deberá contener valores inferiores a 18. Al llegar al nodo hoja se lee la única entrada que contiene. Dicha entrada tiene el valor de 14, que al ser inferior de 16 no es de interés para la consulta. Como el nodo hoja no contiene más entradas, para devolver los valores con identificador de empleado superior a 16 se sigue el apuntador al siguiente nodo hoja del árbol. En dicho nodo se encuentra una nueva entrada del índice con el valor de 18. Como 18 es mayor que 16 el sistema utiliza el RID de la entrada para acceder a los datos del empleado y devolverlos en la consulta. Al no haber más entradas en el nodo hoja y no haber tampoco otros nodos hoja a continuación, la consulta acaba y se retornan los datos del empleado con identificador de empleado 18.

Los árboles B+ son útiles para:

- **Realizar accesos directos por valor**: primero hay que localizar la hoja que tiene la entrada del valor buscado, y después, utilizar el RID de la entrada para encontrar los datos a los que se quiere acceder.

- **Realizar accesos secuenciales por valor**: primero hay que localizar la primera entrada de la secuencia y utilizar su RID para devolver el primer resultado. Posteriormente se obtendrán los siguientes valores de forma ordenada y sus RID navegando por los nodos hoja del índice.

Las explicaciones previas asumen que el árbol B+ es no agrupado. En el caso que fuese agrupado, podemos ganar eficiencia, especialmente en los accesos secuenciales por valor. En este caso, una vez se ha localizado la primera entrada de interés en un nodo hoja del árbol B+, siguiendo su RID podemos acceder al primer resultado en el fichero que guarda las filas que están siendo indexadas. Como dicho fichero está ordenado físicamente según el valor que toma la columna (o columnas) sobre la que se ha construido el árbol B+, es suficiente realizar la lectura de las páginas de dicho fichero y, por lo tanto, no es necesario consultar los nodos hoja restantes del árbol B+.

De forma similar, en las explicaciones, la tabla sobre la cual se ha construido el árbol B+ se encuentra en un único fichero que solo almacena datos de esa tabla. Algunos SGBD (por ejemplo, este sería el caso de Oracle y PostgreSQL) permiten crear índices particionados sobre tablas que han sido, a su vez, fragmentadas de forma horizontal. En este caso, cada fragmento de la tabla se encuentra almacenado en un fichero diferente que solo almacena datos de esa tabla, y cada uno de estos fragmentos dispone de su propio índice en forma de árbol B+, que únicamente indexa los datos contenidos en el fragmento. Para finalizar, otro tipo de índice que puede ser útil en el diseño físico es el basado en funciones. Este tipo de índice no indexa los valores de una columna (o un conjunto de ellas), sino los valores de una función (o la composición de un conjunto de ellas) que se aplica sobre una o más columnas. Esto puede ser muy útil cuando se hace un uso intenso de una determinada función sobre las mismas columnas de una tabla.

Coste de localización de una entrada

Una práctica habitual al crear árboles B+ es hacer coincidir los nodos del árbol con el tamaño de la página. Así, el número de accesos de E/S necesarios para localizar cualquier entrada del árbol es h, siendo h la altura del árbol.

Como la altura del árbol es directamente proporcional al número de accesos necesarios para encontrar una entrada, es importante crear árboles que tengan una altura tan pequeña como sea posible. Por lo tanto, es necesario que los nodos del árbol sean grandes pero sin sobrepasar el tamaño de una página, porque, de lo contrario, no se podrían consultar con una única operación de E/S. Adicionalmente, nos interesa que los nodos estén tan llenos como sea posible, es decir, nos interesa que el orden d del árbol sea tan grande como sea posible. La idea no es solo que los nodos sean grandes, sino también que estén llenos, es decir, que tengan un índice de ocupación elevado.

Ejemplo: Cálculo del orden de un árbol B+ para ajustar los nodos a las páginas de datos

Suponed que nuestro SGBD trabaja con páginas de 8 Kb (como en el caso de PostgreSQL) y que queremos indexar la columna DNI de una tabla que guarda información de personas. La columna DNI ocupa 8 *bytes* y es la clave primaria de la tabla personas. Consideramos que el tamaño del RID y de los apuntadores a los nodos del árbol es de 4 y 3 *bytes*, respectivamente.

Según estos datos, el cálculo del orden *d* del árbol B+ que permite una ocupación máxima de los nodos, se efectúa de la siguiente manera:

- **Nodo hoja**: en un nodo hoja caben un máximo de *2d* entradas y hasta dos apuntadores a nodos hoja. En nuestro ejemplo, para almacenar cada entrada necesitamos 12 *bytes* (8+4). Y para almacenar los dos apuntadores 6 *bytes* (2*3). Cada nodo hoja tiene un tamaño de 8192 *bytes* (8 Kb). En consecuencia:
  $2d*12 + 6 = 8192 \Rightarrow d= 341{,}0833 \Rightarrow d= 341$, dado que *d* es número entero.

- **Nodo no hoja**: en un nodo no hoja caben un máximo de *2d* valores y *2d+1* apuntadores a nodos hijo. Cada nodo no hoja tiene un tamaño de 8192 *bytes* (8 Kb). En consecuencia:
  $2d*8 + (2d+1)*3 = 8192 \Rightarrow d= 372{,}2272 \Rightarrow d= 372$, dado que *d* es número entero.

El orden *d* de un árbol B+ es único y debe satisfacer las necesidades de almacenamiento de los nodos hoja y no hoja. Por lo tanto, en el caso de nuestro ejemplo, el orden máximo *d* del árbol B+ es 341.

Por otro lado, dado el orden del árbol B+ y las propiedades comentadas anteriormente, es fácil calcular cuántas filas de datos se pueden indexar en un índice en forma de árbol B+ en función de la altura del mismo. A continuación presentamos un ejemplo que indica el número de filas que puede indexar el árbol B+ anterior con alturas 1, 2 y 3, respectivamente.

Ejemplo: cálculo del número de filas que se pueden llegar a indexar en función de la altura del árbol B+

Continuando con el árbol B+ anterior, cuyo orden *d* es 341, la cuestión es cuántas filas diferentes de la tabla de personas podemos llegar a indexar como máximo. Esto dependerá de la altura (*h*) del árbol B+.

- **Si h=1**, el árbol B+ tiene un único nodo (que es a la vez nodo raíz y hoja) que contiene 341*2 entradas, es decir, permite indexar 682 personas diferentes.

- **Si h=2**, el árbol B+ tiene un nodo raíz que contiene 341*2 valores y 341*2+1 apuntadores a nodos hijo, que a la vez, son nodos hoja. Por lo tanto, tenemos 683 nodos hoja, cada uno con 682 entradas. En consecuencia, tendremos un total de 683*682 entradas, es decir, el árbol B+ permite indexar 465.806 personas diferentes.

- **Si h=3**, el árbol B+ tiene un nodo raíz que contiene 682 valores y 683 apuntadores a nodos internos. Tenemos, pues, 683 nodos internos. Cada nodo interno, a su vez, tiene 682 valores y 683 apuntadores a nodos hoja. Por lo tanto, tenemos un total de 466.489 (683^2) nodos hoja. El número total de entradas será 466.489*682, es decir, el árbol B+ permite indexar 318.145.498 personas diferentes (notad que este número ya excedería el total de habitantes de España).

La altura de un índice en forma de árbol B+ condiciona el número de operaciones de E/S necesarias para dar respuesta a los accesos por valor. En particular, en una consulta de acceso directo por valor, solo es necesario consultar *h* nodos, siendo *h* la altura del árbol, para identificar si hay alguna fila de una tabla que satisface los criterios de búsqueda. En caso de que sea así, consultar los datos de la fila requerirá de una nueva operación de E/S para cargar los datos de la fila en memoria. A continuación se muestra mediante un ejemplo el número de operaciones de E/S necesarias para resolver una consulta SQL en el índice anteriormente creado.

Ejemplo: Cálculo de operaciones necesarias de E/S para responder a una consulta de acceso directo por valor utilizando un árbol B+

Imaginemos que nuestro árbol B+ (visto en los dos ejemplos anteriores) tiene altura *h* igual a 3, orden *d* de 341 y que los nodos presentan ocupación máxima. Supongamos que queremos resolver la siguiente consulta:

```
SELECT *
   FROM personas
   WHERE DNI='46742377';
```

¿Cuántas operaciones de E/S son necesarias para localizar la persona con DNI 46742377?

Suponiendo que en la tabla de personas existe una persona con el DNI indicado, serían necesarias 4 operaciones de E/S, 3 en el índice, para localizar el nodo hoja que contiene la entrada que nos interesa, más 1 operación de E/S adicional en el fichero de datos para recuperar los datos de la persona deseada. Si no existe ninguna persona con el DNI necesario serían necesarias 3 operaciones de E/S (las del índice). Imaginemos el número de operaciones de E/S para localizar esa persona entre un total de 318.145.498 personas si no tuviésemos el árbol B+. O incluso peor, imaginemos qué pasaría si la persona con el DNI deseado (46742377) no existiese en la base de datos.

Coste de mantenimiento

Los árboles B+ son árboles equilibrados. Este tipo de árboles tienen características muy interesantes que facilitan la búsqueda de elementos, pero mantener los árboles equilibrados tiene un coste asociado.

Como ya hemos visto, los árboles B+ deben satisfacer la restricción de que todos los nodos (excepto el nodo raíz) deben contener como mínimo *d* valores o entradas dependiendo del tipo de nodo, siendo *d* el orden del árbol. Este hecho requiere que la modificación de los valores de la columna indexada y la inserción y eliminación de filas de la tabla indexada implique reestructurar el índice a menudo. Por ello en tablas y/o columnas con una frecuencia de actualización muy alta el uso de este tipo de índice puede tener un coste de mantenimiento muy elevado.

3.1.2. Tablas de dispersión

Otro tipo de estructura de datos que se utiliza en la creación de índices es la de dispersión (o *hash* en inglés). Este tipo de estructuras permite un acceso directo por valor a los datos muy eficiente, incluso más eficiente que usando árboles B+ en muchos casos. No obstante, este tipo de índices no dan soporte al acceso secuencial por valor, es decir, no resulta de utilidad en búsquedas que utilicen operadores distintos a la igualdad (>, >=, <, <=, <>).

La filosofía de estos índices es la misma que hay detrás de las tablas de dispersión utilizadas para almacenar datos en memoria interna. Aunque en nuestro caso el objetivo es utilizar funciones de dispersión para minimizar el número de operaciones de E/S.

En los índices basados en dispersión, existe una función de dispersión (normalmente llamada *h*) que identifica la página de datos donde se ubicarán las filas de la tabla indexada. La función de dispersión *h* es una función que tie-

ne por dominio el conjunto de los posibles valores de la columna indexada y como rang, las referencias de las páginas disponibles en la base de datos para la tabla indexada: $h(v) \rightarrow p$.

Para cada fila de la tabla indexada la función de dispersión calcula, a partir del valor de su columna indexada, la página de datos donde se deberá almacenar la fila.

Como el valor de la función de dispersión establece la ubicación de los datos de la tabla, en principio solo es posible utilizar un índice de dispersión por tabla. No obstante, en algunos casos, se utilizan tablas auxiliares para permitir una indirección y así soportar el uso de más de un índice de dispersión por tabla, a cambio de incrementar en una unidad el número de operaciones de E/S necesarias en la recuperación de los datos.

Debido a sus características, no tiene sentido que los índices de dispersión sean agrupados, ya que la función de dispersión devuelve directamente la página donde almacenar los datos y no garantiza que datos con valores cercanos tengan valores de h próximos. Hay otras características que complicarían también su agrupación, como la gestión de sinónimos que veremos más adelante.

Normalmente, el número de valores posibles que devuelve la función de dispersión debe ser menor que el número de valores posibles de la columna indexada. Eso se debe a dos hechos:

1) El número de posibles valores puede ser muy superior al número de páginas disponibles.

2) La mayoría de los posibles valores no se utilizan.

Suponed por ejemplo un identificador de empleado de 32 *bits*. Los posibles valores de dicho identificador es superior a 4.000 millones (concretamente es igual a 4.294.967.296=2^{32}). En caso de tener una función de dispersión que devuelve el mismo número de valores que la columna que indexa, necesitaríamos 4.294.967.296 páginas para almacenar la tabla indexada. Si el tamaño de página de nuestro SGBD fuera de 8 Kb se requeriría 32 *terabytes* (2^{32}*****8 *kilobytes*) para almacenar la tabla. Además, cada página solo contendría una fila, con lo cual se estaría malgastando espacio. Como puede comprobarse, esta estrategia de almacenamiento no es precisamente eficiente.

Por todo ello, las funciones de dispersión acostumbran a reducir el número de valores posibles y por lo tanto no son inyectivas sino exhaustivas: es decir, dos valores diferentes (v_i y v_j tal que $v_i \neq v_j$) pueden tener el mismo valor de h: $h(v_i)=h(v_j)$. Cuando esto pasa, los valores v_i y v_j se denominan sinónimos.

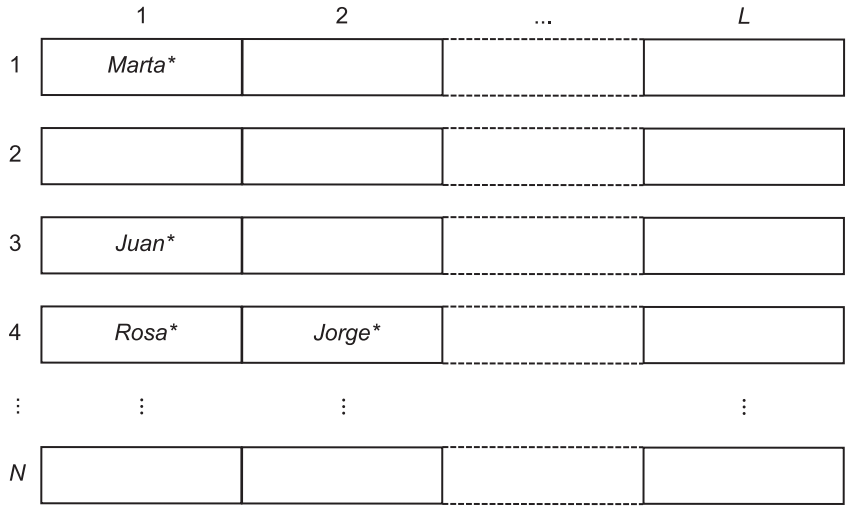
Las filas de los valores sinónimos se almacenan en la misma página. Cuando la página está llena se crea una nueva página, llamada página de excedentes, donde se almacenarán las filas de los nuevos sinónimos y se crea un apuntador en la página original que apunta a la nueva página de excedentes. En caso de que la página de excedentes se llene, se encadenará una nueva página de excedentes, y así sucesivamente. Una situación similar a la descrita también puede acontecer cuando el índice de dispersión se construye sobre una columna que puede tomar valores repetidos. La página inicial que debería contener los datos de interés una vez aplicada la función de dispersión se denomina página primaria, en contraposición a las páginas de excedentes.

Tal y como el lector habrá intuido, la habilidad de la función *h* para evitar valores sinónimos afectará significativamente al número de operaciones de E/S requeridas para la recuperación de los datos de interés.

Ejemplo: Localización de una fila de datos utilizando índices de dispersión

Supongamos que queremos indexar una tabla de empleados con un índice de dispersión según la columna nombre de pila. Supongamos también que *h* tiene los siguientes valores para los siguientes nombres de pila (*Juan* → *3*, *Marta* → *1*, *Rosa* → *4*, *Jorge* → *4*). La siguiente figura nos muestra la distribución de los datos de la tabla en las páginas de datos de la base de datos. En particular, *N* es el número de páginas primarias que tenemos disponibles (que viene determinado por el número de posibles valores de la función de dispersión) y *L* es el número de filas que caben en página. *Nombre** representa los datos completos de la fila de la tabla con valor *Nombre*. Por lo tanto, por ejemplo, en la cuarta página se encuentran los datos de las filas que corresponden a los nombres de *Rosa* y *Jorge*.

Figura 4. Distribución de los datos de una tabla siguiendo un índice de dispersión



Supongamos que el usuario realiza la siguiente consulta:

```
SELECT *
   FROM EMPLEADO
   WHERE nombre = 'Jorge';
```

El SGBD identificaría que la columna *nombre* está indexada según el índice anteriormente mostrado y calcularía *h('Jorge')*. El resultado de dicho cálculo sería el número de página donde se encuentran los datos de la fila (la página 4) o, en el peor de los casos, la página desde donde acceder a sus datos. El sistema cargaría en memoria la cuarta página, consultaría la primera fila de la misma, que equivale a un sinónimo (*Rosa*). Posteriormente consultaría la segunda fila de la página para darse cuenta de que es la que estaba buscan-

do. En ese momento podrían retornar los datos y finalizar con la consulta. El número de operaciones de E/S para resolver la consulta sería 1.

Coste de localización de una entrada

Cuando se utiliza un índice de dispersión, el coste de localizar una fila varía bastante en función de si esta se encuentra en una página de excedentes o no. Si una entrada no se encuentra en una página de excedentes solo habrá que hacer una operación de E/S para obtenerla. En caso contrario habrá que hacer distintas operaciones de E/S hasta encontrar la página de excedentes que contenga la fila buscada.

En consecuencia, para conseguir un buen rendimiento del índice interesa que haya pocas páginas de excedentes. Esto se puede conseguir de dos formas distintas:

- Escoger una función de dispersión que tenga en cuenta la distribución de valores de la columna (o columnas) indexada para minimizar el número de sinónimos.

- Diseñar el índice para reducir el número de páginas de excedentes.

Existe numerosa bibliografía que trata el tema de cómo escoger funciones de dispersión adecuadas para cada caso. Tratar este tema queda fuera de los objetivos de este material, principalmente por la dificultad de definir criterios que sirvan para la mayoría de casos.

Para ver cómo podemos conseguir reducir el número de páginas de excedentes, introduciremos el concepto de factor de carga. Se llama factor de carga al número de entradas que se espera tener, dividido por el número de entradas que caben en las páginas primarias:

$$C = M / (N \times L)$$

M representa el número de entradas que esperamos tener, N el número de páginas primarias (o número de posibles valores de la función de dispersión) y L es el número de filas que caben en cada página. El valor de L es sencillo de calcular ya que se hace dividiendo el tamaño de página por el tamaño de cada fila de la tabla indexada. Idealmente también se debería conocer el valor de M, si no exactamente, al menos sí de forma aproximada.

El factor de carga es un elemento de diseño que permite ajustar los valores de N (modificando la función de dispersión) o de L (cambiando el tamaño de la página de datos) para reducir el número de páginas de excedentes necesarias.

Un factor de carga bajo indica que habrá menos excedentes, minimizando el número de operaciones de E/S necesarias para obtener los datos. En contrapartida, requerirá de más páginas primarias. Por otro lado, un factor de carga

elevado permitirá reducir el número de páginas primarias, pero aumentará el número de páginas de excedentes. Esto puede causar que se requieran más operaciones de E/S para resolver las consultas.

Los índices de dispersión que hemos explicado, que requieren fijar a priori el número *N* de páginas primarias, se conocen también bajo la denominación de índices de dispersión estática. Como alternativa existen los índices de dispersión dinámica. En estos índices no es necesario que el número *N* de páginas primarias esté fijado a priori. En otras palabras, el número de páginas primarias crece en función de las necesidades, a medida que se añaden más filas en la tabla sobre la que se ha construido el índice. Entre los índices de dispersión dinámica más conocidos están el *Extendible hashing* y el *Linear hashing*. Este último está disponible, por ejemplo, en el SGBD Berkeley DB. Asimismo, este tipo de índices, inicialmente diseñados para el ámbito de las bases de datos, también han sido transferidos a lenguajes de programación, como sería el caso de Python que dispone de *Extendible hashing*.

Coste de mantenimiento

En el caso general, las inserciones, modificaciones y las eliminaciones implicará insertar o eliminar filas en las páginas primarias o en las de excedentes. El número de operaciones de E/S dependerán del factor de carga.

Las eliminaciones de filas se pueden hacer de forma lógica (solo marcando la fila afectada como borrada) o de forma física. La eliminación física de una fila puede implicar mover filas de las páginas de excedentes a las páginas primarias, reduciendo la longitud de las cadenas de excedentes.

Las operaciones de modificación sobre una columna indexada implicará mover las filas afectadas a páginas de memoria distintas y, en algunos casos, reestructurar la página primaria con filas de las páginas de excedentes. En este tipo de operaciones el mantenimiento puede llegar a ser más alto que en el caso de los índices basados en árboles B+, debido a que deben moverse las filas de datos de página. En los índices en forma de árbol B+ no agrupados se puede realizar la actualización simplemente modificando la información del índice.

3.1.3. Mapa de *bits*

Otro tipo de índices útiles para indexar columnas con pocos valores posibles que se repiten a menudo son los índices de mapas de *bits* (o índices *bitmap* en inglés). La principal ventaja de este tipo de índice es el poco tamaño que ocupa cuando se usa bajo las condiciones adecuadas, permitiendo incluso tener todo el índice cargado en memoria y evitar operaciones de E/S para consultarlo. Este tipo de índice permite identificar qué filas cumplen una determinada condición utilizando solo operaciones de *bits*. Como las operaciones sobre se-

cuencias de *bits* son mucho más rápidas que sus alternativas sobre otros tipos de datos, este índice permite identificar si una fila cumple las condiciones especificadas más rápidamente.

En un índice de mapa de *bits* se crea una secuencia de *bits* para cada fila de la tabla que indica el valor de la columna indexada. La manera de crear dicha secuencia puede variar, pero en estos materiales explicaremos su forma más simple: tendrá tantos *bits* como posibles valores puede tener la columna indexada. El *bit i*-ésimo de la secuencia de *bits* relativa a una fila tendrá un 1 en caso de que la fila tenga el valor correspondiente a la posición *i* y un 0 en caso contrario. El valor del índice para toda una tabla estará formado por una matriz de *N* filas y *M* columnas, siendo *N* el número de filas y *M* los posibles valores de la columna indexada. Normalmente se denomina mapa de *bits* a la secuencia de *bits* relacionada con el posible valor de un índice (las columnas de la matriz), que indica qué filas de la tabla tienen dicho valor.

Por ejemplo, supongamos que tenemos una tabla de empleados como la que se muestra en la figura 5, con las columnas DNI, nombre, vehículo propio y zona geográfica. Supongamos también que la columna vehículo propio admite valores nulos o bien los valores SÍ/NO, y la columna zona geográfica es obligatoria y admite solo 4 valores (Barcelona, Girona, Lleida, y Tarragona). Como las columnas vehículo propio y zona geográfica tienen pocos valores posibles, podemos considerar adecuado crear dos índices de mapa de *bits*, uno para cada columna.

Figura 5. Ejemplo de representación de dos índices de tipo de mapa de *bits* (columnas vehículo propio y zona geográfica)

Empleado			
DNI	**Nombre**	**Vehículo propio**	**Zona**
88775997	José	SÍ	Barcelona
46781222	María	SÍ	Lleida
E-998272	Jordi	NULL	Girona
87271923	Neus	SÍ	Barcelona
88928187	Elena	No	Tarragona

Vehículo propio	
Sí	**No**
1	0
1	0
0	0
1	0
0	1

Zona geográfica			
Barcelona	**Girona**	**Lleida**	**Tarragona**
1	0	0	0
0	0	1	0
0	1	0	0
1	0	0	0
0	0	0	1

En la figura anterior se puede ver gráficamente la creación de dichos índices. Habrá dos mapas de *bits* del índice sobre la columna vehículo, uno para cada posible valor. Cada mapa de *bits* tendrá 5 *bits*, uno por cada fila de la tabla. El lector podría pensar que se podría utilizar un único mapa de *bits*. No obstante, en este caso es obligatorio utilizar dos mapas de *bits*, ya que la columna admite valores nulos y, por lo tanto, que una fila no tenga un valor cierto no implica necesariamente que sea falso (tal y como muestra la tercera fila de nuestro ejemplo). En la figura también podemos ver el índice de mapa de *bits*

resultante de indexar la columna zona geográfica. Notad que en este caso hay 4 mapas de *bits* que se corresponden con los posibles valores de la columna zona geográfica. Cada mapa de *bits* consta de 5 *bits*.

Una de las ventajas de los índices de mapas de *bits* es el escaso tamaño que ocupan cuando el número de valores diferentes que toma la columna indexada son pocos. En el ejemplo anterior necesitaríamos 10 *bits* (2 valores x 5 filas) para almacenar el primer índice y 20 *bits* (4 valores x 5 filas) para almacenar el segundo. Además, los mapas de *bits* se pueden comprimir y sus ratios de compresión son muy elevadas debido a la distribución homogénea de sus valores.

Este tipo de índice es aplicable solo cuando se realizan operaciones de igualdad y desigualdad (=, <>), inclusión (`IN`, `NOT IN`) o lógicas (`AND`, `OR`, `NOT`). En consecuencia, no es adecuado cuando se realizan operaciones de comparación sobre las columnas indexadas (<, <=, >, >=).

Ejemplo: Resolución de una consulta utilizando índices de mapas de *bits*

Supongamos que el usuario desea consultar los empleados que son responsables de la zona de Barcelona o Lleida y que tienen vehículo propio.

```
SELECT *
   FROM EMPLEADO
   WHERE vehiculo = 'SI' AND zona IN ('Barcelona','Lleida');
```

El SGBD al recibir esta consulta comprobaría que las columnas utilizadas en la cláusula `WHERE` están indexadas por índices de mapas de *bits* (vehículo propio y zona). Suponiendo que el SGBD decidiera utilizar estos índices, los pasos que podría realizar para identificar las filas a devolver serían los siguientes:

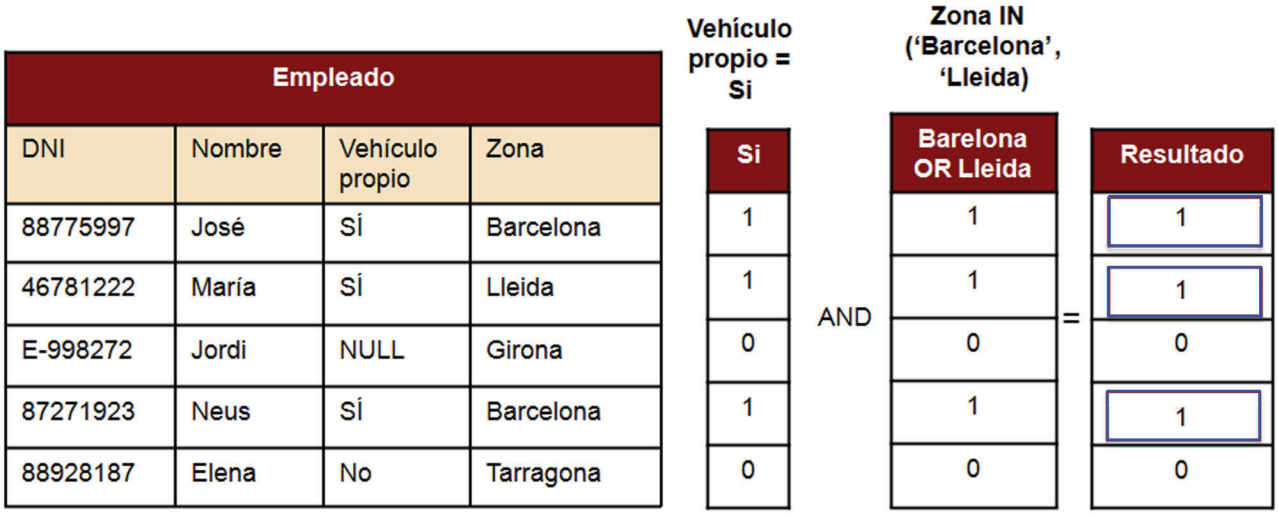
Para identificar los empleados con vehículo propio se debería obtener las filas que tienen un 1 en la primera posición del mapa de *bits*, es decir, las filas con un 1 en el mapa de *bits* del valor *Sí*. El resultado incluye la primera, la segunda y la cuarta fila.

Para identificar a los empleados de las zonas de Barcelona y Lleida, se deben identificar las filas con un 1 en el primer o el tercer mapa de *bits*. Esto se puede hacer usando una operación `OR` lógica entre los mapas de *bits* de los valores Barcelona y Lleida. El resultado incluiría las filas 1, 2 y 4.

Finalmente, se identifican las filas que han satisfecho ambas condiciones: las filas 1, 2 y 4, que se podrían calcular realizando una operación `AND` lógica entre los dos mapas de *bits* resultantes de los pasos 1 y 2.

Podemos ver las operaciones realizadas y el resultado gráficamente en la siguiente figura:

Figura 6. Ejemplo de cómo resolver una consulta utilizando índices de mapas de *bits*



Coste de localización de una entrada

El acceso a los índices de mapa de *bits* es normalmente muy rápido cuando el número de posibles valores de la columna indexada es limitado. A medida que el número de valores posibles de la columna indexada aumenta, el tamaño del mapa de *bits*, y en consecuencia el tiempo de respuesta, se incrementa exponencialmente.

Coste de mantenimiento

Calcular en qué casos es aconsejable un índice de tipo mapa de *bits* puede ser una tarea complicada. Para tomar una decisión con rigor, se debería tener en cuenta, entre otros, el número de filas de la tabla, el número de columnas a indexar, y el número de posibles valores para cada columna. Esta información debería usarse para estimar el tamaño del índice. Normalmente, y para simplificar, se tiende a evitar este tipo de índices cuando el número de valores posibles para la columna a indexar está por encima de los 100, ya que su rendimiento decrece muy rápidamente y su tamaño se incrementa de forma exponencial.

Además, este tipo de índice solo es recomendable para tablas estáticas, es decir, para tablas con pocas actualizaciones. El motivo es que cuando se realiza una modificación sobre una columna indexada, se bloquea todo el índice de mapa de *bits*. Dicho bloqueo impide que se pueda utilizar el índice de nuevo hasta que no se resuelva la transacción que originó el bloqueo. Por lo tanto, si en una tabla de un millón de filas modificáramos el valor de una fila, bloquearíamos el índice entero, provocando que no se pudiera utilizar para acceder a ningún valor de la tabla y ralentizando enormemente otras operaciones que pudieran estar ejecutándose concurrentemente en la base de datos.