Diseño físico de bases de datos

PID_00270599

Blai Cabré i Segarra
Jordi Casas Roma
Dolors Costal Costa
Pere Juanola Juanola
Ivo Plana Vallvé
Àngels Rius Gavidia
Ramon Segret i Sala

Tiempo mínimo de dedicación recomendado: 7 horas

Universitat Oberta de Catalunya

Blai Cabré i Segarra

Ingeniero industrial por la Universidad Politécnica de Cataluña. Ejerce como profesional informático especializado en bases de datos en la empresa IBM.

Jordi Casas Roma

Licenciado en Ingeniería Informática por la Universitat Autònoma de Barcelona (UAB), máster en Inteligencia Artificial Avanzada por la Universidad Nacional de Educación a Distancia (UNED) y doctor en Informática por la UAB. Desde 2009 ejerce como profesor en los Estudios de Informática, Multimedia y Telecomunicación de la Universidad Oberta de Catalunya (UOC), y también como profesor asociado en la UAB. Es director del máster universitario en Ciencia de Datos de la UOC.

Dolors Costal Costa

Doctora en Informática por la Universidad Politécnica de Cataluña. Profesora titular del Departamento de Lenguajes y Sistemas Informáticos de la Universidad Politécnica de Cataluña, asignada a la Facultad de Informática de Barcelona.

Pere Juanola Juanola

Ingeniero técnico de Informática por la Universidad de Gerona (UdG). Tiene más de 20 años de experiencia en Oracle (administración y aplicaciones). Ha administrado entornos de entidades bancarias y ha trabajado en el sector público en la creación de aplicaciones en Oracle. Ha trabajado como responsable de seguridad informática y en calidad informática desarrollando la aplicación de CMMi e ITIL. Desde 1995 ejerce la docencia tanto en universidad presencial como no presencial. Actualmente trabaja como jefe de proyecto en temas de Oracle y bases de datos.

Ivo Plana Vallvé

Ingeniero en informática y máster en la Sociedad de la Información y el Conocimiento por la Universidad Oberta de Cataluña (UOC). Profesional de la informática en la empresa semipública desempeñando las funciones de Director de Informática y Jefe de Transformación Digital. Profesor colaborador de la UOC.

Àngels Rius Gavidia

Doctora en Informática por la Universitat Oberta de Catalunya. Actualmente es profesora de los Estudios de Informática, Multimedia y Telecomunicación de la UOC. Anteriormente ha sido profesora del Departamento de Lenguajes y Sistemas Informáticos de la UPC y personal docente colaborador de los Estudios de Informática, Multimedia y Telecomunicación de la UOC.

Ramon Segret i Sala

Ingeniero industrial y licenciado en Informática. Ha ejercido como profesional informático especializado en bases de datos en la empresa IBM. Actualmente, es profesor de los Estudios de Informática y Multimedia de la UOC.

La revisión de este recurso de aprendizaje UOC ha sido coordinada por la profesora: Àngels Rius Gavidia (2020)

Sexta edición: febrero 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

El diseño físico de bases de datos constituye la cuarta etapa en el proceso de diseño de una base de datos. En las etapas anteriores se ha llevado a cabo el análisis de requisitos, el diseño conceptual y, finalmente, el diseño lógico de la base de datos. En este módulo veremos el proceso de transformación del modelo lógico, obtenido en la etapa anterior, hacia un modelo físico que nos permita obtener una implementación sobre un sistema de gestión de bases de datos (SGBD).

Antes de iniciar esta etapa, hay que seleccionar el SGBD concreto sobre el cual se quiere implementar la base de datos. En este módulo nos centraremos en las bases de datos relacionales; por lo tanto, el objetivo es ver el proceso de transformación de un modelo lógico relacional hacia un modelo físico y concretar un SGBD relacional.

Este módulo se estructura en tres partes:

1) En la primera parte veremos cómo hay que estructurar y almacenar la información de la base de datos en un soporte físico no volátil para que pueda ser recuperada. Presentaremos los niveles lógico, físico y virtual de las bases de datos (apartados 1, 2, 3 y 4).

2) En la segunda parte de este módulo veremos el proceso de transformación del modelo lógico relacional hacia el modelo físico, que nos permitirá una implementación sobre un SGBD concreto de la base de datos (apartado 5).

3) Finalmente, en la tercera parte, veremos el funcionamiento de los métodos principales de acceso a los datos (apartado 6).

Objetivos

En los materiales didácticos de este módulo, encontraremos las herramientas indispensables para alcanzar los objetivos siguientes:

1. Conocer la estructura física que utiliza la base de datos para almacenar los datos de manera no volátil.

2. Conocer la funcionalidad y la estructura del nivel virtual y del nivel físico.

3. Aprender a realizar el diseño físico de la base de datos a partir de su diseño lógico adaptado a las características de un SGBD concreto.

4. Definir los índices necesarios y convenientes en cada tabla para que las aplicaciones tengan un buen rendimiento cuando accedan a la base de datos.

5. Conocer los diferentes métodos de acceso necesarios para efectuar consultas y actualizaciones en los datos almacenados en las bases de datos.

6. Comprender la utilidad de los índices para la implementación de los accesos por valor.

7. Conocer la estructura de los índices de árboles B^+.

1. Conceptos previos

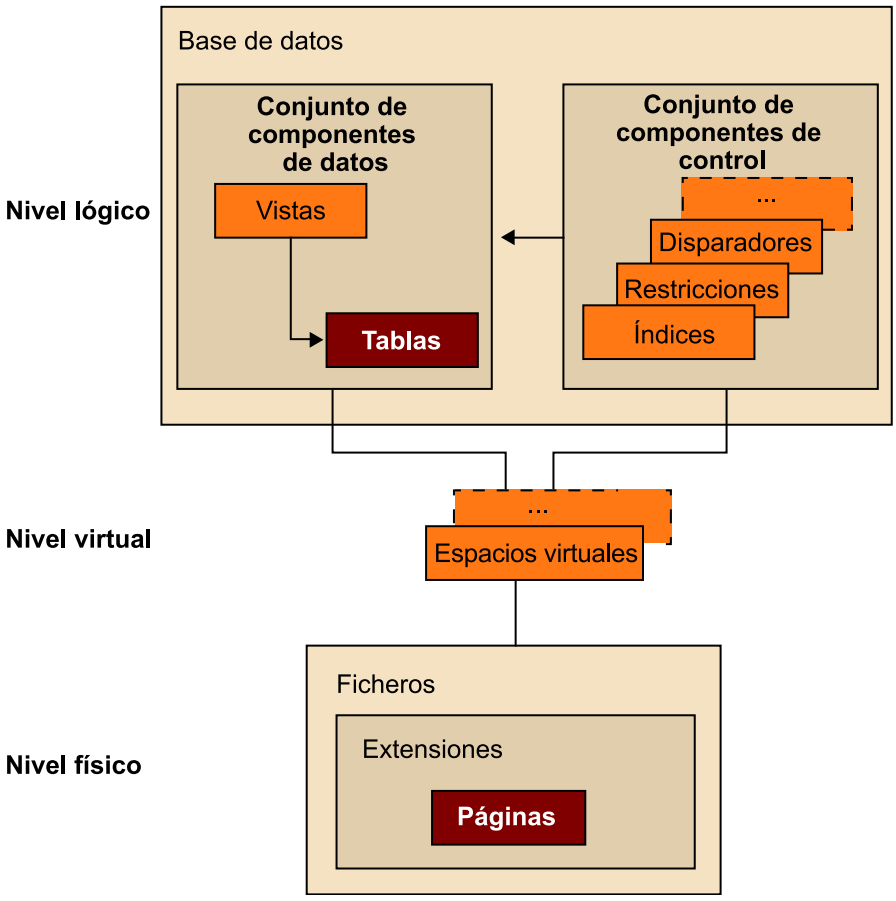
La manera más sencilla de explicar una base de datos (BD) es decir que se trata de un conjunto de datos persistentes relacionados entre sí. La característica de persistencia implica que los datos no han de desaparecer entre ejecuciones sucesivas de programas, aunque entre estas ejecuciones transcurra un intervalo de tiempo largo. Este requisito obliga a su vez a tener los datos entre ejecución y ejecución almacenados en un medio no volátil, normalmente en un disco magnético.

Para comprender qué relación hay entre los datos tal como los ve el programador o el usuario final y tal y como están almacenados en el soporte no volátil, utilizaremos una arquitectura de tres niveles, que denominaremos *arquitectura de los componentes de almacenamiento*, representada en la figura 1.

¿Qué es una arquitectura?

Una arquitectura, como cualquier otra manera de esquematizar la realidad, es una herramienta sencilla y potente ideada para abstraer y entender los rasgos fundamentales de los sistemas más complejos.

Figura 1. Arquitectura de los componentes de almacenamiento



Lo primero que vemos es que la arquitectura de los componentes de almacenamiento es de tres niveles:

1) El **nivel lógico** corresponde a todos los componentes que, de una manera más o menos directa, son conocidos y manipulados por el programador o por el usuario final. Estos componentes forman la interfaz externa o de usuario del sistema de gestión de la base de datos (SGBD). De manera muy simplificada, podemos considerar que en el nivel lógico hay un conjunto de datos estructurados en tablas. De hecho, todos los demás componentes lógicos giran en torno a las tablas.

2) Las tablas deben ser persistentes; por este motivo se almacenan en un soporte no volátil y utilizan unas estructuras determinadas, que son los componentes que constituyen el **nivel físico**.

3) Entre los niveles físico y lógico existe un tercero que denominaremos **nivel virtual**. Este nivel proporciona al SGBD[1] una visión simplificada del nivel físico, como ya veremos.

Ya conocéis los componentes del nivel lógico. En este módulo presentamos los componentes de los otros dos niveles, el virtual y el físico, que denominamos conjuntamente *componentes de almacenamiento* porque controlan la disposición física de los datos en el soporte no volátil.

[1] Abreviamos *sistema gestor de bases de datos* con la sigla SGBD.

Terminología

En este módulo denominamos *programador* a la persona que confecciona programas que interaccionan con la base de datos, normalmente mediante sentencias de SQL, y *usuario final*, a la persona que interacciona con los datos directamente por medio de una interfaz gráfica.

2. El nivel lógico

Aunque los componentes del nivel lógico ya se estudian en otro módulo didáctico, recordaremos brevemente los conceptos más importantes para poder relacionar los componentes lógicos con los de los niveles físico y virtual.

Ved también

Podéis ver los componentes del nivel lógico en el módulo "Diseño lógico de bases de datos" de esta asignatura.

2.1. Componentes lógicos

En la figura 1 podemos ver que el componente lógico más importante es la tabla y, para reflejarlo, la hemos representado con un tramado. El conjunto de tablas es el núcleo fundamental de la base de datos por dos motivos: es el conjunto de datos disponibles para el usuario, y es también el conjunto más importante y voluminoso de datos almacenados en los soportes físicos.

El usuario puede acceder directamente a las tablas o bien puede acceder a estas de manera indirecta por medio de las vistas; por este motivo, tablas y vistas están agrupadas como el conjunto de componentes de datos.

Una base de datos tiene, aun así, otros componentes que no son las tablas y las vistas, pero que están estrechamente relacionados con éstas. Se trata, entre otros, de las restricciones[2], que definen ciertas reglas que tienen que cumplir los datos, o los disparadores[3], los cuales indican acciones que se tienen que ejecutar si se cumplen ciertas condiciones. Denominamos todos estos otros componentes *conjunto de componentes de control*, puesto que en cierto modo permiten controlar los datos dinámicamente.

(2)En inglés, *constraints*.

(3)En inglés, *triggers*.

También se puede considerar que los índices pertenecen al conjunto de componentes de control, aunque con un matiz ligeramente diferente, dado que su función fundamental es de rendimiento: facilitan el acceso a los datos en un tiempo razonable.

3. El nivel físico

Como ya hemos dicho, los datos se almacenan en soportes no volátiles, normalmente en discos magnéticos en la mayoría de los sistemas informáticos (desde los grandes sistemas hasta los ordenadores personales). Estos soportes son controlados por el sistema operativo de la máquina, que es el que realmente efectúa la lectura y la escritura física de los datos. Los SGBD no reimplementan estas funciones, sino que utilizan las rutinas especializadas del sistema operativo (SO) para leer y escribir los datos y también para la gestión física de los dispositivos.

Los sistemas operativos gestionan los datos en los discos magnéticos a partir de unas unidades globales denominadas *ficheros*[4]. Normalmente, el sistema operativo no reserva una gran cantidad de espacio de disco para cada fichero, sino que lo va adquiriendo a medida que lo necesita. La unidad de adquisición es la denominada *extensión*[5], otro de los componentes de este nivel. Finalmente, la extensión es un múltiplo entero del componente físico más pequeño, que es la página. La *página*[6] es el elemento que contiene y almacena los datos del nivel lógico, y por este motivo también la hemos tramado en la figura 1. A continuación veremos estos tres componentes, del menor al mayor.

(4)En inglés, *files*.

(5)En inglés, *extent*.

(6)En inglés, *page*.

3.1. La página

Para entender qué es una página de una BD[7] o de un SGBD, no partiremos de una definición, sino de una descripción de lo que es una página en una BD desde dos puntos de vista diferentes, aunque complementarios:

(7)Abreviamos *base de datos* con la sigla BD.

- Los datos se almacenan en dispositivos externos pero, por otro lado, sabemos que para efectuar cualquier operación tienen que estar presentes en la memoria principal del ordenador. Debe haber, pues, un transporte de datos entre la memoria externa (que normalmente es un disco) y la memoria interna (o memoria principal). Este transporte se hace empleando una unidad discreta de transporte de datos que en los SO[8] se denomina *bloque* y en los SGBD recibe el nombre de *página*.

(8)Abreviamos *sistema operativo* con la sigla SO.

- Paralelamente al hecho de ser la unidad de entrada/salida, la página también es la unidad de organización de los datos almacenados. El espacio del disco siempre se asigna a un número múltiple de páginas, y cada página puede estar direccionada de manera individual.

Concretando, diremos que la **página** es la unidad mínima de acceso y transporte del sistema de entrada/salida (E/S) de un SGBD, lo cual la hace ideal para ser a su vez la unidad de organización más importante de los datos almacenados.

Observad que en la definición anterior hemos definido la página como "unidad mínima de acceso y transporte". La razón es que, del mismo modo que nunca se accede a menos de una página, sí que se puede acceder al mismo tiempo a un número entero de páginas consecutivas. Esta cuestión, que veremos más adelante, no invalida la explicación que hemos dado hasta ahora de la página ni su importancia dentro de la arquitectura de componentes. Se trata sólo de una opción de rendimiento del sistema.

Que la página sea la unidad de organización del nivel físico no significa que no tenga su propia estructura interna. De hecho, en una página hay diferentes componentes. A estos componentes que hay en el interior de la página sólo pueden acceder las rutinas del SGBD, una vez la página ya está en la memoria principal del ordenador. En cambio, a la página como unidad accede (para leer y escribir) el subsistema de E/S[9] del SO de la máquina.

[(9)]Abreviamos la expresión *entrada/salida* con la sigla E/S.

Un símil artístico

Imaginad que la página es un cuadro. El SGBD sería entonces el pintor; es quien ve el interior del cuadro y lo llena (lo pinta). Una vez listo, lo envuelve y llama a su mozo (el sistema operativo), que toma el cuadro y, sin verlo, lo transporta al almacén.

3.1.1. Estructura de una página

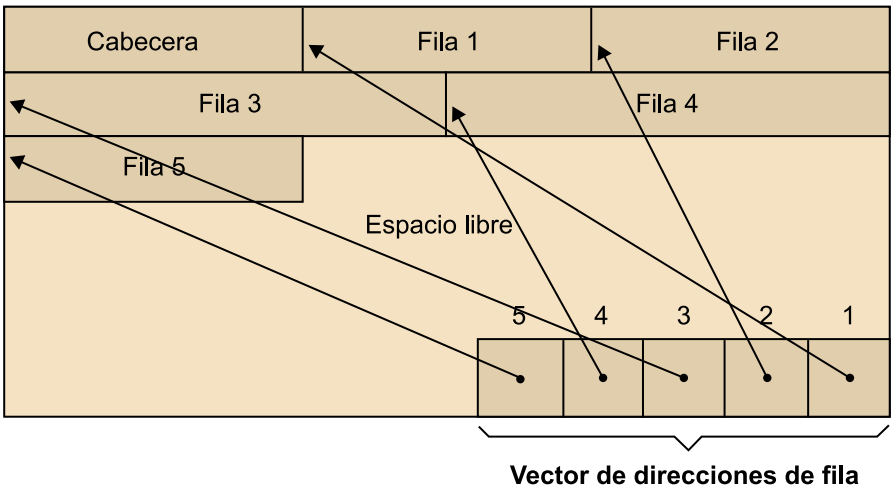
Para facilitar la explicación, ahora nos centraremos en las páginas que almacenan tablas. Estas páginas se suelen denominar *páginas de datos*. Más adelante, ampliaremos algunos detalles para generalizar la estructura de la página.

El tamaño de una página

Los valores más corrientes del tamaño de una página actualmente son 2 kB, 4 kB y 8 kB. Posiblemente, 4 kB es el tamaño más habitual.

La página es de longitud fija. Hay SGBD que permiten elegir el tamaño de la página entre un pequeño repertorio. Otros solo tienen un tamaño único. La estructura de la página, que se muestra en la figura 2, es muy similar en todos los SGBD.

Figura 2. Estructura de una página



Una página consta de los elementos siguientes, enumerados desde la dirección más baja de la página hasta la más alta:

a) Una **cabecera**, con información de control de la página.

b) Un cierto número de registros, que aquí denominaremos **filas** porque corresponden a las filas de las tablas del nivel lógico. Este número puede variar con el tiempo, y en un momento concreto puede ser igual a cero.

c) Un **espacio libre**.

d) Un **vector de direcciones de fila** (**VDF**), que tiene tantos elementos como filas hay en la página. Cada elemento del VDF[10] contiene la dirección de una fila dentro de la página (apunta a la fila). El elemento que ocupa la posición de dirección más alta en el VDF apunta a la fila que ocupa la dirección más baja en la página, tal y como podéis ver en la figura 2. La fila siguiente (a continuación de la primera) está apuntada por el elemento del VDF a la izquierda del primer elemento.

[10]Abreviamos la expresión *vector de direcciones de fila* con la sigla VDF.

Así pues, las filas se van colocando de izquierda a derecha en el orden creciente de las direcciones dentro de la página, mientras que los elementos del VDF se van colocando de derecha a izquierda desde la posición más alta y siguiendo en el orden decreciente de las direcciones. De este modo, el espacio libre siempre queda hacia la mitad de la página.

3.1.2. Estructura de una fila

El elemento que hemos denominado *fila* cuando hemos hablado de la estructura de la página (nivel físico) generalmente registra toda la información correspondiente a una fila de una tabla (nivel lógico). Por esta razón, su estructura es sencilla. Se trata de una secuencia de campos (nivel físico), cada uno de los cuales registra la información del campo de la fila correspondiente (nivel lógico).

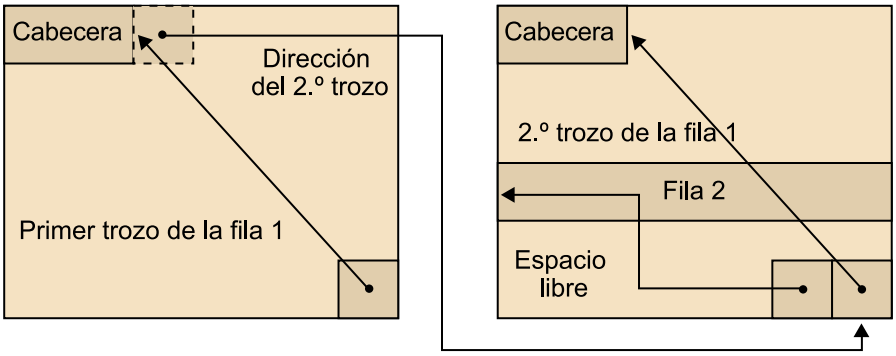
Así pues, en cuanto a filas y campos, la correspondencia entre el nivel lógico y el físico es muy directa.

Es posible que el SGBD necesite guardar alguna información de control en lo que respecta a la fila y, por este motivo, antes de la secuencia de campos encontramos una cabecera de fila. Podéis ver la estructura de la fila en la figura 3.

Figura 3. Estructura de una fila

Normalmente, las filas de una tabla son de longitud inferior al tamaño de las páginas y por este motivo caben en las mismas perfectamente. De todos modos, si la longitud de la fila es superior a la de la página, los SGBD parten la fila en trozos. Cada trozo se almacena en una página diferente y apunta al trozo siguiente, como podéis observar en la figura 4.

Figura 4. Almacenamiento de una fila más larga que una página



No os preocupéis ahora si en la figura no acabáis de entender la manera concreta como un trozo de fila apunta al siguiente. Lo veréis más adelante.

3.1.3. Estructura de un campo

Hemos visto que una página está formada por filas y que una fila está formada por campos. Un campo de una fila de una tabla (nivel lógico) se almacena como uno de los campos de la fila dentro de la página (nivel físico).

Un campo normalmente está formado por una cabecera y un contenido, tal como indicamos en la figura 5.

Figura 5. Estructura de un campo

La **cabecera** del campo permite guardar ciertas características del campo, como por ejemplo:

a) La indicación de si el contenido en cada momento es nulo o no lo es. Esta indicación es necesaria si el campo está definido de manera que admite el valor nulo.

b) La longitud del campo, también en un momento concreto. Esta información es necesaria cuando el campo se ha definido con un tipo de dato de longitud variable, como por ejemplo el tipo *CHARACTER VARYING*.

CHARACTER VARYING

CHARACTER VARYING es el nombre del estándar SQL:1999 para el tipo de dato "cadena de caracteres de longitud variable", que en varios SGBD comerciales se denomina *VARCHAR*.

El **contenido** del campo es el resultado de almacenar el valor del campo según las normas de codificación de cada SGBD en cada arquitectura de máquina concreta. A continuación, encontraréis el formato de almacenamiento de los tipos de datos más comunes en los diferentes SGBD:

[11]Bytes codificados en código ASCII o bien en el código que utilice la máquina donde funciona el SGBD.

- Tipo *SMALLINT*: número binario entero de 16 bits.
- Tipo *INTEGER*: número binario entero de 32 bits.
- Tipo *FLOAT*: número en coma flotante de 64 bits.
- Tipo *CHARACTER*(*n*) o, de manera abreviada, *CHAR*(*n*): cadena de *n*bytes[11], donde *n* es la longitud definida en el tipo de dato.
- Tipo *CHARACTER VARYING*: como el caso anterior, con la excepción de que aquí *n* es la longitud real del valor concreto que el campo tenga en cada momento.
- Tipo *DATE*: aunque externamente es una cadena de 8 dígitos (4 para el año, 2 para el mes y 2 para el día), internamente se puede almacenar de manera que ocupe mucho menos espacio, por ejemplo 4 bytes, siguiendo un patrón de codificación propio de cada SGBD.

Ahorrar espacio en disco

Los SGBD intentan ahorrar espacio en el disco y tiempo de proceso siempre que sea posible. Por esto, si el campo no admite nulos y es de longitud fija (por ejemplo, *CHAR*(3)), entonces esta información no es necesaria y podemos prescindir de la cabecera. En este caso, el campo ocupa solo el espacio necesario para almacenar su valor.

3.1.4. Gestión de la página

Después de ver cómo se organiza el espacio de una página y cuáles son sus contenidos, describiremos brevemente las principales operaciones que un SGBD hace de este espacio y estos contenidos. Son las siguientes:

1) Formateo de una página. Cuando el SGBD necesita más espacio dentro de un fichero para almacenar datos, adquiere una nueva extensión. Las páginas de esta extensión se formatean, es decir, se inicializan de una manera adecuada para que posteriormente las utilice el SGBD. La **inicialización** consiste fundamentalmente en escribir la cabecera y dejar el resto de la página como espacio libre.

2) Carga inicial de filas. Si la extensión inicializada se utiliza para una carga inicial o masiva de filas de tablas, el SGBD empieza a usar el espacio libre de la página de la manera siguiente:

- La primera fila se coloca detrás de la cabecera, se crea un elemento del VDF que apunta a esta fila y se coloca al final de todo de la página.

- La segunda fila ocupa el lugar inmediatamente detrás de la primera, y su elemento del VDF ocupa el lugar justo delante del elemento de la fila anterior.

El proceso se repite sucesivamente, como ya habéis visto en la explicación de la estructura de una página.

De este modo el espacio libre va disminuyendo, pero siempre ocupa un lugar en torno al centro de la página. El administrador de la base de datos (DBA[12]) normalmente habrá fijado un tanto por ciento de espacio libre mínimo que el SGBD tiene que respetar en una carga masiva de filas. Esto significa que, cuando el proceso de carga detecta que el espacio que queda libre en una página es precisamente este espacio libre mínimo, ya no coloca más filas y empieza a hacerlo en la página siguiente.

(12)Abreviamos *administrador de la base de datos* con la sigla DBA, de la expresión inglesa *database administrator*.

Administrador de la BD

El administrador de la BD (*database administrator*, DBA) es la persona -o equipo de personas- encargada, entre otras cosas, de la gestión de los componentes de almacenamiento.

La finalidad de esta limitación es dejar un cierto espacio libre en todas las páginas para que sea aprovechado para las operaciones que veremos a continuación.

Añadir una fila

Las filas se añaden con la sentencia *INSERT*.

3) Alta posterior de una nueva fila. Para añadir una fila a una tabla de la BD, el SGBD en el nivel físico localiza primero la página donde tiene que añadir la fila. Esta página se denomina *página candidata*. Una vez el SGBD sabe cuál es la página candidata, utiliza el espacio libre disponible en la página para colocar la fila y dejarla apuntada por un nuevo elemento del VDF. La fila y el elemento del VDF se colocan según el criterio explicado anteriormente.

Puede llegar un momento en el que ya no quede espacio libre suficiente en la página candidata para colocar la fila. Entonces, la fila se coloca en otra página según algoritmos propios de cada SGBD.

Reflexión

Las diferentes formas de localizar una página quedan fuera del alcance de este módulo.

4) Baja de una fila. La baja de una fila consiste en liberar el espacio ocupado por la fila y el elemento del VDF. En este momento, o más adelante, el SGBD reorganiza la página para que el espacio liberado por la baja se una al resto del espacio libre de la página. Habrá un desplazamiento de filas y de elementos del VDF para que la estructura de la página continúe siendo tal y como hemos descrito anteriormente.

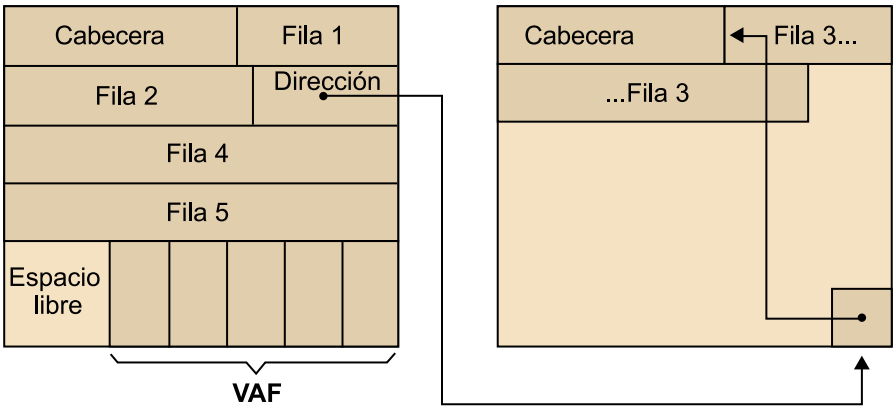
5) Cambio de longitud de una fila. Un cambio de longitud de una fila puede afectar de tres maneras a la reorganización del espacio de la página:

a) Si el cambio tiene como resultado una disminución de la longitud de la fila, el espacio liberado se unirá al espacio libre según el mecanismo que acabamos de explicar.

b) Si, por el contrario, da lugar a un aumento de la longitud y el espacio extra requerido está disponible en forma de espacio libre dentro de la página, la fila quedará en su lugar ocupando más espacio y las filas que la siguen se desplazarán hacia la derecha (hacia direcciones más altas).

c) En caso de que la necesidad de espacio no se pueda servir a partir del espacio libre de la página, esta fila se traslada hacia otra página que disponga de espacio suficiente. En la página original y en la posición original se crea una dirección que apunta hacia la nueva posición de la fila, como se indica en la figura 6.

Figura 6. Fila situada fuera de la página original



Como en el caso de las filas que ocupan más de una página, no os preocupéis tampoco si no acabáis de entender, a partir de la figura, la manera como se apunta a la fila que se ha trasladado de la página original a la nueva. Lo veréis más adelante.

3.1.5. Otros tipos de páginas

Hasta ahora nos hemos centrado en las páginas que contienen filas de tablas, las páginas de datos. Cómo sabéis muy bien, en el nivel lógico hay otros componentes. De todos modos, la estructura de página que acabáis de ver sirve de base para entender los otros tipos de páginas.

En efecto, dejando los índices aparte, todos los demás componentes (vistas, restricciones, disparadores, etc.) son definiciones que alguien ha proporcionado al SGBD mediante el lenguaje SQL. Estas definiciones, como explicamos más adelante, se guardan en unas tablas diseñadas especialmente para contenerlas, pero que a todos los efectos que aquí nos interesan son tablas como las que contienen los otros datos. Así pues, una vez en el nivel físico, todo lo que hemos visto hasta ahora es válido para las tablas que contienen estos componentes.

Los índices son un caso ligeramente distinto, y también lo son las tablas con datos del tipo objeto grande. La estructura del bloque que utilizan es similar, pero varía su contenido.

Reflexión

Queda fuera del alcance de este módulo el estudio en profundidad de los diferentes tipos de páginas.

3.2. La extensión

Una **extensión** es un número entero de páginas, en principio consecutivas, que el sistema operativo adquiere a petición del SGBD cuando éste detecta que necesita más espacio para almacenar datos. Esta adquisición es automática (sin intervención humana explícita).

Algunas estrategias del SO

Algunas de las estrategias que utiliza el SO cuando no puede asignar páginas físicamente consecutivas son, por ejemplo, partir la extensión en trozos o devolver un aviso que indique que no hay suficiente espacio.

Es importante notar que cuando a petición del SGBD, el SO gestiona los datos en el soporte no volátil de los periféricos, lo hace siempre dentro del marco de un fichero. Así, la adquisición de una extensión también tiene lugar dentro de este marco. Esto significa que cuando se detecta que falta espacio en un fichero, se adquiere una extensión para este fichero. La extensión está, pues, asociada a un fichero, es parte de él. Y el fichero es, entre otras cosas, un conjunto de extensiones.

Las páginas de una extensión deben ser físicamente consecutivas. Si el SO no las puede asignar porque físicamente no están en el dispositivo, seguirá estrategias diferentes.

La razón de la contigüidad de las páginas de una extensión

La razón principal por la cual las páginas en una extensión deben ser contiguas es que, de este modo, los componentes que son lógicamente consecutivos (por ejemplo, las filas de una tabla) se pueden almacenar de manera consecutiva y se pueden recuperar del mismo modo, y así se favorecen los tratamientos que impliquen la recuperación consecutiva de todos los componentes (tratamientos que son frecuentes en las BD relacionales).

Por otro lado, los mecanismos más avanzados de mejora de rendimiento que encontraréis esbozados en el subapartado que nos da una visión general de la E/S en un SGBD, tienen sentido sólo si las páginas sobre las que actúan son consecutivas.

3.3. El fichero

El **fichero** es la unidad que usan los SO para gestionar el espacio en los dispositivos periféricos. También es un conjunto de extensiones.

Los SGBD aprovechan la funcionalidad que proporcionan los SO y no interaccionan directamente con los ficheros. Este es el motivo de que muchas veces cueste encontrar el término *fichero* de manera explícita en libros o manuales de SGBD. Incluso hay SGBD relativamente pequeños en los que todos los datos que controlan están almacenados en un único fichero, lo cual desvirtúa la importancia de este componente.

De todos modos, los ficheros están siempre presentes en el almacenamiento: los distintos datos de las BD están en las páginas; éstas se agrupan en extensiones, y estas últimas, en ficheros. En resumen, podemos decir que el fichero es para el SGBD un conjunto de páginas que se han ido creando de extensión en extensión.

Extensiones de un fichero

Normalmente, los SGBD permiten definir dos tipos de extensiones. Una se denomina **extensión primaria** y la otra, **extensión secundaria**. Estas dos extensiones pueden ser –y normalmente lo son– de longitud diferente, pero ambas son un múltiplo de la página.

Cuando un fichero necesita espacio por primera vez, se adquiere el número de páginas correspondientes a la extensión primaria. A partir de este momento, las extensiones sucesivas que se vayan necesitando en este fichero adquirirán el número de páginas de la extensión secundaria. Para un fichero solo hay una extensión primaria (la primera), pero hay varias secundarias (las siguientes). El número máximo de extensiones secundarias lo fija el sistema operativo o el SGBD, y suele ser una potencia de dos (16, 128, 256, etc.).

3.4. Visión general de las E/S en un SGBD

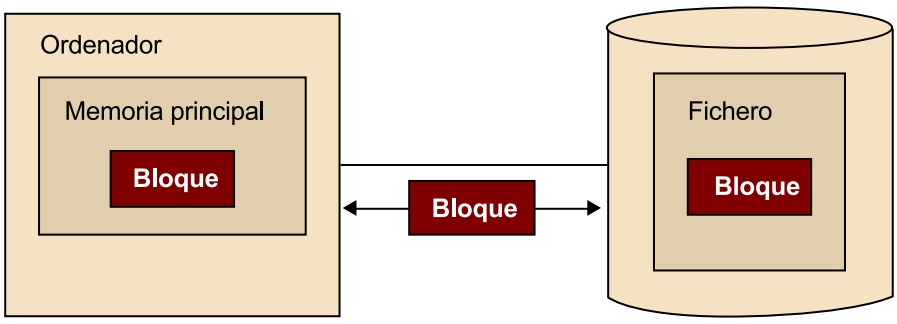
Recordad que hemos definido la página como una unidad de almacenamiento y de transporte. Dado que hasta ahora sólo hemos hablado de almacenamiento, para finalizar la exposición del nivel físico trataremos el transporte.

El procesamiento de datos persistentes, que se efectúa en la memoria principal del ordenador, como ya habéis visto, requiere un transporte de datos entre unidad central y periféricos. A continuación, explicamos brevemente este transporte y el comportamiento de la entrada/salida en el caso de los ficheros clásicos y en los SGBD.

1) Entrada/salida de ficheros clásicos

En el caso de la gestión de los denominados *ficheros clásicos* o *ficheros tradicionales* –puesto que no pertenecen a una base de datos– este transporte es muy sencillo conceptualmente, tal y como podéis ver en la figura 7.

Figura 7. E/S en los ficheros clásicos



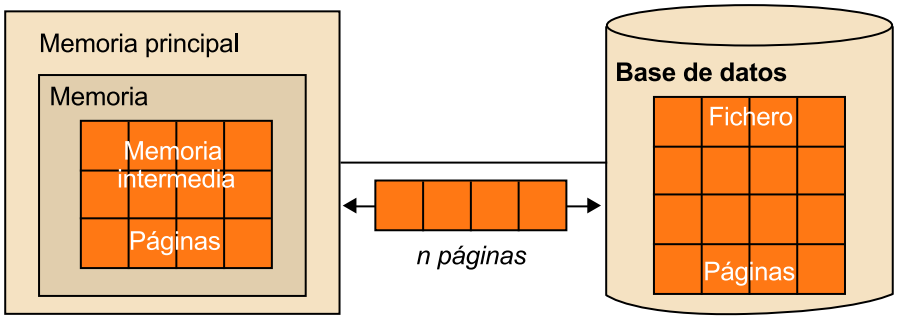
Para leer datos, por ejemplo, el SO desencadena una operación de E/S para llevar hacia la memoria principal el bloque que contiene los datos que se necesitan. Este bloque es una unidad físicamente delimitada y direccionable en el conjunto de los datos grabados en el soporte no volátil (el disco magnético).

Después, el bloque viaja por el canal que une el periférico con la unidad central del ordenador y, finalmente, el bloque se coloca en la memoria principal y puede ser tratado por los programas.

2) Entrada/salida en las bases de datos

En el caso de los SGBD, este mecanismo, aunque básicamente es el mismo, es algo más elaborado. Tiene unas características propias que muestra la figura 8 y que describimos a continuación:

Figura 8. E/S en las bases de datos



a) La longitud de la página es fija. Normalmente, la página tiene la misma longitud en todos los ficheros que constituyen las BD. Esto se contrapone al caso de los ficheros clásicos, en los que la longitud del bloque suele ser diferente para cada uno, puesto que se elige la longitud que parece más adecuada para los datos almacenados.

Entonces, y respecto a los SGBD, podemos plantearnos la pregunta siguiente: si los componentes que encontramos en las páginas (como por ejemplo, las filas o las entradas de índice) son, en principio, de longitud variable, ¿por qué los encapsulamos en un marco de longitud fija (la página)? La respuesta es fácil: al basar el almacenamiento en una unidad fija para todo el SGBD, la gestión que este tiene que hacer de la E/S puede ser mucho más sencilla, lo que finalmente da lugar a un rendimiento mejor.

b) La optimización del tiempo de E/S, que permite ofrecer tiempos razonables de procesamiento cuando se tratan cantidades importantes de datos, es otra característica de los SGBD. Los diferentes SGBD han elaborado distintos mecanismos, algunos bastante sofisticados. Mencionamos brevemente algunos:

- Aprovechar la operación física de E/S para leer y llevar a la memoria más de una página consecutiva a la vez, lo cual ahorra tiempo de transporte por unidad leída (página).

- Avanzar en el tiempo la lectura de una página cuando se prevé que se necesitará próximamente. Así, en el momento en que un proceso la necesite, no tendrá que esperar el tiempo de lectura, puesto que la página ya estará en la memoria.

- Retener en la memoria principal páginas modificadas, incluso después de ser escritas en el soporte no volátil, con el fin de ahorrarse una nueva lectura si algún otro proceso las necesita.

Recordad

Cuando hemos definido la página habéis visto que, a pesar de que normalmente es la unidad de E/S, muchas veces, por razones de rendimiento, el SGBD puede leer o escribir un número entero de páginas de una vez.

c) Las técnicas de transmisión simultánea de varias páginas en una sola operación de E/S implican la necesidad de dedicar una parte de la memoria principal del ordenador a recibir, gestionar y almacenar páginas de las BD. Esta memoria, gestionada por las rutinas adecuadas del SGBD, se denomina **memoria intermedia**[13] y está estructurada como un conjunto de *unidades de memoria intermedia*[14].

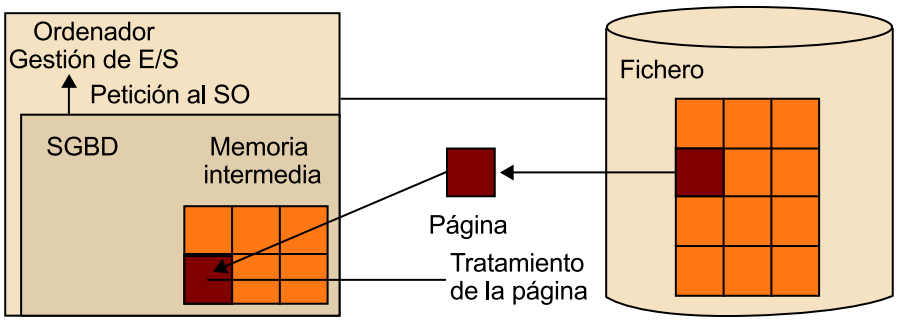
(13)En inglés, *buffer pool.*

(14)En inglés, *buffers.*

Cada unidad de memoria intermedia es un trozo de memoria que actúa como marco para contener una página. El conjunto de todas estas unidades puede ser bastante grande, y dimensionarlas correctamente es uno de los principales factores de refinamiento del rendimiento de una instalación con SGBD.

Finalmente, para ilustrar las funciones propias de el SGBD y las del sistema operativo a la entrada/salida, pasamos a comentar la figura 9.

Figura 9. Funcionamiento del SO y del SGBD a la entrada/salida de una página



Observamos que el SGBD se sirve del SO para ciertas funciones. Cuando el SGBD detecta la necesidad de tratar una página que no tiene en la memoria intermedia, hace una petición a las rutinas de gestión de E/S del sistema operativo para obtenerla. Estas rutinas son las que gestionan la lectura física de la página y la sitúan en una unidad libre del conjunto de las unidades de memoria intermedia. A continuación, el SGBD trata la página, puesto que es quien conoce su organización y sus contenidos.

4. El nivel virtual

Para empezar, justificaremos la existencia del nivel virtual y después veremos la estructura del espacio virtual, que es el componente que materializa las funciones del nivel virtual.

4.1. Justificación de la existencia del nivel virtual

En una primera aproximación podríamos pensar que cada tabla se almacena en un fichero, es decir, que hay una relación biunívoca entre tabla y fichero, con lo cual desaparecería la necesidad de un nivel intermedio que haga corresponder componentes lógicos con componentes físicos, puesto que la correspondencia sería fija.

La realidad, sin embargo, es más compleja que la suposición anterior. A continuación, presentamos algunos casos:

a) Hay tablas muy grandes que nos interesará fragmentar y almacenar cada fragmento en un dispositivo diferente con el fin de mejorar el acceso a los mismos.

b) Por el contrario, podemos encontrar un conjunto de tablas muy pequeñas y que convenga guardarlas todas en un mismo fichero para no consumir tantos recursos del sistema.

c) Cada vez más se usan tablas que, además de campos con datos de tipo tradicional (cantidades numéricas, cadenas de caracteres que representan nombres y direcciones, fechas, etc.), tienen otros que almacenan tipos de datos diferentes, como por ejemplo gráficos, imágenes o algunos minutos de audio o de vídeo. Estos datos, que necesitan muchos más bytes para ser almacenados, son los objetos grandes y que interesa almacenar en ficheros diferentes de los que se emplean para los datos tradicionales, para mejorar los tiempos de acceso.

d) A pesar de que hasta ahora sólo hemos mencionado las tablas, todos sabéis que hay otros componentes, como por ejemplo índices, definiciones de disparadores, etc. Al igual que en el punto anterior, seguramente interesará almacenar cada componente en ficheros estructurados de manera distinta, para obtener el máximo rendimiento de ellos.

Recursos limitados

El interés por no querer consumir más recursos de los imprescindibles reside en el hecho de que hay algunos SGBD que soportan sólo un número determinado de ficheros.

Todos estos ejemplos tienen que advertirnos de que resulta bastante útil disponer de un nivel intermedio. Nos proporciona un grado elevado de flexibilidad (o independencia) para asignar componentes lógicos a los componentes físicos, puesto que no hay que hacerlo de una manera estrictamente biunívoca

(una tabla, un fichero). Concretamente, nos permite decidir dónde se almacenará cada tabla o fragmento. De este modo, elegimos en qué máquina, disco o trozo de disco tendremos los diferentes datos de la base de datos.

4.2. El espacio virtual y sus asociaciones

Hemos visto que la función del nivel virtual es proporcionar un grado elevado de independencia entre los niveles lógico y físico. Denominamos **espacio virtual**(**EV**[15]) al componente que implementa esta funcionalidad.

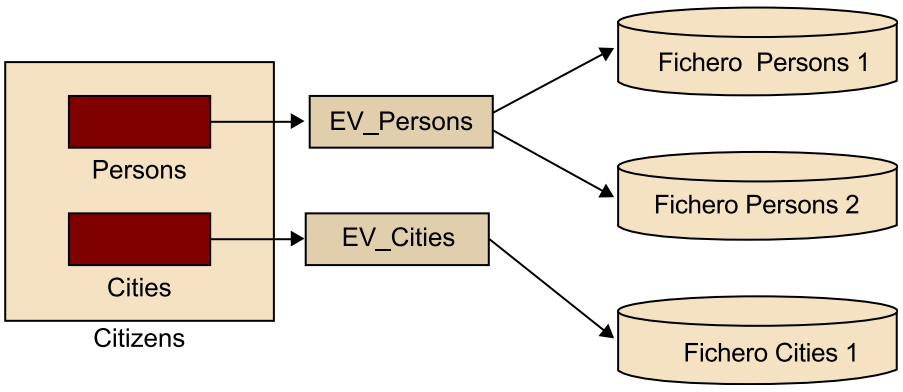
[15] Abreviamos *espacio virtual* con la sigla EV.

Los diferentes SGBD comerciales otorgan distintos nombres a este componente, entre los cuales los más empleados son *dbspace* y *tablespace*. También cabe decir que la funcionalidad que veremos implementada para el espacio virtual es una simplificación de la realidad de muchos SGBD comerciales, los cuales utilizan una estructura algo más compleja en este nivel virtual. Aun así, la idea del nivel virtual es simple y se puede explicar correctamente a partir de un único componente, el espacio virtual, que describimos a continuación.

Normalmente, en todos los SGBD una tabla se asocia a un único EV. Por el contrario, un EV puede estar asociado a una o más tablas. Por otro lado, la asociación entre EV y fichero suele ser de muchos a muchos: un EV se asocia a uno o más ficheros y un fichero se corresponde uno o más EV.

La asociación entre tabla y EV se hace en tiempo de definición de la tabla. Junto con otros atributos, se indica con qué EV la asociamos. El espacio virtual también se debe definir y, al definirlo, se mencionan el fichero o los ficheros con los cuales lo asociamos.

Figura 10. Ejemplo del nivel virtual



Ejemplo del nivel virtual

La figura 10 muestra una base de datos de ciudadanos (*Citizens*). En este ejemplo podemos ver la asociación entre el nivel lógico (con dos tablas: *Persons* y *Cities*) y el nivel físico (con tres ficheros: *Persons1*, *Persons2* y *Cities1*) por medio de dos espacios virtuales (*EV_Persons* y *EV_Cities*).

A continuación, se muestran las sentencias necesarias para la creación de estas asociaciones en el SGBD Oracle:

```
CREATE TABLE Person (idPerson integer, name varchar2(30), ...)
TABLESPACE EV_Persons;

CREATE TABLESPACE EV_Persons DATAFILE '/db/filas/Persons1.dbf' SIZE
100M;
```

La sentencia *CREATE TABLE* permite crear una tabla en un EV determinado, haciendo uso de la cláusula *TABLESPACE*. La segunda sentencia, *CREATE TABLESPACE*, permite crear un EV y asignarle un fichero.

Espacio virtual de agrupación

El espacio virtual de agrupación es un tipo de espacio virtual al que se asocian dos o más tablas y que se corresponde con un único fichero. Este tipo de espacio virtual permite tener los datos de tablas relacionadas en un mismo fichero, así como tener la combinación (operación de *JOIN*) preconstruida en el disco. Este tipo de espacio virtual tiene sentido si el acceso a las tablas se hace siempre conjunto, puesto que penaliza las consultas a las tablas individuales.

4.3. Estructura del espacio virtual

El EV es una visión diferente de las páginas del nivel físico. Al igual que una vista en el modelo relacional es otra manera de ver los datos de una tabla sin que esto represente duplicar físicamente los datos de la tabla, el espacio virtual es una manera distinta de organizar las páginas que hemos visto en el nivel físico, pero sin duplicarlas materialmente.

Las únicas páginas que realmente existen son las del nivel físico, agrupadas en extensiones y almacenadas en ficheros. Ahora bien, estas páginas no deben estar necesariamente ordenadas en el soporte físico. Incluso páginas que contienen elementos que en el nivel lógico son consecutivos (por ejemplo, las filas de una tabla) pueden estar separadas unas de otras en el nivel físico (por ejemplo, ocupando extensiones diferentes); en algunos casos, el resultado de la combinación de varias tablas se puede almacenar en un espacio virtual, si conviene tenerlo precalculado.

Estructura de un espacio virtual

La estructura de un espacio virtual es similar a la de otros mecanismos que ya conocéis, como por ejemplo:

a) La correspondencia que hay en un ordenador entre memoria virtual y memoria real.

b) Las vistas del modelo relacional. Los datos de una vista son los que realmente existen en una o más tablas, pero dispuestos de otro modo, sin que esto represente una duplicación de estos datos.

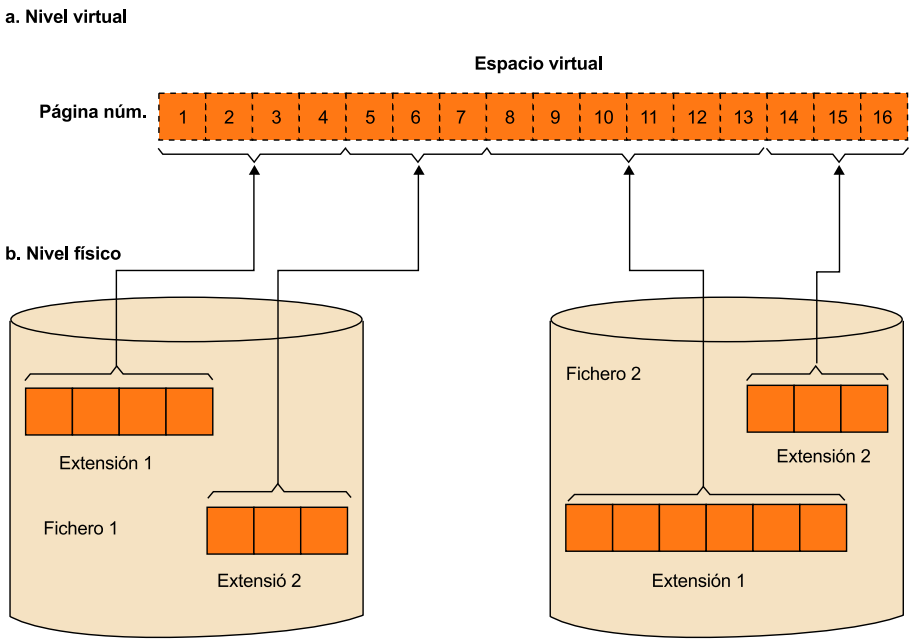
Como veremos enseguida, es conveniente que las páginas estén ordenadas ocupando posiciones consecutivas. Dado que esto físicamente no siempre es posible, los SGBD implementan una ficción, que es lo que denominan *espacio virtual*.

El espacio virtual está constituido por las imágenes de las páginas reales (imágenes que denominaremos *páginas virtuales*) ordenadas secuencialmente y situadas de manera contingua (sin saltos en medio).

Así pues, podemos definir el **espacio virtual** como una secuencia de páginas virtuales que se corresponden una por una con páginas reales del nivel físico.

Podemos ilustrar la estructura del espacio virtual mediante la figura 11.

Figura 11. Correspondencia entre páginas reales y páginas virtuales

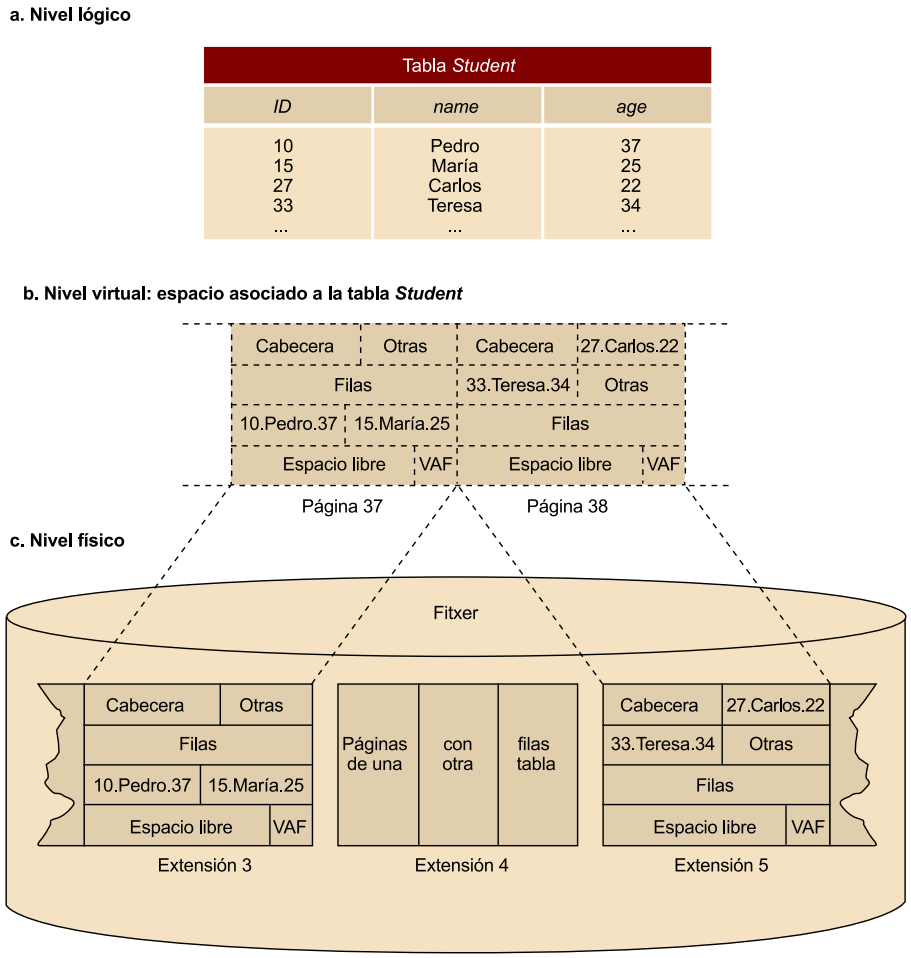


Observad que las distintas páginas que contienen, por ejemplo, las filas de una tabla, están distribuidas entre extensiones diferentes de ficheros distintos. El EV es una ficción para ver y tratar todas estas páginas como si estuvieran ordenadas de manera consecutiva.

En la figura 12 representamos con más detalle y mediante un ejemplo la relación que hay entre los tres niveles de la arquitectura que hemos visto hasta ahora.

En el nivel lógico, tenemos la tabla *Student* con sus filas. Esta tabla se almacena en páginas de datos en el nivel físico. Por la manera como se ha ido creando la tabla y por la disponibilidad de espacio libre en el disco, puede haber sucedido que no todas las filas de la tabla hayan quedado en páginas consecutivas. Observamos que las páginas que contienen estas filas pertenecen a dos extensiones diferentes, separadas por otra extensión que puede ser de otro fichero y, evidentemente, puede contener otros datos.

Figura 12. Un ejemplo de los tres niveles



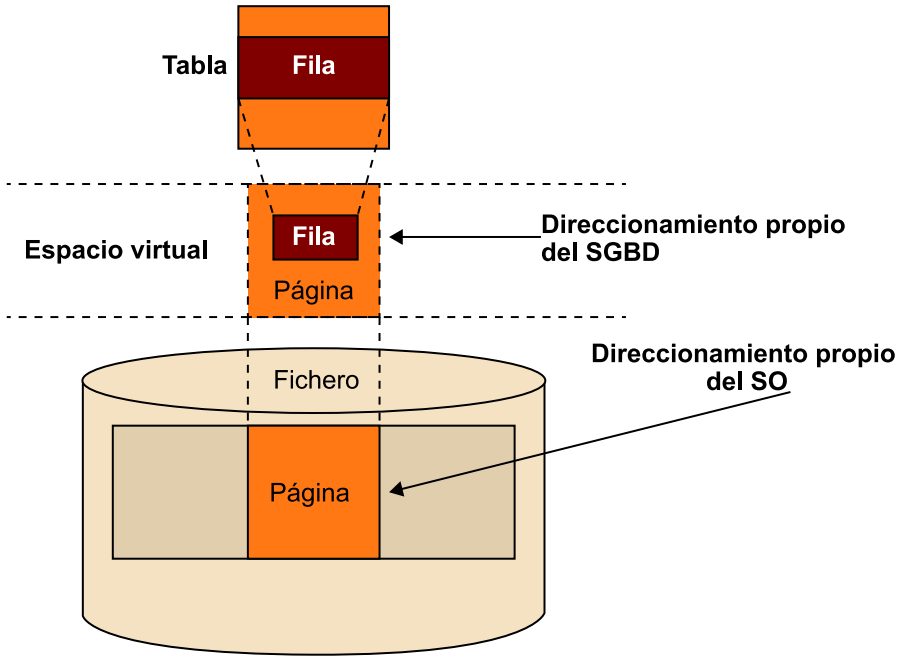
No podemos asegurar la contigüidad de las filas del nivel lógico en las páginas del nivel físico. Entonces, el EV nos permite ver todas las páginas que contienen las filas de esta tabla de manera consecutiva y sin saltos.

4.3.1. Direccionamiento en un SGBD

Ahora estamos en condiciones de analizar cómo es el direccionamiento en un SGBD.

En el nivel lógico, el usuario manipula tablas, filas y campos. Estos componentes están almacenados en ficheros. ¿Cómo encuentra el SGBD uno de estos componentes en el nivel físico? ¿Cómo lo direcciona? Responderemos a esta pregunta en dos pasos, puesto que en realidad hay un direccionamiento doble, que se corresponde con la arquitectura que hemos estado viendo y que se ilustra en la figura 13.

Figura 13. Un direccionamiento doble



1) Primer paso: del nivel lógico al virtual

Puesto que la unidad más pequeña que dirige de manera explícita un SGBD es la fila, lo que hace realmente el SGBD es encontrar las filas dentro del espacio virtual. El direccionamiento del SGBD es, pues, del nivel lógico al nivel virtual.

Tratamiento de los campos

Una vez la fila está en la memoria, el SGBD, dado que conoce su estructura, puede localizar todos sus campos y tratarlos individualmente.

Para localizar las filas en el EV, los SGBD utilizan una dirección que denominaremos **identificador de fila** (**RID**). El RID, como podemos ver en la figura 14, tiene dos partes diferenciadas:

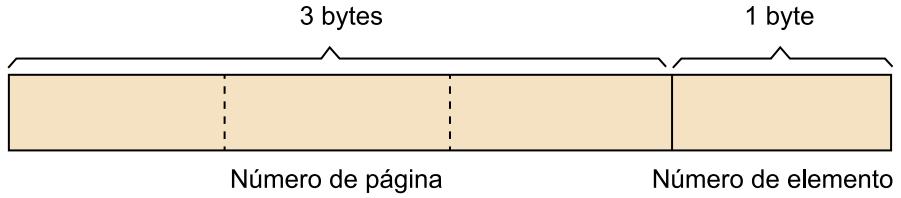
El identificador de fila (RID)

RID es un acrónimo del término inglés *row identifier*. Este identificador recibe diferentes nombres según las implementaciones de cada SGBD. Por ejemplo, en el caso de Oracle recibe el nombre de *ROWID*.

- el número de la página que contiene la fila dentro del espacio virtual asociado a la tabla a la cual pertenece la fila,
- el número de elemento del VDF de esta página que apunta a la fila.

Tal como indicamos en la figura 14, el RID normalmente es una dirección de 4 bytes, 3 para el número de página y 1 para el número de elemento del VDF.

Figura 14. Estructura del RID



Este sistema de direccionamiento merece algunos comentarios:

a) Una fila, una vez dada de alta en una página, no puede cambiar de página mientras esté viva, puesto que su dirección también cambiaría. Dado que esta dirección puede estar guardada en más de un lugar de la base de datos, interesa

no hacer este cambio para evitar el mantenimiento inútil y costoso de otras estructuras de datos (por ejemplo, en los índices que se hayan construido sobre esta tabla).

b) En cambio, una fila se puede desplazar dentro de su página original. Cambiando solo el contenido del elemento de VDF correspondiente (que indica el desplazamiento dentro de la página donde empieza la fila), tendremos siempre la fila correctamente direccionada, y esto sin ningún cambio en su RID. Por esto, es posible una gestión del espacio libre de una página sin cambios en los RID de las filas de esta página. La gestión de espacio libre, como ya habéis visto, tiende a agrupar todo el espacio libre de una página y, para ello debe desplazar filas dentro de la página.

c) Por el mismo motivo que mencionábamos en el punto **a**, si una fila aumenta su longitud de manera que no cabe en su página original y se tiene que colocar en otra página, ya hemos visto que en la página original se mantiene una dirección, que es un RID, el cual desde la página original apunta a la nueva página. Así pues, los RID almacenados en otras estructuras, como los índices que apuntaban a la fila, la continúan encontrando sin que se hayan tenido que modificar.

Ahora ya habréis podido comprender que las direcciones o apuntadores que hemos mencionado en puntos anteriores de este módulo son en realidad RID. Es el caso de las filas demasiado largas que no cabían en una página (que se han partido en trozos y cada trozo apunta al siguiente) o bien el de las filas que crecen de longitud (que se tienen que trasladar a otra página y se les apunta desde la página original).

2) Segundo paso: del nivel virtual al físico

En este paso, normalmente intervienen los SO y no lo veremos en detalle. Destacaremos solamente que la unidad de localización en el nivel físico es la página.

De este modo, el SGBD pide al SO que le lea/guarde una página concreta y este lo hace empleando una dirección muy diferente del RID. Por lo tanto, en este paso hay una transformación de direcciones: del RID a una dirección física que depende de la arquitectura del dispositivo.

Antes de acabar el tema del direccionamiento, podemos reflexionar sobre la utilidad del direccionamiento propio de los SGBD, el RID.

Al prescindir de muchos de los detalles del nivel físico, un direccionamiento como el RID permite que el SGBD pueda emplear más de un tipo diferente de dispositivos de almacenamiento, puesto que las diferencias son gestionadas

en otro nivel y por otro componente (normalmente, el sistema operativo). Este mecanismo simplifica los SGBD y aumenta su potencia, puesto que les proporciona la independencia de los dispositivos físicos concretos.

Y finalmente, no debemos olvidar que el nivel virtual es el que posibilita, por medio del EV, el direccionamiento de tipo RID.

5. Transformación del modelo lógico en el modelo físico

A partir del diseño lógico de una base de datos debemos llegar a su diseño físico, pasando por el nivel virtual. La forma de implementar un diseño lógico en un SGBD concreto depende de las características propias de cada uno de los SGBD. Y las características propias de cada SGBD son un reflejo del entorno:

- Características del hardware.
- Sistema operativo y software básico.
- Diseño del SGBD.

Cada SGBD ha desarrollado un lenguaje propio, hecho a medida por el propio constructor, para implementar el diseño físico de la base de datos, de acuerdo con las características del entorno y para obtener el máximo rendimiento del hardware, del sistema operativo y del propio gestor. De hecho, se podría considerar como una ampliación del lenguaje SQL estándar, con las cláusulas propias que cada gestor necesita para definir los componentes del diseño físico. Sin embargo, hay una gran similitud o equivalencia entre buena parte de los componentes de los diferentes gestores.

El estándar SQL incorpora la definición de todos los componentes del diseño lógico de la base de datos. En cambio, no incorpora ningún elemento del diseño físico.

En la transformación del modelo lógico en el modelo físico, seguiremos los pasos siguientes:

1) Transformar las tablas con las correspondientes claves primarias, claves foráneas y claves alternativas, a partir del diseño lógico de la base de datos que hemos obtenido en el paso anterior.

2) A continuación relacionamos cada tabla con un espacio para tablas, y cada índice con un espacio para índice. Es el nivel virtual.

3) Para terminar, relacionamos cada espacio virtual con un fichero físico y definimos sus características. Esto constituye el diseño físico de la base de datos.

De manera adicional es preciso completar esta definición con los índices necesarios para garantizar un acceso correcto a los datos, además de las restricciones necesarias sobre los datos (valores nulos, valores únicos de los campos, etc.).

A continuación, veremos el proceso de transformación. Como ejemplo de SGBD para la parte que no queda incluida en el estándar SQL, se utiliza el sistema gestor Oracle. Aun así, gran parte de la sintaxis es similar a la que se utiliza en otros SGBD y basta consultar un manual de referencia para ver cómo se aplican las diferentes instrucciones en cada SGBD.

5.1. Tabla

La tabla es un componente del diseño lógico de la base de datos y, como tal, está definida en el estándar SQL. Ahora bien, es importante remarcar que la parte que afecta al nivel virtual y físico no está incluida en el estándar y, por lo tanto, cada SGBD implementa estas funciones de manera diferente.

La sentencia *CREATE TABLE* de todos los SGBD incorpora las cláusulas del estándar SQL y, además, el enlace con los elementos físicos propios de cada uno.

```
CREATE TABLE table_name
  ( column_definition )
  <unique_constraint>
  <referential_constraint>
  <check_constraint>
  <extent_specs>
TABLESPACE table_space_name
```

Las cláusulas siguientes de la sentencia *CREATE TABLE* pertenecen al SQL estándar: *column_definition*, *unique_constraint*, *referential_constraint*, *check_constraint*. Estas cláusulas son definiciones del diseño lógico. Todos los SGBD las incorporan con una sintaxis casi idéntica.

Por otro lado, las cláusulas *<extent_specs>* y *TABLESPACE* son específicas de Oracle. Sirven para relacionar la definición de la tabla (nivel lógico) con el espacio para tablas (nivel virtual), que se denomina *Tablespace* en Oracle. Además, la cláusula *<extent_specs>* puede incorporar otros parámetros que permiten controlar el porcentaje de ocupación de las páginas del espacio para tablas.

Ejemplo de creación de una tabla en Oracle

A continuación, veremos un ejemplo de creación de una tabla en el SGBD Oracle:

```
CREATE TABLE Employees (
  employeeId NUMBER(6) PRIMARY KEY,
  firstName VARCHAR2(20),
  lastName VARCHAR2(25),
  email VARCHAR2(25) NOT NULL,
```

```
    phoneNumber VARCHAR2(20),
    salary NUMBER(8,2) NOT NULL,
    commissionPct NUMBER(2,2),
    departmentId NUMBER(4),
    CONSTRAINT empSalaryMinDemo CHECK (salary > 0),
    CONSTRAINT empEmailUKDemo UNIQUE (email)
) TABLESPACE users_data;
```

En el ejemplo, se crea la tabla *Employees* con diferentes atributos y algunas restricciones. Algunas consideraciones sobre el ejemplo:

- La clave primaria de la tabla es el atributo *employeeId*.
- Dos de los atributos no admiten el valor nulo (*email* y *salary*).
- Las restricciones que encontramos son:
  - *CHECK*: diferentes validaciones, como por ejemplo que el campo sea superior a cero o que cumpla determinadas condiciones.
  - *UNIQUE*: unicidad del campo indicado, es decir, que no puede haber dos registros en esta tabla con el mismo valor en este campo.

A continuación, veremos con más detalle las definiciones relativas a clave primaria y clave alternativa, índice y algunas de las restricciones más relevantes.

5.1.1. Clave primaria y clave alternativa

Las definiciones de claves primarias, foráneas y alternativas actualmente forman parte del estándar SQL. No obstante, no siempre ha sido así; en el año 1986 todavía no se habían definido y el estándar SQL86 no las incluyó cuando se publicó. En el estándar SQL89 se mencionan las claves primarias por primera vez, y no fue hasta 1992 cuando de hecho se definieron y se incorporaron al estándar SQL92.

Como ya sabemos, la clave primaria tiene que ser obligatoriamente una clave única y que no admita valores nulos. La teoría del modelo de bases de datos relacionales sugiere que cada tabla debería tener una clave primaria que identificara de manera unívoca la entidad que describe. A pesar de esto, el estándar SQL no obliga a la existencia de una clave primaria en cada tabla, sino que es opcional. Lógicamente, sin embargo, cada tabla puede tener como máximo una clave primaria.

El resto de las claves únicas y que no admiten valores nulos son claves alternativas a la clave primaria. Esto tampoco está legislado en el estándar SQL.

Tal como muestra el ejemplo anterior, se indica la clave primaria mediante la instrucción *PRIMARY KEY*.

5.1.2. Índice

Los índices son unos elementos del diseño físico de la base de datos que tienen como finalidad mejorar el rendimiento de las aplicaciones cuando acceden a las tablas. Los índices no forman parte del diseño lógico de la base de datos y, por lo tanto, no están incluidos en el estándar SQL.

Aun así, la sentencia *CREATE INDEX* está presente en todos los SGBD con opciones muy similares. A continuación, veremos el detalle de esta sentencia sobre el SGBD Oracle. Además, incorpora cláusulas para definir el espacio para índices y cláusulas de enlace con los elementos físicos propios (los ficheros de índices).

```
CREATE [UNIQUE] INDEX index_name
   ON table_name ( column [ ASC | DESC ] [ ,..n ])
  [ CLUSTER cluster_name ]
  [ < extent_specs > ]
  [ TABLESPACE table_space_name ]
  [ ..... ]

< extent_specs > ::=
  [ PCTFREE nn ]
  [ PCTUSED nn ]
  [ INITRANSnn ]
  [ MAXTRANSnn ]
  [ STORAGE < storage_clause > ]

< storage_clause > ::=
  INITIAL nn
  [ NEXT nn ]
  [ MAXEXTENTS nn ]
  [ PCTINCREASE nn ]
  [ OPTIMAL nn ]
  [ ..... ]
```

Donde:

- *index_name* es el nombre lógico del índice definido sobre la tabla especificada en la cláusula *ON*.
- *UNIQUE* y *CLUSTER* son características del índice.
- *table_space_name* es el nombre del espacio para índice y se define con la sentencia *CREATE TABLESPACE*, que hemos visto anteriormente.
- Cuando se crea el espacio para índice (*CREATE TABLESPACE*), se asocia a un fichero físico con características propias de nombre, tamaño, ubicación física en disco, etc.
- La cláusula *<extent_specs>* especifica condiciones de porcentaje de ocupación de las páginas del espacio para índice.

- La cláusula <*storage_clause*> define características de almacenamiento, tamaño inicial, tamaño incremental, extensiones mínimas y máximas, etc.

Ejemplo de creación de un índice

El ejemplo siguiente muestra cómo se crea un índice mediante la instrucción *CREATE INDEX* de Oracle:

```
CREATE INDEX indexDepName
ON Employees (departmentId) TABLESPACE users_ind;
```

5.1.3. Restricciones

A continuación, veremos algunas de las principales restricciones que podemos definir sobre los atributos de las tablas y cómo se implementan en el SGBD de Oracle.

Estas restricciones nos permiten definir una lógica que nos ayude a validar los datos de nuestro modelo, de manera que cumplan determinadas condiciones.

Algunos ejemplos típicos que podemos resolver mediante estas restricciones son:

- Verificar que un número sea superior a cero.
- Verificar que un atributo contenga una cadena de texto de un tamaño determinado.
- Verificar que no haya valores duplicados para un cierto atributo.

a) Check

Esta restricción permite indicar ciertas condiciones que el valor del registro debe cumplir para ser admitido en la tabla.

Por ejemplo, podemos definir una tabla con información sobre trabajadores, que contenga como atributos el nombre y el sueldo. Evidentemente, el sueldo debe ser positivo. Para indicarlo, en la creación de la tabla utilizamos la restricción *CHECK*.

```
CREATE TABLE Employees (
 name VARCHAR2 (30),
 salary NUMBER CHECK (salary> 0)
) TABLESPACE data_employees;
```

b) Not null

Esta restricción permite definir atributos que deben contener información obligatoriamente, es decir, que deben contener datos. Indicamos esta restricción en el SGBD por medio del parámetro *NOT NULL*.

```
CREATE TABLE Employees (
name VARCHAR2 (30) NOT NULL,
salary NUMBER
) TABLESPACE data_employees;
```

c) Unique

Esta restricción posibilita validar que no existen valores duplicados en un determinado atributo. Permite definir las claves candidatas.

Continuando con el ejemplo anterior, esta restricción nos permite definir que cada trabajador debe tener un identificador único (por ejemplo, el DNI en el caso de España), el cual debe ser único para cada persona.

```
CREATE TABLE Employees (
 id VARCHAR2 (9) UNIQUE,
 name VARCHAR2 (30) NOT NULL,
  salary NUMBER
) TABLESPACE data_employees;
```

d) Foreign key / References

Esta restricción permite crear claves foráneas que referencian atributos de otras tablas.

Por ejemplo, si queremos crear una tabla que contenga los departamentos de la empresa (*Department*) e indicar a qué departamento pertenece cada trabajador (*Employees*), será preciso crear la nueva tabla y añadir un atributo que referencie la nueva tabla para cada registro de la tabla *Employees*.

```
CREATE TABLE Department
( id NUMBER PRIMARY KEY
  name VARCHAR2 (50)
) TABLESPACE data_departments;

CREATE TABLE Employees
( id VARCHAR2 (9) UNIQUE,
  name VARCHAR2 (30) NOT NULL,
  salary NUMBER,
  departId NUMBER,
  CONSTRAINT fkDep FOREIGN KEY (departId) REFERENCES Department (id)
) TABLESPACE data_employees;
```

5.2. Espacio para tablas

El espacio para tablas es un componente del nivel virtual. No pertenece al diseño lógico de la base de datos y, por lo tanto, no está incluido en el estándar SQL. La sentencia de creación de un espacio para tablas es diferente en cada SGBD e incorpora las cláusulas de enlace con los elementos físicos propios de cada uno de estos SGBD (los ficheros).

El nivel virtual recibe en Oracle el nombre de *tablespace*. La sentencia *CREATE TABLESPACE* permite crear espacios virtuales en este SGBD.

```
CREATE TABLESPACE table_space_name
   DATAFILE < filespec > [ ,...n ]
   DEFAULT STORAGE < storage_clause >

 < filespec > ::=
   'file_name'
   [ SIZE nn ]
 < storage_clause > ::=
   INITIAL nn
   [ NEXT nn ]
   [ MINEXTENTS nn ]
   [ MAXEXTENTS nn ]
   [ PCTINCREASE nn ]
   [ OPTIMAL nn ]
   [ ..... ]
```

Donde:

- *table_space_name* es el nombre del espacio para tablas, que se define con esta sentencia y que está relacionada con la cláusula *TABLESPACE* de la sentencia *CREATE TABLE*.

- Cada espacio para tablas se asocia a uno o más ficheros físicos <*filespec*>.

- La definición del fichero físico viene dada por su nombre externo *file_name*, su ubicación en disco y su tamaño, *size*.

- La cláusula <*storage_clause*> define sus características de almacenamiento, tamaño inicial, tamaño incremental, extensiones mínimas y máximas, etc.

Ejemplo de creación de un espacio para tablas

En el ejemplo siguiente crearemos un espacio para tablas con el nombre *users_data* formado por el fichero '/db/users/users_data1.dbf' y con un tamaño total de 100 MB.

```
CREATE TABLESPACE users_data
DATAFILE '/db/users/users_data1.dbf' SIZE 100M;
```

Algunos parámetros son opcionales y no es necesario que sean informados en la sentencia de creación. En estos casos, se asigna el valor por defecto del parámetro. En el ejemplo anterior el tamaño de la página no se especifica y, por lo tanto, se utilizará el valor por defecto.

5.3. Base de datos

Como en el caso del espacio para tablas, la base de datos no pertenece al diseño lógico de la base de datos y, por lo tanto, tampoco está incluida en el estándar SQL.

La sentencia *CREATE DATABASE* es diferente en cada SGBD e incorpora cláusulas de definición de elementos físicos propios de cada uno (diario, catálogo, ficheros, etc.).

En Oracle, la sintaxis de esta sentencia es como se indica a continuación:

```
CREATE DATABASE database_name
   [ CONTROLFILE REUSE ]
   [ LOGFILE < filespec > [ ,...n ] ]
   [ MAXLOGFILES nn ]
   [ MAXLOGMEMBERS nn ]
   [ MAXLOGHISTORY nn ]
   [ DATAFILE < filespec > [ ,...n ] ]
   [ MAXDATAFILES nn ]
   [ MAXINSTANCES nn ]
   [ CHARACTER SET charset ]
   [ ..... ]
```

Donde:

- *database_name* es el nombre de la base de datos que se asocia a un conjunto de ficheros físicos que contienen los espacios para tablas explicados en el subapartado anterior.

- Los ficheros físicos que contienen los espacios para tablas se relacionan en la cláusula *DATAFILE < filespec >*.

- La cláusula *LOGFILE <filespec>* especifica el nombre de los diarios que se definen para que el gestor registre todas las actualizaciones de las tablas de esta base de datos y posibilitar, así, su recuperación en caso necesario.

- El detalle de la definición de los ficheros físicos *<filespec>* se ha explicado en el subapartado de los espacios para tablas.

- Otros parámetros limitan el número máximo de ficheros de cada tipo: *MAXLOGFILES*, *MAXLOGMEMBERS*, *MAXDATAFILES*, etc.

Ejemplo de creación de bases de datos

En el siguiente ejemplo crearemos una base de datos para una universidad, con unos diarios (*logfiles*) de 10 MB cada uno y con unos límites de 5 *logfiles* y 100 *datafiles*.

```
CREATE DATABASE university
   USER SYS IDENTIFIED BY pass1
   USER SYSTEM IDENTIFIED BY pass2
   LOGFILE GROUP 1 ('/db/oracle/oradata/university/redo01.log')
        SIZE 10M,
      GROUP 2 ('/db/oracle/oradata/university/redo02.log') SIZE 10M,
      GROUP 3 ('/db/oracle/oradata/university/redo03.log') SIZE 10M
   MAXLOGFILES 5
   MAXDATAFILES 100;
```

6. Implementación de métodos de acceso

En una base de datos hay datos almacenados a los que se debe acceder para hacer consultas y actualizaciones. Por este motivo, una de las funciones de los SGBD es la de proporcionar el acceso a los datos que gestionan. La problemática de cómo ofrecer este acceso es el objeto de estudio de este apartado.

Inicialmente, veremos de manera breve los diferentes métodos de acceso a los datos. A continuación, profundizaremos en el estudio de los métodos de acceso por posición, por valor y por varios valores. Finalmente, veremos cómo implementa estas funciones el SGBD Oracle.

6.1. Los métodos de acceso a una BD

Una de las funciones de los SGBD es proporcionar el acceso a los datos que gestionan. El acceso a los datos debe permitir consultar y actualizar los datos almacenados. Siempre que se lee o se graba algún dato en una BD se hace mediante alguno de los métodos de acceso de los que dispone el SGBD.

Reflexión

Debemos entender "actualizaciones de datos" en sentido amplio; es decir, su inserción, eliminación y modificación.

6.1.1. Los datos

En este subapartado, para presentar los métodos de acceso a los datos de manera comprensible, supondremos siempre que los datos a los que hay que acceder se perciben según la visión que proporciona el nivel virtual, y no según la del nivel físico. Así pues, casi siempre que en este módulo se emplea la palabra *página* tenemos que interpretar que nos referimos a páginas virtuales; si nos referimos a páginas reales, lo mencionaremos de manera explícita.

Con objeto de simplificar, también supondremos que todos los accesos se hacen a datos almacenados en una sola tabla, situada en un único EV. Así pues, no consideramos los casos en que es necesario combinar datos de tablas (y espacios) diferentes para obtener los datos a los que queremos acceder.

6.1.2. Los accesos por posición

En este subapartado, veremos los tipos de acceso más simples que utilizan los SGBD.

El acceso **directo por posición** consiste en obtener una página que dispone de un número de página determinado dentro de un espacio.

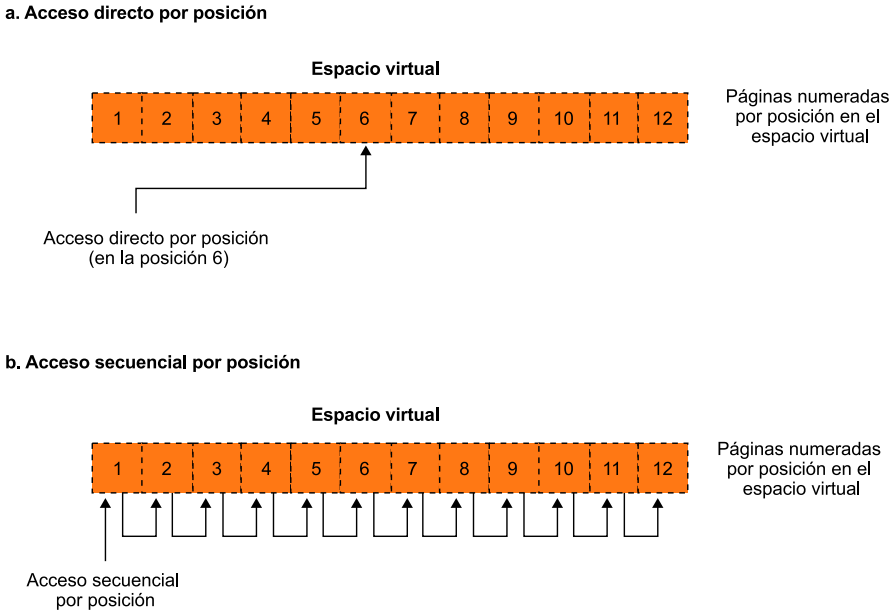
El acceso **secuencial por posición** consiste en obtener las páginas de un espacio siguiendo el orden establecido por sus números de página.

Accesos por posición

Los accesos por posición son viejos conocidos. El acceso secuencial y el acceso directo por posición se utilizan también en la tecnología de los ficheros. En concreto, el fichero secuencial ofrece el acceso secuencial por posición y el fichero relativo ofrece tanto el acceso directo por posición como el acceso secuencial por posición.

La figura 15 muestra estos dos tipos de acceso por posición.

Figura 15. Los accesos por posición



Estos tipos de acceso simples son suficientes para casos en los que solo haya que efectuar consultas y actualizaciones muy sencillas en la BD.

Ejemplo de acceso directo por posición

Supongamos que disponemos de una BD que contiene la tabla *Employees*(*emplId, emplName, officeNum, salary*). Supongamos también que esta tabla se almacena en un espacio determinado y que las filas de los empleados se sitúan en el espacio mencionado por orden de inserción. Consideremos ahora la actualización siguiente, descrita en SQL:

```
INSERT INTO Employees
VALUES (25, 'Juan Tarrago', 150, 2000);
```

Esta actualización se puede efectuar fácilmente. El SGBD tiene grabado el número de la primera página del espacio con capacidad de absorber nuevas filas. Entonces, se usa el acceso directo por posición para insertar el empleado en la página mencionada (añadiendo la fila correspondiente al empleado nuevo).

Ejemplo de acceso secuencial por posición

En la base de datos anterior podríamos realizar una consulta para recuperar los datos de todos los empleados, como por ejemplo la siguiente:

```
SELECT * FROM Employees;
```

Para resolver esta consulta, el SGBD puede utilizar el acceso secuencial por posición, mediante el cual el SGBD va obteniendo todas las páginas que contienen las filas de los empleados. De este modo, puede proporcionar los empleados en el mismo orden en que los obtiene. Observad que la consulta no requiere ninguna ordenación particular de los

empleados y, entonces, el SGBD los puede proporcionar en el mismo orden en el que están almacenados.

6.1.3. Los accesos por valor

En este subapartado, explicamos algunos tipos de acceso más complejos que los accesos por posición que acabamos de ver.

El **acceso directo por valor** consiste en obtener todas las filas que contienen un valor determinado por un atributo.

El **acceso secuencial por valor** consiste en obtener varias filas por el orden de los valores de un atributo.

Ejemplos de acceso directo por valor

Consideremos las sentencias de manipulación siguientes:

- Sentencia 1:

```
SELECT * FROM Employees WHERE officeNum = 150;
```

- Sentencia 2:

```
UPDATE Employees SET salary = 2500 WHERE officeNum = 200;
```

- Sentencia 3:

```
DELETE FROM Employees WHERE officeNum = 150;
```

Observad que en los tres ejemplos hay que buscar las filas de los empleados que tienen un número de despacho (*officeNum*) determinado, para mostrarlos, modificarlos o borrarlos, respectivamente. Es decir, en los tres casos hay que hacer búsquedas de un valor por un atributo determinado.

De estos ejemplos, se deduce que el acceso directo por valor es necesario para ejecutar consultas y actualizaciones sobre filas que contienen un determinado valor de un atributo.

Ejemplos de acceso secuencial por valor

Consideremos las sentencias de manipulación siguientes:

- Sentencia 1:

```
SELECT * FROM Employees ORDER BY officeNum;
```

- Sentencia 2:

```
SELECT * FROM Employees
WHERE officeNum >= 100 AND officeNum <= 300;
```

- Sentencia 3:

```
UPDATE Employees SET salary = 2500
WHERE officeNum >= 100 AND officeNum <= 300;
```

- Sentencia 4:

```
DELETE FROM Employees
WHERE officeNum >= 100 AND officeNum <= 300;
```

En todos estos ejemplos, es necesario conseguir algunas filas de la tabla de empleados en una secuencia ordenada por el atributo *officeNum*.

En la primera de las sentencias hay que obtener los empleados ordenados por el número de despacho, debido a la cláusula *ORDER BY officeNum*. Para ejecutar las tres sentencias restantes también debe disponer del acceso secuencial por valor, aunque la causa es quizá menos evidente. La cláusula *WHERE officeNum >= 100 AND officeNum <= 300* implica que sea necesario localizar todos los empleados que tienen un número de despacho entre el 100 y el 300 para consultar los datos, modificarlos o borrarlos, respectivamente. Una manera de localizar todos los empleados es disponer de algún mecanismo de acceso a cada uno de ellos por orden de número de despacho a partir del valor 100 y hasta el 300. Este mecanismo es el acceso secuencial por valor.

De los ejemplos anteriores, se desprende que el acceso secuencial por valor se utiliza para ejecutar consultas en las que el resultado debe estar ordenado por un atributo y para consultas o actualizaciones que afecten a un conjunto de filas que contienen el valor de un atributo que se encuentra dentro de un rango determinado de valores.

6.1.4. Los accesos por varios valores

Hemos considerado accesos por valor de un solo atributo. Nos falta considerar los casos en los que se quiere acceder según los valores de varios atributos: los accesos por varios valores. Los accesos por varios valores pueden ser, como los accesos por un solo valor, directos o secuenciales.

Los accesos por varios valores

Los accesos por varios valores no tienen su correspondiente en tecnología de los ficheros, a diferencia de lo que sucedía con los otros accesos que hemos explicado.

Ejemplos de acceso por varios valores

Consideremos los ejemplos siguientes de acceso por varios valores:

- Sentencia 1:

```
SELECT *
FROM Employees
WHERE officeNum = 150 AND salary = 2000;
```

- Sentencia 2:

```
SELECT *
FROM Employees
ORDER BY officeNum, salary;
```

- Sentencia 3:

```
DELETE *
FROM Employees
WHERE officeNum >= 100 AND salary >= 1800;
```

En estos ejemplos, se accede a los datos según los valores de dos atributos: el atributo *officeNum* y el atributo *salary*. La primera sentencia corresponde a un acceso directo por varios valores y las dos siguientes, a un acceso secuencial por varios valores.

También puede ocurrir, sin embargo, que los accesos por varios valores sean mixtos.

Los **accesos mixtos** por varios valores son accesos en los que se combina el acceso directo por valores de algunos atributos y el acceso secuencial por valores de otros atributos.

Ejemplos de accesos mixtos por varios valores

Consideremos las sentencias siguientes:

- Sentencia 1:

```
SELECT *
FROM Employees
WHERE officeNum = 150 AND salary >= 2000;
```

- Sentencia 2:

```
SELECT *
FROM Employees
WHERE salary = 2000
ORDER BY officeNum;
```

- Sentencia 3:

```
DELETE *
FROM Employees
WHERE officeNum >= 100 AND officeNum <= 300 AND salary = 2000;
```

La primera de las sentencias combina un acceso directo por valor del atributo *officeNum* con un acceso secuencial por valor del atributo *salary*. Las dos siguientes combinan un acceso secuencial por valor del atributo *officeNum* con un acceso directo por valor del atributo *salary*.

6.2. Implementación de los accesos por posición

Para la implementación de los accesos por posición, los SGBD se basan casi completamente en el SO. Las rutinas de gestión de E/S del SO permiten obtener una página real si tenemos su dirección y también permiten grabar en el disco una página real concreta. Podríamos decir que el SO implementa el acceso por posición en páginas reales.

De acuerdo con el punto de partida que acabamos de presentar, la implementación de los accesos por posición es muy simple.

En el caso del **acceso directo por posición**, sólo es preciso que el SGBD transforme los números de posición de las páginas virtuales en direcciones de páginas reales almacenadas al disco.

Si el SGBD necesita realizar un **acceso secuencial por posición**, el mismo procedimiento que hemos explicado para el acceso directo por posición sirve para ir accediendo a todas las páginas, de la primera a la última.

6.3. Implementación de los accesos por valor

La implementación de los accesos por valor es más compleja que la de los accesos por posición. La dificultad proviene del hecho de que si el SGBD se basara únicamente en el SO para implementarlos, no siempre obtendría un buen rendimiento. Será necesario, por lo tanto, que el SGBD disponga de mecanismos propios especializados que los implementen de manera eficiente.

En este apartado veremos que para implementar de una manera eficiente el acceso directo y el acceso secuencial por valor, los SGBD usan unas estructuras de datos auxiliares que se denominan *índices*. Para la implementación de los accesos por valor con índices, partiremos del hecho de que ya disponemos de los accesos por posición que hemos explicado. Los SGBD utilizan los accesos por posición para implementar los accesos por valor.

6.3.1. Necesidad de los índices

Empezaremos explicando por qué los índices son necesarios si se desea obtener un buen rendimiento al realizar accesos por valor. Para ello, presentaremos algunas implementaciones que no los utilizan y veremos los problemas que comportan.

Consideremos la sentencia siguiente, que requiere un acceso directo por valor del atributo *officeNum*:

```
SELECT * FROM Employees WHERE officeNum = 150;
```

Hay que tener en cuenta que las filas de la tabla *EMPLEADOS* se encuentran en un espacio virtual. Por lo tanto, ya que disponemos del acceso secuencial por posición a las páginas del espacio virtual, el SGBD puede emplear este acceso para obtener todas las páginas una a una, comprobando para cada una, qué filas contienen el valor buscado.

El inconveniente de la solución anterior es que requerirá un gran número de E/S, sobre todo si hay muchos empleados en la BD. Habrá que consultar todas las páginas de la BD que tienen filas de empleados para conseguir únicamente los empleados del despacho 150, que pueden ser muy pocos.

Ahora analizamos otro ejemplo que requiere un acceso secuencial por valor; podéis observar la sentencia siguiente:

```
SELECT * FROM Employees ORDER BY officeNum;
```

Para obtener los empleados por orden de número de despacho, convendría que estuvieran almacenados según este mismo criterio. Entonces, utilizando el acceso secuencial por posición que nos facilita el SO, se conseguirían los empleados en el orden deseado.

El problema de esta solución es que la ordenación de los empleados por número de despacho iría muy bien para la consulta anterior, pero muy mal si también se quisiera ejecutar, por ejemplo, la sentencia siguiente:

```
SELECT * FROM Employees ORDER BY emplId;
```

Puesto que en una BD tienen que convivir habitualmente consultas y actualizaciones que requieren ordenaciones diferentes de unos mismos datos, la solución que hemos presentado no es satisfactoria para todas las sentencias que será necesario ejecutar de manera eficiente.

Las soluciones que hemos analizado para implementar los accesos por valor no nos permiten conseguir un rendimiento lo suficientemente aceptable en muchos casos. A continuación, analizaremos la manera como los índices nos permitirán mejorar esta situación.

6.3.2. Características generales de los índices

Existen índices de muchos tipos diferentes, pero todos tienen algunas características generales comunes como implementaciones adecuadas de los accesos por valor. Los índices de los SGBD tienen una utilidad parecida a la de los índices que contienen los libros para localizar rápidamente un apartado determinado.

El índice de un libro

Este módulo tiene un índice al principio. Si queremos localizar con rapidez el subapartado dedicado a los árboles B^+, lo que debemos hacer es consultar el índice. Allí encontraremos fácilmente el número de página donde está situado el subapartado de los árboles B^+, porque el índice no es muy largo y no tardaremos en leerlo. Una vez conocido el número de página, solo debemos localizar la página que lleva aquel número.

Los **índices** que emplean los SGBD son unas estructuras de datos auxiliares que, como los índices de los libros, facilitan las búsquedas sobre unos determinados datos.

Uno de los motivos por los cuales las búsquedas pueden ser más rápidas si se realizan mediante un índice en lugar de acceder directamente a los datos, es que los índices suelen ocupar menos espacio que los datos.

Las filas que contienen los datos generalmente son voluminosas porque almacenan muchos atributos. Los índices, en cambio, normalmente contienen sólo una colección de parejas, denominadas *entradas*, formadas por un valor y un RID.

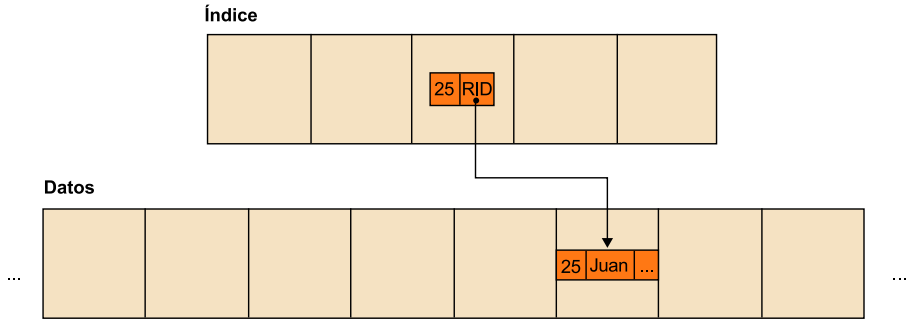
Espacio ocupado por los índices y por los datos

Las filas de una tabla de empleados podrían registrar el número de empleado, su nombre, DNI, dirección, teléfono, sueldo, número de despacho, departamento donde está asignado, etc. En cambio, si queremos tener un índice para acceder a los empleados según su número de empleado, las entradas del índice tendrán que estar formadas sólo por un número de empleado y un RID.

Una búsqueda mediante un índice consiste en localizar primero los valores en el índice para conocer el RID. Una vez se dispone del RID, se emplea un acceso directo por posición para conseguir la página que contiene la fila de los datos que buscábamos.

La figura 16 ilustra la consulta del empleado número 25 con un índice.

Figura 16. Consulta mediante un índice



De los párrafos anteriores, podemos deducir que los SGBD utilizan el acceso por posición para implementar los accesos por valor.

Un índice tiene que estar organizado de alguna manera que facilite las búsquedas de los valores que contiene. Hay varias maneras de organizar los índices: los árboles B^+, las funciones de dispersión, etc. Algunos de estos tipos de índice sólo facilitan el acceso directo por valor (por ejemplo, las funciones de dispersión); otros sirven tanto para el acceso directo como para el acceso secuencial por valor (por ejemplo, los árboles B^+).

Ejemplo

Sobre una tabla que contiene datos de empleados podemos tener un índice que facilite el acceso por número de empleado, otro que facilite el acceso por nombre, otro que facilite el acceso por número de despacho, etc.

Una característica muy útil de la mayoría de los tipos de índice es que permiten la posibilidad de tener varios índices sobre unos mismos datos.

Las peculiaridades de los índices sugieren que es mejor gestionarlos de manera separada de los datos de las tablas. Por este motivo, hay un tipo de espacio virtual para contenerlos, denominado *espacio de índices*.

Más adelante, estudiamos los tipos de índices que utilizan más a menudo los SGBD para implementar los accesos por valor: los estructurados como árboles B^+ y los estructurados según funciones de dispersión. Cada SGBD tiene sus particularidades propias en la implementación de un tipo de índice determinado. Dado que sería impracticable intentar explicar exhaustivamente y con todos los detalles las implementaciones que hay, solo estudiamos los principios comunes a la mayoría de las implementaciones de índices estructurados como árboles B^+ y los de los índices estructurados mediante funciones de dispersión. Estos principios nos facilitan la comprensión de las implementaciones particulares que proporcionan los SGBD del mercado.

Observación

Observad que los índices también se pueden usar para implementar los accesos que requieren los ficheros por valor.

El cálculo del coste de ejecución de los accesos a los datos mediante la utilización de índices es un aspecto que deberemos considerar de ahora en adelante para evaluar la bondad de los diferentes tipos de índice. Para este cálculo, tomamos las convenciones siguientes:

1) Consideramos sólo el coste de las E/S y no otros componentes del coste, como por ejemplo el que corresponde a los cálculos de la UCP[16]. El motivo es que el coste de las E/S es normalmente el componente dominante en el coste de las operaciones que se llevan a cabo en las BD y nos proporciona una buena aproximación a los costes reales.

(16)UCP es la sigla del término *unidad central de procesamiento*.

2) Para contabilizar el coste de las E/S adoptamos la simplificación de contar el número de páginas que se leen o que se graban en el disco. Esta simplificación nos sirve porque suponemos que todas las E/S transportan únicamente una página (es el caso más habitual) y que para acceder a una página siempre hay que hacer una E/S, a pesar de que en algunos casos concretos podría no ser necesario (por ejemplo, si ya se ha accedido a la página con anterioridad y todavía se tiene una copia de la misma en la memoria interna).

6.3.3. Árboles B+

En los subapartados siguientes, presentamos un tipo de índice que sirve para facilitar el acceso directo y el secuencial por valor de un atributo: los árboles B^+.

Los **árboles B^+** son un tipo particular de árbol de búsqueda. El objetivo primordial de la estructura de los árboles B^+ es el de conseguir que las búsquedas se efectúen con un número pequeño de E/S.

Es importante tener en cuenta que, a diferencia otros tipos de árboles, los árboles B^+ que estudiamos aquí están orientados a realizar búsquedas en el disco.

El estudio de los árboles B^+ es muy interesante porque la mayoría de los sistemas comerciales (como por ejemplo Oracle, PostgreSQL, Informix, DB2, etc.) los implementan.

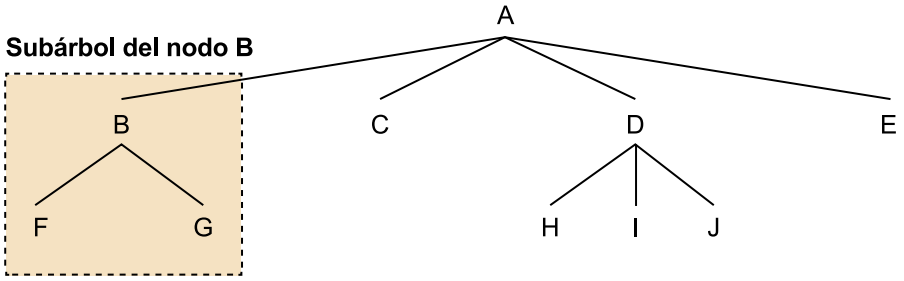
Terminología de las estructuras de datos de árbol

Empezamos presentando brevemente la terminología que se emplea al hablar de las estructuras de datos en forma de árbol.

Un **árbol** se compone de nodos. Cada **nodo del árbol**, excepto un nodo especial denominado *raíz*, tiene un nodo padre y varios (cero o más) nodos hijos. El **nodo raíz** no tiene padre. Los nodos que no tienen hijos se denominan **nodos hoja** y los nodos que no son hojas se denominan **nodos internos**. El nivel de un nodo es siempre el nivel de su padre más uno, y el nivel del nodo raíz es uno. Un **subárbol de un nodo** consiste en el nodo y todos sus nodos descendentes: sus nodos hijos, los nodos hijos de sus hijos, etc.

La figura 17 ilustra la estructura de datos de árbol que acabamos de presentar.

Figura 17. Estructura de un árbol



Ejemplo

En la figura 17 podemos ver los siguientes elementos del árbol:

- El nodo raíz de la figura es A.
- Los nodos hijos de A son B, C, D y E.
- Los nodos hoja son F, G, C, H, Y, J y E.
- El subárbol del nodo B es el marcado en la figura.

Estructura de los nodos

En primer lugar, hay que considerar que todo árbol B^+ tiene un número de orden *d* que indica la capacidad de sus nodos. Más concretamente, si un árbol B^+ tiene orden *d*, entonces sus nodos contienen como máximo $2d$ valores.

En un árbol B^+, los nodos internos y los nodos hoja tienen una estructura diferente. Las causas de esta diferencia estructural son tres:

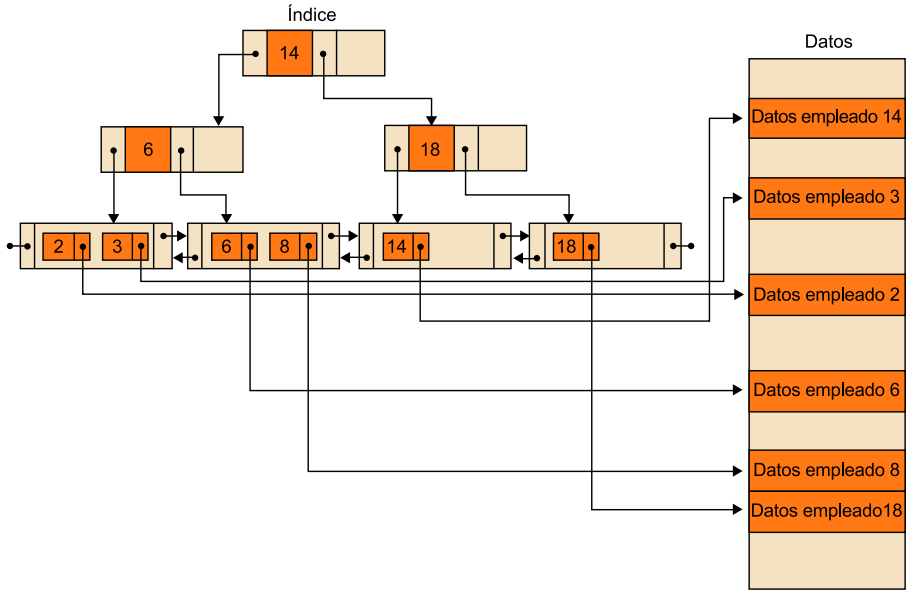
1) Los nodos hoja son los que contienen todas las entradas del índice (es decir, las parejas de valor y RID).

2) Los nodos internos tienen como objetivo dirigir la búsqueda de la hoja que tiene la entrada correspondiente a un valor determinado. El acceso directo por valor consistirá, pues, en hacer un recorrido del árbol que empezará en la raíz e irá bajando por los nodos del árbol hasta llegar a la hoja adecuada (la que contiene, si existe, la entrada correspondiente al valor buscado).

3) Los nodos hoja están conectados por apuntadores, que sirven para facilitar el acceso secuencial por valor (el recorrido de las hojas siguiendo los apuntadores proporciona las entradas ordenadas por valor).

La figura 18 ilustra lo que acabamos de explicar.

Figura 18. Índice del árbol B^+



Ejemplo de estructura de un árbol B^+

La figura 18 muestra un índice de árbol B^+ de orden $d = 1$ que sirve para acceder a datos de empleados según el valor del atributo "número de empleado". Observad que los RID que apuntan a los datos están sólo 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.

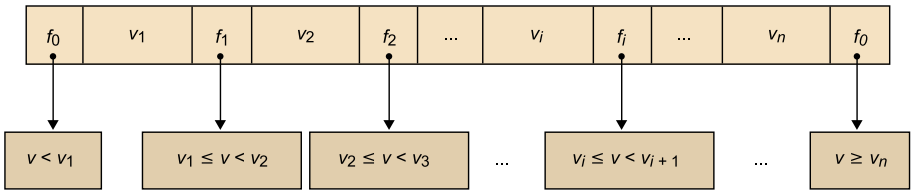
A continuación, describiremos la estructura de los nodos internos y después la estructura de los nodos hoja de los árboles B^+.

1) Estructura de un nodo interno

Un **nodo interno** contiene valores y apuntadores hacia sus nodos hijos (una manera de implementar un árbol es tener en cada nodo apuntadores hacia todos sus nodos hijos).

La estructura de un nodo interno que contiene n valores (donde n es menor o igual que $2d$) es la que muestra la figura 19.

Figura 19. Estructura de un nodo interno



Cada v_i indica un valor y cada f_i indica un apuntador a un nodo hijo del árbol. Si un nodo contiene n valores, entonces tiene $n + 1$ apuntadores a otros nodos del índice.

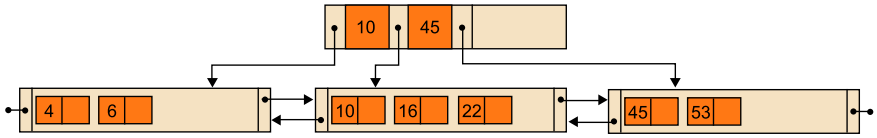
Tal como indica la figura 19, se cumple que el subárbol del nodo apuntado por f_i contiene valores v tales que:

- $v_i \leq v < v_{i+1}$, si $i > 0$ e $i < n$.
- $v < v_1$, si $i = 0$.
- $v \geq v_n$, si $i = n$.

Ejemplo de estructura de un nodo interno

La figura 20 muestra varios nodos de un índice de árbol B^+ de orden 2 por el atributo número de empleado.

Figura 20. Ejemplo de árbol B^+ de orden 2



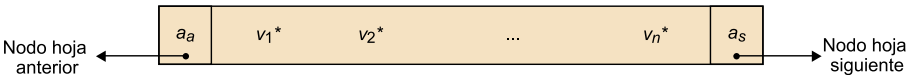
Considerad el nodo raíz: tiene tres apuntadores que apuntan a sus nodos hijos. Observad que el subárbol apuntado por el primero de estos apuntadores contiene valores menores que 10. El subárbol apuntado por el segundo contiene valores mayores o iguales que 10 y menores que 45. Finalmente, el subárbol apuntado por el tercero contiene valores mayores o iguales que 45.

2) Estructura de un nodo hoja

Los **nodos hoja** de los árboles B^+ contienen entradas formadas por los valores a los cuales se tiene que acceder y el RID que apunta a los datos, un apuntador al nodo hoja anterior y un apuntador al nodo hoja siguiente.

La estructura de un nodo hoja que contiene n valores (donde n es menor o igual que $2d$) es la que muestra la figura 21. En la figura, cada v_i corresponde a la entrada del valor v_i (por tanto, v_i y su RID), a_a indica el apuntador al nodo hoja anterior y a_s indica el apuntador al siguiente.

Figura 21. Estructura de un nodo hoja



De manera adicional, es preciso que los nodos hoja cumplan las condiciones siguientes:

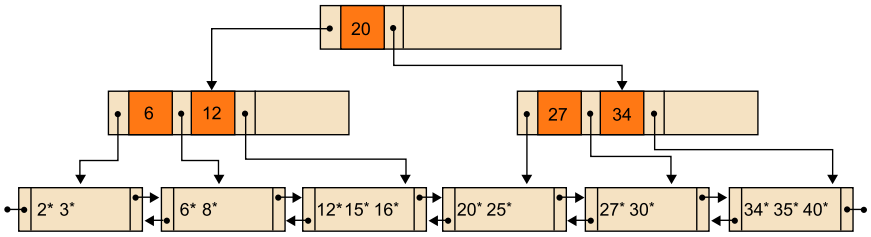
- Todos los valores de un nodo hoja son menores que los valores del nodo hoja siguiente.

- Todos los valores de algún nodo interno del árbol están repetidos en alguna hoja del árbol (así, las hojas contienen todos los valores del árbol).

Ejemplo de estructura de un nodo hoja

La figura 22 muestra varios nodos de un índice de árbol B^+ de orden 2 por el atributo número de empleado.

Figura 22. Ejemplo de estructura de un nodo hoja



Considerad, por ejemplo, el nodo hoja que está situado más a la izquierda. Contiene las entradas del empleado número 2 y del empleado número 3. También tiene los apuntadores en la hoja anterior y en la siguiente del árbol (en este caso concreto, no existe hoja anterior). Observad que todos los valores del nodo considerado (los valores 2 y 3) son menores que los valores de la hoja siguiente (6 y 8). Finalmente, observad que el valor 6 de la segunda hoja está repetido en un nodo interno. El motivo es que en un árbol B^+ las hojas deben contener todos los valores del árbol.

Acceso directo por valor

Para realizar un **acceso directo por valor** con un índice del tipo árbol B^+ se requiere primero localizar la hoja que tiene la entrada del valor buscado, y después usar el RID de la entrada para localizar los datos a los que se quiere acceder.

Ejemplo de localización de la hoja que tiene la entrada del valor buscado

Mostraremos cómo se puede implementar la localización de un valor *v* en el árbol con un ejemplo de la figura 22. Supongamos que queremos encontrar el valor 8 en el árbol del ejemplo anterior.

Accedemos primero al nodo raíz. Después, seguimos el primer apuntador porque $8 < 20$. A continuación, seguimos el segundo apuntador, dado que $6 \leq 8 < 12$, y localizamos el nodo hoja que contiene la entrada del valor deseado.

Supongamos ahora que buscamos el valor 26 en el mismo árbol. Accederemos igualmente al nodo raíz. Seguiremos el apuntador al hijo de la derecha, porque $26 \geq 20$. Después seguiremos el primer apuntador porque $26 < 27$. El nodo hoja que obtenemos esta vez no contiene la entrada del valor 26, lo cual indica que el 26 no está en nuestros datos.

Acceso secuencial por valor

Para realizar un acceso secuencial por valor con un índice del tipo árbol B^+ es necesario primero localizar ordenadamente todas las entradas de los valores a los cuales se quiere acceder y, para cada una, usar el RID que apunta a los datos para encontrar el valor.

La localización ordenada de las entradas del árbol es sencilla. Conviene recordar que las hojas contienen todos los valores del árbol, que cada hoja tiene un apuntador a la hoja siguiente y que los valores de un nodo hoja son menores que los valores del nodo hoja siguiente. Entonces, solo hay que acceder primero a la hoja situada más a la izquierda y, después, ir accediendo cada vez a la hoja siguiente hasta que se acabe la cadena de hojas.

Observación

Si seguimos el procedimiento de acceso secuencial por valor explicado en este subapartado con el árbol de la figura 21, iremos obteniendo todas las entradas del árbol de manera ordenada.

Propiedades destinadas a mejorar el rendimiento

Consideremos un acceso directo por valor que se implementa con un árbol B^+. Para localizar la hoja que contiene la entrada del valor, el número de nodos que hay que recorrer es igual al del nivel de la hoja mencionada. En consecuencia, si reducimos el nivel de las hojas de un árbol B^+, conseguiremos reducir el número de nodos que hay que recorrer cuando se hace un acceso directo por valor.

Una propiedad de los árboles B^+ destinada a reducir el nivel de las hojas es que todos los nodos del árbol (excepto la raíz) tienen que estar llenos, como mínimo, hasta un 50% de su capacidad. Esta norma equivale a exigir que en un árbol de orden *d* todos los nodos excepto la raíz contengan, al menos, *d* valores.

Esta norma evita tener árboles B^+ donde muchos de los nodos estén prácticamente vacíos. Si conseguimos que los nodos de un árbol no estén demasiado vacíos, el árbol tendrá menos nodos y también menos niveles.

Con el objetivo de mejorar el rendimiento de un índice, suele ser deseable que el número de nodos que haya que recorrer sea siempre más o menos el mismo, independientemente de cuál sea el valor al que se quiere acceder.

Por el motivo anterior, los árboles B^+ cumplen una segunda propiedad: ser equilibrados. Un árbol es equilibrado si todas sus hojas están en el mismo nivel.

En un árbol equilibrado, el nivel de las hojas es lo que se denomina **altura del árbol**. La localización de cualquier hoja siempre requerirá recorrer un número de nodos que coincidirá con la altura del árbol.

Observación

Observad que el árbol de la figura 22 es un árbol equilibrado y su altura es 3.

Almacenamiento del árbol y coste de localización de una entrada

Tal y como ya hemos visto, por las peculiaridades de los índices es aconsejable gestionarlos de manera separada de los datos de las tablas. Por este motivo, existe un tipo de espacio virtual para contener los índices denominado *espacio de índices*.

Una cuestión importante sobre el almacenamiento de un árbol es la elección de un tamaño adecuado para sus nodos (que depende, por lo menos, del orden del árbol). Ya hemos explicado que interesa que un árbol B^+ tenga una altura pequeña; para ello, es preciso que los nodos del árbol sean grandes aunque sin sobrepasar el tamaño de una página, porque de lo contrario no se podrían consultar con una única E/S.

En consecuencia, el tamaño habitual de los nodos de un árbol B^+ coincide con el tamaño de las páginas y cada nodo del árbol se almacena en una página virtual diferente. Observad que, según esta forma de almacenamiento, en un árbol B^+ de altura h son necesarias h E/S para localizar una entrada del árbol (por ejemplo, si $h = 3$, harán falta tres E/S).

El hecho de que los nodos sean del tamaño de una página (que almacena normalmente 4 kB) permite tener típicamente árboles de órdenes entre 50 y 100.

Reflexión

Si pensamos que el árbol B^+ se construye sobre el número de empleado, y suponiendo que este número es la clave primaria de la relación de empleados, nuestra relación podrá contener hasta 338.100 empleados diferentes.

Orden habitual de un árbol y volumen de datos indexados

a) Supongamos que disponemos de páginas de 4 kB y los valores por indexar ocupan 20 bytes. Consideremos también que el tamaño del RID es de 4 bytes y los apuntadores a páginas del árbol ocupan 3 bytes. Según estos datos, en los nodos internos caben 177 valores y 178 apuntadores a otros nodos del árbol; en los nodos hoja caben 170 entradas (parejas de valor y RID) y los 2 apuntadores a las hojas anterior y siguiente. Puesto que la capacidad en valores debe ser la misma en todos los nodos, la restringiremos a 170 (el peor caso). El orden del árbol será, pues, la mitad de 170, es decir, $d = 85$. Observad que hemos obtenido un orden que se encuentra entre 50 y 100.

b) Consideremos ahora un árbol de orden $d = 50$, de altura $h = 3$ y donde todos los nodos tienen una ocupación del 69%. Si $d = 50$, la capacidad máxima de los nodos es de 100 y una ocupación del 69% supone tener 69 valores en cada nodo. En este árbol, tendremos el número de valores siguiente en cada nivel:

- Nivel 1: 1 nodo con 69 valores y 70 apuntadores.
- Nivel 2: 70 nodos con 69 valores cada uno y 70 apuntadores cada uno.
- Nivel 3: 4.900 nodos con 69 entradas cada uno.

Es decir, 338.100 entradas en total. Según esto, el árbol B^+ anterior, que tiene una altura de solo 3 niveles, nos permite indexar un volumen de datos de 338.100 filas.

De este ejemplo se desprende que con órdenes $d = 50$ y una altura de $h = 3$ se puede indexar una cantidad de datos considerable.

Inserciones y supresiones

Las inserciones y supresiones se tienen que hacer de manera que el árbol resultante satisfaga todas las propiedades de los árboles B^+. Esto implica en algunos casos realizar una reestructuración del árbol.

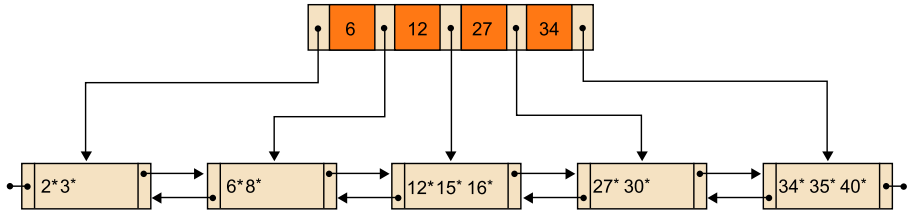
A) Inserciones

Describimos un algoritmo que, dada una entrada, localiza el nodo hoja que le corresponde y, si este nodo tiene espacio libre, se la inserta. En el supuesto de que este nodo hoja no tenga espacio libre para la nueva entrada, reestructura el árbol para encontrarle un lugar.

Comentaremos algunos ejemplos que nos ayudarán a entender con más exactitud qué debe hacer este algoritmo de inserción, cuya descripción detallada queda fuera del alcance de este módulo.

Consideremos el árbol B^+ de orden 2 de la figura 23.

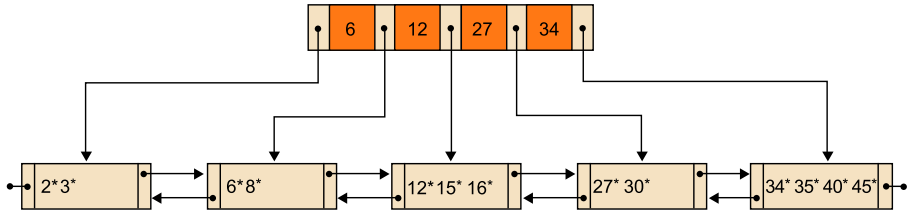
Figura 23. Árbol de partida I



A partir de este árbol, consideraremos los casos de inserción que exponemos a continuación:

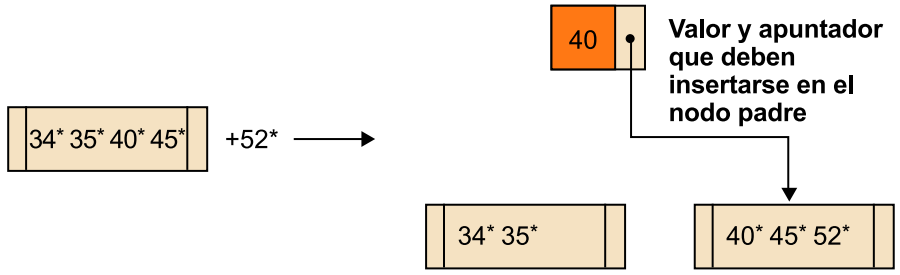
a) Supongamos que se quiere insertar en este árbol la entrada 45*. El 45 tiene que pertenecer a la hoja situada más a la derecha. Dado que esta hoja tiene espacio libre, la entrada 45* se puede colocar en la hoja que le corresponde y no hay que hacer ninguna reestructuración del árbol. Se obtiene el árbol que muestra la figura 24.

Figura 24. Inserción sin necesidad de reestructuración



b) Ahora consideramos que se quiere insertar la entrada 52* en el árbol que acabamos de obtener. Le corresponde la hoja situada más a la derecha, pero no tiene espacio libre. Para insertar la nueva entrada, hay que dividir la hoja en dos. Las dos hojas resultantes de la división se muestran en la figura 25.

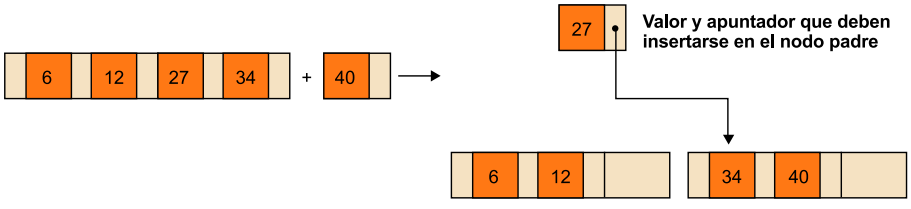
Figura 25. División de un nodo hoja



Cuando se divide un nodo, hay que insertar en su nodo padre un valor y un apuntador al nuevo hijo. En nuestro ejemplo, este valor es el 40, porque es el valor más pequeño del segundo nodo hoja que ha resultado de la división. Es necesario, pues, insertar el 40 y el apuntador en el nodo padre. El nodo padre no tiene espacio y habrá que dividirlo.

Cuando se divide un nodo no-hoja, sus primeros d valores se dejan en el nodo (el 6 y el 12), los d valores más grandes se colocan en un nuevo nodo (el 34 y el 40) y el valor del medio se inserta en el nodo padre (el 27), junto con un apuntador hacia el nuevo nodo. La figura 26 nos muestra esta división de un nodo no-hoja.

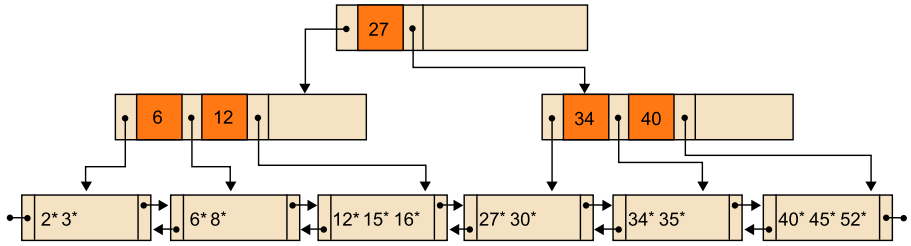
Figura 26. División de un nodo no-hoja



Observad las diferencias que hay entre esta división y la división de un nodo hoja.

Puesto que acabamos de dividir el nodo raíz del árbol, hay que crear un nuevo nodo raíz en el que se colocan el valor 27, el apuntador al nuevo nodo y también un apuntador a la raíz antigua. El árbol que se obtiene finalmente es el que podéis ver en la figura 27.

Figura 27. Inserción con reestructuración



B) Supresiones

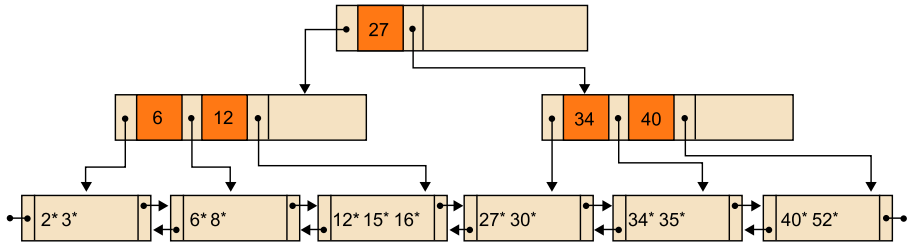
Describiremos un algoritmo que, partiendo de una entrada, localiza el nodo hoja que le corresponde, lo borra y, si hace falta, reestructura el árbol para que todos los nodos del árbol (excepto la raíz) tengan, por lo menos, *d* valores.

Comentaremos algunos ejemplos para entender de manera más precisa cómo funciona el algoritmo, cuya descripción detallada queda fuera del alcance de este módulo.

Consideremos el árbol de la figura 27 y supongamos los casos de supresión siguientes:

a) Supongamos que queremos borrar el valor 45 del árbol. La hoja que contiene la entrada del valor 45 es la que está situada más a la derecha. Después de borrar 45*, la hoja todavía contiene dos valores y, por lo tanto, no hay que hacer ninguna reestructuración. El árbol obtenido es el que muestra la figura 28.

Figura 28. Supresión sin necesidad de reestructuración



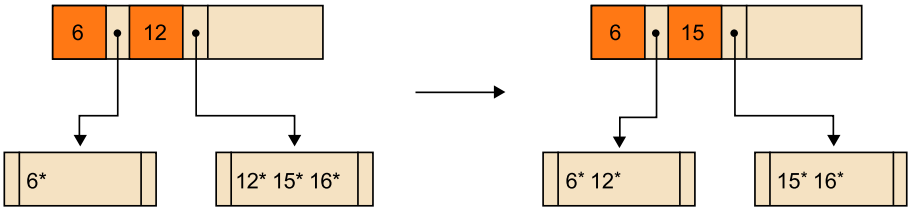
b) Ahora supondremos que queremos borrar el valor 8 del árbol que acabamos de obtener. Su entrada está situada en la segunda hoja del árbol. Después de borrarla, la hoja se queda con menos de dos entradas, por lo que hay que corregirlo. Para garantizar que la ocupación de un nodo sea correcta, es decir, que la estructura resultante después de la supresión siga siendo un árbol B^+, siempre utilizamos su nodo hermano de la derecha; solo si no tiene hermano a la derecha usaremos el de la izquierda.

Nodos hermanos

Dos nodos son hermanos si comparten el mismo nodo padre.

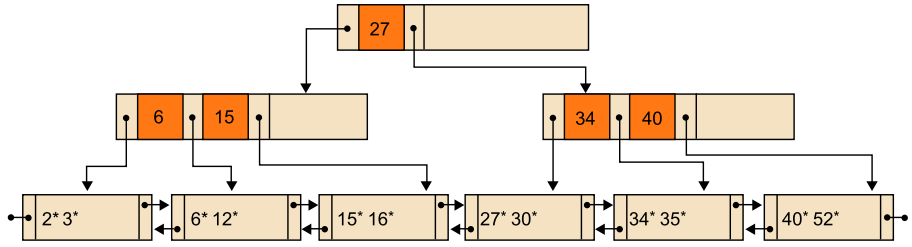
En nuestro ejemplo, puesto que el nodo hermano de la derecha tiene tres entradas, se hace una redistribución de entradas entre los dos nodos hoja. La figura 29 ilustra esta redistribución.

Figura 29. Redistribución con nodos hoja



Observad que la redistribución afecta al contenido del nodo padre de los dos nodos participantes. El valor 12 del nodo padre se cambia por el 15. El árbol que se obtiene finalmente es el que se muestra en la figura 30.

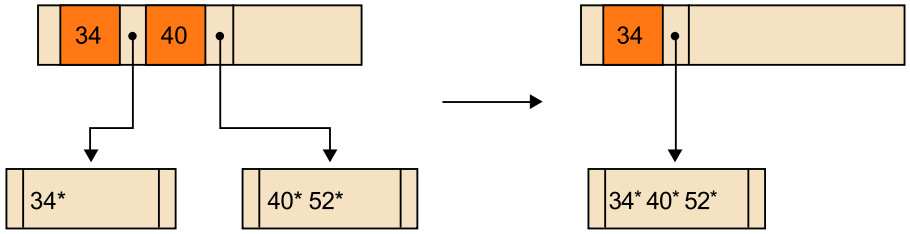
Figura 30. Supresión con reestructuración



c) Consideremos ahora el caso del borrado del 35 del árbol que acabamos de obtener. La entrada del 35 está situada en la penúltima hoja. Después de borrarla, la hoja se queda con menos de dos entradas, de modo que hay que llevar a cabo una reestructuración. En este caso, su nodo hermano de la derecha tiene solo dos entradas y, por lo tanto, no se hará una redistribución como en el caso anterior (el nodo hermano no tiene entradas sobrantes para redistribuir), sino que se hará una fusión de los dos nodos hoja en un único nodo.

La figura 31 nos muestra esta fusión de los nodos hoja.

Figura 31. Fusión de nodos hoja



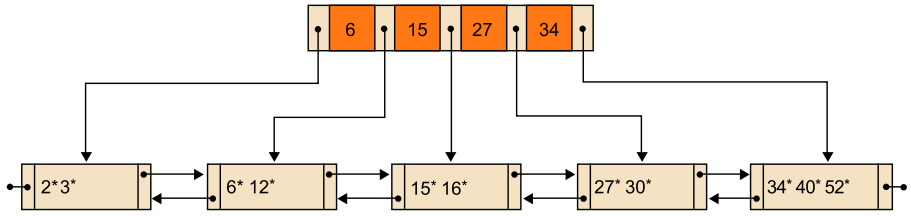
Observad que cuando se fusionan dos nodos, su nodo padre pierde un valor. Debido a esta pérdida, el nodo padre de nuestro ejemplo nos ha quedado con menos de dos valores. En este caso, será preciso hacer todavía otra reestructuración del árbol para corregirlo. El nodo padre no tiene ningún hermano a la derecha; por lo tanto, usaremos el de la izquierda para reestructurar el árbol.

El hermano de la izquierda tiene solo dos valores, por lo que se hará una fusión de ambos nodos. La figura 32 nos ilustra esta fusión de dos nodos no-hoja.

Conviene observar las diferencias respecto de la fusión anterior de dos hojas.

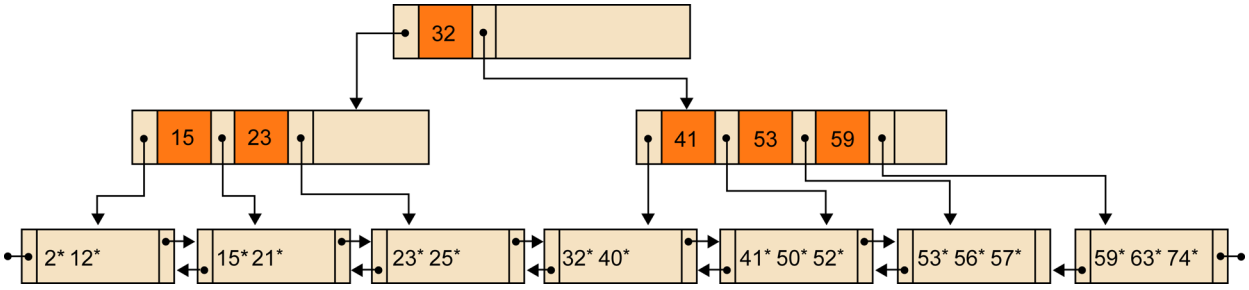
Después de esta última fusión, nos ha quedado un árbol con la raíz vacía. En estos casos hay que descartar la raíz antigua y construir la raíz nueva con el nodo resultante de la fusión. Finalmente, se obtiene el árbol que muestra la figura 33.

Figura 33. Supresión con reestructuración y pérdida de un nivel



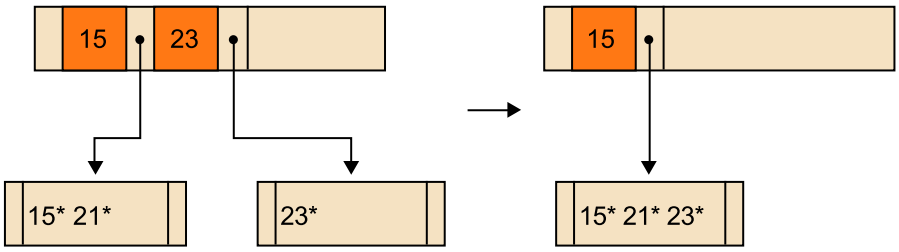
d) Para el último ejemplo de supresión, elegimos el árbol B^+ de partida que muestra la figura 34.

Figura 34. Árbol de partida II



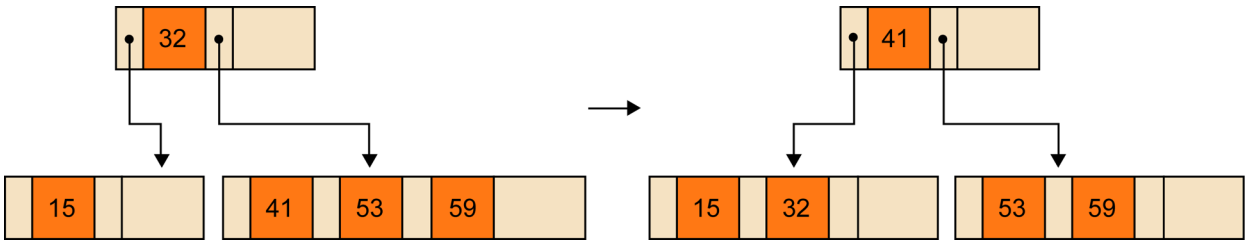
Consideremos el borrado del valor 25 del árbol. La entrada del 25 está en la tercera hoja. Después de borrarlo, la hoja se queda con menos de dos entradas. Esta hoja no tiene ninguna hoja hermana a la derecha y hay que fusionarla con la de la izquierda, como muestra la figura 35.

Figura 35. Fusión de nodos hoja



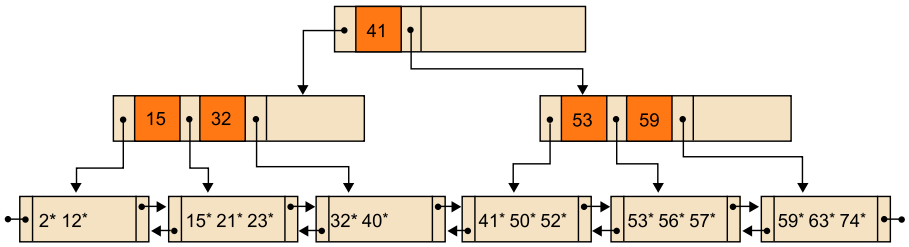
Después de la fusión, el nodo padre se ha quedado con menos de dos valores. Será necesario hacer una redistribución con su nodo hermano derecho (que tiene más de dos valores). La figura 36 muestra esta redistribución de valores entre nodos no-hoja.

Figura 36. Redistribución con nodos no-hoja



Entonces, el árbol B^+ que se obtiene finalmente es el que muestra la figura 37.

Figura 37. Supresión con redistribución de nodos

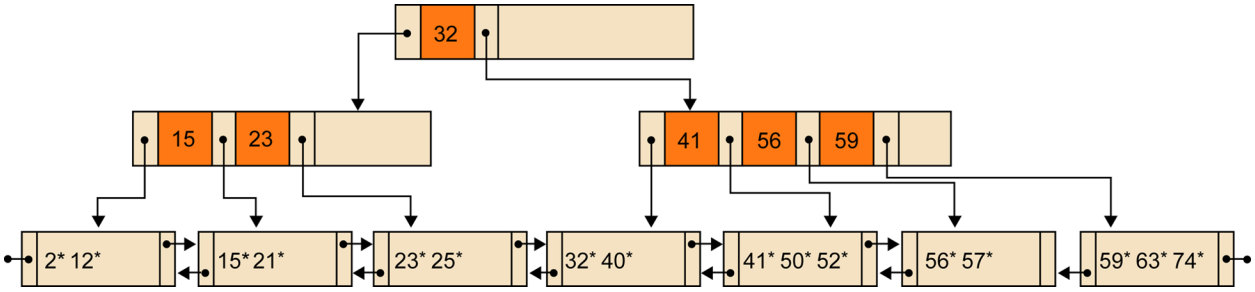


C) Supresiones de valores que tienen una copia en nodos internos

Hemos analizado un procedimiento de borrado de los valores de un árbol B^+ que sólo aparecen en un nodo hoja del árbol. Como ya sabemos, un árbol B^+ puede tener valores que figuran en dos lugares: en un nodo hoja y en un nodo no-hoja (nodo interno). Para borrar los valores que tienen una copia en nodos internos, podemos utilizar el mismo procedimiento si hacemos previamente una sustitución del valor copiado en el nodo interno por el valor del árbol que le sigue según el orden de los valores. Una vez hecho esto, podremos emplear el procedimiento de supresión anterior para eliminar el valor de la hoja.

Borramos el valor 53 de la figura 34, que tiene la entrada en la penúltima hoja del árbol y una copia en el nodo padre de la hoja mencionada. Sustituimos la copia del 53 por el valor que le sigue, que es el 56. Una vez hecho esto, borramos el 53 de la hoja. El árbol que nos queda se muestra en la figura 38.

Figura 38. Supresión de valores que tienen una copia en nodos internos



Valores repetidos

En las búsquedas, inserciones y supresiones de los árboles B^+ que acabamos de explicar no hemos considerado la posibilidad de que pudiera haber valores repetidos en los datos; hemos considerado que indexábamos los valores de atributos identificadores.

Existen varias soluciones para el tratamiento de los valores repetidos en los árboles B^+; comentaremos dos brevemente:

1) Una posibilidad es permitir la existencia de entradas diferentes con el mismo valor en el árbol. En este caso, hojas diferentes pueden tener entradas correspondientes a un mismo valor. El algoritmo de acceso directo por valor tendrá que localizar primero la entrada situada más a la izquierda del valor, y después todas las posibles entradas adicionales del mismo valor. Estas entradas adicionales se pueden encontrar en otras hojas y se localizan siguiendo los apuntadores entre hojas del árbol. Las inserciones y supresiones también deben adaptarse.

2) Otra posibilidad es tener una sola entrada que contenga varios RID (uno para cada aparición del valor). Esta posibilidad provoca que el tamaño de las entradas sea variable y que su gestión resulte más compleja. Por el contrario, se ahorra espacio porque el valor en entradas diferentes no se repite, y, en consecuencia, se ahorran operaciones de E/S.

6.3.4. Dispersión

Otros tipos de índices sirven para facilitar el acceso directo por valor, pero no el acceso secuencial por valor: son los índices basados en la dispersión[17].

[17] En inglés, *hashing*.

A continuación, analizaremos la utilización de la dispersión para implementar búsquedas en el disco, donde es fundamental la minimización del número de E/S.

Los **índices basados en la dispersión** consiguen normalmente un rendimiento algo mejor que los índices de árbol B^+ cuando se trata de implementar los accesos directos por valor. Por el contrario, no permiten implementar los accesos secuenciales por valor, mientras que los árboles B^+ sí lo permiten. En consecuencia, algunos sistemas comerciales proporcionan solo implementaciones basadas en los árboles B^+.

Los sistemas comerciales

Algunos sistemas comerciales, como por ejemplo Oracle, proporcionan solo implementaciones basadas en los árboles B^+. Otros, como por ejemplo el sistema PostgreSQL, proporcionan ambos tipos de índices.

Introducción a la dispersión

La característica fundamental de los índices basados en la dispersión es que las entradas se colocan en las páginas según una función de dispersión *h*.

La **función de dispersión *h*** es una función que a partir de un valor *v* devuelve un número de página *p*, es decir, $p = h(v)$. El **número de página *p*** será un entero del intervalo [1, ..., *N*]. Entonces, la entrada correspondiente al valor *v*, que representamos por *v**, se situará en la página número *p* del intervalo.

Cuando se quiera acceder al valor *v*, podremos acceder a su entrada, *v**, de la manera siguiente:

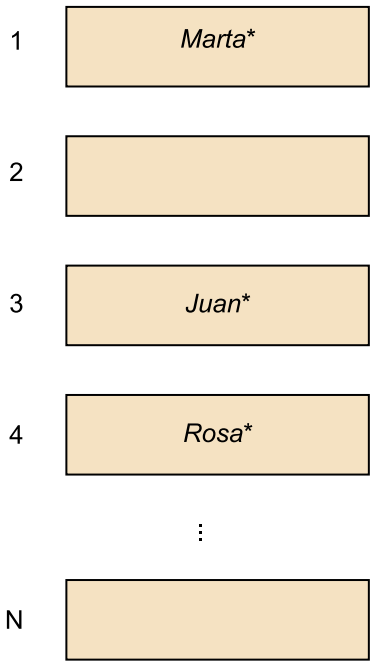
- Se calcula el número de página *p*, tal que $p = h(v)$.
- Se accede a la página *p* para localizar la entrada del valor *v*.

Podemos utilizar diferentes funciones *h* de dispersión que transforman un valor numérico o alfanumérico en un número entero del intervalo [0, ..., $N - 1$]. Podremos usar cualquiera de estas funciones siempre que sumemos 1 al resultado que nos den, porque nos interesa obtener números de página que estén en el intervalo [1, ..., *N*].

Localización de una entrada con un índice basado en la dispersión

Queremos indexar unos datos por el valor de un atributo que representa nombres de pila. Supongamos que $h(Juan) = 3$, $h(Marta) = 1$ y $h(Rosa) = 4$. Entonces, las entradas de estos valores quedarán colocadas en las páginas del índice tal y como se muestra en la figura 39.

Figura 39. Ejemplo de página de índice



Para localizar la entrada del valor *Rosa* en el índice anterior, calcularemos $h(Rosa)$. Este cálculo nos dará el número de página 4. Luego, accederemos a la página 4, donde encontraremos la entrada correspondiente a *Rosa*.

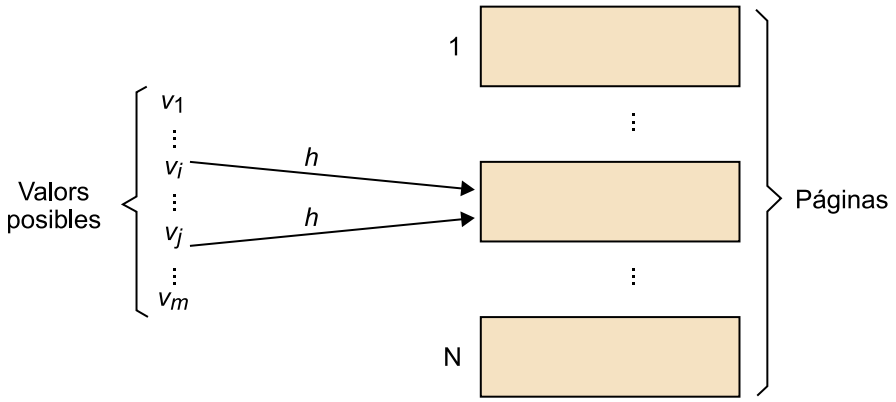
Observad que el número de valores posibles puede ser muy grande, mientras que el número de páginas que puede ocupar el índice en el disco suele ser mucho más limitado. Esto hace que el número de valores v posibles sea mucho mayor que el número N de páginas posibles. La función de dispersión h transforma valores de un intervalo muy grande en posiciones de un intervalo menor (el intervalo [1, ..., N]).

Ejemplo

Pensad en valores de un atributo que representa un DNI. Los valores posibles del atributo se encuentran entre 0 y 99.999.999. En cambio, el número de páginas que puede ocupar el índice en el disco difícilmente llegará a los cien millones.

Por este motivo, es posible que para dos valores diferentes, v_i y v_j, suceda que $h(v_i) = h(v_j)$, es decir, que la función devuelva la misma página para los dos valores, como se aprecia en la figura 40.

Figura 40. Valores sinónimos

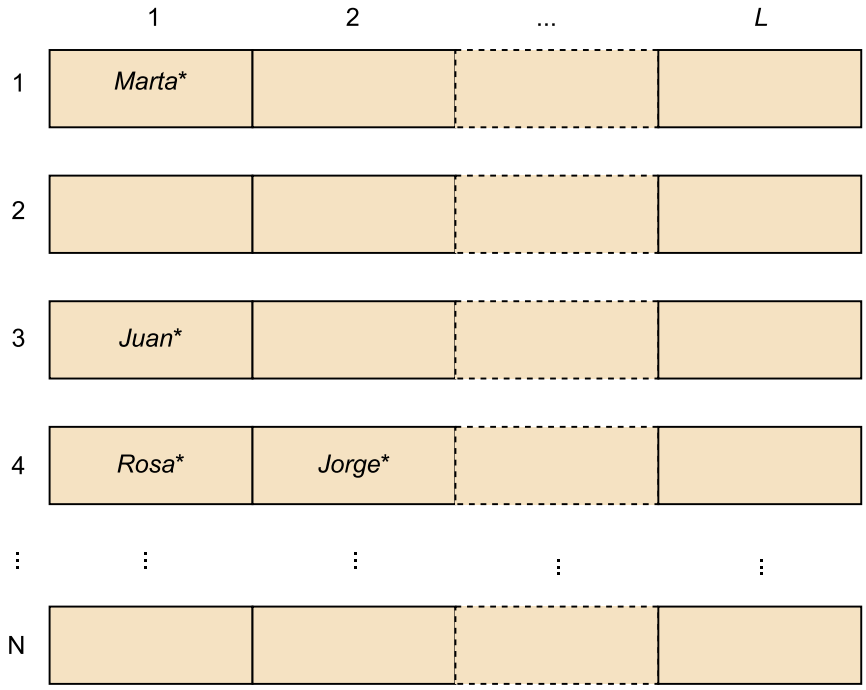


En este caso, los valores v_i y v_j se denominan *sinónimos*. Las entradas correspondientes a los valores sinónimos deben estar en la misma página. Las páginas, por lo tanto, deben poder almacenar varias entradas. Denominamos L al número de entradas que caben en una página.

Ejemplo de valores sinónimos

Si al índice por el atributo "nombre de pila" anterior añadimos el valor *Jorge* y $h(Jorge) = 4$, entonces tendremos la situación que se muestra en la figura 41.

Figura 41. Ejemplo de página de índice



A veces, una página puede ser insuficiente para contener todos los sinónimos que deberían colocarse. Esto sucederá cuando a una página le correspondan más de L sinónimos.

Cuando un sinónimo no tiene espacio en la página que le corresponde, se denomina *excedente* y se coloca en alguna otra página, que se elige según una determinada política de gestión de excedentes. Existen diversas maneras alternativas de gestionar los excedentes; a continuación, explicaremos una.

Gestión de excedentes

La política de gestión de excedentes que estudiamos se basa en que el índice lo forman dos tipos disjuntos de páginas:

- **Páginas primarias**: páginas donde colocaremos todas las entradas que no son excedentes (habrá N).
- **Páginas de excedentes**: páginas destinadas a guardar las entradas excedentes.

Cuando una página primaria *p* se llena y se le inserta un nuevo valor que según la función de dispersión debería ir en la misma página *p*, la entrada del valor se coloca en una página de excedentes *e* y se añade a la página primaria *p* un apuntador hacia la página *e*.

Esta página de excedentes *e* se utilizará solo para contener los posibles excedentes de la página primaria *p* que se inserten posteriormente. Si la página *e* también se llena, se le encadenará una nueva página de excedentes *e'* donde se podrán colocar más excedentes de la página primaria *p*, y así sucesivamente.

De este modo, para cada página primaria con excedentes, se construye una cadena de páginas de excedentes. El primer apuntador de la cadena está situado en la página primaria que lo ha originado.

Puesto que hay una cadena separada de páginas de excedentes para cada página primaria, esta política de gestión de excedentes a veces se denomina **encadenamiento separado**[18].

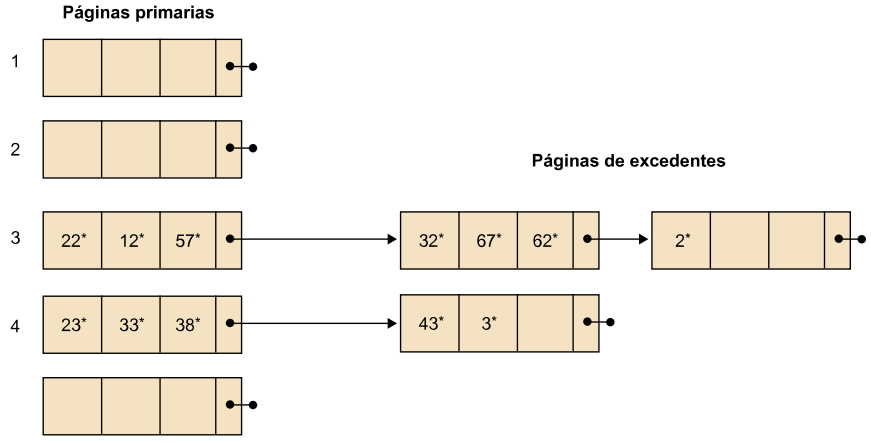
[18] En inglés, *separate chaining*.

Las supresiones de los valores del índice pueden hacerse de varias maneras. Una posibilidad es marcar el valor con una señal especial que indique que está borrado, sin reorganizar nada. Otra posibilidad es aprovechar estas supresiones de valores para reducir la longitud de las cadenas de excedentes.

Ejemplo de gestión de excedentes con encadenamiento separado

Supongamos que tenemos un índice con cinco páginas primarias ($N = 5$), que todas las páginas tienen una capacidad de tres entradas ($L = 3$) y que la función de dispersión es $h(x) = (x \bmod 5) + 1$. Si insertamos los valores: 22, 23, 12, 57, 33, 38, 32, 67, 62, 43, 2, 3, el índice quedará como muestra la figura 42.

Figura 42. Ejemplo de página de excedentes



Almacenamiento y coste de localización de una entrada del índice

Los índices basados en la dispersión se almacenan en un tipo de espacio virtual denominado *espacio de índices* (al igual que los índices de árbol B^+).

Es fácil darse cuenta de que el coste de localizar una entrada en el índice almacenado en las páginas de un espacio de índices varía bastante según si esa entrada es excedente o no:

- Si una entrada no es excedente y, por lo tanto, se encuentra en su página primaria, solo habrá que hacer una E/S para obtenerla.
- Para las entradas excedentes, en cambio, siempre será necesario hacer más de una E/S.

En consecuencia, para conseguir un buen rendimiento del índice interesa que el índice tenga pocas entradas de excedentes. Una manera de reducir el número de entradas de excedentes es disponer de más espacio del que se requiere para las páginas primarias; es decir, conseguir que las páginas primarias estén poco cargadas de entradas.

Se denomina **factor de carga** al número de entradas que se espera tener dividido por el número de entradas que caben en las páginas primarias:

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

donde M representa el número de entradas que esperamos tener y $N \times L$ es el número de entradas que caben en las páginas primarias.

Cuanto más bajo sea el factor de carga, menos excedentes habrá y menos E/S serán necesarias. En contrapartida, si el factor de carga es muy bajo, se utiliza más espacio.

Ejemplo de cálculo de *L*

Si disponemos de páginas de 4 kB, los valores indexados ocupan 40 bytes, los RID, 4 y los apuntadores a páginas, 3. Entonces, el número L de entradas será 93, dado que se tiene que cumplir que 4 kB = $(40 + 4)L + 3$ (la página debe contener hasta L entradas formadas por el valor y un RID cada una, así como el apuntador a la primera página de excedentes).

Introducción a la dispersión dinámica

La dispersión que acabamos de explicar es una dispersión estática, en el sentido de que el número de páginas primarias N es fijo. Un problema que tiene este tipo de dispersión es que, si el número de datos indexados crece más de lo previsto, puede ocurrir que el factor de carga C aumente excesivamente y el rendimiento del índice empeore.

Una solución es crear un nuevo índice con un número N' de páginas primarias mayor que N, cambiar de función de dispersión para que devuelva valores de $[0, ..., N' - 1]$ en lugar de valores de $[0, ..., N - 1]$, y reinsertar todas las entradas en el nuevo índice empleando la nueva función de dispersión. Sin embargo, esta solución es muy costosa.

La dispersión dinámica tiene el objetivo de admitir el incremento dinámico del número de páginas primarias sin que sea necesaria la reinserción de todas las entradas.

Algunas técnicas de dispersión dinámica permiten modificar la función de dispersión de manera dinámica para acomodarse al aumento o la disminución de los datos indexados. En este módulo, sin embargo, no describimos las técnicas de dispersión dinámica, dos de las cuales, la dispersión extensible y la dispersión lineal, las podéis encontrar explicadas con todos los detalles en la obra de Ramakrishnan (2002).

Lectura complementaria

Encontraréis la explicación detallada de la dispersión extensible y la lineal en la obra siguiente:

R. Ramakrishnan; J. Gehrke (2002). *Database management systems*. Boston: McGraw-Hill.

6.3.5. Índices agrupados

Los índices que permiten implementar los accesos secuenciales por valor (como por ejemplo los árboles B^+) pueden ser índices agrupados[19] o índices no agrupados[20].

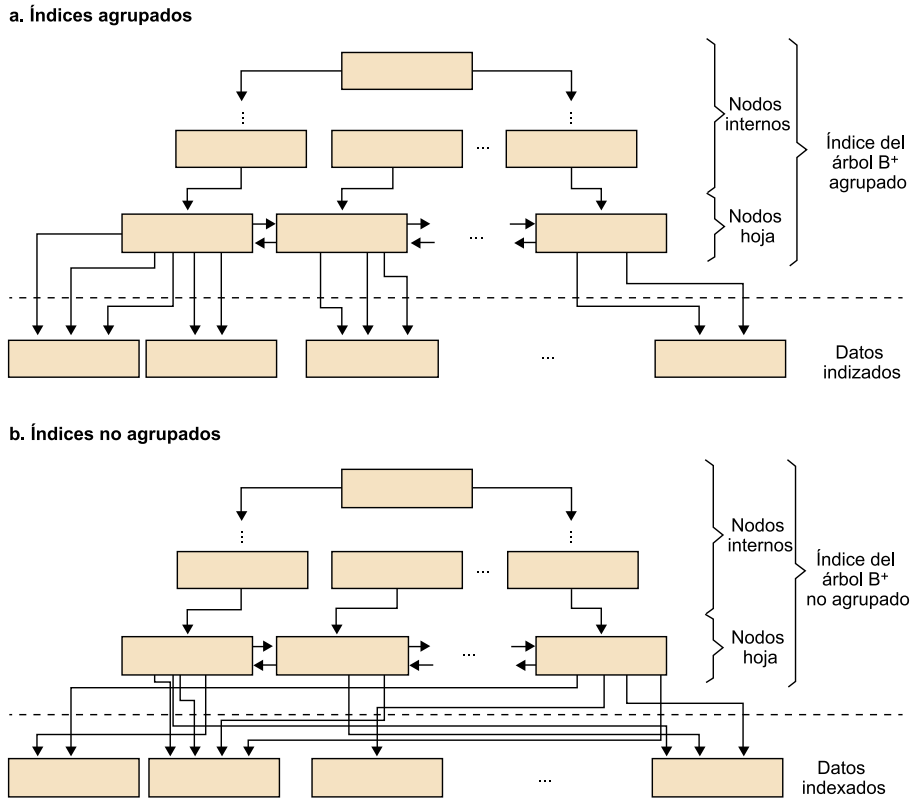
(19)En inglés, *clustered*.

(20)En inglés, *unclustered*.

Un **índice agrupado** es aquel en el que los datos que indexa están ordenados físicamente según el acceso secuencial por valor que proporciona el índice. En cambio, un **índice no agrupado** es un índice en el que los datos indexados están colocados de manera aleatoria.

La figura 43 ilustra las diferencias entre los índices agrupados y los no agrupados.

Figura 43. Diferencias entre índices agrupados y no agrupados



Reflexión

Aunque podemos tener varios índices sobre unos datos, solo uno puede ser agrupado, porque los datos pueden tener una única ordenación física.

El **coste de un acceso secuencial por valor** varía mucho según si el índice que nos proporciona el acceso es agrupado o no:

- Si el índice es agrupado, los RID que se obtengan consecutivamente apuntarán a filas contiguas. Entonces, será necesario hacer muchos accesos seguidos a filas de la misma página (y se podrán portar conjuntamente con una sola E/S).

- En índices no agrupados puede pasar que la mayoría de los RID consecutivos apunten a filas de páginas diferentes. Esta situación puede llegar a suponer tantas E/S como filas a las cuales haya que acceder.

Un aspecto que hay que tener en cuenta en los índices agrupados es que la ordenación física de sus datos es difícil de mantener cuando se producen inserciones y supresiones. En la práctica, esta dificultad se resuelve de la manera siguiente:

1) Inicialmente, se deja espacio libre en las páginas que contienen los datos para absorber inserciones futuras.

2) Si el espacio libre de una página se agota, las inserciones futuras a la página se realizan en páginas de excedentes que se le encadenan.

3) Si hay muchas páginas de excedentes, los datos se reorganizan para mejorar el rendimiento.

6.4. Implementación de los accesos por varios valores

La implementación de los accesos por varios valores es en algunos casos muy similar a la implementación de los accesos por un solo valor, que ya hemos estudiado en el subapartado anterior. En otros casos, no obstante, presentan algunas dificultades adicionales. Explicaremos en primer lugar la implementación de los accesos directos por varios valores y a continuación analizaremos los accesos secuenciales y mixtos por varios valores.

6.4.1. Implementación de los accesos directos

Considerad la relación *Employees*(*emplId*, *emplName*, *officeNum*, *salary*), que tiene un índice de árbol B^+ definido para facilitar los accesos por valor del atributo *officeNum* y otro índice de árbol B^+ para facilitar los accesos por valor del atributo *salary*.

Supongamos que se quiere ejecutar la sentencia siguiente, que requiere un acceso directo por varios valores, concretamente por los valores de *salary* y de *officeNum*:

```
SELECT *
FROM Employees
WHERE officeNum = 150 AND salary = 1200;
```

Una buena estrategia para procesar la consulta anterior es la que presentamos a continuación:

1) Usar el índice del atributo *officeNum* para obtener todos los RID de las filas de empleados que tiene el despacho 150.

2) Usar el índice del atributo *salary* para encontrar todos los RID de filas de empleados que tienen el sueldo 1.200.

3) Realizar la intersección entre los dos conjuntos de RID. Los RID de la intersección corresponden a los empleados que tienen al mismo tiempo el despacho 150 y un sueldo de 1.200.

Esta es una buena estrategia porque aprovecha el hecho de tener dos índices para los datos de los empleados. Aun así, en algunos casos concretos puede tener un mal rendimiento.

Por ejemplo, si hay muchos empleados en el despacho 150, muchos empleados con un sueldo de 1.200 y muy pocos empleados que cumplan las dos condiciones a la vez, habrá que examinar muchos RID para localizar pocos empleados. Una solución más eficiente para este caso es definir un único índice –en lugar de dos, como en el caso anterior– que utilice valores compuestos de los atributos *officeNum* y *salary*.

Un **índice de valores compuestos por los atributos** $[A_1, ..., A_i, ..., A_n]$ tiene la misma estructura que los otros índices. La única diferencia es que los valores del índice serán, de hecho, listas de elementos $[v_1, ..., v_i, ..., v_n]$ donde v_i es un valor del atributo A_i.

La gestión del índice implica necesariamente establecer una relación de orden entre los valores compuestos del índice. Se ordenan según el primer elemento de la lista y, si el primer elemento coincide, se ordenan de acuerdo con el segundo; si este también coincide, se ordenan según el tercero, etc.

Ejemplo de índice de valores compuestos

El índice de valores compuestos por los atributos [*officeNum*, *salary*] tiene valores como por ejemplo [150, 1.200], [150, 1.500], etc.

Para establecer una relación de orden entre los valores compuestos del índice hay que deducir para los dos valores anteriores [150, 1.200], [150, 1.500] cuál es el más grande. Aquí, puesto que el primer elemento de la lista coincide, se utiliza el segundo para ordenar. Se obtiene que [150, 1.200] < [150, 1.500].

En general, el orden entre dos valores compuestos $v = [v_1, ..., v_i, ..., v_n]$ y $w = [w_1, ..., w_i, ..., w_n]$ se establece de la manera siguiente:

- Si $v_1 > w_1$, entonces $v > w$; y si $v_1 < w_1$, entonces $v < w$.
- Si $v_1 = w_1$ y $v_2 > w_2$, entonces $v > w$; y si $v_1 = w_1$ y $v_2 < w_2$, entonces $v < w$.
- Si $v_1 = w_1, ..., v_{i-1} = w_{i-1}$ y $v_i > w_i$, entonces $v > w$; y si $v_1 = w_1, ..., v_{i-1} = w_{i-1}$ y $v_i < w_i$, entonces $v < w$.
- Si $v_1 = w_1, ..., v_i = w_i, ..., v_n = w_n$, entonces $v = w$.

6.4.2. Implementación de los accesos secuenciales y mixtos

Hemos visto que los índices de valores compuestos por los atributos $[A_1, ..., A_i, ..., A_n]$ constituyen una buena implementación de los accesos directos por varios valores. En este subapartado comentaremos su aplicación para implementar accesos secuenciales y mixtos por varios valores y veremos que presenta algunas deficiencias.

Considerad otra vez la relación *Employees*(*emplId*, *emplName*, *officeNum*, *salary*) y recordad que tiene un índice de árbol B^+ de valores compuestos por los atributos [*officeNum*, *salary*].

Supongamos que se quieren ejecutar las sentencias siguientes:

- Sentencia 1:

```
SELECT *
FROM Employees
ORDER BY officeNum, salary;
```

Esta sentencia corresponde a un acceso secuencial por varios valores. El orden de presentación de los empleados que requiere esta sentencia coincide con el orden de los valores compuestos del índice por [*officeNum*, *salary*]. Así pues, el índice nos permitirá procesar la sentencia de manera eficiente.

- Sentencia 2:

```
SELECT *
FROM Employees
WHERE officeNum = 100 AND salary > 960;
```

Esta sentencia corresponde a un acceso mixto por varios valores. Observad que el orden de los valores compuestos del índice por [*officeNum*, *salary*] concuerda con el orden en el que nos interesa obtener los empleados para procesar la consulta. De este modo, el índice nos permite también procesar la consulta de manera eficiente.

Desgraciadamente, no sucederá lo mismo con otros ejemplos de accesos secuenciales o mixtos por varios valores, porque no habrá concordancia entre el orden del índice por [*officeNum*, *salary*] y el orden que requiera la consulta. Este es el caso de las sentencias siguientes:

- Sentencia 3:

```
SELECT *
FROM Employees
ORDER BY salary, officeNum;
```

- Sentencia 4:

```
SELECT *
```

```
FROM Employees
WHERE salary = 1200 AND officeNum > 100;
```

Para las sentencias 3 y 4 necesitaríamos un índice [*salary*, *officeNum*], en lugar del índice [*officeNum*, *salary*].

Existen tipos de índices, denominados **índices multidimensionales por varios atributos [A_1, ..., A_i, ..., A_n]**, que no imponen un orden lineal en el conjunto de valores indexados. Éstos organizan los datos según un cierto vínculo espacial: cada valor se considera un punto de un espacio de dimensión *n*, donde *n* es el número de atributos del índice.

Utilidad de un índice multidimensional

Con un solo índice multidimensional por los atributos *salary* y *officeNum* podríamos procesar todas las sentencias de consulta 1, 2, 3 y 4 presentadas en este subapartado.

Se han propuesto varios tipos de índices multidimensionales: los árboles R, los archivos en retícula, etc. Se utilizan sobre todo en SGBD destinados a almacenar información geográfica, pero pocos sistemas relacionales los incorporan.

Lectura complementaria

Podéis encontrar la descripción de los árboles R y de los archivos en retícula en la obra siguiente:

A. Silberschatz; H. F. Korth; S. Sudarshan (2010). *Fundamentos de bases de datos* (3.ª ed.). Madrid: McGraw-Hill.

6.5. Índices del sistema Oracle

El sistema Oracle pone al alcance del diseñador algunos de los índices que hemos estudiado en este módulo.

Documentación Oracle

Para obtener los detalles de la implementación, así como una guía de la sintaxis y ejemplos de uso, se puede consultar la documentación en línea del SGBD Oracle.

6.5.1. Accesos por valor

Para implementar accesos por valor directos o secuenciales el sistema Oracle proporciona índices de árbol B^+ por un atributo, que pueden ser agrupados o no.

Por ejemplo, si tenemos una relación *Employees*(*emplId*, *emplName*, *officeNum*, *salary*), podemos declarar un índice para acceder por el valor del atributo denominado *officeNum* así:

```
CREATE INDEX indexOfficeNum ON Employees (officeNum);
```

Si se desea que el acceso secuencial sea por orden descendente, debe declararse de manera explícita (porque, por defecto, el orden se considera ascendente):

```
CREATE INDEX indexOfficeNum ON Employees (officeNum DESC);
```

Podemos conseguir que el índice sea agrupado si declaramos:

```
CREATE INDEX CLUSTER indexOfficeNum ON Employees (officeNum);
```

Finalmente, si queremos que el índice no admita valores repetidos, la sentencia será:

```
CREATE UNIQUE INDEX indexOfficeNum ON Employees (officeNum);
```

6.5.2. Accesos por varios valores

Para implementar accesos por varios valores, el sistema Oracle proporciona índices de árbol B^+ de valores compuestos, que también pueden ser agrupados o no.

Por ejemplo, si en la relación *Employees*(*emplId*, *emplName*, *officeNum*, *salary*) queremos declarar un índice de valores compuestos por los atributos [*officeNum*, *salary*], haremos:

```
CREATE INDEX indexOfficeSalary ON Employees (officeNum, salary);
```

Si queremos que el acceso secuencial por alguno de los atributos sea descendente, hay que declararlo de manera explícita. Por ejemplo, si deseamos que la ordenación de los sueldos sea descendente:

```
CREATE INDEX indexOfficeSalary ON Employees (officeNum, salary DESC);
```

También podemos conseguir que el índice sea agrupado:

```
CREATE INDEX CLUSTER indexOfficeSalary ON Employees (officeNum, salary);
```

Finalmente, si se desea que el índice no admita valores repetidos, habrá que declarar:

```
CREATE UNIQUE INDEX indexOfficeSalary ON Employees (officeNum, salary);
```

6.5.3. Consideraciones del planificador de ejecución para el uso de los índices

Los índices se pueden crear para cualquier columna de cualquier tabla, pero no siempre es beneficioso o eficiente, ya que es necesario mantenerlos actualizados y esto implica consumo de recursos del SGBD. El mismo Oracle crea índices automáticamente en determinados casos, como por ejemplo para las columnas que definen la clave primaria (con restricción *PRIMARY KEY*) o alternativa (con restricción *UNIQUE*).

Para construir consultas óptimas utilizando recursos mínimos, hay que seguir algunas buenas prácticas. Una de ellas es tener el índice adecuado para cada tabla, ya que la existencia de índices inapropiados o no utilizado causa ineficiencias en el momento de insertar, borrar o modificar datos en/de las tablas, especialmente si hay varios índices y las tablas son grandes. Esto se debe a que,

en el caso de inserción o borrado de datos en una tabla, el SGBD debe actualizar todos los índices existentes sobre las columnas de esta tabla, mientras que, en el caso de modificación de datos, el SGBD solo tiene que actualizar los índices de la columna o las columnas implicada/s.

Así, como regla general, la tablas sobre las que se prevé un elevado número de inserciones, borrados o modificaciones, se recomienda que tengan un número reducido de índices, mientras que las tablas sobre las que principalmente se realizan consultas (pocas actualizaciones) no hay problema si tienen más índices.

En el caso de Oracle, si buscamos el rendimiento óptimo del SGBD, debemos tener en cuenta las comprobaciones realizadas por el planificador de ejecuciones para determinar los índices que conviene crear:

- Si la consulta realizada filtrando los valores de la columna con índices, recupera menos del 15% (aproximadamente) de las filas de una tabla, se recomienda usar el índice.

- En las consultas sobre columnas con valores únicos o con pocos duplicados, se utilizará casi siempre el índice existente.

- Los valores de tipo *null* existentes en las columnas no se indexan, pero en el caso de consultas sobre el resto de valores de una columna, también será habitual usar el índice.

- En el caso de consultas sobre tablas pequeñas, que caben en un bloque, no se suele utilizar ningún índice, ya que se optará por llevar toda la tabla a memoria para hacer un recorrido sobre ella.

- El Oracle no crea índices automáticamente para las columnas que son clave externa a otras tablas, y como se utilizarán para realizar combinaciones, es aconsejable crear índices para estas columnas, ya que se utilizarán habitualmente.

Si nos encontramos con los casos descritos, debemos analizar la existencia de índice y, si fuese necesario, crearlo. Tenemos los diferentes tipos de índices para elegir:

- Simples: son los creados por omisión por el Oracle, para una sola columna y de tipo ascendente.

- Compuestos: son aquellos en los que interviene más de una columna de una tabla. Es importante situar como primera columna la que tenga más

valores diferentes. En determinados casos, también pueden ser usados por el SGBD al realizarse búsquedas para solo una de las columnas del índice.

- Ascendentes/descendentes: un índice ascendente (creado por omisión) no puede ser utilizado cuando se solicitan resultados ordenados de forma descendente. En estos casos, será necesario crear un índice específico de tipo *DESC*.

- Calculados: se crean utilizando funciones, expresiones aritméticas, llamadas a C, funciones SQL, entre otras posibilidades. A continuación, se proporcionan un par de ejemplos:

```
CREATE INDEX IDX_UPP_Nombre ON UPPER(nombre);
```

o

```
CREATE INDEX IDX_result ON (saldo/numMovimientos);
```

Veamos más recomendaciones para el uso de los índices.

En el caso de los índices calculados, es necesario asegurarse de que las consultas utilizarán la misma expresión que se utilizó al crear el índice. Esto garantiza que el Oracle encontrará un índice compatible para la consulta. Además, no se pueden utilizar índices calculados que no sean deterministas, es decir, que no devuelvan siempre el mismo valor.

Los índices deben ser deterministas, su valor puede no variar en función del momento en el que se calcule. Por ejemplo, si tuviéramos un campo llamado *fechaNacimiento*, no deberíamos crear un índice para el campo de *edad* que se calculara en función de la diferencia entre la edad actual y la fecha de nacimiento, ya que Oracle usaría la expresión del momento de su creación y al día siguiente ya dejaría de ser correcto.

Otro tema a considerar es si puede ser mejor utilizar un índice de tipo B+ o tipo *bitmap* (estos últimos no disponibles en el Oracle Express Edition). Podemos adoptar esta norma: si hay existen muchos valores diferentes en una columna, el tipo de índice más adecuado es el árbol B+, mientras que, si hay pocos valores diferentes en una columna, lo más recomendable es utilizar el índice de tipo *bitmap*.

El momento de crear los índices también es importante ya que, por ejemplo y de modo general, es mejor crear los índices de una tabla después de realizar los procesos de carga de datos (SQL Loader), modificaciones masivas o simi-

lares, cuando el contenido de los campos e incluso la cantidad de registros varíe sustancialmente, que tenerlos creados antes y provocar la actualización continuada durante estos procesos.

Finalmente, algunas restricciones a tener en cuenta al utilizar/crear índices:

- No se pueden usar índices sobre columnas de tipo *LONG* o *LONG RAW*.

- La medida de la entrada de un índice no puede exceder (aproximadamente) la mitad del espacio de un bloque, por lo que no se podrán crear índices sobre columnas con valores muy grandes.

En los planes de ejecución de las consultas es donde se muestra el uso de los índices. Por lo tanto, el modo más sencillo de comprobar cuál es el plan de ejecución previsto para una consulta en Oracle, por ejemplo, es presionando F10 en el SQL Developer. A continuación se muestra un plan de ejecución de consultas en el que se ve que el optimizador de Oracle, para tener acceso a los datos, utilizará el índice *IDX_cityName*.

Figura 44. Plan de ejecución que muestra la recomendación de usar el índice llamado *IDX_cityName* dada una consulta

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	2
TABLE ACCESS	CITY	BY INDEX ROWID	1	2
INDEX	IDX_CITYNAME	RANGE SCAN	1	1
Access Predicates				
CITYNAME='Brasilia'				

6.6. Claves sintéticas en el modelo físico

Al finalizar el proceso de diseño de una base de datos es necesario asegurarse de que el resultado obtenido es óptimo tanto en términos de estructura lógica (la definición de las columnas es coherente con el tipo de datos que almacenarán) como de recursos utilizados (mínimo de recursos utilizados). Además, también es importante asegurarse de que el diseño desarrollado será fácil de mantener. Estas comprobaciones implican hacer algunas reflexiones que pueden comportar realizar adaptaciones y refinamientos del modelo físico obtenido.

Uno de los temas que se suelen revisar son los valores que contienen las claves primarias como identificadores unívocos de los datos.

Por definición, una clave primaria debe contener un valor que sea único en todas las filas de la tabla, ya que tiene que servir para identificar de forma unívoca cada una de ellas e, idealmente, este valor debe ser constante a lo largo del tiempo. Sin embargo, nos encontramos con que en el mundo cotidiano pocos datos hay que sean identificadores, inamovibles y/o inmutables, como para ser considerados buenos candidatos a clave primaria.

6.6.1. Necesidad de uso de claves sintéticas

Hay casos en los que el uso de claves primarias puede implicar ineficiencias que se pueden resolver con claves sintéticas.

Supongamos que queremos identificar un vehículo:

- Si elegimos la matrícula para identificarlo, se puede dar el caso de que el vehículo se vuelva a matricular, por ejemplo, pasando de matrícula no europea a matrícula europea, o en el caso de una operación de compra/venta, cuando se cambia de país.

- Si escogemos el número de bastidor del vehículo, puede ocurrir que el vehículo sufra un accidente de importancia y se le tenga que cambiar el chasis (donde lleva estampado el número de bastidor).

- Podríamos pensar en usar la combinación de número de matrícula + número de bastidor...

O puede que nos interese identificar a una persona:

- Si tomamos el DNI como identificador, tenemos que los menores de edad sin DNI no se podrán identificar. También nos encontraríamos con que las personas de otro país no tienen DNI (tienen número de pasaporte) y que, en cualquier caso, una persona que es inmigrante y se nacionaliza puede tener primero NIE (número de identidad de extranjero) y posteriormente, al obtener la nacionalidad, le pueden cambiar este número por otro correspondiente al DNI.

- Se podría pensar en utilizar la combinación de nombre + apellidos + fecha de nacimiento. Pese a ello, en los casos de nombres muy comunes, puede ocurrir que estos valores coincidan con los de otras personas y, por lo tanto, que no sean únicos.

Problemas por homonimia

La coincidencia de nombres y apellidos se llama *homonimia*, y puede causar problemas muy importantes, ya sean legales, médicos...

En los casos de ejemplo, y si se produjeran las situaciones descritas, se deberían cambiar los valores elegidos como clave primaria, con la dificultad que esto puede conllevar en un sistema complejo, en el que puede haber docenas de tablas relacionadas. Cambiar el valor de una clave primaria para variar el identificador de una persona, por ejemplo, puede significar tener que actualizar decenas de tablas, ya que el valor modificado como clave primaria debe constar como clave foránea en las tablas relacionadas. Un modo de no encontrarse con esta problemática es utilizar lo que se denomina *clave sintética*.

Una clave primaria sintética no es más que un campo declarado como clave primaria, que contiene un valor no repetido que se genera de modo secuencial. Por ejemplo, si consideramos la tabla *Persona* y tenemos una clave sintética, esta tomará valores consecutivos desde '1', para la primera fila que se inserte con datos de persona, '2' para la segunda y así sucesivamente.

Normalmente, para crear una clave sintética se utiliza un disparador (*trigger*) y una secuencia definida en la base de datos. Con cada nueva inserción, el disparador recuperará de la secuencia un nuevo valor y lo usará para rellenar la columna, independientemente del posible valor inicial que se haya dado en la instrucción de inserción.

Sin embargo, una vez determinada la necesidad de utilizar un valor sintético como clave primaria, se debe elegir qué otro/s valor/es pueden servir para identificar unívocamente cada registro, ya que habitualmente no se harán las consultas por clave sintética, sino por alguna otra clave que aparentemente parezca más lógica, al menos por el propio nombre y significado.

Por ejemplo, volviendo al caso de la tabla *Persona* considerado anteriormente, si definiéramos una clave primaria sintética se debería definir también una clave alternativa con la cláusula *UNIQUE,* como por ejemplo el NIF. La primera clave permitiría relacionar las filas de esta tabla con las de otras tablas, y la segunda serviría para realizar búsquedas dentro de la tabla *Persona* y para evitar repeticiones de valores.

Las ventajas de usar esta técnica son diversas:

- Las filas de las tablas se identifican unívocamente desde su inserción hasta su borrado con un valor identificativo único e inmutable.

- Es posible disponer de una clave alternativa, que también permite identificar cada fila (al menos, evitar duplicidades) el valor de la cual es fácilmente actualizable, ya que solo consta en una tabla.

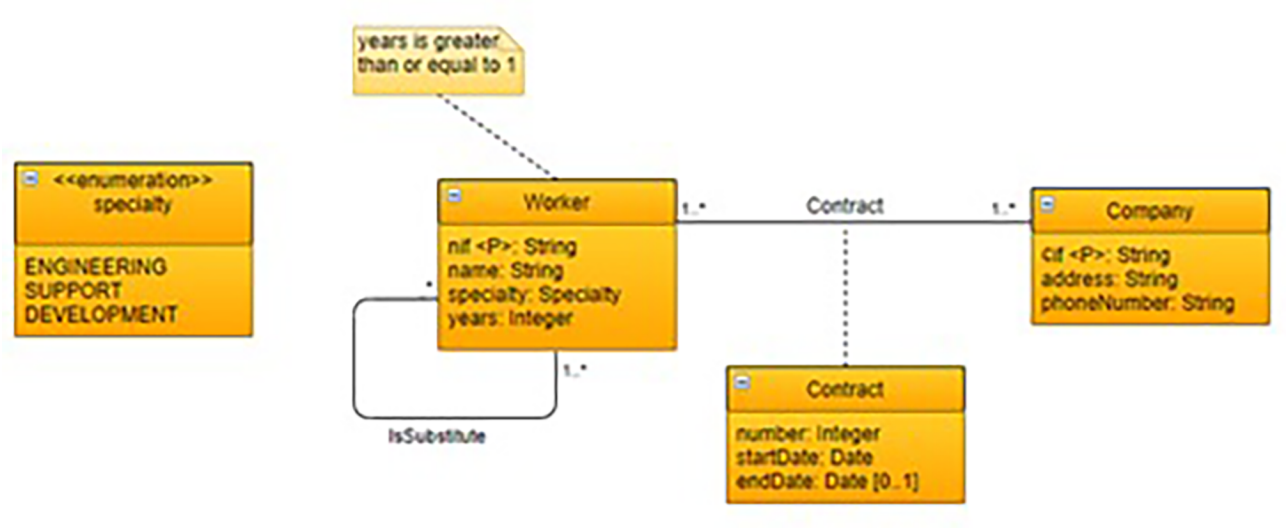
- La ocupación de espacio en los índices de la base de datos por el atributo que es clave primaria y que será el que más se usará al hacer consultas, será muy compacto, ya que con toda probabilidad será un campo de tipo *INTEGER*. Si usásemos los otros identificadores descritos, como el NIF, la matrícula del vehículo o la dirección, el número de bytes que ocuparía la clave primaria sería un orden de magnitud superior.

La determinación del uso de una clave primaria sintética se realiza en la última etapa de diseño, en el paso de transformación del diseño lógico relacional a físico, y de modo general, se utilizará siempre una clave primaria sintética para las tablas que identifican objetos del mundo físico, utilizándose como clave alternativa los atributos que en la etapa del diseño conceptual se habían detectado como claves primarias.

6.6.2. Caso práctico

Nos dicen que tenemos el siguiente modelo conceptual:

Figura 45. Ejemplo de modelo conceptual para modelar la contratación de trabajadores de una empresa



Y su transformación a modelo lógico relacional es:

Company (cif, **address, phoneNumber**)

Worker (nif, **name, specialty, years**)

Contract (cifCompany, nifWorker, **number, startDate,** endDate)
{cifCompany} is foreign key to Company
{nifWorker} is foreign key to Worker

IsSubstitute (nifWorker, nifSubstitute)
{nifWorker} is foreign key to Worker
{nifSubstitute} is foreign key to Worker

Siendo el código SQL para la transformación a físico en el SGBD Oracle, el siguiente:

```
CREATE TABLE Company (
  cif VARCHAR2(10 CHAR) CONSTRAINT PK_Company PRIMARY KEY,
  address VARCHAR2(100 CHAR) CONSTRAINT NN_Address NOT NULL,
  phoneNumber VARCHAR2(50 CHAR) CONSTRAINT NN_PhoneNumber NOT NULL
);
```

```
CREATE TABLE Worker (
  nif VARCHAR2(10 CHAR) CONSTRAINT PK_Worker PRIMARY KEY,
  name VARCHAR2(100 CHAR) CONSTRAINT NN_WorkerName NOT NULL,
```

```
  specialty VARCHAR2(11 CHAR) CONSTRAINT NN_WorkerSpecialty NOT NULL
  CONSTRAINT CH_WorkerSpecialty CHECK (specialty = 'ENGINEERING' OR
              specialty = 'SUPPORT' OR
              specialty = 'DEVELOPMENT'),
  years INTEGER CONSTRAINT NN_WorkerYears NOT NULL
     CONSTRAINT CH_WorkerYears CHECK (years >= 1)
);

CREATE TABLE Contract (
  cifCompany VARCHAR2(10 CHAR),
  nifWorker VARCHAR2(10 CHAR),
  "number" INTEGER CONSTRAINT NN_ContractNumber NOT NULL,
  startDate DATE CONSTRAINT NN_ContractStartDate NOT NULL,
  endDate DATE,
  CONSTRAINT PK_Contract PRIMARY KEY (cifCompany, nifWorker),
  CONSTRAINT FK_CompCont FOREIGN KEY (cifCompany) REFERENCES Company (cif),
  CONSTRAINT FK_WorkerCont FOREIGN KEY (nifWorker) REFERENCES Worker (nif),
  CONSTRAINT CH_ContractDates CHECK (endDate IS NULL OR endDate >= startDate)
);

CREATE TABLE IsSubstitute (
  nifWorker VARCHAR2(10 CHAR),
  nifSubstitute VARCHAR2(10 CHAR),
  CONSTRAINT PK_Substitutes PRIMARY KEY (nifWorker, nifSubstitute),
  CONSTRAINT FK_SubstWorker1 FOREIGN KEY (nifWorker) REFERENCES Worker (nif),
  CONSTRAINT FK_SubstWorker2 FOREIGN KEY (nifSubstitute) REFERENCES Worker (nif),
  CONSTRAINT CH_Substitutes CHECK (nifWorker <> nifSubstitute)
);
```

Se observa que tanto la tabla *Worker* como *Company* son susceptibles de usar una clave sintética. Nos centraremos en *Worker*.

Tal como se ha transformado el modelo lógico relacional en diseño físico, nos encontramos con que, si un trabajador varía su NIF, porque antes de la nacionalizarse tiene un NIE (número de identidad de extranjero) y luego un DNI, la modificación debería hacerse en las tablas *Worker*, *Contract* e *isSubstitute* por el hecho de estar relacionadas. Por lo tanto, se debe crear una transacción, con el trabajo que esto conlleva, para asegurar que la base de datos quede íntegra tras los cambios.

Si quisiéramos introducir una clave sintética en la tabla *Worker* en el momento de realizar la transformación del modelo lógico relacional a modelo físico, lo haríamos así:

```
CREATE TABLE Company (
  cif VARCHAR2(10 CHAR) CONSTRAINT PK_Company PRIMARY KEY,
  address VARCHAR2(100 CHAR) CONSTRAINT NN_Address NOT NULL,
```

```
  phoneNumber VARCHAR2(50 CHAR) CONSTRAINT NN_PhoneNumber NOT NULL
);

CREATE TABLE Worker (
  idWorker INTEGER CONSTRAINT PK_Worker PRIMARY KEY,
  nif VARCHAR2(10 CHAR) CONSTRAINT AK_Worker UNIQUE,
  name VARCHAR2(100 CHAR) CONSTRAINT NN_WorkerName NOT NULL,
  specialty VARCHAR2(11 CHAR) CONSTRAINT NN_WorkerSpecialty NOT NULL
CONSTRAINT CH_WorkerSpecialty CHECK (specialty = 'ENGINEERING' OR
           specialty = 'SUPPORT' OR
           specialty = 'DEVELOPMENT'),
  years INTEGER CONSTRAINT NN_WorkerYears NOT NULL
     CONSTRAINT CH_WorkerYears CHECK (years >= 1)
);
```

```
CREATE TABLE Contract (
  cifCompany VARCHAR2(10 CHAR),
  idWorker INTEGER,
  "number" INTEGER CONSTRAINT NN_ContractNumber NOT NULL,
  startDate DATE CONSTRAINT NN_ContractStartDate NOT NULL,
  endDate DATE,
  CONSTRAINT PK_Contract PRIMARY KEY (cifCompany, idWorker),
  CONSTRAINT FK_CompCont FOREIGN KEY (cifCompany) REFERENCES Company (cif),
  CONSTRAINT FK_WorkerCont FOREIGN KEY (idWorker) REFERENCES Worker (idWorker),
  CONSTRAINT CH_ContractDates CHECK (endDate IS NULL OR endDate >= startDate)
);

CREATE TABLE IsSubstitute (
  idWorker INTEGER,
  idSubstitute INTEGER,
  CONSTRAINT PK_Substitutes PRIMARY KEY (idWorker, idSubstitute),
  CONSTRAINT FK_SubstWorker1 FOREIGN KEY (idWorker) REFERENCES Worker (idWorker),
  CONSTRAINT FK_SubstWorker2 FOREIGN KEY (idSubstitute) REFERENCES Worker (idWorker),
  CONSTRAINT CH_Substitutes CHECK (idWorker <> idSubstitute)
);

-- Creación de la secuencia para obtener el valor en curso del trabajador a insertar
CREATE SEQUENCE s_Worker
  INCREMENT BY 1
  START WITH 1;

-- Creación del disparador que insertará automáticamente el valor a idWorker
CREATE OR REPLACE TRIGGER t_sintetic_idWorker
  BEFORE INSERT ON Worker
  FOR EACH ROW
BEGIN
  SELECT s_Worker.NEXTVAL INTO :NEW.idWorker
```

```
    FROM DUAL;
END t_sintetic_idWorker;
```

Después, comprobaríamos que:

- Las consultas que involucren diferentes tablas serían más eficientes, ya que los índices de las claves primarias ocuparían menos espacio.

- Si se quiere variar el NIF de *Worker* se podría hacer con una única operación de modificación (*UPDATE*), de modo muy rápido y sencillo.

- Es posible realizar búsquedas de personas por NIF ya que este campo es clave alternativa.

Debe tenerse en cuenta que al realizar inserciones en la tabla *Worker* no tendremos que inicializar el valor del campo *idWorker,* ya que el disparador generará automáticamente el valor para informarlo de acuerdo con una secuencia dada.

Imaginemos que queremos insertar los datos de un trabajador con esta sentencia SQL:

```
INSERT INTO Worker (nif, name, specialty, years, cifCompany)
  VALUES ('39893123U', 'John Smith', 'ENGINEERING', 7, ' G12343212'');
```

Observad que no se da ningún valor explícito al *idWorker*. Este se inicializará en el momento de la inserción a través del disparador correspondiente

Resumen

En la primera parte de este módulo didáctico, hemos presentado los componentes que usan los SGBD en el almacenamiento de los datos que gestionan. Hemos visto el nivel físico de la arquitectura de una base de datos, en el que los SGBD utilizan ficheros, y el nivel virtual, que es un nivel propio de los SGBD que permite simplificar la visión que tienen de los datos almacenados.

En la segunda parte de este módulo, hemos tratado la manera de implementar el diseño físico de la base de datos.

Finalmente, en la tercera y última parte de este módulo didáctico hemos identificado métodos de acceso que son necesarios para llevar a cabo diferentes tipos de consultas y actualizaciones en las BD. Estos métodos de acceso son los accesos directos y secuenciales por posición, los accesos directos y secuenciales por valor y, finalmente, los accesos por varios valores, que pueden ser directos, secuenciales o mixtos.

Hemos explicado que la implementación de los accesos por posición se fundamenta casi completamente en los servicios proporcionados por el SO y que, en cambio, la implementación eficiente de los accesos por uno o varios valores es más compleja y requiere que el SGBD disponga de estructuras propias especializadas.

Hemos descrito algunas de las estructuras que los SGBD utilizan para implementar los accesos por uno o varios valores: índices del tipo árbol B^+, índices estructurados según funciones de dispersión, índices agrupados e índices de valores compuestos. Hemos mostrado el impacto que pueden tener estas estructuras en el número de E/S necesario para implementar los accesos a los datos y hemos visto cómo el optimizador del SGBD puede ayudar a construir consultas que optimicen los recursos.

Glosario

acceso directo por posición *m* Método de acceso que consiste en obtener una página que tiene un número de página determinado dentro de un espacio.

acceso directo por valor *m* Método de acceso que consiste en obtener todas las filas que contienen un determinado valor por un atributo.

acceso por varios valores *m* Método de acceso que consiste en obtener varias filas según los valores de varios atributos. Puede ser directo, secuencial o mixto.

acceso secuencial por posición *m* Método de acceso que consiste en ir obteniendo las páginas de un espacio siguiendo el orden definido por sus números de página.

acceso secuencial por valor *m* Método de acceso que consiste en obtener varias filas por orden de los valores de un atributo.

árbol B^+ *m* Estructura de datos que se emplea para organizar índices que permiten implementar el acceso directo y el acceso secuencial por valor.

arquitectura de componentes de almacenamiento *f* Esquema de tres niveles (lógico, físico y virtual) con el cual clasificamos y describimos cada componente de los SGBD, especialmente aquellos que están relacionados con el almacenamiento de los datos.

bloque *m* Unidad de transferencia de datos entre la memoria del ordenador y los dispositivos o ficheros externos. El sistema operativo de la máquina, concretamente la parte especializada en la E/S, es quien lleva a cabo esta transferencia.

catálogo *m* Conjunto de tablas que contienen los metadatos de la base de datos.

dispersión *f* Manera de organizar valores que se puede utilizar en índices que implementan el acceso directo por valor.

entrada *f* Elemento de un índice que consiste en una pareja formada por un valor y un RID.

espacio virtual *m* Secuencia de páginas virtuales. Proporciona una visión ordenada y contigua de las páginas físicas. Según su funcionalidad específica tiene características ligeramente diferentes, en función de las cuales puede ser un espacio para tablas, fragmentado, de agrupación, de objetos grandes, de índices, temporal, etc.
sigla **EV**

EV *m* Podéis ver **espacio virtual**

extensión *f* Unidad de asignación de espacio en un dispositivo periférico. Cada extensión es un número entero de páginas consecutivas y está contenida dentro de un fichero. Normalmente existe una extensión primaria, que se adquiere la primera vez que el fichero se extiende, y una extensión secundaria, que corresponde a las extensiones siguientes.

fichero *m* Unidad de gestión del espacio en los dispositivos periféricos. Normalmente, el sistema operativo gestiona los ficheros en lugar del SGBD.

identificador de fila *m* Dirección uniforme que utilizan los SGBD para grabar las referencias internas de sus estructuras de datos. Otros nombres equivalentes son ROWID y, últimamente, OID (en la medida en que los SGBD relacionales se convierten en los denominados *object-relational*).
sigla **RID**

índice *m* Estructura de datos auxiliar que los SGBD usan para facilitar las búsquedas necesarias para implementar los accesos por uno o varios valores.

índice agrupado *m* Índice que proporciona el acceso secuencial por valor (y también el acceso directo por valor) y que indexa datos que están ordenados físicamente según el orden del acceso secuencial por valor proporcionado.

índice de valores compuestos *m* Índice de valores compuestos por los atributos $[A_1, ..., A_i, ..., A_n]$. Tiene la misma estructura que los otros índices, pero con la diferencia de que los valores del índice son, de hecho, listas de valores $[v_1, ..., v_i, ..., v_n]$ en las que cada v_i es un valor del atributo A_i.

memoria intermedia *f* Espacio de la memoria principal del ordenador, normalmente bastante grande, dedicado a contener todas las unidades de memoria intermedia que gestiona el SGBD.

método de acceso *m* Tipo de acceso a los datos almacenados por un SGBD.

nivel físico *m* Nivel que engloba los componentes físicos (fichero, extensión y página) dentro de la arquitectura de componentes de almacenamiento.

nivel lógico *m* Nivel que engloba los componentes lógicos (base de datos, tablas, vistas, restricciones, etc.) dentro de la arquitectura de componentes de almacenamiento.

nivel virtual *m* Nivel que engloba el componente virtual (el espacio virtual) dentro de la arquitectura de componentes de almacenamiento.

página *f* Unidad de transferencia de datos entre la memoria del ordenador y los ficheros de una BD. El SO de la máquina es lo que lleva a cabo esta transferencia y pasa la página al SGBD para que este le gestione la información que contiene, puesto que el SGBD es lo que entiende la estructura interior de la página. La página también es la unidad principal de grabación de los datos de una base de datos en los dispositivos periféricos. En este módulo, a veces la denominamos *página física* o *página real* para distinguirla de la página virtual.

página virtual *f* Imagen de la página real o visión que desde el nivel virtual se tiene de la página real sin materializarla físicamente. Hay una relación biunívoca entre página real y página virtual.

RID *m* Podéis ver **identificador de fila**

SGBD *m* Sigla de **sistema de gestión de bases de datos**

SO *m* Sigla de **sistema operativo**

vector de direcciones de fila *m* Estructura que ocupa las posiciones más altas dentro de una página. Es un vector con tantos elementos como filas hay en la página. Cada elemento apunta a una de estas filas. El último elemento apunta a la primera fila, el penúltimo a la segunda, y así sucesivamente.
sigla **VDF**

VDF *m* Podéis ver **vector de direcciones de fila**

Bibliografía

Elmasri, Ramez; Navathe, Shamkant, B. (2007). *Fundamentos de sistemas de bases de datos* (5.ª ed.). Madrid: Pearson Educación.

Hansen, G. W.; Hansen, J. V. (1997). *Diseño y administración de bases de datos* (2.ª ed.). Prentice Hall.

Ramakrishnan, Raghu; Gehrke, Johannes (2003). *Database management systems* (3.ª ed.). Boston: McGraw-Hill Higher Education.

Silberschatz, A.; Korth, H. F.; Sudarshan, S. (2006). *Fundamentos de bases de datos* (5.ª ed.). Madrid: McGraw-Hill. Edición de 2011 en eBook.

Teorey, T. J. (2011). *Database Modeling & Design* (5.ª ed.). San Francisco: Morgan Kaufmann Publishers, Inc.