Ficheros

José Ramón Paramá Gabía

Capítulo 1

Ficheros

Las bases de datos se almacenan físicamente como ficheros (o archivos) de registros, los cuales se almacenan normalmente en discos magnéticos. Este capítulo trata sobre la organización de las bases de datos en el espacio de almacenamiento y las técnicas para acceder a ellas de forma eficiente utilizando varios algoritmos, algunos de los cuales requieren estructuras auxiliares de datos llamadas índices.

1.1. Conceptos básicos de ficheros

1.1.1. Registros y tipos de registros

Los archivos se organizan lógicamente como secuencias de registros. Cada registro consta de una colección de *valores* o *elementos* de datos relacionados, donde cada valor está formado de uno o más bytes y corresponde a un determinado *campo* del registro. En general, los registros describen entidades y sus atributos. Por ejemplo, un registro de ALUMNO representa una entidad alumno, y cada valor de campo del registro especifica algún atributo de ese alumno, como NOMBRE, FECHA-NACIMIENTO, CURSO o TITULACIÓN.

Definición 1.1.1 Una colección de nombres de campos y sus tipos de datos correspondientes constituye una definición de *tipo de registro*.

El tipo de datos de cada campo, especifica el tipo de valores que el campo puede tomar. El tipo de datos de un campo es casi siempre uno de los tipos de datos estándar empleados en programación. Por ejemplo podríamos definir el registro ALUMNO, utilizando la sintaxis C, como sigue:

```
struct alumno{
       char nombre[30];
       date fecha-nacimiento;
       int curso;
       char titulación[50];
};
```

En las aplicaciones recientes de bases de datos, puede haber necesidad de almacenar elementos de datos que consisten en objetos grandes no estructurados, que representan

imágenes, vídeo, audio o ficheros de cualquier tipo. Estos se denominan *BLOB* (*binary large objects*). Incluso las más modernas reconocen tipos de datos especiales de datos como áreas, polígonos, texto en distintos formatos (word, wordperfect, ...), etc.

Ficheros, registros de longitud fija y registros de longitud variable

Un *fichero* es una **secuencia** de registros. En la mayoría de los casos todos los registros son de un mismo tipo. Si todos los registros son del mismo tamaño, se dice que el fichero se compone de *registros de longitud fija*. Si diferentes registros del fichero tienen tamaños distintos, se dice que el fichero está constituido por *registros de longitud variable*.

Existen varias posibilidades como se puede observar en la Figura 1.1:

- Los registros del fichero son todos del mismo tipo, pero uno o más campos son de tamaño variable (*campos de longitud variable*).
- Los registros del fichero son todos del mismo tipo, pero uno o más campos pueden tener múltiples valores para un campo. Se dice que este campo es un *campo repetitivo*, y el grupo de valores se denomina *grupo repetitivo*.
- Los registros del fichero son de distintos tipos. Por ejemplo, si nuestro fichero de empleados tiene dos tipos de empleados, los que cobran a fin de mes (que tienen un campo con su salario), y los que cobran por horas. Estos últimos necesitan además de un campo para indicar lo que se les paga por hora (campo más pequeño que el de un salario mensual) otro campo para indicar las horas trabajadas el último mes.

Nombre	Dirección	Dept	Puesto trabajo	Salario
30 bytes	50 bytes	10 bytes	20 bytes	18 bytes

Registro de empleado de tamaño fijo

Nombre	Dirección	Dept	Puesto trabajo	Salario Mes	
Nombre	Dirección	Dept	Puesto trabajo	Salario Hora	Horas trabj

Registro de longitud variable de distinto formato.

Nombre	Dirección	Dept	Puesto trabajo	Salario
Nombre	Dirección	Dept	Puesto trabajo	Salario

Registro de longitud variable con campos de longitud variable.

Nombre	Dirección	Dept	Puesto trabajo	Nombre Hijo 1	
Nombre	Dirección	Dept	Puesto trabajo	Nombre Hijo 1	Nombre Hijo 2

Registro de longitud variable con grupos repetitivos.

Figura 1.1: Registros de longitud variable

En general, los registros de longitud variable son menos comunes que los de longitud fija por las dificultades de manejo. En primer lugar, debemos saber la longitud del registro. Esta información puede ser proporcionada por el sistema de ficheros como un parámetro separado de los datos, o puede ser incluido como un campo más del registro.

Cuando los campos son de longitud variable, es necesario obtenerla. Una vez más puede haber un campo en el registro que lo indique, o se puede utilizar un carácter especial (como # o $) que actúe como *fin de campo*.

Cuando el formato del registro cambia, se puede utilizar el valor de un campo para identificar el tipo de registro.

1.1.2. Claves

Definición 1.1.2 Una *clave* (en el entorno de ficheros) es un campo o conjunto de campos usados para identificar u ordenar los registros de un fichero.

En algunos textos, denominan a este campo o campos *campo/s de ordenamiento*. Una clave *externa*[1] es aquella que no es compuesta por uno o más campos de los registros que forman el fichero, sino que se deriva de alguna característica física del registro almacenado, por ejemplo, su posición dentro del fichero. Así, por ejemplo, el primer registro del fichero tiene como clave externa el 0, el segundo registro el 1, y así sucesivamente. Este tipo concreto de clave externa se conoce comúnmente como *número de registro*.

Definición 1.1.3 Una *clave primaria* es aquella clave tal que no existen dos registros que puedan tener el mismo valor de clave primaria.

Estas definiciones de clave las utilizaremos en el entorno de ficheros, veremos más adelante como estos mismos términos tienen un significado ligeramente distinto en el entorno de las bases de datos.

1.2. Restricciones del medio físico

1.2.1. La jerarquía de memoria

La memoria en una computadora actual se estructura en una jerarquía como se muestra en la Figura 1.2. En la parte superior, tenemos el *almacenamiento primario*, que está formado por la cache y la memoria principal, y proporciona acceso muy rápido a los datos. A continuación, el almacenamiento secundario, que está formado por dispositivos más lentos como memorias flash o discos magnéticos. El almacenamiento terciario es la clase más lenta de almacenamiento; por ejemplo, discos ópticos y cintas.

Actualmente, el costo de una determinada cantidad de memoria principal es sobre 100 veces más caro que la misma cantidad de almacenamiento en un disco, y las cintas aún son más baratas. El almacenamiento terciario, es decir, discos ópticos y cintas, juega un papel muy importante en los Sistemas de Gestión de Bases de Datos porque la cantidad de datos es normalmente muy grande. Dado que comprar suficiente memoria principal para almacenar toda la información es extremadamente caro, debemos almacenar información en cintas y discos y, construir sistemas de bases de datos que puedan recuperar los datos de los niveles

[1] No confundir con la clave externa del modelo relacional.

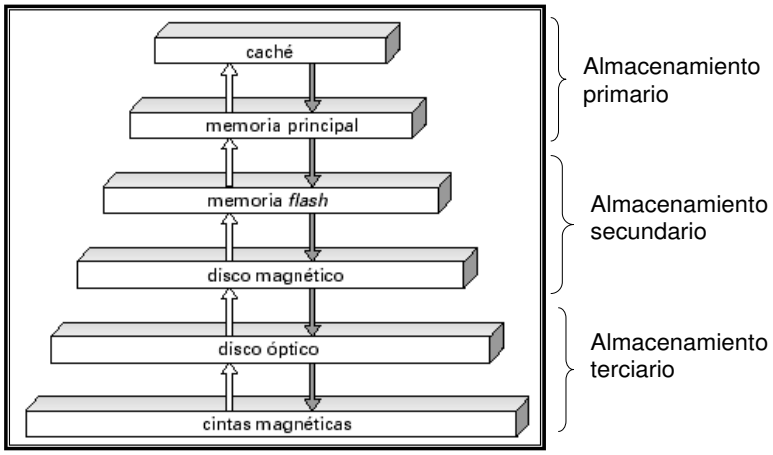

Figura 1.2: Jerarquía de memoria

bajos de la jerarquía de memoria hacia la memoria principal, según se va necesitando para ser procesada.

Hay otras razones además del costo para almacenar información en el almacenamiento secundario y terciario. Sobre sistemas de 32 bits, sólo se pueden direccionar 2^{32} posiciones de memoria principal, sin embargo, el número de objetos almacenados en una base de datos puede exceder con mucho esta cantidad. Además, la información debe permanecer (persistir) entre ejecuciones de programas. Esto requiere dispositivos de almacenamiento que mantengan la información después de un reinicio (debido a un apagado normal o un fallo); estos dispositivos se suelen denominar *no volátiles*. El almacenamiento primario es volátil, mientras que el secundario y terciario es no volátil.

Las cintas y discos ópticos, son baratos y pueden almacenar gran cantidad de información. Son una buena elección para el almacenamiento de información que debe ser mantenida por mucho tiempo, pero no es previsible que se consulte muy frecuentemente. Esto es debido a los tiempos de acceso demasiado lentos, así para el almacenamiento no volátil de la información *operacional* (que se accede frecuentemente) en los sistemas de bases de datos se suele utilizar los discos magnéticos.

Algunas de las más recientes tecnologías (como los discos ópticos, DVD, y jukeboxes de cintas) es probable que ofrezcan alternativas viables al uso de discos magnéticos. Pero por ahora es importante estudiar las características principales de los discos magnéticos que son los que más comúnmente almacenan persistentemente las bases de datos.

Las técnicas usadas para almacenar gran cantidad de datos estructurados en disco son importantes para los diseñadores de bases de datos, para el administrador de la base de datos (ABD) y para quienes implementan los Sistemas de Gestión de Bases de Datos (SGBD). Los diseñadores de bases de datos y el ADB deben conocer las ventajas e inconvenientes de cada una de las distintas alternativas de cada técnica de almacenamiento para diseñar, implementar y operar con la base de datos en un SGBD específico. Por lo general, el SGBD ofrece varias opciones para organizar los datos, y el proceso de *diseño físico de la base de datos* implica elegir las técnicas de organización de datos idóneas para los requisitos de la aplicación que se está tratando.

Antes de conocer dichas técnicas de almacenamiento debemos conocer las restricciones

del medio físico sobre las que se implementarán. Nos centraremos en los discos magnéticos que son, como hemos mencionado anteriormente, los más utilizados para almacenar bases de datos.

1.2.2. Discos magnéticos

La unidad básica de almacenamiento de un disco es el bit, si magnetizamos un área del disco de cierta manera, podemos entender que se almacena un 1, y si se magnetiza en con el polo opuesto, un 0.

Todos los discos están hechos de un material magnético en forma de disco circular delgado. A fin de aumentar la capacidad de almacenamiento los *discos duros* realmente son un paquete de discos o platos (ver Figura 1.3).

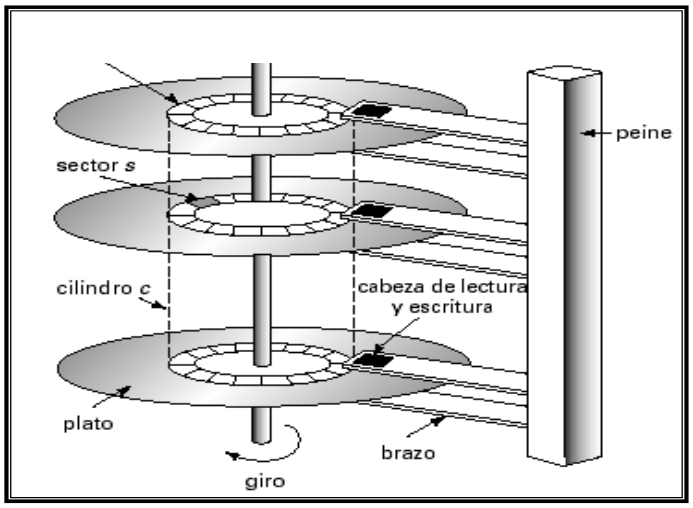

Figura 1.3: Paquete de discos

La información se almacena en la superficie del disco en círculos, generalmente, concéntricos de pequeña anchura, cada uno de los cuales tiene diferente diámetro. Cada círculo se denomina *pista*. Todas las pistas del mismo diámetro forman un cilindro. El concepto de cilindro tiene mucha importancia porque los datos almacenados en el mismo cilindro se pueden leer con mucha mayor rapidez como veremos más adelante.

Dado que una pista puede contener gran cantidad de información, se le divide en bloques más pequeños denominados *sectores*. La división de una pista en sectores está codificada permanentemente en la superficie del disco y no se puede modificar.

La división de una pista en *bloques de disco* o *bloques físicos* de igual tamaño la establece el sistema operativo durante el formateo del disco. Un bloque físico es siempre una secuencia continua de sectores de una sola pista de un plato. Los datos se transfieren entre el disco y la memoria principal en unidades de bloques físicos. Los niveles inferiores del sistema de archivos del SO traduce la dirección de un bloque en en una combinación del número de superficie, número de pista (dentro de la superficie) y el número de bloque (dentro de la pista). Es importante seleccionar el número de sectores por bloque con mucho cuidado como veremos más adelante.

Para realizar una operación de Lectura/Escritura, además de la dirección del bloque a leer/escribir, también se proporciona la dirección de un *buffer*, un área contigua reservada en la memoria principal en la que cabe un bloque. En el caso de una operación de lectura, el bloque de disco se copia en el buffer; si la operación es de escritura, el bloque que está en el buffer se copia en el disco.

La lectura y escritura de los sectores es realizada por cabezas. Normalmente hay una cabeza por cada superficie de disco (como se puede observar en la Figura 1.3). Las cabezas no tocan la superficie del disco, sino que vuelan sobre una brisa que se genera entre el disco y la cabeza. Si las cabezas llegan a tocar la superficie, se desprenderá la superficie magnética del plato que pasará a estar libre dentro del disco. Lo más normal es que este trozo haga chocar a otras cabezas con sus respectivas superficies arruinando por completo el paquete de discos.

Cuando se requiere leer un sector determinado, todas las cabezas deben moverse conjuntamente hasta colocarse en el cilindro donde se encuentra el sector en cuestión, las cabezas están alineadas verticalmente, es decir, todas las cabezas están alineadas en el mismo cilindro. Existen unos discos más caros denominados discos de cabezas fijas, que tiene una cabeza por pista.

Una vez que la cabeza está sobre la pista adecuada, el plato gira hasta que el sector pase por debajo de la cabeza que realiza la lectura. Los discos actuales, generalmente sólo permiten que una de las cabezas lea/escriba en un momento dado. Puede parecer que es una limitación absurda, si todas pudieran leer/escribir al mismo tiempo sería un gran beneficio, pero la razón es que sería muy difícil asegurar que todas las cabezas están perfectamente alineadas en las pistas correspondientes. Aunque existe alguna aproximación comercial, son caras y más propensas a fallos.

Como se puede apreciar, en el proceso intervienen dispositivos mecánicos que son siempre mucho más lentos que cualquier dispositivo electrónico. Esto va penalizar de modo severo los tiempos de acceso a disco, siendo mucho más lento (del orden de milisegundos, por lo regular entre 12 y 60 mseg) que el acceso a memoria principal (del orden de nanosegundos). Por lo tanto, la localización de los datos en el disco es un cuello de botella importante en las aplicaciones de bases de datos. De este modo las estructuras de fichero siempre van intentar minimizar las transferencias de bloques entre disco y memoria principal.

Rendimiento en discos magnéticos

El tiempo necesario para acceder a un bloque físico tiene varios componentes. *Tiempo de búsqueda* es el tiempo necesario para mover las cabezas hasta la pista donde está el bloque deseado. El *retardo rotacional* es el tiempo necesario para que el disco rote hasta que el sector deseado esté debajo de la cabeza lectora, de media es la rotación de la mitad del disco y es mucho menos importante que el tiempo de búsqueda. El *tiempo de transferencia* es tiempo para leer/escribir los datos el el bloque físico una vez la cabeza está posicionada, esto es, el tiempo para que el disco rote sobre el bloque físico.

$$tiempo\ de\ acceso = tiempo\ de\ busqueda + retardo\ rotacional + tiempo\ de\ transferencia$$

Esto implica que el tiempo que requieren las operaciones con la base de datos (o fichero) está afectado significativamente por cómo está almacenada la información en el disco. El tiempo de transferencia desde/hacia disco domina el tiempo que necesitan las operaciones en bases de datos (o ficheros). Para minimizar este tiempo, es necesario colocar los datos

en disco estratégicamente. Básicamente, si dos registros van ser frecuentemente utilizados conjuntamente, debemos colocarlos cerca uno del otro. Por "cerca", hablando de discos, nos referimos a que deberían estar en el mismo bloque físico. Decreciendo en orden de cercanía, deberían estar en la misma pista, mismo cilindro o en un cilindro adyacente. La idea es que el tiempo de acceso al bloque del segundo registro (una vez leído el primero) sea el menor posible.

Dispositivos de acceso aleatorio

Normalmente se denominan a los discos dispositivos de acceso aleatorio, a diferencia de las cintas que se denominan dispositivos de acceso secuencial. Sin embargo, un auténtico dispositivo de acceso aleatorio es aquel en el que todas las direcciones son igualmente accesibles. Esto no es cierto en los discos (a no ser que sean de cabezas fijas, y aún en ese caso, no son exactamente dispositivos de acceso aleatorio). Los sectores que son más accesibles son aquellos que pasarán próximamente por debajo de las cabezas lectoras. Aquellos sectores que están en cilindros distantes, requerirán más tiempo debido al tiempo necesario para mover el brazo que contiene a las cabezas. Por lo tanto es más apropiado referirse a los discos como dispositivos de acceso pseudoaleatorio.

Acceso paralelo a disco mediante RAID

Debido al aumento espectacular en el rendimiento y capacidad de las memorias y procesadores, las necesidades de disponer de almacenamiento secundario con más capacidad, fiabilidad y velocidad han crecido. A pesar de las mejoras en las tecnologías de discos, el avance en la tecnología de procesadores y memorias ha sido mucho mayor. Así, la respuesta de la tecnología de almacenamiento secundario está representado por los RAID (Redundant Array of Inexpensive Disks).

El RAID fue propuesto originalmente por Patterson, Gibson y Katz [PGK88]. El principal objetivo de los RAID es igualar los niveles de mejora en el rendimiento entre procesadores (mucho mayor) y discos. Las mejoras de capacidad de los discos sí son buenas, pero los tiempos de acceso no mejoran en la misma medida.

La solución es un array de pequeños discos independientes actuando como un único disco lógico de alto rendimiento. Se utiliza un concepto llamado *data striping (fraccionamiento de datos)* que utiliza el paralelismo para mejorar el funcionamiento del disco. El fraccionamiento de datos distribuye los datos de forma transparente sobre varios discos haciendo que aparezca como un único y gran disco rápido. El fraccionamiento mejora el funcionamiento de E/S permitiendo que varias operaciones de E/S se puedan atender en paralelo, permitiendo así altos porcentajes de transferencia.

Hay dos posibilidades de fraccionamiento. En primer lugar, el *fraccionamiento a nivel de bit* consiste en dividir un byte de datos y escribir el bit *j-ésimo* en el disco *j-ésimo*, es decir, necesitamos 8 discos. Cada disco participa en cada petición de E/S. El fraccionamiento a nivel de bit se puede generalizar a un número de discos múltiplo de 8 o factor de 8. Así, en un array de 4 discos, el bit n va al disco que es (n mod 4).

En segundo lugar, la otra posibilidad es el *fraccionamiento en el nivel de bloque*. Los distintos bloques de un fichero se reparten por los discos que conforman el RAID.

Por otra parte, la fiabilidad puede ser mejorada almacenando información redundante sobre discos. La desventaja de la redundancia es clara; más operaciones de E/S, más necesidad

de cómputo para mantener la redundancia y más capacidad de disco para almacenar la información redundante.

Una técnica para introducir la redundancia es la denominada *reflejo o sombra*. Los datos se escriben redundantemente en dos discos físicos idénticos que son tratados como un único disco lógico. Otra solución consiste en almacenar información extra que normalmente no se utiliza pero que puede ser usada para reconstruir la información perdida en caso de fallo de disco. Si se incorpora la redundancia se deben considerar dos problemas: (1) la selección de una técnica que calcule la información redundante, y (2) la selección de un método de distribución de la información redundante a través de del array de discos. El primer problema se soluciona usando códigos Hamming. Bajo este esquema de paridad, se considera que el disco redundante contiene la suma de todos los otros discos. Cuando un disco falla, se puede reconstruir la información perdida mediante un proceso similar a la resta.

Para el segundo problema, las dos mejores soluciones son o almacenar la información redundante en pocos discos o distribuirla uniformemente a través de todos los discos. Esto último da lugar a un mejor equilibrio de carga.

Existen distintos niveles de RAID que escogen una combinación de estas opciones para implementar la redundancia.

1.2.3. Bloques y extensiones

Los registros de un fichero se deben asignar a bloques del disco porque el bloque es la unidad de transferencia de datos entre disco y memoria. Si el tamaño del bloque es mayor que el del registro, cada bloque contendrá numerosos registros, aunque inusualmente algunos ficheros pueden tener registros grandes que no caben en un bloque. Supongamos que el tamaño de bloque es de *bloq* bytes. Para un fichero con registros de longitud fija con tamaño R, siendo $bloq \geq R$, podemos colocar $B = \lfloor bloq/R \rfloor$ registros por bloque. El valor B se denomina factor de bloqueo del fichero. En general, R no será un divisor exacto de *bloq*, de modo que tendremos cierto espacio desocupado en cada bloque igual a:

$$bloq - (B * R)\ bytes$$

Los sistemas de ficheros que no son capaces de aprovechar este espacio se denominan *no extendidos*. En estos casos se debe seleccionar con cuidado el número de sectores por bloque.

Hasta el momento hemos utilizado la palabra bloque para referirnos a bloques físicos, pero también existe el concepto de *bloque lógico*, que es el conjunto de registros almacenados en un bloque físico.

No existe una fórmula para calcular el factor de bloqueo adecuado. Existen dos límites uno superior y otro inferior. El límite inferior viene impuesto por la necesidad de evitar un número excesivo de transferencias de información cuando se está leyendo el fichero. Esto es muy importante cuando el fichero es procesado secuencialmente o cuando se procesa un grupo de registros contiguos.

El límite superior en el tamaño del bloque físico viene determinado por el tamaño del buffer de memoria principal o el tiempo para transmitir los datos, puede que sólo necesitemos un pequeño registro, pero se debe leer el bloque completo, con el consiguiente consumo de tiempo de transferencia. Sin embargo, observe que cuando se están procesando registros que no están colocados contiguamente en el fichero, un factor de bloqueo alto no implica una reducción significativa de las lecturas físicas, mientras que el tiempo de transferencia y el tamaño del buffer crecen, lo cual es contraproducente.

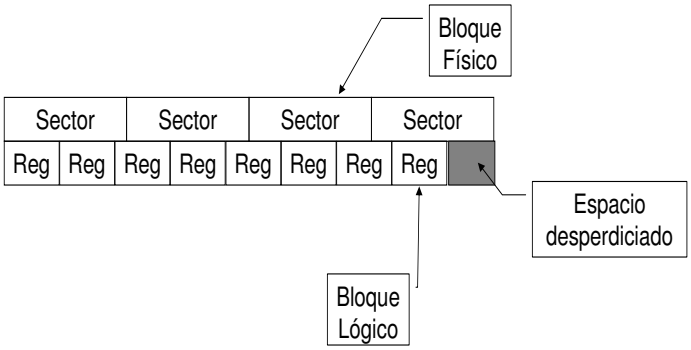

Figura 1.4: Encajando Sectores y registros.

Además dependiendo del encaje entre el tamaño del bloque físico, y el bloque lógico, se puede desperdiciar más o menos espacio en disco como se puede observar en la Tabla 1.1.

Factor de bloqueo	Tamaño del bloque lógico	Sectores por bloque	Tamaño del bloque físico	Eficiencia del almacenamiento (%)
1	160	1	256	63
2	320	2	512	63
3	480	2	512	94
4	640	3	768	83
5	800	4	1024	78
6	960	4	1024	94
7	1120	5	1280	88
8	1280	5	1280	100

Tabla 1.1: Tabla con los tamaños de bloque para un registro de 160 bytes con sectores de 256 bytes.

Para aprovechar el espacio desperdiciado al final de cada bloque físico, algunos sistemas de ficheros pueden almacenar parte de un registro en un bloque físico y el resto en otro. Un puntero, al final del primer bloque, apuntará al bloque que contiene el resto del registro en caso de que no sea el siguiente bloque consecutivo de disco. Esta organización se llama extendida, porque los registros pueden extenderse más allá del final de un bloque. Siempre que el registro sea mayor que el bloque físico, es necesario emplear una organización extendida.

Asignación en el disco de los bloques de un fichero

Existen varias técnicas estándar para asignar los bloques de un fichero en disco. En la *asignación contigua* los bloques del fichero se asignan a bloques consecutivos del disco. Esto agiliza notablemente la lectura de todo el fichero, teniendo en cuenta el esquema de tiempos

comentado en la sección 1.2.2, pero dificulta la expansión del fichero. En la *asignación enlazada* cada bloque del fichero contiene un puntero al siguiente bloque de ese fichero. Esto facilita la expansión del fichero pero vuelve más lenta la lectura del fichero debido a los movimientos del brazo que soporta a las cabezas del disco. Una combinación de las dos asigna grupos de bloques de disco consecutivos, y luego enlaza los grupos. A estos grupos se les llama en ocasiones *segmentos de fichero* o *extensiones*. Otra posibilidad es utilizar la *asignación indexada*, donde uno o más bloques de índice contiene punteros a los bloques que forman el fichero actualmente.

1.2.4. Operaciones básicas sobre ficheros

Las operaciones con ficheros se realizan utilizando las *operaciones básicas* o *primitivas* que proporcionan los sistemas de ficheros de los sistemas operativos. Son un conjunto de operaciones básicas que pueden variar de un sistema a otro, pero por lo general incluyen las que vamos a presentar aquí. Los detalles sintácticos pueden variar, pero los principios serán los mismos. Estas operaciones primitivas son usadas para construir los algoritmos de E/S en subsiguientes secciones.

Existe un concepto común a cualquier conjunto de operaciones básicas, el *puntero actual*, que apunta al registro actual en el fichero.

Por lo general, los programas de alto nivel, como los SGBD, acceden a los registros utilizando estas instrucciones, por eso algunas veces en las siguientes descripciones haremos referencia a variables de programa:

- *Abrir(Nombre fichero)*: prepara el fichero para que se pueda leer o escribir sobre él. Coloca el puntero justo antes del primer registro.
- *Cerrar(Nombre fichero)*: finaliza el acceso al fichero, es decir, libera la zona de memoria donde estaba el puntero actual.
- *Leer-siguiente(Nombre fichero, buffer)*:

  1. Avanza el puntero al siguiente registro ocupado.
  2. Copia el registro al buffer indicado.
  3. Si no existe ningún registro ocupado, la variable EOF se pone a `CIERTO` y no se transfiere información.

  El *buffer* es un espacio de memoria del usuario (no del sistema operativo).

- *Leer-directo(Nombre-fichero, buffer, número-registro)*:

  1. Pone el puntero actual en el registro *número-registro*.
  2. Copia el registro en el *buffer*.
  3. Si el *número-registro* está fuera del rango del fichero o está vacío, la variable VALID se pone a `FALSE`.

- *Actualizar(Nombre-fichero, buffer)*: Copia un registro del *buffer* a la posición indicada por el *puntero actual*. La variable VALID se pone a `FALSO` si el puntero actual no apunta a un registro válido.

- *Añadir(Nombre-fichero, buffer)*:
  1. Pone el puntero actual justo después del último registro del fichero.
  2. Añade un registro nuevo.
  3. Copia el registro del buffer a la posición apuntada por el puntero actual.
- *Escribir-directo(Nombre-fichero, buffer, número-registro)*
  1. Pone el puntero actual en el registro *número-registro*.
  2. Copia el registro que está en el *buffer* al fichero.
  3. Si el *número-registro* está fuera del rango del fichero, la variable VALID se pone a `FALSE`.
- *EOF(Nombre-Fichero)*: Se pone a `TRUE` si el puntero actual está más allá del último registro del fichero, en otro caso, contiene `FALSE`.
- *VALID(Nombre-Fichero)*: Se pone a `FALSE` si cualquier operación falla o si el registro que se ha leído está borrado lógicamente, en otro caso contiene `TRUE`. VALID siempre informa de la última operación realizada.
- *Asignar(Nombre-fichero, puntero)*:
  1. Busca en el disco un segmento de tamaño igual al de un bloque físico.
  2. Asigna este bloque físico al fichero.
  3. Devuelve un *puntero* que apunta al nuevo bloque físico.
- *Liberar(Nombre-fichero, puntero)*
  1. Borra el bloque físico indicado por el puntero del fichero.
  2. Devuelve el bloque físico a la piscina de bloques libres del sistema de ficheros.

1.2.5. Las seis tareas básicas sobre ficheros

Son muchas las tareas que se pueden realizar sobre un fichero, pero la mayor parte del procesamiento de ficheros consiste en una de las seis tareas básicas que presentamos en esta sección. En las siguientes secciones, iremos describiendo los algoritmos necesarios para realizar estas tareas dependiendo de las distintas *organizaciones de ficheros*[2] utilizando la operaciones básicas presentadas en la sección anterior.

1. *Añadir*. Añadir un nuevo registro al fichero.
2. *Borrar*. Borrar un registro del fichero.
3. *Leer todos los registros en cualquier orden*.
4. *Leer todos los registros en el orden de clave*.
5. *Leer un registro con un valor específico de clave*.
6. *Actualizar el registro actual*.

[2]Se refiere a la organización de los datos de un fichero en registros, bloques y estructuras de acceso; esto incluye la forma en la que los registros y los bloques físicos se colocan en el almacenamiento y se interconectan.

1.3. Ficheros secuenciales cronológicos

Esta organización también es conocida como *ficheros de registros no ordenados o ficheros de montón*. Este es el tipo más simple y básico de organización de ficheros. Los registros se colocan en el fichero en el orden en que se insertan, y los registros nuevos se insertan al final del fichero. Muchos de los algoritmos para las tareas básicas que vamos ver para esta organización, son también válidos para otras organizaciones de ficheros como veremos más adelante.

1.3.1. Las tareas básicas

Añadir

Esta tarea es muy eficiente en esta organización. Se realiza con una única operación primitiva, *Añadir*. Se requiere una única escritura física. Sin embargo, cuando se quieren añadir varios registros al mismo tiempo, el sistema de ficheros puede agrupar los registros en el buffer y realizar una escritura física por bloque físico (que incluye la escritura de varios registros lógicos). De este modo, añadir M registros requerirá M/B (redondeado hacia arriba) escrituras físicas.

Borrar

No existe ningún modo razonable para borrar un registro de un fichero secuencial cronológico. Lo más razonable es realizar un borrado lógico del registro, marcándolo como "borrado". Esto puede ser realizado mediante la asignación de un valor específico a algún atributo del registro, o más comúnmente, utilizando un campo especial creado para este fin con un valor binario que indique si está borrado lógicamente o no.

Sea cual sea el método escogido, el borrado pasa a ser un caso especial de actualización que utiliza la primitiva *Actualizar* para establecer la marca de borrado. Esto requiere que primero se lea el registro para luego realizar la operación de actualización. Por lo tanto son necesarias dos transferencias lógicas (de disco a memoria y de memoria a disco). Si se están borrando varios registros al mismo tiempo, el factor de bloqueo puede reducir el número de lecturas/escrituras físicas necesarias dependiendo del orden en el que los registros se deben borrar.

El borrado lógico de registros por marca impone una limitación a esta organización, dado que después de un periodo de uso el fichero se verá atestado de registros marcados como borrado, con lo que se desperdiciará espacio en disco. Para solucionar este problema el fichero se debe *reorganizar* cada cierto tiempo. Esta reorganización se realiza copiando todos los registros que no están marcados como borrados a un nuevo fichero como se ve en el siguiente algoritmo.

```
abrir(fich-viejo);
abrir(fich-nuevo);
WHILE not EOF(fich-viejo) DO BEGIN
      LEER-SIGUIENTE(fich-viejo, buffer);
      IF VALID(fich-viejo) THEN AÑADIR(fich-nuevo, buffer);
END;
```

Observe que si el fichero viejo tiene N registros, incluyendo los borrados lógicamente, y M sin marcar como borrados, el número total de accesos a disco de la reorganización será $N/B + M/B$ (siendo B el factor de bloqueo). Por lo tanto un factor de bloqueo alto favorece este algoritmo.

Leer todos los registros en cualquier orden

Esta tarea es la que mejor realiza esta organización. El fichero es simplemente leído desde el principio al final.

```
abrir(fich);
WHILE not EOF(fich) DO BEGIN
      LEER-SIGUIENTE(fich, buffer);
      IF VALID(fich) THEN /* Procesar el contenido del buffer */;
END;
```

En un fichero de N registros, incluyendo los que están marcados como borrados, el número total de lecturas físicas es N/B. Es decir, un factor de bloqueo alto beneficia a esta tarea.

Leer un registro con un valor de clave específico

Hay tres posibilidades para este problema: (1) que exista un único registro que tenga el valor de clave buscado; (2) que existan varios registros con el valor de clave buscado; y (3) que no exista ningún registro con el valor de clave buscado.

Para el primer caso, el fichero es leído secuencialmente hasta que se encuentre el registro con la clave buscada. De media, se leerán mitad de los registros del fichero, por lo tanto la búsqueda media requerirá $N/2B$ lecturas físicas.

En el resto de los casos, se debe recorrer el fichero por completo el fichero, por lo tanto habrá que realizar N/B lecturas físicas. Como es fácil de ver, el factor de bloqueo alto favorece estos algoritmos.

Finalmente podemos concluir que esta organización es la peor para esta tarea.

Leer todos los registros por orden de clave

Como en el caso anterior, esta no es la mejor organización para realizar esta tarea. La mejor solución para este caso es la utilización del algoritmo *Merge Sort*.

Este algoritmo tiene dos fases. La fase Sort consiste en dividir el fichero en porciones que quepan en memoria principal. Cada porción se ordena en memoria principal utilizando cualquier algoritmo de ordenación en memoria como el *quick sort*. Una vez se ordena cada porción en memoria, se reescribe en disco la porción, pero ahora ordenada.

En una segunda fase Merge, abre todos los archivos con porciones del archivo (ahora ordenadas):

1. Al abrirse se coloca un puntero al primer registro de cada fichero (porción).
2. Se comparan los primeros registros de todas las porciones, y el que tenga el valor más pequeño de clave se escribe en un nuevo fichero, avanzando el puntero de la porción procesada al siguiente registro.

3. Se compara el valor de la clave de los registros apuntados por el puntero de cada porción, y el registro con valor de clave menor se escribe en el nuevo fichero avanzando el puntero de la porción procesada.

4. Repetir 3 hasta que los punteros de todas las porciones lleguen al final.

Ejemplo 1.3.1 Supongamos que disponemos del archivo compuesto por los registros con las siguientes claves:

5	12	10	7	2	15	18	20	3	9	1	6	14	11	4

Supongamos que caben en memoria cuatro registros. Por lo tanto creamos cuatro porciones:

Porción 1				Porción 2				Porción 3				Porción 4			
5	12	10	7	2	15	18	20	3	9	1	6	14	11	4	

Ordenamos uno a uno, en memoria, las cuatro porciones:

Porción 1				Porción 2				Porción 3				Porción 4			
5	7	10	12	2	15	18	20	1	3	6	9	4	11	14	

A continuación se abren los cuatro fichero, leyendo el primer registro en cada una de ellas.

Porción 1				Porción 2				Porción 3				Porción 4			
5 ↑	7	10	12	2 ↑	15	18	20	1 ↑	3	6	9	4 ↑	11	14	

Se escoge el registro de clave más pequeño (en el orden escogido) de los cuatro registros apuntados actualmente, y se escribe en un nuevo fichero, haciendo avanzar el puntero de la porción procesada.

Porción 1				Porción 2				Porción 3				Porción 4			
5 ↑	7	10	12	2 ↑	15	18	20	1	3 ↑	6	9	4 ↑	11	14	

Nuevo fichero

1															

Ahora se escoge el siguiente registro de entre los apuntados por el puntero de cada porción, en este caso el 2.

Porción 1				Porción 2				Porción 3				Porción 4			
5 ↑	7	10	12	2	15 ↑	18	20	1	3 ↑	6	9	4 ↑	11	14	

Nuevo fichero

1	2												

Ahora el proceso continúa del mismo siguiendo el mismo procedimiento, el siguiente elemento es el 3 (en la porción 3).

Porción 1				Porción 2				Porción 3				Porción 4			
5 ↑	7	10	12	2	15 ↑	18	20	1	3	6 ↑	9	4 ↑	11	14	

Nuevo fichero

1	2	3											

El proceso termina cuando todos los registros se han procesado. □

Un modo de acortar el procedimiento, es introducir en memoria, a la hora de realizar el ordenamiento de cada porción, solamente el valor de la clave y el número de registro. Con esto conseguimos introducir muchos más elementos en memoria, y así se deben crear menos porciones.

Al finalizar el proceso tenemos un archivo ordenado por clave, donde para cada registro contiene la clave y el número de registro en el antiguo fichero, con esa información podemos construir un nuevo fichero con los registros completos en orden de clave.

1.4. Fichero relativos o directos

Los ficheros relativos son aquellos en los cuales los registros pueden ser accedidos utilizando en número de registro como clave externa. Si los registros del fichero (de longitud fija) se numeran como $0, 1, \ldots, r-1$, y los registros de cada bloque $0, \ldots, B-1$, donde B es el factor de bloqueo, el i-ésimo registro del fichero se encontrará en el bloque $\lfloor (i/B) \rfloor$ del fichero y será el ($i \ mod \ B$)-ésimo registro de ese bloque.

El sistema, sin embargo, asume el cálculo de la dirección del sector del bloque físico donde está el registro que se está buscando. De este modo el fichero relativo es similar a un array unidimensional donde cada elemento del array es un registro y el índice del array el número de registro, a cada posición de ese imaginario array le denominamos slot.

Los números de registro actúan como clave de los registros en el fichero. Por ejemplo, una compañía puede asignar un número de empleado a cada uno de sus trabajadores, este número será en número del registro que almacena su información en el fichero.

Claramente la ventaja es que se puede acceder directamente a un registro dado si conocemos su clave externa. Pero a cambio, tenemos que tener en cuenta dos restricciones. (1) Un slot no puede ser leído antes de que se escriba un registro en él, para evitar una excepción a esta regla cuando se crea el fichero, se marcan como vacíos todos los slots. (2) Los números de registro debe estar dentro de un rango de valores $0, \ldots, N-1$. Intentar leer un registro en una posición fuera de ese rango dará lugar a un error. Es decir el tamaño del fichero (como el caso de un array) es fijo.

1.4.1. Las tareas básicas

Añadir

Los registros pueden ser añadidos en cualquier momento en cualquier slot. Para añadir un registro se usa una operación de la primitiva ESCRIBIR-DIRECTO. Es responsabilidad del programador controlar que ningún registro válido se destruya al ser sobrescrito.

Borrar

Los registros se marcan como borrados. Algunos sistemas de ficheros tienen una operación específica para borrar registros y reinicializar el estado del slot a vacío. En caso de que no se disponga de esta posibilidad, con una operación UPDATE o ESCRIBIR-DIRECTO se puede marcar como borrado el slot. Como los slots se puede reutilizar, no hay necesidad de reorganización.

Leer todos los registros en cualquier orden

Esta tarea utiliza el mismo algoritmo que el caso de los ficheros secuenciales cronológicos, una lectura secuencial del fichero desde el inicio hasta el final. La primitiva LEER-SIGUIENTE saltará los slots vacíos en la mayoría de los sistemas de ficheros.

Leer un registro con un valor de clave específico

Como en el caso del fichero secuencial cronológico hay tres posibilidades para este problema: (1) que exista un único registro que tenga el valor de clave buscado; (2) que existan varios registros con el valor de clave buscado; y (3) que no exista ningún registro con el valor de clave buscado.

En el caso de que el número de registro no sirva como clave de búsqueda, estaríamos ante la misma situación que en el caso del fichero secuencial cronológico. Si el número de registro actúa como clave externa, con una operación de LEER-DIRECTO (y por lo tanto una lectura física) se obtiene el registro deseado. Incluso si en una lectura anterior quedó en el buffer el bloque físico que contiene la lectura actual, puede que no sea necesaria una lectura física.

Leer todos los registros en orden de clave

Una vez más, si el número de registro no es la clave por la que queremos realizar la lectura, debemos utilizar el algoritmo del Merge-Sort presentado en la sección 1.3.1 para primero ordenar el fichero, y luego leer secuencialmente el fichero una vez ordenado. Para tratar de mejorar esta solución, podemos intentar mantener los registros del fichero ordenados por la clave. Por ejemplo, podemos intentar mantener los empleados ordenados alfabéticamente dentro del fichero (usando el campo nombre como clave). Así una lectura secuencial (con LEER-SIGUIENTE) del fichero sería suficiente.

El problema aparece cuando tenemos que insertar un nuevo empleado y el en lugar que le corresponde no hay un slot vacío. Podríamos intentar, cuando insertamos empleados nuevos, dejar huecos para futuras inserciones, pero siempre puede llegar un punto en el que se produzca la situación anterior, además de desperdiciar espacio libre.

Por otro lado, si el número de registro actúa como clave externa, entonces una sencilla lectura secuencial del fichero bastará.

1.5. Ficheros ordenados

Podemos ordenar físicamente los registros de un fichero en disco basándonos en los valores de uno de sus campos, (en terminología de ficheros) es decir la clave, aunque también se le denomina en algunos textos *campo de ordenación*. De un modo más preciso, para cada registro i, donde i no es ni el primer registro del fichero ni el último, $KEY(i-1) \leq KEY(i) \leq KEY(i+1)$, donde $KEY(i)$ devuelve la clave del registro i.

Hay dos tipos de ficheros ordenados, ficheros secuenciales ordenados y ficheros relativos ordenados. Los *ficheros secuenciales ordenados* son exactamente iguales a los fichero secuenciales cronológicos pero con los registros ordenados. Los *ficheros relativos ordenados* son iguales a los ficheros relativos pero con tres restricciones:

1. Están ordenados, tal y como se describió anteriormente.
2. Todos los registros deben contener un valor de la clave. Esto debe ser cierto aunque el registro se marque como borrado.
3. El número de registro no puede ser utilizado como clave. El número de registro puede ser utilizado por los algoritmos que operen sobre el fichero, pero no puede ser utilizado como clave externa debido a las reorganizaciones, que como veremos, son necesarias en esta organización.

La mayor ventaja que se obtiene con esta organización, es la posibilidad de utilizar la búsqueda binaria para localizar un registro dado, lo que constituye una mejora sobre las búsquedas lineales.

1.5.1. Las tareas básicas

Añadir y borrar

Estas dos tareas comparten el hecho de estar prohibidas por las restricciones expuestas en la sección anterior. Si se utiliza el fichero ordenado relativo, se puede marcar los registros como borrados, pero manteniendo el valor de la clave intacta. Añadir registros también es difícil debido a que el fichero puede estar lleno o no haber un slot vacante en el lugar adecuado para mantener la secuencia de la clave. Por lo tanto, es necesario reorganizar el fichero para añadir o borrar registros, o para cambiar el valor de la clave de un registro.

El fichero se reorganiza mezclando el fichero original con el *fichero de cambios* para crear un nuevo fichero. El fichero original y el de cambios se puede borrar una vez construido el nuevo. El fichero de cambios contiene registros idénticos a los que contiene el fichero original con la única diferencia de que contienen un campo más para indicar la naturaleza del cambio. Este campo indica si el registro debe ser borrado o añadido durante la reorganización.

El fichero de cambios está ordenado por el mismo campo de ordenamiento que el fichero original. Ambos son leídos secuencialmente, y el nuevo fichero se escribe también secuencialmente. El algoritmo se puede observar en la Figura 1.5.

Leer todos los registros en cualquier orden

Consiste en una lectura secuencial del fichero como en el caso del fichero secuencial cronológico.

```
abrir(fich-antiguo);
abrir(fich-nuevo);
abrir(fich-cambios);
LEER-SIGUIENTE(fich-antiguo, buff-ant);
LEER-SIGUIENTE(fich-cambios, buff-camb);
WHILE not EOF(fich-ant) or not EOF(fich-cambios) DO BEGIN
      IF not EOF(fich-antiguo) and not EOF(fich-cambios) THEN
             IF ORDEN(buff-ant)=ORDEN(buff-cmab) THEN BEGIN
                IF FLAG(fich-cambios)=anadir THEN BEGIN
                   ESCRIBIR-SIGUIENTE(fich-nuevo, fich-cambios);
                   ESCRIBIR-SIGUIENTE(fich-nuevo, fich-antiguo);
                   /*Pone ambos registros en el nuevo fichero*/
                END;
              LEER-SIGUIENTE(fich-antiguo, buff-ant);
             /*Vuelve leer un registro de cada fichero*/
             LEER-SIGUIENTE(fich-cambios, buff-camb);
             END ELSE IF ORDEN(buff-ant)<ORDEN(buff-cmab) THEN BEGIN
                ESCRIBIR-SIGUIENTE(fich-nuevo, fich-antiguo);
                LEER-SIGUIENTE(fich-antiguo, buff-ant);
             END ELSE IF ORDEN(buff-ant)>ORDEN(buff-cmab) THEN BEGIN
                IF FLAG(fich-cambios)=anadir THEN
                   ESCRIBIR-SIGUIENTE(fich-nuevo, fich-cambios);
                ELSE /* Error se está intentado borrar un registro no existente*/
                LEER-SIGUIENTE(fich-cambios, buff-camb);
             END;
      ELSE IF EOF(fich-cambios) THEN BEGIN
              ESCRIBIR-SIGUIENTE(fich-nuevo, fich-antiguo);
              LEER-SIGUIENTE(fich-antiguo, buff-ant);
      END ELSE IF EOF(fich-antiguo) THEN BEGIN
                  IF FLAG(fich-cambios)=anadir THEN
                     ESCRIBIR-SIGUIENTE(fich-nuevo, fich-cambios);
                     ELSE /* Error se está intentado borrar un registro no existente*/
                     LEER-SIGUIENTE(fich-cambios, buff-camb);
END;
cerrar(fich-antiguo);
cerrar(fich-nuevo);
cerrar(fich-cambios);
```

Figura 1.5: Algoritmo de reorganización

Leer un registro con un valor específico de clave

Este algoritmo es una de las razones para escoger esta organización, dado que permite realizar búsquedas binarias sobre el fichero.

Si hay un sólo registro que tenga el valor buscado en el campo de búsqueda, en el caso de un fichero secuencial cronológico necesitábamos $N/2B$ lecturas físicas, siendo N el número de registros del fichero y B el factor de bloqueo. Partiendo de la teoría de Shannon [Sha48], se puede calcular que aproximadamente son necesarias $\log_2(N)$ (este valor es una aproximación, el factor de bloqueo puede hacer variar ligeramente este valor, haciéndole restar algunos accesos a esta fórmula) lecturas para encontrar el registro, lo cual es un avance claro sobre $N/2B$.

Leer todos los registros en orden de clave

Dado que el fichero está ordenado por la clave (campo de ordenación), una simple lectura secuencial es suficiente. Por lo tanto es otra de las razones para utilizar esta estructura.

Si deseásemos leer el fichero por orden distinto al del campo de ordenación, tendríamos que utilizar una vez más el algoritmo del merge sort.

1.6. Ficheros de acceso directo (HASH)

Uno de los primeros artículos de ficheros de acceso directo es [Mor68]. El fichero de acceso directo es otra variación del fichero relativo. Las diferencias con el fichero relativo se encuentran en el modo en el que se asignan los números de registro y cómo se localizan los registros. En lugar de asignar arbitrariamente los registros a los slots, el número de registro que le corresponde a un registro dado se deriva de su clave, aplicando sobre dicha clave una *función de hash*. A partir de aquí vamos a representar la función de hash como **H(clave)**.

Una buena función de hash debe tener tres características:

1. Los números de registro que genere deben estar distribuidos uniformemente y aleatoriamente sobre el fichero.

2. Pequeñas variaciones en el valor de la clave deben causar grandes variaciones en el valor de H(clave). Todas las partes de la clave deben ser utilizadas por la función de hash.

3. La función de hash debe minimizar la creación de sinónimos. Un *sinónimo* es una clave que al serle aplicada la función de hash el resultado es el mismo que el de una clave diferente. Es decir, si $H(X) = H(Y)$, entonces X e Y son sinónimos, también denominadas *colisiones*.

1.6.1. Funciones hash

El problema de encontrar una buena función de hash está relacionado con el de encontrar un buen generador de números pseudoaleatorios. Esto es debido a que en ambos casos el resultado debe ser un conjunto de números que estén estadísticamente uniformemente y aleatoriamente distribuidos sobre un rango dado. La diferencia de la función de hash está en que debe ser repetible; dada la misma clave, la función de hash debe devolver el mismo número de registro siempre.

La función de hash se puede ver como una asignación del espacio de valores de una clave al espacio de direcciones de un fichero. Un fichero de N slots tendrá un espacio de direcciones de N direcciones, mientras que normalmente la clave tendré un espacio de direcciones mucho más grande. Por lo tanto no existe modo alguno de asignar a todo valor del espacio de direcciones de la clave una dirección de fichero única (supóngase que la clave es el número de seguridad social de los empleados, cuando hablamos del fichero de empleados de una empresa).

Antes de poder aplicar la función de hash puede ser necesario transformarla para poderla tratar de forma numérica. Por ejemplo, si es el nombre de una persona. Una práctica común y sencilla es tomar los valores numéricos de los códigos ASCII de los caracteres que forman la clave, y aplicar sobre ellos alguna operación matemática sobre ellos (por ejemplo, una simple suma) para obtener un único valor numérico. Esto se denomina *plegado de la clave*.

Una vez que se dispone de un valor numérico, la fórmula más utilizada por su sencillez y efectividad es:

$$H(K) = K \text{ mód } N \tag{1.1}$$

donde K es la forma numérica de la clave y N es el número de slots del fichero. Observe que $0 \leq H(K) < N$. Para obtener buenos resultados K debe ser mucho mayor (varios órdenes de magnitud) que N.

Aunque esta fórmula tiene la ventaja de ser muy simple, puede dar malos resultados si el valor de N no se escoge con cuidado. Supongamos que la clave es un número de 10 dígitos. Si $N = 1000$, $H(K)$ será siempre en los tres dígitos de menor orden. Los otros seis dígitos no tendrán ninguna influencia en el resultado de la función de hash. Obsérvese que este problema no depende de la base que utiliza la máquina para representar los números, el problema sería el mismo siempre que N sea un múltiplo pequeño de un entero elevado a otro entero. Para solucionar esto, simplemente escogemos un número primo como N. En las cercanías de todo número se puede encontrar siempre un número primo.

Estudios exhaustivos han demostrado que la Fórmula (1.1) funciona muy bien siempre que N sea un número primo.

1.6.2. Estructuras para manejar sinónimos

Como se ha comentado, es inevitable la creación de sinónimos. Sea M el número de registros almacenados en un fichero de N slots, el *factor de carga* es $(M \div N)$, es decir el porcentaje de ocupación del fichero. La Tabla 1.2 muestra, con una distribución aleatoria de claves y con distintos factores de carga, las probabilidades de que la función de hash deje slots vacíos o que se envíe un uno, dos y hasta 8 registros al mismo slot. Cuando el factor de carga se acerca al 100 %, la fracción de slots a los que la función de hash no asigna ningún registro se acerca a $1/e$ (0.368). Del mismo modo, la fracción de slots a los que la función de hash asigna un único registro es otra vez $1/e$. Esto quiere decir que a los restantes $(1 - 2/e)$ slots (o 26,4 % de los slots), la función de hash les asigna los restantes $(1 - 1/e)$ (o 63,2 %) registros. Es decir, de media se asignan 2,39 registros a cada uno de esos slots. Dado que sólo se puede almacenar un registro por slot, se debe diseñar algún método para utilizar los slots vacíos.

Número de registros	Factor de carga									
k	20 %	50 %	70 %	80 %	85 %	90 %	95 %	98 %	99 %	100 %
0	.81873	.60653	.49659	.44933	.42741	.40657	.38674	.37531	.37158	.36788
1	.16375	.30327	.34761	.35946	.36330	.36591	.36740	.36780	.36786	.36788
2	.01637	.07582	.12166	.14379	.15440	.16466	.17452	.18022	.18209	.18394
3	.00109	.01264	.02839	.03834	.04375	.04940	.05526	.05887	.06009	.06131
4	.00005	.00158	.00497	.00767	.00930	.01111	.01313	.01442	.01487	.01533
5	.00000	.00016	.00070	.00123	.00158	.00200	.00249	.00283	.00294	.00307
6	.00000	.00001	.00008	.00016	.00022	.00030	.00039	.00046	.00049	.00051
7	.00000	.00000	.00001	.00002	.00003	.00004	.00005	.00006	.00007	.00007
8	.00000	.00000	.00000	.00000	.00000	.00000	.00000	.00001	.00001	.00001

Tabla 1.2: Probabilidades de hash con una distribución aleatoria

Fichero de overflow

Una solución simple al problema de los sinónimos es colocar a los sinónimos en otro fichero, llamado *fichero de overflow*. Si el factor de carga del fichero principal se mantiene bajo, la proporción de sinónimos será baja, y el fichero de overflow no necesitará mucho espacio. Este fichero puede organizarse como se desee. La forma más sencilla es organizarlo como un fichero secuencial cronológico. Mientras su tamaño sea pequeño, los retardos serían aceptables.

Se podría organizar como otro fichero de acceso directo, pero más pequeño. De este modo necesitaríamos a su vez otro fichero de overflow, una vez más, todavía más pequeño.

Existen dos desventajas para el fichero de overflow. (1) El fichero principal no debe tener un factor de carga demasiado alto o los tiempos de acceso serán demasiado altos dado el tamaño del fichero de overflow. (2) Dado que el fichero principal debe tener un factor de carga bajo, supondrá un desperdicio de espacio libre considerable debido a los slots vacíos. Por estas razones este método no es muy recomendable.

Direccionamiento abierto

Sería mucho más conveniente almacenar todos los registros en un único fichero y aprovechar 36,7 % de slots que siempre están vacíos (a un 100 % de factor de carga). Pero para ello se debe establecer algún método para almacenar los sinónimos en los (en otro caso) slots vacíos.

Una aproximación válida para este fin es usar la técnica de *direccionamiento abierto*. El direccionamiento abierto se basa en la existencia de una secuencia predecible de slots a examinar cuando se está buscando un slot vacío para introducir un registro. Cuando toque buscar ese registro (en el caso de que no ocupe el slot que por función hash le corresponde), dado que la secuencia es predecible, es repetible, por lo tanto se recorre del mismo modo que se hizo cuando se buscaba el slot vacío, hasta dar con el registro. Hay dos métodos de direccionamiento abierto: prueba lineal y rehashing.

Prueba lineal simplemente establece que cuando estamos ante un sinónimo, el hueco se buscará en los slots siguientes a la posición natural del registro, es decir, si el registro debería estar en el slot i y está ocupado, se buscará en los slots $i+1, i+2, \ldots$ hasta que se encuentre un slot vacío.

La *posición natural* de un registro de clave K es el slot $H(K)$. En el caso de que la

posición natural esté cerca del final del fichero, y todos los slots hasta el fin del mismo estén ocupados, se continúa buscando slots vacíos desde el principio del fichero. Con esto acabamos de describir la primera tarea común en un fichero de acceso directo con prueba lineal para ubicar los sinónimos, *añadir*.

Evidentemente el algoritmo para buscar un registro con una clave determinada es similar.

```
boolean busca-reg(fichero,buffer,clave-buscada,n)
k:=PLEGAR(clave-buscada); /*Plegado de la clave*/
r:= k mod n; /*Cálculo de la dirección natural*/
LEER-DIRECTO(fichero,buffer,r); /*Leer dirección natural*/
WHILE VALID(fichero) AND CLAVE(buffer) ≠ clave-buscada DO BEGIN
/* Mientras no se encuentra el registro seguimos buscando*/
      r:=(r+1) mod n; /*Sigue la prueba lineal*/
      LEER-DIRECTO(fichero,buffer,r);
END;
IF CLAVE(buffer) = clave-buscada THEN RETURN TRUE
ELSE RETURN FALSE;
```

Ejemplo 1.6.1 Sea el fichero:

Nº slot	0	1	2	3	4	5	6	7	8	9	10
Datos	9	3	17	19					4	5	7

Supongamos que el siguiente registro a insertar tiene clave 11, $H(11) = 11 \text{ mód } 11 = 0$. Sin embargo, el slot 0 está ocupado. Por prueba lineal se buscaría a partir del slot 0, en los slots 1, 2 y 3, hasta llegar al slot 4 que está vacío, por lo tanto aquí se coloca el registro.

Nº slot	0	1	2	3	4	5	6	7	8	9	10
Datos	9	3	17	19	11				4	5	7

El proceso de búsqueda del registro, como hemos visto consiste en realizar la misma secuencia.□

Existen varios problemas con el uso de la prueba lineal. Primero, cuando el factor de carga del fichero se acerca al 100 %, puede que algunas áreas estén demasiado congestionadas, con zonas donde todos los slots están ocupados. Estudios empíricos han revelado que con un 99 % de factor de carga, la búsqueda de un registro representa inspeccionar de media 25 slots, llegando en los peores casos a tener que buscar en 1000 slots (para ficheros grandes obviamente). Sin embargo, si el factor de carga se mantiene por debajo del 85 %, el rendimiento es aceptable.

Dado que la prueba lineal realiza una búsqueda *secuencial* desde la posición natural del registro, un factor de bloqueo alto beneficia esta técnica. En el caso anterior (con un factor de carga cercano al 100 %) si en lugar de tener un factor de bloqueo 1, pasamos a tener 50 registros por bloque, el número medio de lecturas físicas pasa de las 25 a 1.5.

Un segundo problema de la prueba lineal es que no es fácil borrar registros. El algoritmo *busca-reg* presentado en esta sección interpreta un registro vacío como la prueba de que un registro no está en el fichero, es decir, es el fin de la búsqueda. Mientras no se borren registros

esta en una suposición válida. Sin embargo, en el momento en que se borren registros esta ya no es una marca válida de fin de búsqueda.

Una solución simple pero costosa es continuar la búsqueda hasta que se encuentre el registro o se haya recorrido todo el fichero. Una solución más razonable sería marcar los slots como abandonados, pero no borrarlos. El algoritmo *busca-reg* no terminaría su búsqueda al encontrar un slot "abandonado", sin embargo, ese slot podría ser utilizado a la hora de insertar un nuevo registro.

De las tareas básicas, quedan para el caso del fichero de acceso directo con prueba lineal, la lectura de todos los registros en cualquier orden (lectura secuencial de todo el fichero) y leer todos los registros en orden de clave. Para esta tarea disponemos de dos posibilidades; una es utilizar el algoritmo Merge-Sort de la Sección 1.3.1. La otra opción consistiría en las siguientes fases:

- Hacer una lectura secuencial del fichero para obtener todos los valores de clave en el fichero.
- Ordenarlas en un array en memoria con cualquier método de ordenación en memoria.
- Leer el array por orden, y aplicar para cada clave el algoritmo *busca-reg*.

Rehashing es otro método de direccionamiento abierto. Necesita dos funciones de hash. La primera función es tal y como la hemos presentado anteriormente: $H(K) = K$ mód N, donde K es una clave numérica (o plegada). Si el slot apuntado por $H(K)$ está ocupado, se aplica la segunda función de hash $D = (K \text{ mód } J) + 1$. J es un número primo diferente y más pequeño que N. D se utiliza como un desplazamiento sobre $H(K)$. Dado que $H(K) + D$ puede ser mayor que N, esta suma se realiza módulo N.

Con esto se evita el problema de la prueba lineal que tendía a congestionar demasiado zonas del fichero, y de este modo, las búsquedas de un registro requerirán menos lecturas físicas debido a las pruebas de distintos slots. Sin embargo, dado que D casi siempre va ser mayor que el factor de bloqueo, las ventajas asociadas se pierden.

Cuando se desea añadir un registro, se intenta insertar en el slot $H(K)$, si está ocupado, se intenta en $(H(K) + D)$ mód N, si éste también lo está, se trata de insertar en $(H(K) + 2D)$ mód N y así sucesivamente hasta encontrar un slot vacío.

Cabría esperar tres problemas con esta estrategia. Primero, cualquier par de claves que son sinónimos con la función de hash principal, es altamente deseable que no lo sean en el segunda, sino estaría siempre compitiendo por los mismos slots. Segundo, D siempre debe ser mayor que cero o el procedimiento iterará por siempre. Esto se asegura al añadir un $+1$ en la expresión que calcula D. Tercero, podríamos pensar que el algoritmo que inserta un nuevo registro puede entrar en un bucle sin fin a pesar de haber slots vacíos. Sin embargo, al ser N primo, y ser el desplazamiento (dado que se suma a $H(K)$ en módulo N) menor que N, esto nunca puede ocurrir.

Como en el caso anterior leer un registro con una clave específica, consiste en repetir el proceso de añadir un registro.

Leer todos los registros en cualquier order y en orden de clave se realiza del mismo modo que en el caso de la prueba lineal.

Encadenamiento sin reemplazamiento

Las técnicas de direccionamiento abierto requieren búsquedas largas cuando los factores de carga son altos, debido a que hay que recorrer la secuencia de registros hasta encontrar el deseado. Esta secuencia incluye en muchos casos registros que no son sinónimos del registro buscado.

El *encadenamiento sin reemplazamiento* es un método que evita examinar esas listas de registros que no son sinónimos del registro buscado. Para ello utiliza punteros para construir listas enlazadas en el fichero. Cuando se añade un registro, y su dirección natural está ocupada, se añade a una lista enlazada cuya cabeza está en su dirección natural. A la hora de recuperar el registro, sólo es necesario recorrer la lista para comprobar todos los sinónimos de la dirección natural.

El problema de donde colocar los sinónimos es el mismo que en el caso del direccionamiento abierto. Se puede usar cualquiera de las dos estrategias presentadas en la sección anterior.

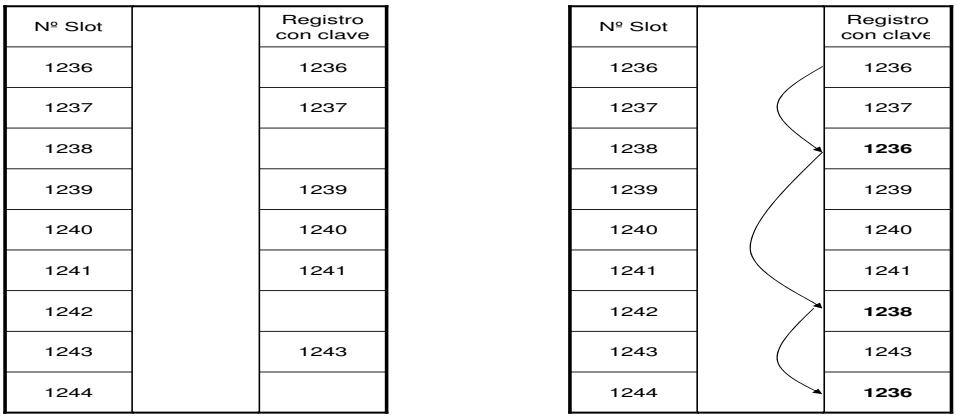

Figura 1.6: Cadenas de sinónimos. Antes de inserción y después de inserción.

El problema de este método es que se pueden *fusionar* varias cadenas en una, conteniendo sinónimos de distintas direcciones, lo que da origen a búsquedas más largas. La Figura 1.6 muestra un ejemplo. En la parte izquierda muestra una porción de un fichero donde todos los registros se han colocado en sus direcciones naturales. En la parte derecha muestra la misma parte del fichero después de varias inserciones.

La primera inserción es la del registro que tiene como dirección natural la 1236, que ya está ocupada. Por prueba lineal se encuentra un slot vacío en la dirección 1238. A continuación se coloca un puntero del slot 1236 al slot 1238 enlazando al registro recién insertado con su dirección natural. Supongamos que ahora se tiene que insertar un registro cuya dirección natural es 1238, una vez más ocupado, pero por un registro cuya dirección natural es 1236 (y por lo tanto está enlazado con ese slot). El algoritmo de inserción, por prueba lineal, encuentra un espacio en el slot 1242, donde se inserta en nuevo registro. A continuación la cadena se extiende.

Por último, se inserta un tercer registro con dirección natural 1236. La prueba lineal

encuentra un slot vacío en el slot 1244, en donde se inserta el nuevo registro. La cadena que comienza en el 1236, continúa en el slot 1238, ahora se extiende al slot 1244. Como se puede observar, esta cadena son en realidad dos cadenas *fusionadas*, una con sinónimos del slot 1236, y otra con sinónimos del 1238.

Según el factor de bloqueo va aumentando, las cadenas se van enmarañando, esto fuerza reorganizaciones periódicas después de muchas inserciones y borrados.

El costo de mantener los punteros se compensa con las búsquedas. En las condiciones comentadas para el caso de la prueba lineal, es decir, un factor de carga cercano al 100 % y un factor de bloqueo de 1, el número medio de lecturas físicas (según estudios empíricos) es de 1.77, en comparación con las 25 lecturas físicas que requería el caso de la prueba lineal.

El borrado de un registro vuelve ser un problema complicado de solucionar. Hay dos problemas, el primero es que al borrar un registro hay que tener cuidado de no romper la cadena. Esto se soluciona conectando el puntero del slot que precede (en la cadena) al borrado, con el slot al que apunta el puntero del registro borrado.

El otro problema es borrar el registro que es la cabeza de la cadena. Cuando se buscase un registro que está en dicha cadena, el algoritmo comienza la búsqueda en la posición natural (del registro buscado) y al encontrar ese slot vacío, no continuaría la búsqueda por la cadena.

Podríamos pensar que para solucionar el problema, lo que podemos hacer es reemplazar el registro que se está borrando con el siguiente registro en la cadena. Sin embargo esto mueve el problema un paso en la cadena. Lo que es peor, el siguiente registro, puede estar en su posición natural. Si lo movemos de ese lugar haríamos imposible su localización.

Como no hay un modo sencillo de solucionar este problema, tendremos que realizar el borrado, marcando los registros como "abandonados". Los punteros seguirán colocados tal y como estaban antes del borrado. Con lo cual, a pesar de que estos slots se pueden reutilizar para las nuevas inserciones, son necesarias reorganizaciones cada cierto tiempo.

Para buscar un registro con una clave determinada, aplicamos la función de hash, buscamos en su dirección natural, y si no está ahí continuamos siguiendo la cadena. El problema de de cuándo determinar que un registro buscado no está en el fichero es sencillo, en cuanto se llegue al final de la cadena que comienza en su dirección natural, sin encontrar dicho registro, podemos concluir que el registro no está en el fichero.

Para leer todos los registros en cualquier orden y en orden de clave utilizaremos las mismas técnicas que para el caso de la prueba lineal.

Encadenamiento con reemplazamiento

Aunque el encadenamiento sin reemplazamiento consigue reducir el número medio de lecturas físicas a la hora de buscar un sinónimo, el problema de los borrados no está bien resuelto. La solución a este problema, presentada en esta sección, además va a reducir el número medio de lecturas en las búsquedas como veremos a continuación.

El *encadenamiento con reemplazamiento* difiere del encadenamiento sin reemplazamiento en que cuando se inserta un registro, si el slot de su dirección natural está ocupado por un registro que no está en su posición natural, este registro se recolocará en otra posición de modo que el nuevo registro se pueda colocar en su dirección natural. Este simple cambio tendrá dos efectos beneficiosos:

- La probabilidad de encontrar un registro en su dirección natural aumenta considerablemente.

- La cadena de sinónimos no puede *fusionarse* como ocurría con el encadenamiento sin reemplazamiento. Por lo tanto no hay problemas con los borrados. Otro beneficio del hecho de que las cadenas no se fusionen, es que éstas son más cortas, y por lo tanto da lugar a búsquedas más rápidas.

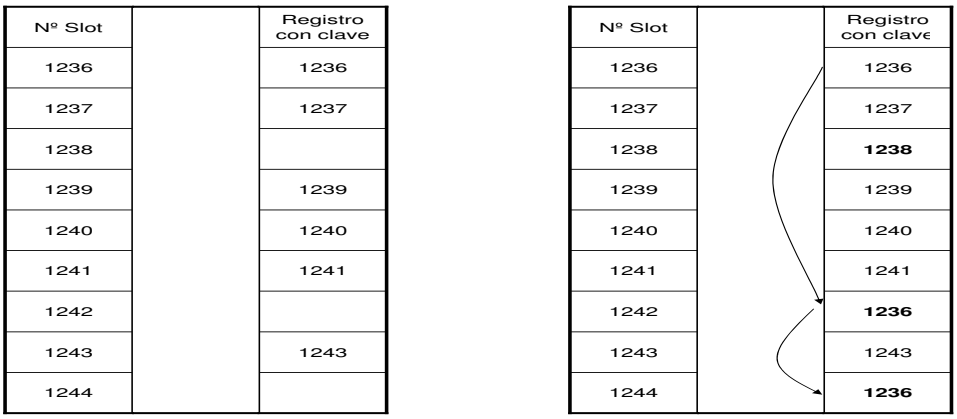

Figura 1.7: Cadenas de sinónimos. Antes de inserción y después de inserción.

En la Figura 1.7 se puede observar el mismo ejemplo que en la Figura 1.6, pero con encadenamiento con reemplazamiento. Después del estado inicial (izquierda de la figura) se inserta el primer registro cuya posición natural es 1236, que ya está ocupada. Por prueba lineal se encuentra un hueco en el slot 1238, donde, por lo tanto, se inserta inicialmente.

El siguiente registro que se inserta, tiene por dirección natural el slot 1238, que en ese momento está ocupado, pero por un registro cuya dirección natural es el slot 1236. Por lo tanto este último (el registro con dirección natural 1236) se reubica en otro slot, y el registro que se está insertando, se ubica en su dirección natural (1238). Se busca por prueba lineal un slot para registro que se debe realojar, y se encuentra en el slot 1242, a donde se mueve el registro. Se debe, por supuesto, ajustar convenientemente los punteros de la cadena de sinónimos de la dirección 1236.

Finalmente, un tercer registro también debería ser insertado en la dirección 1236. Como está ocupada, por prueba lineal se encuentra un slot vacío en la dirección 1244, donde es colocado y enlazado con la cadena de sinónimos de la dirección 1236 una vez más.

El borrado de un registro ahora no tiene mayor complicación, cuando se borra la cabeza de la cadena, uno de los registros que la conforman debe reemplazar al registro borrado. En cualquier otro lugar, simplemente se debe tener cuidado con mantener la cadena enlazada.

Para leer un registro con una clave determinada, leer todos los registros en cualquier orden y leer todos los registros en orden de clave, se sigue los mismos algoritmos que en el caso anterior de encadenamiento sin reemplazamiento.

1.6.3. Una mejora

Normalmente para minimizar la generación de sinónimos, cada slot del fichero es en realidad un bloque físico (o incluso varios bloques físicos), por lo tanto, por lo general cabrá en dicho slot más de un registro lógico del fichero, todos ellos sinónimos. El ordenamiento dentro del bloque es irrelevante dado que una vez realizada la lectura física, todo el bloque estará en el buffer de la memoria principal y ahí la búsqueda es extremadamente rápida.

1.6.4. Cuando es adecuada esta técnica

Observe que la principal ventaja de esta técnica está en el algoritmo de *buscar un registro con una clave determinada*, dado que los ficheros de acceso directo proporcionan un acceso muy rápido a los registros que cumplan una condición de igualdad por un único campo, que suele ser el campo clave del fichero.

Es por lo tanto una organización adecuada para responder consultas de este tipo, pero inadecuada para consultas de tipo rango (los elementos mayores y/o menores que un valor).

Por otro lado, hay que tener en cuenta que la búsqueda de un registro, dado el valor de campo que no es clave, es tan costosa como en un fichero secuencial cronológico.

1.7. Ficheros hash extensibles

Esta evolución de los ficheros de acceso directo se debe a Fagin et al. [FNPS79]. Su origen está en la principal limitación de los ficheros de acceso directo; *su tamaño es fijo*. Este hecho viene forzado por el uso de la función de hash que usa el tamaño del fichero para distribuir uniformemente los registros a lo largo de sus slots.

Si se cambia el tamaño del fichero, se debe cambiar la función de hash, y todos los registros antes ubicados según la función de hash antigua deben ser recolocados según la nueva función de hash. Esto no es más que mover los registros de un fichero estático a otro igualmente estático.

Sería mucho más adecuado disponer de una estructura dinámica que se expandiera y contrajera automáticamente según el número de registros en el fichero. Idealmente, la estructura de un fichero dinámico tendría tres propiedades:

1. El fichero se expandirá automáticamente según las necesidades de alojamiento de nuevos registros. La expansión no requerirá la reorganización del fichero o la reubicación de más de una pequeña parte de los registros.

2. El fichero se contraerá cuando sea necesario, de modo que la probabilidad de que el factor del carga del fichero baje del 50 % sea muy baja. Como en el caso de la expansión, la contracción no requerirá una reorganización del fichero o la recolocación de una porción significativa de registros.

3. La estructura del fichero permitirá la recuperación de un registro por su clave primaria con un acceso a disco.

El hash extensible tiene estas tres característica, pero para lograrlo, en lugar de hacer una única transformación de la clave, con esta técnica es necesario realizar varias transformaciones.

1.7.1. Transformaciones de la clave

La secuencia de transformaciones se puede observar en la Figura 1.8.

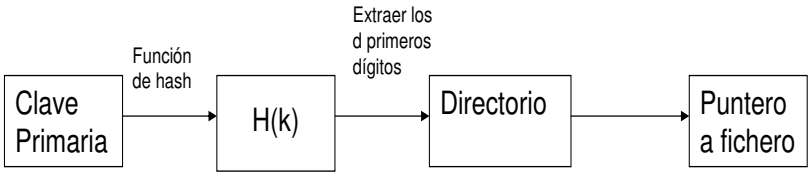

Figura 1.8: Transformaciones de la clave en el hash extensible.

La primera transformación es una función hash que asigna aleatoriamente a las claves un valor en un espacio de direcciones fijo. Los primeros d dígitos de este resultado se extraen para utilizarlos como un índice de un directorio. El directorio contiene finalmente el puntero al registro deseado.

Como comentamos en la Sección 1.6.3, los registros están en cubos. Así los punteros del directorio apuntan a dichos cubos. Como comentamos anteriormente, un cubo se lee en una lectura física, es decir, es un bloque físico, y normalmente aloja a varios registros (sinónimos) disminuyendo de este modo las colisiones.

La primera transformación es similar a la que vimos en los ficheros de acceso directo. Las únicas diferencias son que el espacio de direcciones producido por la función de hash debe ser próximo a una potencia de 2 y, dado que ahora el espacio de direcciones de la función de hash ya no son los slots del fichero, el rango de la función es ahora algo arbitrario (siempre teniendo en cuenta que debe ser un número primo próximo a una potencia de 2).

La segunda transformación extrae una porción relativamente pequeña de dígitos de $H(k)$. Normalmente se utilizan números binarios y se extraen los dígitos de mayor orden. Estos dígitos se usan como un índice dentro de un array unidimensional de punteros. Este array se llama *directorio* y contiene 2^d entradas, una para cada combinación de d bits.

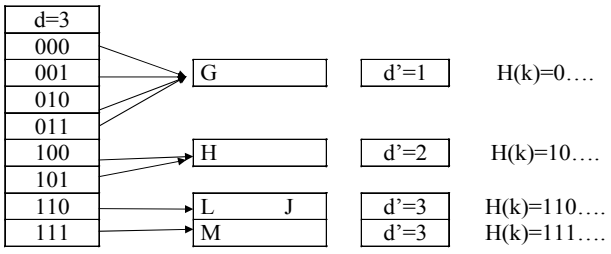

Figura 1.9: Orden de directorio $d = 3$ con cuatro cubos.

En la Figura 1.9 el directorio tiene orden tres, es decir, se utilizan los tres primeros dígitos de $H(k)$ para escoger una de las 8 (2^3) entradas del directorio. Por ejemplo, el registro de clave

G, en este ejemplo, necesariamente al aplicársele la función de hash ($H(G)$), el resultado debe ser un número binario cuyo bit de mayor orden sea un 0 (según la Figura 1.9). Supongamos que es 00101101..., en tal caso, se toman los tres primeros dígitos (001), que se utilizan para inspeccionar el directorio, el puntero de la entrada correspondiente nos direcciona al primer cubo del fichero, en donde se encuentra el registro con clave G.

Así la secuencia completa para buscar un registro consiste en cinco pasos:

1. Se le aplica la función de hash sobre la clave, $H(k)$.
2. Se extraen los primeros d dígitos de $H(k)$.
3. Se utilizan los d dígitos para buscar en el directorio el puntero a cubo apropiado.
4. El puntero se utiliza para leer el cubo y traerlo a memoria principal.
5. El registro deseado se localiza dentro del cubo.

1.7.2. Expansión y contracción del fichero

La razón de la complejidad de la estructura mostrada en la sección anterior es permitir la posibilidad de la expansión y contracción del fichero según el número de registros almacenados en el mismo varía.

Considérese el fichero en la Figura 1.9. Si deseamos insertar un registro, digamos, con clave K y la aplicación de la función de hash genera, por ejemplo, $H(K) = 10100111\ldots$, K debe ser ubicado en el segundo cubo (ocupado únicamente por el registro con clave H). Si suponemos que en cada cubo caben dos registros, K cabe en este cubo, se inserta ahí, y no hay nada más que hacer, el resultado se muestra en la Figura 1.10.

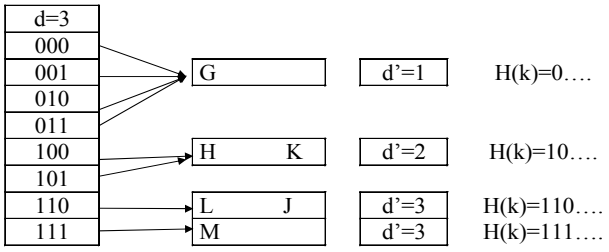

Figura 1.10: Una vez insertado el registro con clave K.

Si ahora insertamos, digamos, el registro con clave U, cuya función de hash es $H(U) = 1011001\ldots$, el directorio nos indica que debería ser insertado en el cubo ocupado actualmente por H y K, que ya está lleno. Entonces se debe añadir un nuevo cubo al fichero a donde irán los registros cuya función de hash empiece por 101 y en el antiguo cubo quedarán los registros cuya función de hash empiece por 100. Así en la Figura 1.11 podemos ver (suponiendo que $H(H) = 1000\ldots$, lo cual no tiene porque ser cierto, pero veremos más adelante cómo solucionar este caso) como quedaría el fichero.

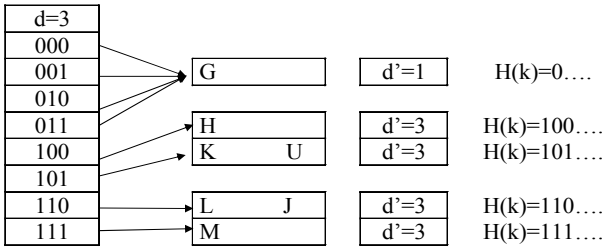

Figura 1.11: Una vez insertado el registro con clave U.

Observe que cada cubo tiene un parámetro d' que indica el número de dígitos de $H(K)$ cuyo valor es común a todos los registros del cubo. Este valor siempre debe ser menor o igual a d. El número de punteros que apuntan a un cubo determinado es $2^{(d-d')}$.

Esta división de cubos, puede continuar según el fichero crezca y mientras existan al menos 2 punteros que apuntan al cubo que se debe dividir. Llegados al punto de necesitar insertar un registro en un cubo lleno y con un sólo puntero apuntándole (es decir $d = d'$) el directorio se debe duplicar.

Supongamos que el fichero de la Figura 1.11, se desea insertar el registro con clave V, y que $H(V) = 1100\ldots$. V debería ser insertado en el cubo ocupado actualmente por L y J, que está lleno y además está apuntado por un único puntero. Es necesario, por tanto, duplicar el directorio.

Cada puntero en el directorio original es duplicado y ocupa dos posiciones consecutivas en el nuevo directorio. En la Figura 1.12, podemos ver, por ejemplo, como el puntero que en la Figura 1.11 apuntaba al cubo que contiene el registro M, ahora se ha duplicado, ahora apuntan a ese registro dos punteros, los correspondientes a las entradas 1110 y 1111. Lo mismo ocurre con el resto, salvo el caso del cubo que se está dividiendo (el que contenía los registros L y J). Ahora es posible dividir este cubo de modo que un puntero apuntará al cubo que contiene los registros cuya función de hash empieza por 1100 y otro puntero hará lo propio con los que empiezan por 1101.

El proceso de contracción es lo contrario de lo visto hasta ahora. Dos cubos se pueden recombinar si se dan tres condiciones:

1. El factor de carga medio de los dos cubos no puede exceder el 50 %.
2. Los cubos a combinar deben tener el mismo valor de d'.
3. Los valores de la función de hash aplicada a las claves de los registros en los dos cubos deben ser iguales en los $d' - 1$ primeros bits.

Por ejemplo, supongamos que en el archivo de la Figura 1.13, se borra el registro con clave X. Quedan dos cubos (el que contiene el registro con clave W y el contiene al registro con clave M) que cumplen todas las características para la fusión de 2 cubos:

1. Ambos tienen un factor de carga del 50 %.

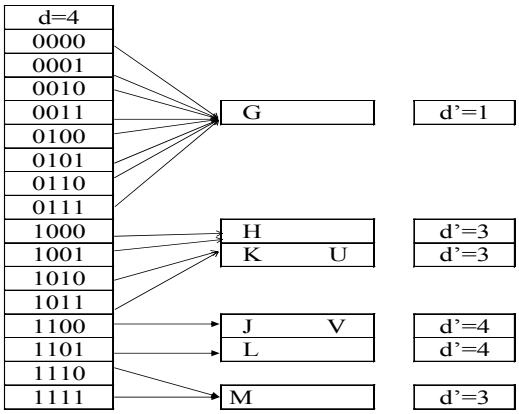

Figura 1.12: Duplicación del directorio.

2. Ambos tienen el mismo valor de d'.

3. Al aplicar la función de hash sobre las claves de los registro en su interior, los $d' - 1$ primeros bits son iguales (111).

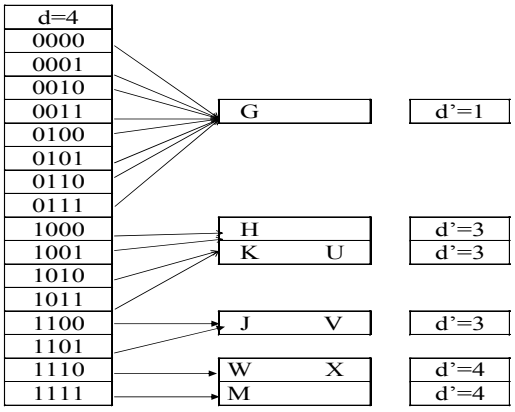

Figura 1.13: Borrado de X (antes).

Por lo tanto se pueden recombinar en un único cubo, quedando como resultado que se muestra en la Figura 1.14.

El directorio se debe contraer siempre que todos los pares de punteros (empezando desde la posición cero del directorio) tengan el mismo valor. Este es el caso del directorio de la

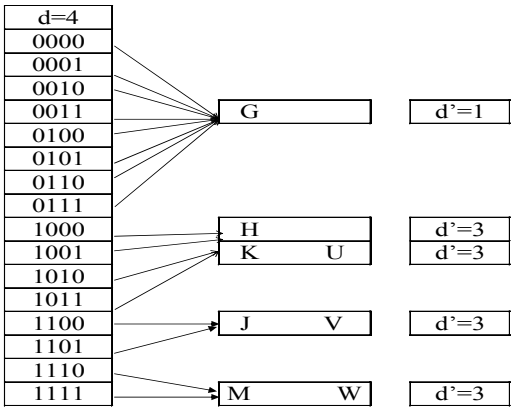

Figura 1.14: Borrado de X (después).

Figura 1.14. Los punteros 0 y 1 apuntan al mismo cubo, el 2 y el 3 también y así hasta los punteros 14 y 15. El resultado se puede ver en la Figura 1.15.

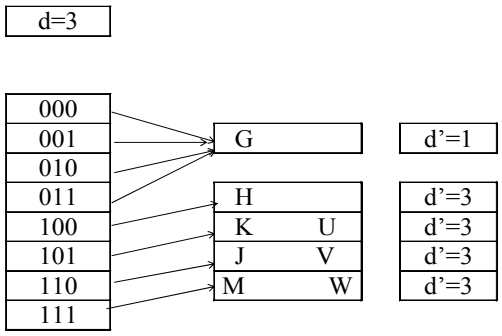

Figura 1.15: Contracción del directorio.

Bibliografía

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible hashing-a fast access method for dynamic files. *TODS*, 4(3), septiembre 1979.

[Mor68] R. Morris. Scatter storage techniques. *CACM*, 11(1):38–43, enero 1968.

[PGK88] D. Patterson, G. Gibson, and R. Katz. Raid: redundant arrays of inexpensive disks. In *Proceedings ACM SIGMOD Int. Conference on Management of Data*, 1988.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell Syst.Tech. J.*, 27:379–423, 623–656, 1948.