TEMA 1: Introducción a los archivos y bases de datos.

Autor: Enrique V. Bonet Esteban

1.1 Introducción a los archivos.

Entenderemos como *archivo* o *fichero* a un conjunto de informaciones sobre un mismo tema tratado como una unidad de almacenamiento y organizado de forma estructurada para la búsqueda de un dato individual. Por tanto, un archivo no es más que una agrupación de datos, cuya estructura interna es la que el usuario, programador o sistema operativo le haya conferido implícitamente, pues la organización del archivo no es intrínseca a la existencia del mismo.

Es necesario aclarar en este punto que los archivos son independientes de los programas y de hecho, un mismo archivo creado por un programa puede ser usado por otros. El único requisito para obtener información de un archivo cualquiera es conocer como se han organizado los datos en el mismo.

Un archivo, desde una concepción clásica, esta formado por *registros*, que son las unidades elementales que componen el archivo. Un registro es una colección de información generalmente relativa a una entidad particular (un estudiante, un vehículo, etc.). Los registros pueden contener varios datos, siendo cada uno de estos datos los *campos* de los registros. Un campo puede tener una longitud fija o variable, debiendo existir en el segundo caso alguna manera de indicar su longitud. El término *longitud del campo* hace referencia al tamaño máximo de un campo. Los campos pueden ser de diferente tipo: numérico, cadena, fecha, etc. Como consecuencia de ello, un registro puede tener también una longitud variable, refiriendo el término *longitud del registro* al tamaño máximo de un registro. Un ejemplo de todo esto puede verse en la figura 1.1.1.

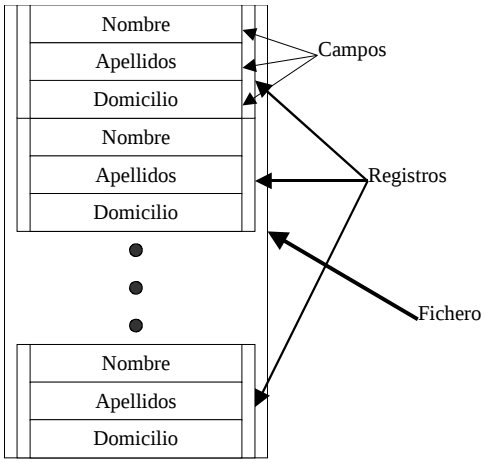

Figura 1.1.1: Ejemplo de archivo formado por registros.

La forma más común de identificar un registro es eligiendo un campo dentro del registro que llamaremos *clave del registro*. La localización de un registro se hace a través de los valores que tiene el campo clave, siendo a veces necesario utilizar varios campos como claves, bien por existir valores iguales de la clave en distintos registros, bien por estar interesados en buscar los registros por distintos campos. Por esta razón se

pueden utilizar más de un campo como clave de un registro, denominándose los campos *clave primaria*, *clave secundaria*, etc.

Desde el punto de vista hardware, para acceder a un archivo sólo existen *direcciones físicas*. Por ejemplo, en un disco la información se lee o escribe en forma de bloques (sectores) referenciados por unidad/superficie/pista/sector. El sistema operativo se encarga de transformar la *dirección lógica* de un dato en el archivo (posición del dato dentro del archivo) a la dirección física de forma transparente para el usuario.

Desde un punto de vista físico, el acceso a los datos puede realizarse de dos formas:

- *Acceso secuencial.* Se caracteriza porque el acceso a la información de un archivo se produce en un orden determinado y fijo. El acceso a los bloques de datos se hace de forma sucesiva, uno tras otro, tal como se reproducen las canciones grabadas en una cinta.

- *Acceso directo.* Se caracteriza porque el acceso a determinada información contenida en el archivo puede hacerse sin acceder a las informaciones anteriores o posteriores, como ocurre al acceder a una canción grabada en un CD-ROM.

1.1.1 Organización lógica de un archivo.

Existen diferentes formas de estructurar u organizar los registros que componen un archivo sobre un soporte de información. La eficiencia en la utilización del archivo depende de la organización del mismo; por ello se debe optar por una u otra organización atendiendo a la forma en que se va a usar el archivo. Las principales organizaciones de archivos son:

- *Secuencial.* Los registros se encuentran de forma consecutiva y han de ser leídos, necesariamente, según este orden. En la organización secuencial los registros carecen de un orden especial, estando situados según el orden temporal de su inclusión en el archivo. Por lo general, en un archivo secuencial, al final del archivo físico, se graba la marca de final de archivo (end of file, EOF). Los archivos secuenciales son los que ocupan menos memoria y son útiles cuando se desconoce a priori la cantidad de datos a almacenar. Un caso especial de los archivos secuenciales son los archivos de texto, donde cada registro es simplemente un carácter o código de control.

- *Secuencial indexada.* En esta organización se dispone de una tabla de índices adicional, donde por índice entendemos una referencia que permite obtener de forma automática la ubicación de la zona del archivo físico donde se encuentra el registro buscado. Esto permite localizar un registro por medio de su clave sin leer previamente la clave de todos los que le preceden. La organización secuencial indexada implica un mantenimiento de las tablas de índices y una previsión inicial de la cantidad máxima de registros que va a contener.

- *Directa.* La ubicación del registro en el soporte físico se obtiene directamente a partir de funciones que la obtienen a partir del valor de la clave, mediante un algoritmo de transformación (hashing) de ésta. Un archivo, para que pueda estar

dotado de una organización directa tiene que tener registros de longitud fija y prefijado su propio tamaño, lo que determina la distribución de la información, al tiempo que limita la cantidad de registros que podrá contener.

La propiedad principal que debe cumplir una función de hash es que distribuya aleatoria, pero uniformemente, las claves entre las direcciones disponibles. De esta forma se consigue que una concentración en los valores de la clave no se traslade a una concentración en el almacenamiento al aplicar la función de hash. Las funciones de hash más importantes son cuatro: truncamiento, plegado, multiplicación y división.

En una función hash de truncamiento, a la clave se le eliminan algunos dígitos[1] del comienzo o del final. Podemos construir una función de hash eliminando los primeros k o los últimos m dígitos de una clave de n dígitos, o de ambos extremos, debiendo suceder siempre que n > k+m. Así, por ejemplo, si la clave es un número decimal de 8 dígitos, podemos construir una función hash de truncamiento eliminando los 2 primero y los dos últimos, de forma que f(12345678) = 3456. A todos los números que tengan los mismos dígitos en sus cuatro cifras centrales les corresponderá la misma dirección según la tabla de hash.

En una función hash de plegado, se divide la clave en dos o más partes y luego se suman. Siguiendo con el ejemplo anterior, f(12345678) = 1234 + 5678 = 6912. Si como resultado de la suma aparece un dígito más, este puede eliminarse mediante un truncamiento o no, según se desee.

Una función hash de multiplicación divide la clave en varias partes y multiplica entre si las partes resultantes, truncando el resultado si es demasiado grande. Por ejemplo, f(12345678) = 1234 * 5678 = 70006652. Como función hash, la multiplicación esparce de forma aleatoria las claves, evitando las colisiones entre claves próximas. Sin embargo, si las claves terminan frecuentemente en 000 o 001, por ejemplo, la multiplicación no esparce adecuadamente las claves.

La división es el método hash más sencillo y efectivo. Consiste en dividir la clave entre un número primo que se aproxime al tamaño de la tabla, de forma que el resto de la división entera es la dirección relativa. La forma general de estas funciones hash es f(N) = N mod T. Así, f(12345678) = 12345678 mod 1009 = 563.

1.1.2 Operaciones sobre archivos.

Desde el punto de vista del programador, los archivos interesan porque los programas típicamente operan sobre ellos, para leer ó escribir en los mismos. Para ello, los programadores emplean procedimientos o funciones para comunicarle al sistema operativo la operación a realizar y obtener una respuesta de éste.

El sistema operativo mantiene información sobre cada archivo que manipula, tales como el soporte donde se encuentra el archivo, el tipo de organización del mismo,

[1] Si la clave esta formada por caracteres, se suele convertir a una clave numérica mediante la suma de los valores ASCII, UNICODE, etc., que forman la clave.

el lugar donde éste empieza o la posición actual dentro del archivo. Todo ello se encuentra contenido en un *descriptor de fichero* asociado a cada archivo que se esté utilizando en un momento determinado.

Los procedimientos básicos que se pueden llevar a cabo sobre los distintos tipos de archivos son:

- *Apertura de un archivo*. Para que un programa pueda operar sobre un archivo, la primera operación que debe realizar es la apertura del mismo. En la misma, el programa emplea una subrutina identificando el archivo con el que quiere trabajar (mediante un nombre y, en algunos casos, el soporte donde se encuentra) y el modo en que va a emplearlo (lectura, escritura, etc.). A partir de estos datos el sistema operativo construye un descriptor de fichero, de manera que a partir de ese instante el programa ya no se referirá al archivo por su nombre sino por el descriptor de fichero creado. El modo de apertura de un archivo determina las operaciones que se podrán realizar sobre el mismo (por ejemplo, si ha sido abierto para lectura no podrá escribirse en él). Además, cuando se abre un archivo, la posición actual dentro del archivo es el comienzo del mismo.

- *Cierre de un archivo*. Cuando un programa no vaya a acceder más a un archivo, es necesario indicar al sistema operativo este hecho. Con ello el sistema operativo libera el descriptor de fichero y se asegura que el archivo queda debidamente almacenado en la memoria secundaria. Para cerrar un archivo simplemente se utiliza una subrutina de cierre indicando el archivo por medio de su descriptor.

- *Creación de un archivo*. Para utilizar un archivo, evidentemente, éste tiene que existir. Por ello el archivo debe ser creado en algún momento mediante una llamada al sistema operativo. Si se produce un intento de lectura en un archivo que no existe se producirá inevitablemente un error. Sin embargo, en un intento de escritura en el archivo, en muchos lenguajes de programación el archivo se creara de forma automática, por lo cual no es posible deducir de antemano un error en la operación.

- *Lectura y escritura de un archivo.* La lectura de un archivo consiste en transferir información del archivo a la memoria principal, mientras que la escritura en un archivo es la transferencia de información de la memoria principal a un archivo. Para ello, se invoca la subrutina de lectura o escritura respectiva, indicando el archivo al que queremos acceder mediante el descriptor de fichero. Además, para una lectura debemos indicar el lugar de la memoria principal donde se desean situar los datos procedentes del archivo; mientras que para una escritura debemos indicar el lugar de la memoria principal donde se encuentran los datos que se quieren escribir en el archivo.

  En los archivos secuenciales, la lectura o escritura se realizan en el archivo a partir de la posición actual en el mismo (contenida en el descriptor de fichero). Esta posición es, en principio, la posición siguiente a la del último acceso al archivo que se hubiera producido. En caso de que el soporte lo permita (soporte direccionable), podremos llamar a una subrutina de posicionamiento y efectuar

el acceso a partir de la nueva posición. En la lectura de los archivos es especialmente importante detectar cuando se ha alcanzado el final del archivo y no hay más datos que leer. En algunos lenguajes la propia subrutina de lectura indica cuando se encuentra la marca de final de archivo. En otros se dispone de una subrutina de detección de fin de archivo que conviene consultar antes de realizar una lectura.

En los archivos secuenciales indexados los accesos no utilizan la posición actual, sino que deben indicar el valor del campo clave para buscar el registro, de modo que si no se encuentra un registro con dicha clave, hay que tener en cuenta esta circunstancia. Igualmente la escritura se realiza indicando el valor de la clave y el registro se sitúa en el archivo en la posición adecuada, actualizando las tablas de índices si fuera necesario.

En los archivos de organización directa, que vimos tenían un tamaño prefijado y registros de longitud fija, usando la función hash que utilice el archivo, a partir de cada clave se obtiene la posición del archivo donde debe encontrarse el registro. El problema de esta organización (genérico de toda función hash) es que la relación no es biunívoca, con lo cual distintas claves pueden dar lugar a la misma posición. Esto se soluciona de forma parcial con la existencia de una zona de desborde donde se emplazan los registros que han colisionado. Para la lectura indicamos el registro con su campo clave, lo cual determina la posición en el archivo. Sin embargo, si un campo con clave distinta ocupa esa posición a consecuencia de tener el mismo resultado de la función hash, se busca el registro en la zona de desborde. Si tampoco se encuentra allí se indica el error correspondiente. Para la escritura, con el campo clave se calcula, mediante la función hash, la posición en el archivo. Si dicha posición ya se encuentra ocupada por otro registro, este nuevo registro se escribe en la zona de desborde, siempre y cuando exista espacio.

1.2 Operaciones sobre archivos en C.

Expondremos a continuación como es posible realizar los procedimientos básicos sobre archivos en C.

1.2.2 Apertura de un archivo.

Como se comento de forma general, un archivo debe ser abierto antes de poder leer y/o escribir en él. En C la apertura de un archivo se realiza mediante la función *fopen()*, la cual admite como argumentos dos cadenas de caracteres, conteniendo la primera el nombre del archivo y la segunda el modo de apertura del archivo (lectura, escritura, etc.). La función *fopen()* devuelve un descriptor del fichero abierto mediante un puntero a una estructura de tipo *FILE*, o bien *NULL* si la apertura falla. El código que realiza la apertura es:

```
#include <stdio.h> /* Archivo cabecera con la declaración de funciones
                      y del tipo FILE */
......
FILE *fp; /* Declaramos una variable de tipo puntero a la estructura
             FILE */
fp=fopen(nombre,modo);
```

No siempre es posible abrir un archivo, pues puede no existir, no tener permisos (permiso denegado), etc., por lo cual siempre es necesario comprobar que el archivo ha podido ser abierto. Por tanto, la operación de apertura de un archivo debe realizarse siempre de la siguiente forma:

```
#include <stdio.h> /* Archivo cabecera con la declaración de funciones
                      y del tipo FILE */
......
FILE *fp; /* Declaramos una variable de tipo puntero a la estructura
             FILE */
if ((fp=fopen(nombre,modo))==NULL)
   /* Error de apertura del archivo */
```

1.2.3 Cierre de un archivo.

El cierre de un archivo debe producirse siempre que se no sea necesario trabajar más con el citado archivo. En C, el cierre del archivo se produce mediante la llamada a la función *fclose()*, la cual admite como argumento el puntero a la estructura de tipo *FILE* que fue devuelta en la llamada a la función *fopen()*. Por tanto, para cerrar un archivo previamente abierto con éxito, solo debemos ejecutar el siguiente código:

```
fclose(fp) /* Cierra el archivo cuyo descriptor viene dado por el
              puntero fp */
```

1.2.4 Creación de un archivo.

La creación de un archivo en C se realiza mediante la llamada a la ya comentada función *fopen()*. La creación puede realizarse mediante unos valores determinados del argumento modo de la función. En concreto el argumento modo puede tomar los valores "*w*" o "*a*". El valor "*w*" indica que deseamos crear el archivo desde el principio, esto es, en caso de que ya exista el archivo, este será destruido y creado de nuevo, perdiendo los datos previamente existentes. Sin embargo, con el valor "*a*" le indicamos que queremos insertar nuevos datos en el archivo, de forma que si este ya existe no será creado, mientras que si no existe se creará como si hubiéramos utilizado "*w*".

1.2.5 Lectura y escritura.

Una vez un archivo ha sido abierto para leer y/o escribir en el mismo, podemos proceder a efectuar las operaciones deseadas. Todas las operaciones que deseemos realizar debemos efectuarlas indicando el archivo sobre el cual queremos efectuar las operaciones mediante el descriptor de fichero devuelto por la función *fopen()*.

Un archivo puede se leído o escrito en formato de texto o binario. Intuitivamente, un fichero en formato de texto es aquel que contiene texto, y en formato binario es aquel que contiene datos binarios, como por ejemplo un programa ejecutable. Sin embargo, en ciertos casos no es tan intuitivo. Por ejemplo, un número entero puede ser escrito en formato texto o formato binario. En formato de texto, el número entero es almacenado mediante los símbolos del 0 al 9, siendo representado en el archivo tal y como lo hacemos nosotros sobre una hoja de papel. Por otro lado, en formato binario, el número es almacenado exactamente igual que se encuentra representado en la memoria del ordenador, esto es, mediante una secuencia de bits, secuencia que no se corresponde

con la representación que realizamos en una hoja de papel. Así, el número 25889 es almacenado en formato de texto como la secuencia de caracteres *25889*, mientras que en formato binario corresponde a la cadena de bits 0110010100100001 que se mostrarían como la secuencia de caracteres *e!*.

Para poder leer o escribir los archivos, el lenguaje C proporciona dos grupos de funciones, unas para archivos de tipo texto y otras para archivos de tipo binario.

Los archivos de tipo texto son leídos con la función *fscanf()* y escritos con la función *fprintf()*. Los archivos de tipo binario son leídos con la función *fread()* y escritos con la función *fwrite()*.

La función *fscanf()* posee como primer argumento el identificador del archivo, a continuación una cadena de caracteres que indica los datos y tipo de los datos a leer y por último un número variable de punteros a las direcciones de memoria donde deseamos guardar los valores leídos. El número de punteros vendrá dado por el número de datos que hemos indicado que queremos leer en la cadena de caracteres. La función *fscanf()* devuelve el número de datos leídos y asignados de forma correcta.

La función *fprintf()* es similar a la función fscanf(), siendo sus dos primeros argumentos iguales y a continuación un número variable de datos con los valores que deseamos guardar. Devuelve el número de bytes escritos o un valor negativo en caso de error.

La función *fread()* lee datos binarios de un archivo. Puede leer cualquier número de datos. Para ello, sus argumentos son un puntero al vector donde desean guardarse los datos (el vector puede ser un puntero a un solo elemento si solo desea leerse un elemento), el tamaño que tiene cada elemento individual, el número de elementos que deseamos leer y por último, el descriptor de fichero de donde deseamos hacer la lectura. La función devuelve el número de elementos (no bytes) que se han leído.

Por último, la función *fwrite()* escribe datos binarios en un archivo. Sus argumentos son idénticos a los de la función *fread()*, devolviendo el número de elementos que se han escrito.

En último lugar, explicar que en C existe una función, *fseek()*, que permite posicionarnos en la posición del archivo deseada. La función fseek() posee tres parámetros. El primero es el descriptor de fichero en el cual queremos posicionarnos, el segundo es el desplazamiento respecto al origen que queremos realizar. El tercer parámetro indica el origen del desplazamiento, puede tomar los valores *SEEK_SET*, *SEEK_CUR* o *SEEK_END* (valores 0, 1 y 2 respectivamente), e indican el comienzo del archivo, la posición actual en el archivo y el final del archivo.

1.2.6 Ejemplo de manejo de archivos en C.

Veremos a continuación algunos ejemplos completos sobre como operar con archivos en C.

1.2.6.1 Lectura de un archivo en formato texto.

Supongamos un archivo en formato texto que contiene una serie de líneas, cada una de las cuales corresponde a un número en coma flotante. El programa para leer los elementos del archivo es:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      FILE *fp;
      float v;

      if ((fp=fopen("datos.txt","rt"))==NULL) /* Abrimos el archivo
                                       para lectura en modo texto */
      {
            fprintf(stderr,"\nError abriendo el archivo\n");
            /* Mostramos un mensaje*/
            exit(0); /* Salimos del programa */
      }
      while (fscanf(fp,"%f\n",&v)==1); /* Mientras leamos un
                                          elemento */
      fclose(fp); /* Cerramos el archivo */
      return 0; /* Terminamos el programa */
}
```

Si los elementos nos interesa almacenarlos en un vector para su posterior uso, podemos realizar un bucle inicial como el anterior contando los elementos existentes, o bien, como en el siguiente ejemplo, suponer que el archivo contiene una primera línea que nos indica el número de elementos a leer.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      FILE *fp;
      float *vector,v;
      int num,i;

      if ((fp=fopen("datos.txt","rt"))==NULL) /* Abrimos el archivo
                                       para lectura en modo texto */
      {
            fprintf(stderr,"\nError abriendo el archivo\n");
            /* Mostramos un mensaje*/
            exit(0); /* Salimos del programa */
      }
      if (fscanf(fp,"%d\n",&num)!=1) /* Leemos el entero que indica el
                                        número de elementos */
      {
            fclose(fp); /* Cerramos el archivo */
            fprintf(stderr,"\nError leyendo el archivo");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      if ((vector=(float *)calloc(num,sizeof(float)))==NULL)
      /* Reservamos memoria */
      {
            fclose(fp); /* Cerramos el archivo */
            fprintf(stderr,"\nError reservando memoria\n");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      for(i=0;i<num;i++) /* Para todo el número de elementos */
```

```
      {
            if (fscanf(fp,"%f\n",&v)!=1); /* Leemos un elemento */
            {
                  fclose(fp); /* Cerramos el archivo */
                  fprintf(stderr,"\nError leyendo el archivo\n");
                  /* Mostramos un mensaje */
                  exit(0); /* Salimos del programa */
            }
            vector[i]=v; /* Lo guardamos en su posición del vector */
      }
      fclose(fp); /* Cerramos el archivo */
      free(vector); /* Eliminamos la memoria reservada */
      return 0; /* Terminamos el programa */
}
```

1.2.6.2 Lectura de un archivo en formato binario.

La lectura de un archivo en formato binario elemento a elemento se realiza de forma similar a como se realiza en modo texto. Para ello supongamos un archivo binario que contiene guardados un conjunto de números en coma flotante. El programa para leer dicho archivo es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      FILE *fp;
      float v;

      if ((fp=fopen("datos.bin","rb"))==NULL) /* Abrimos para lectura
                                                 en modo binario */
      {
            fprintf(stderr,"\nError abriendo el archivo\n");
            /* Mostramos un mensaje*/
            exit(0); /* Salimos del programa */
      }
      while (fread(&v,sizeof(float),1,fp)==1); /* Mientras leamos un
                                                  elemento */
      fclose(fp); /* Cerramos el archivo */
      return 0; /* Terminamos el programa */
}
```

Si los elementos nos interesa almacenarlos en un vector para su posterior uso, podemos realizar un bucle inicial como el anterior contando los elementos existentes, o bien, como en el siguiente ejemplo, suponer que el archivo contiene una primera línea que nos indica el número de elementos a leer. Obsérvese la diferencia en la lectura, pues ahora podemos leer todos los elementos del vector con una sola operación de lectura.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      FILE *fp;
      float *vector,v;
      int num;

      if ((fp=fopen("datos.bin","rb"))==NULL) /* Abrimos para lectura
                                                 en modo binario */
```

```
      {
            fprintf(stderr,"\nError abriendo el archivo\n");
            /* Mostramos un mensaje*/
            exit(0); /* Salimos del programa */
      }
      if (fread(&num,sizeof(int),1,fp)!=1) /* Leemos el número de
                                              elementos */
      {
            fclose(fp); /* Cerramos el archivo */
            fprintf(stderr,"\nError leyendo el archivo");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      if ((vector=(float *)calloc(num,sizeof(float)))==NULL)
      /* Reservamos memoria */
      {
            fclose(fp); /* Cerramos el archivo */
            fprintf(stderr,"\nError reservando memoria\n");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      if (fread(vector,sizeof(float),num,fp)!=num) /* Leemos los
                                                      elementos */
      {
            fclose(fp); /* Cerramos el archivo */
            fprintf(stderr,"\nError leyendo el archivo\n");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      fclose(fp); /* Cerramos el archivo */
      free(vector); /* Eliminamos la memoria reservada */
      return 0; /* Terminamos el programa */
}
```

1.2.6.3 Escritura de un archivo en formato texto.

La escritura de un archivo en formato texto es similar a la lectura en formato texto, tan solo hemos de tener en cuenta las dos posibles formas de abrir el archivo. Con la opción *w* el archivo será creado siempre y, en caso de existir previamente, se perderán los datos que contenga. Sin embargo, con la opción *a* podemos abrir el archivo para añadir datos después de los datos contenidos. Veamos un sencillo ejemplo de escritura de un archivo en modo texto.

```
#include <stdio.h>
#include <stdlib.h>
#define N 1000
int main(void)
{
      FILE *fp;
      float vector[N];
      int num,i;

      /* Suponemos que num contiene el número de elementos del vector
         que deseamos guardar, por supuesto dicho número es menor o
         igual que N */
      if    ((fp=fopen("datos.txt","wt"))==NULL)    /*    Creamos    para
escritura
                                                  en modo texto */
      {
            fprintf(stderr,"\nError creando el archivo\n");
            /* Mostramos un mensaje*/
```

```
            exit(0); /* Salimos del programa */
      }
      for(i=0;i<num;i++)
      if (fprintf(fp,"%f\n",vector[i])<0)
      {
            fclose(fp); /* Cerramos el archivo */
            fprintf(stderr,"\nError escribiendo el archivo\n");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      fclose(fp); /* Cerramos el archivo */
      return 0; /* Terminamos el programa */
}
```

1.2.6.4 Escritura de un archivo en formato binario.

La escritura de un archivo en formato binario es similar tan sencilla como era la lectura en dicho formato. Al igual que en el caso anterior, tan solo sólo hemos de tener en cuenta las dos posibles formas de abrir el archivo (w o a). Un ejemplo puede verse a continuación:

```
#include <stdio.h>
#include <stdlib.h>
#define N 1000
int main(void)
{
      FILE *fp;
      float vector[N];
      int num;

      /* Suponemos que num contiene el número de elementos del vector
         que deseamos guardar, por supuesto dicho número es menor o
         igual que N */
      if    ((fp=fopen("datos.bin","wb"))==NULL)    /*    Creamos    para
escritura
                                                    en modo texto */
      {
            fprintf(stderr,"\nError creando el archivo\n");
            /* Mostramos un mensaje*/
            exit(0); /* Salimos del programa */
      }
      if (fwrite(vector,sizeof(float),num,fp)!=num) /* Escribimos los
                                                       datos */
      {
            fclose(fp); /* Cerramos el archivo */
            fprintf(stderr,"\nError escribiendo el archivo\n");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      fclose(fp); /* Cerramos el archivo */
      return 0; /* Terminamos el programa */
}
```

1.2.6.5 Posicionamiento en un archivo.

Aunque la operación de posicionamiento no esta limitada a archivos binarios, pudiendo realizarse en archivos de modo texto, en estos últimos pueden producirse

aparentes errores debido a que secuencias de caracteres como “retorno de carro” y “nueva línea” (dos caracteres) son leídas como un solo carácter.

Como ejemplo de posicionamiento en un archivo, supongamos que tenemos un archivo binario que contiene números enteros y que queremos leer el entero número 101 del archivo (suponemos que existe). El código que realiza dicha operación es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      FILE *fp;
      int valor;

      if ((fp=fopen("datos.bin","rb"))==NULL) /* Abrimos para lectura
                                                 el archivo */
      {
            fprintf(stderr,"\nError abriendo el archivo\n");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      if (fseek(fp,100*sizeof(int),SEEK_SET)!=0) /* Nos posicionamos
                                                    en el archivo */
      {
            fclose(fp); /* Cerramos el archivo */
            fprintf(stderr,"\nError posicionando en el archivo\n");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      if (fread(&valor,sizeof(int),1,fp)!=1) /* Leemos el valor */
      {
            fclose(fp); /* Cerramos el archivo */
            fprintf(stderr,"\nError leyendo el archivo\n");
            /* Mostramos un mensaje */
            exit(0); /* Salimos del programa */
      }
      fclose(fp); /* Cerramos el archivo */
      return 0; /* Terminamos el programa */
}
```

1.3 Bases de datos.

Antes de proceder a realizar una introducción a las bases de datos, veremos su antecedente más inmediato, los sistemas de gestión de archivos, programa o conjunto de programas que se encargan de gestionar un conjunto de archivos de datos.

1.3.1 Sistemas de gestión de archivos.

En una aplicación convencional con archivos, éstos se diseñan siguiendo las instrucciones de los correspondientes programas. Esto es, una vez planteado se decide si debe existir ó no archivos, cuántos deben ser, qué organización contendrá cada uno, qué programas actuarán sobre ellos y cómo lo harán. Esto tiene la ventaja, en principio, de que los programas son bastante eficientes, ya que la estructura de un archivo esta pensada “para el programa” que lo va a usar. Sin embargo, esta forma de actuar está llena de graves inconvenientes. Por un lado, los programas que se realizan con posterioridad a la creación de un archivo pueden ser muy lentos, al tener que usar una

organización pensada y creada “a la medida” de otro programa previo. Por otra parte, si se toma la decisión de crear nuevos archivos para los programas que se han de realizar, se puede entrar en un proceso de degeneración de la aplicación, ya que:

- Gran parte de la información aparecerá duplicada en más de un archivo (redundancia) ocupando la aplicación más espacio del necesario.

- Al existir la misma información en varios archivos, los procesos de actualización se complican de forma innecesaria, dando lugar a una propagación de errores.

- Se corre el riesgo de tener datos incongruentes entre los distintos archivos. Por ejemplo, tener dos domicilios diferentes de la misma persona en dos archivos distintos (por estar uno actualizado y el otro no).

En estas aplicaciones convencionales con archivos, el conocimiento acerca del contenido de un archivo (qué datos contiene y como están organizados) esta incorporado a los programas de aplicación que utilizan el archivo. Por ejemplo, en la figura 1.3.1.1 se puede ver una aplicación de nóminas de una empresa donde cada uno de los programas que procesan el archivo maestro de empleados contienen una *descripción de archivo* (DA) que describe la composición de los datos del archivo. Si la estructura de los datos cambiaba, todos los programas que accedían al archivo tenían que ser modificados. Como el número de archivos y programas crecía con el tiempo, todo el esfuerzo de un departamento se perdía en mantener aplicaciones existentes en lugar de desarrollar otras nuevas.

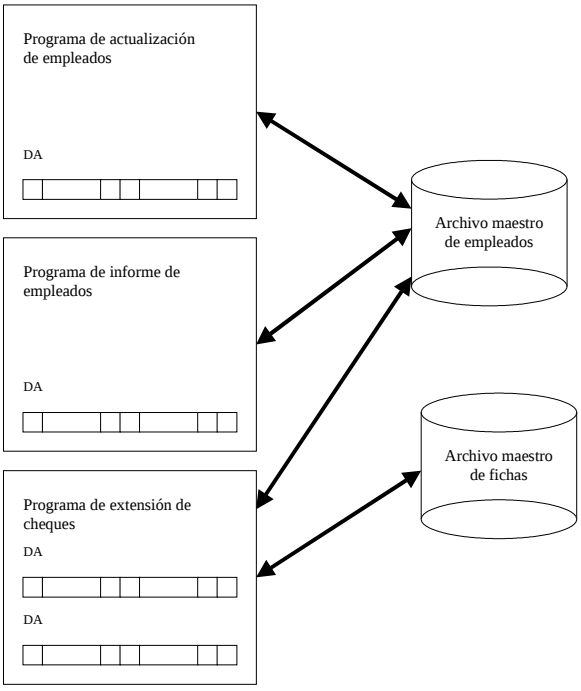

Figura 1.3.1.1: Ejemplo de utilización de sistema de gestión de archivos.

Los problemas de mantener grandes sistemas basados en archivos condujeron a finales de los sesenta al desarrollo de los sistemas de gestión de bases de datos. La idea detrás de estos sistemas es sencilla: tomar la definición de los contenidos de un archivo y la estructura de los programas individuales, y almacenarla, junto con los datos, en una base de datos. Utilizando la información de la base de datos, el sistema gestor de la base

de datos (SGBD o DBMS en inglés) que la controla puede tomar un papel mucho más activo en la gestión de los datos y en los cambios a la estructura de la base de datos.

1.3.2 Bases de datos.

Las bases de datos surgen como alternativa a los sistemas de archivos, intentando eliminar o al menos reducir sus inconvenientes. Podemos definir una base de datos de la siguiente forma:

Una base de datos es un sistema formado por un conjunto de datos y un paquete software para la gestión del mismo, de tal modo que se controla el almacenamiento de datos redundantes, los datos resultan independientes de los programas que los usan, se almacenan las relaciones entre los datos junto con éstos y se puede acceder a los datos de diversas formas.

En una base de datos se almacenan las relaciones entre datos junto a los datos. Esto, y el utilizar como unidad de almacenamiento el campo además del registro, es el fundamento de la independencia con los programas de datos. Los requisitos que debe cumplir un buen sistema de base de datos son:

- *Acceso múltiple.* Diversos usuarios pueden acceder a la base de datos, sin que se produzcan conflictos ni visiones incoherentes.

- *Utilización múltiple.* Cada usuario podrá tener una imagen o visión particular de la estructura de la base de datos.

- *Flexibilidad.* Se podrán usar distintos métodos de acceso, con tiempos de respuesta razonablemente pequeños.

- *Confidencialidad y seguridad.* Se controlará el acceso a los datos (incluso a nivel de campo), impidiéndoselo a los usuarios no autorizados.

- *Protección contra fallos.* Deben existir mecanismos concretos de recuperación en caso de fallo de la computadora.

- *Independencia física.* Se puede cambiar el soporte físico de la base de datos sin que esto repercuta en la base de datos ni en los programas que la usan.

- *Independencia lógica.* Capacidad para que se puedan modificar los datos contenidos en la base, las relaciones existentes entre ellos o incluir nuevos datos, sin afectar a los programas que los usan.

- *Redundancia controlada.* Los datos se almacenan una sola vez.

- *Interfaz de alto nivel.* Existe una forma sencilla y cómoda de utilizar la base, al menos se cuenta con un lenguaje de programación de alto nivel, que facilita la tarea.

- *Interrogación directa (“query”).* Existen facilidades para que se pueda tener acceso a los datos de forma conversacional.

En una base de datos se almacena información de una serie de objetos o elementos. Estos objetos reciben el nombre de *entidades*. Entidad es cualquier ente sobre el que se almacena información. Así, en una base de datos académicos podrá haber información de las entidades alumno, profesor, asignatura, centro, plan de estudios, curso, etc. En una base de datos comerciales de una empresa aparecerán las entidades cliente, producto, vendedor, etc. De cada entidad se almacenan una serie de datos que se denominan *atributos* de la entidad. Puede ser atributo de una entidad cualquier característica o propiedad de está que se considere relevante para la aplicación. Así son atributos de la entidad alumno para una aplicación administrativa el DNI, apellido y nombre, sexo, fecha de nacimiento, etc.

Entidades y atributos son conceptos abstractos. En una base de datos, aunque la tecnología evoluciona constantemente, la información de cada entidad se almacena en *registros*, y cada atributo en *campos* de dicho registro, de forma análoga al almacenamiento en archivos. Sin embargo, cada entidad necesita registros con una estructura específica, mientras que en un archivo, todos los registros tienen la misma estructura. Esto es, en una base de datos hay diferentes tipos de registros, uno por entidad. Normalmente se reserva el nombre "registro" para especificar un "tipo de registro", usándose *ocurrencia de registro* para especificar cada una de las apariciones de ese registro en la base de datos. Con esta terminología se puede decir que en la base de datos de uso para una empresa, hay un registro de vendedor y tantas "ocurrencias" de dicho registro como vendedores tenga la empresa.

Normalmente no es necesario conocer los valores de todos los atributos de una entidad para determinar si dos elementos son iguales. Por lo general, es suficiente con conocer el valor de uno o varios atributos para identificar un elemento. Pues bien, diremos que un conjunto de atributos de una entidad es un *identificador* de dicha entidad si el valor de dichos atributos determina de forma unívoca cada uno de los elementos de dicha entidad y no existe ningún subconjunto de él que sea identificador de la entidad. Por ejemplo, en la entidad alumno, el atributo DNI es un identificador de esa entidad.

Frecuentemente es necesario buscar una ocurrencia en un registro de una base de datos, conociendo el valor de uno o varios campos. Para que esta operación sea rápida, estos campos deben estar definidos en la base de datos como *clave* de búsqueda de dicho registro. En general, podemos decir que una clave es un campo o conjunto de campos, cuyos valores permiten localizar de forma rápida ocurrencias de un registro. La clave puede corresponderse con un identificador de la entidad. Si esto no ocurre, podrá haber varias ocurrencias de registro con el mismo valor de clave. Se dice entonces que la clave admite duplicados.

En una base de datos se almacena, además de las entidades, las *relaciones* existentes entre ellas. Así, por ejemplo, en la base de datos académicos antes citada hay relaciones entre las entidades curso y alumnos, alumnos y profesores, profesores y asignaturas, etc.

1.4 Clasificación de las bases de datos.

Las bases de datos se clasifican tradicionalmente en tres grupos: *jerárquicas, en red* y *relacionales*. Las dos primeras se diferencian en los tipos de relaciones que permiten. Puede decirse que la estructura jerárquica es un caso particular de la estructura en red. Cualquier esquema que se cree para una base de datos jerárquica se puede utilizar para una base en red. Las bases de datos relacionales son conceptualmente distintas, pues en ellas las relaciones se almacenan y manipulan de forma completamente distinta. Veamos a continuación con más detalle los tres grupos de bases de datos.

1.4.1 Base de datos jerárquicas.

Una de las aplicaciones más importantes de los sistemas de gestión de base de datos primitivos era el planeamiento de la producción de empresas de facturación. Si un fabricante de automóviles decidía producir 10.000 unidades de un modelo de coche y 5.000 unidades de otro modelo, necesitaba saber cuántas piezas pedir a sus proveedores. Para responder a la cuestión, el producto (un coche) tenía que descomponerse en ensamblajes (motor, chasis, etc.), que a su vez se descomponían en sub-ensamblajes (válvulas, cilindros, bujías, etc.) y luego en sub-sub-ensamblajes, etc. El manejo de esta lista de piezas, conocido como una *cuenta de materiales*, era un trabajo a medida para los ordenadores.

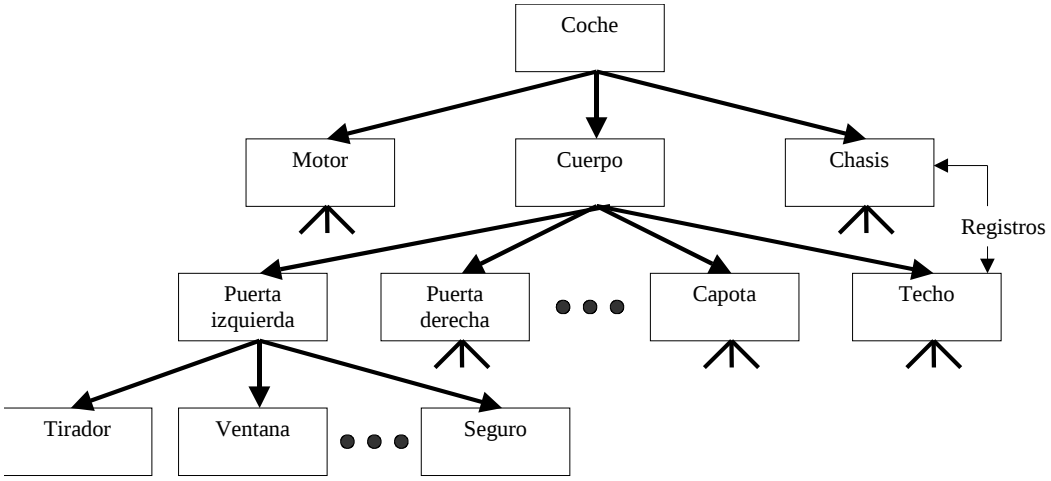

Figura 1.4.1.1: Ejemplo de base de datos jerárquica.

La cuenta de materiales para un producto tenía una estructura jerárquica natural. Para almacenar estos datos, se desarrollo el modelo de datos *jerárquico* (figura 1.3.1.1). En este modelo, cada *registro* de la base de datos representa una pieza específica. Los registros tenían relaciones *padre/hijo*, que ligaba cada pieza a su sub-pieza, y así sucesivamente.

Para acceder a los datos en la base de datos, un programa podría:

- Hallar una pieza particular mediante su número (como por ejemplo la puerta izquierda).
- Descender al primer hijo (el tirador de la puerta).

- Ascender hasta su padre (el cuerpo).

- Moverse de lado hasta el siguiente hijo (la puerta derecha).

La recuperación de los datos en una base de datos jerárquica requería, por tanto, navegar a través de los registros, moviéndose hacia arriba, hacia abajo y hacia los lados un registro cada vez.

Uno de los sistemas de gestión de base de datos jerárquica más populares fue el Information Management System (IMS) de IBM, introducido en 1968. Las ventajas del IMS y su modelo jerárquico son las siguientes:

- Estructura simple. La organización de una base de datos IMS era fácil de entender. La jerarquía de la base de datos se asemejaba al diagrama de organización de una empresa o un árbol familiar.

- Organización padre/hijo. Una base de datos IMS era excelente para representar relaciones padre/hijo, tales como “A es pieza de B” o “A es propiedad de B”.

- Rendimiento. IMS almacenaba las relaciones padre/hijo como punteros físicos de un registro de datos a otro, de modo que el movimiento a través de la base de datos era rápido. Puesto que la estructura era sencilla, IMS podía colocar los registros padre e hijo cercanos unos a otros en el disco, minimizando la entrada/salida de disco.

IMS sigue siendo el DMBS más ampliamente instalado en los maxi-computadores IBM, utilizándose en un 25% aproximadamente.

1.4.2 Bases de datos en red.

La estructura sencilla de una base de datos jerárquica se convertía en una desventaja cuando los datos tenían estructura más compleja. En una base de datos de procesamiento de pedidos, por ejemplo, un simple pedido podría participar en tres relaciones padre/hijo diferentes, ligando el pedido al cliente que lo remitió, al vendedor que lo aceptó y al producto ordenado, tal como se muestra en la figura 1.4.2.1. La estructura de datos simplemente no se ajustaría a la jerarquía estricta de IMS.

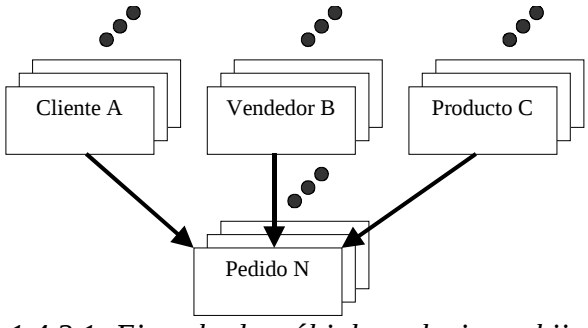

Figura 1.4.2.1: Ejemplo de múltiples relaciones hijo/padre.

Para manejar aplicaciones tales como el procesamiento de pedidos, se desarrolló un nuevo modelo de datos en red. El modelo de datos en red extendía el modelo jerárquico permitiendo que un registro participara en múltiples relaciones padre/hijo, como se muestra en la figura 1.4.2.2. Estas relaciones eran conocidas como *conjuntos* en el modelo de red. En 1971 se publicó un estándar oficial para bases de datos en red, que se conoció como el modelo CODASYL. IBM nunca desarrolló un DBMS en red por sí mismo, sino que extendió el IMS a lo largo de los años. Sin embargo, durante los años setenta, otras compañías de software crearon productos que implementaban el modelo de red, tales como el IDMS de Cullinet, el Total de Cincom y el DBMS Adabas.

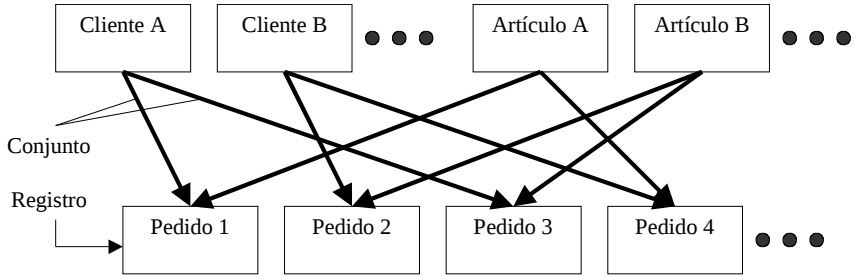

Figura 1.4.2.2: Ejemplo de base de datos en red (CODASYL).

Para un programador, acceder a una base de datos en red era muy similar a acceder a una base de datos jerárquicos. Un programa de aplicación podía:

- Hallar un registro padre específico mediante una clave (como por ejemplo un número de cliente).

- Descender al primer hijo en un conjunto particular (el primer pedido remitido por ese cliente).

- Moverse lateralmente de un hijo al siguiente dentro del conjunto (la orden siguiente remitida por el mismo cliente).

- Ascender desde un hijo a su padre en otro conjunto (el vendedor que aceptó el pedido).

Una vez más el programador tenía que recorrer la base de datos registro a registro, especificando esta vez qué relación recorrer además de indicar la dirección.

Las bases de datos en red tenían varias ventajas:

- Flexibilidad. Las múltiples relaciones padre/hijo permitían a una base de datos en red representar datos que no tuvieran una estructura jerárquica sencilla.

- Normalización. El estándar CODASYL reforzó la popularidad del modelo de red, y los vendedores de mini-computadoras como Digital Equipment Corporation y Data General implementaron bases de datos en red.

- Rendimiento. A pesar de su superior complejidad, las bases de datos en red reforzaron el rendimiento aproximándolo al de las bases de datos jerárquicas. Los conjuntos se representaron mediante punteros a registros de datos físicos, y

en algunos sistemas, el administrador de la base de datos podía especificar la agrupación de datos basada en una relación de conjunto.

Las bases de datos en red tenían sus desventajas también. Igual que las bases de datos jerárquicos, resultaban muy rígidas. Las relaciones de conjunto y la estructura de los registros tenían que ser especificadas de antemano. Modificar la estructura de la base de datos requería típicamente la reconstrucción de la base de datos completa.

Tanto las bases de datos jerárquicas como en red eran herramientas para programadores. Para responder a una pregunta como "Cuál es el producto más popular ordenado por el cliente A", un programador tenía que escribir un programa que recorriera su camino a través de la base de datos. La anotación de las peticiones para informes a medida duraba con frecuencia semanas o meses, y para el momento en que el programa estaba escrito la información que se entregaba con frecuencia ya no merecía la pena.

1.4.3 Bases de datos relacionales.

Las desventajas de los modelos jerárquico y en red condujeron a la propuesta de un nuevo modelo de datos, propuesto por el Dr. Codd en 1970. El modelo relacional era un intento de simplificar la estructura de las bases de datos. Eliminaba las estructuras explícitas padre/hijo de la base de datos y en su lugar representaba todos los datos en la base de datos como sencillas tablas fila/columna de valores de datos. En la figura 1.4.3.1 puede verse una sencilla base de datos relacional.

Tabla ASIGNATURAS

CODIGO	NOMBRE
10000	Tratamiento de datos
10001	Análisis estadístico
10002	Cálculo numérico
…	…

Tabla AULAS

EDIFICIO	NUMERO	CAPACIDAD
E1	11	100
E1	12	120
E2	11	110
…	…	…

Tabla PROFESORES

CODIGO	NOMBRE	APELLIDOS
H0001	Antonio	García García
H0002	Amparo	Pérez Pérez
H0003	Isabel	Fernández Fernández
…	…	…

Figura 1.4.3.1 Ejemplo de base de datos relacional.

Sin embargo, la definición práctica de "¿Qué es una base de datos relacional?" resulta menos clara que la definición matemática recogida en el artículo de Codd de 1970. Por ello el propio Dr. Codd escribió en 1985 un artículo estableciendo doce reglas a seguir por cualquier base de datos que fuera "relacional". Las doce reglas de Codd han sido aceptadas como la definición de un DBMS verdaderamente relacional y se analizaran en el siguiente punto. Sin embargo, una definición más informal es:

Una base de datos relacional es una base de datos en donde todos los datos visibles al usuario están organizados estrictamente como tablas de valores, y en donde todas las operaciones de la base de datos operan sobre estas tablas.

La definición anterior elimina estructura tales como los punteros incorporados de una base de datos jerárquica o en red. Un DBMS relacional puede representar relaciones padre/hijo, pero éstas se representan estrictamente por los valores contenidos en las tablas de la base de datos.

El principio de organización de una base de datos relacional es la *tabla*, una disposición rectangular fila/columna de los valores de datos. Cada tabla de una base de datos tiene un *nombre de tabla* único que identifica sus contenidos. En realidad, cada usuario puede elegir sus propios nombres de tablas sin preocuparse acerca de los nombres elegidos por otros usuarios.

La estructura fila/columna de una tabla puede verse claramente en la figura 1.4.3.2. Cada *fila* horizontal de la tabla PROFESORES representa una única entidad física. Juntas todas las filas de la tabla representan todos los profesores de la universidad. Todos los datos de una fila particular de la tabla se aplican al profesor representado mediante esa fila. Cada fila de una tabla contiene exactamente un valor en cada columna. Una tabla puede contener cualquier número de filas, incluso cero filas, denominándose en tal caso *tabla vacía*. Una tabla vacía sigue teniendo una estructura, impuesta por las columnas, solo que simplemente no contiene datos.

Tabla PROFESORES

CODIGO	NOMBRE	APELLIDOS	CATEGORIA	ANTIGUEDAD
H0001	Antonio	García García	Catedrático	15/02/1983
H0002	Amparo	Pérez Pérez	Ayudante	01/09/1997
H0003	Isabel	Fernández Fernández	Titular	19/04/1991
…	…	…	…	…

Figura 1.4.3.2: Ejemplo de una tabla de una base de datos relacional.

Cada *columna* vertical de la tabla PROFESORES representa un atributo que está almacenado en la base de datos para cada profesor. Por ejemplo la columna CATEGORIA contiene la categoría profesional de cada profesor. Para cada columna de una tabla, todos los valores de esa columna contienen el mismo atributo. El conjunto de valores que una columna puede contener se denomina el *dominio* de la columna. Así, el dominio de la columna NOMBRE es cualquier nombre, mientras que el dominio de la columna CATEGORIA es solo de tres valores, “Catedrático”, “Titular” y “Ayudante” (suponemos que esas son las tres únicas categorías profesionales del personal docente en una universidad).

Cada columna de una tabla tiene un nombre de columna que se escribe generalmente como encabezamiento en la parte superior de la columna. Todas las columnas de una tabla deben tener nombres diferentes, pero no está prohibido que columnas de tablas diferentes tengan nombres idénticos (véanse las tablas ASIGNATURAS y PROFESORES de la figura 1.4.3.1). Una tabla tiene como mínimo una columna. No existe un número máximo de columnas en una tabla, estando este limitado por el producto comercial que se utilice.

Como las filas de una tabla relacional no están ordenadas, no se puede seleccionar una fila específica por su posición en la tabla, sino que debemos usar un identificador, que recordemos son un conjunto de atributos de una entidad que

determinan de forma unívoca cada uno de los elementos de dicha entidad. Este identificador suele ser conocido en la terminología de las bases de datos relacionales como *clave primaria*. En la figura 1.4.3.3 podemos ver dos ejemplos de clave primaria. En uno de ellos la clave primaria esta formada por un solo atributo mientras que en el otro caso esta formada por la combinación de dos atributos.

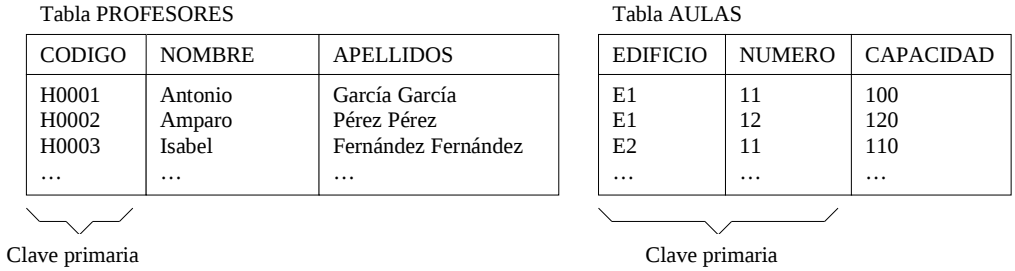

Figura 1.4.3.3: Dos ejemplos de claves primarias en una base de datos relacional.

Una de las principales diferencias entre el modelo relacional y los modelos de datos anteriores es que los punteros explícitos, tales como las relaciones padre/hijo de una base de datos jerárquica, están prohibidos en las bases de datos relacionales. Obviamente estas relaciones siguen existiendo, pero no mediante punteros explícitos sino mediante *valores de datos comunes* almacenados en cada una de las tablas. Por ejemplo, en la figura 1.4.3.4 la tabla ASIGNATURAS contiene la columna PROFESOR, que indica el profesor de la asignatura mediante su clave primaria. Una columna de una tabla cuyo valor coincide con una clave primaria de alguna otra tabla se denomina *clave foránea*. La clave foránea obviamente estará formada por uno o varios atributos según suceda en la clave primaria con la cual esta relacionada. Todas las relaciones de una base de datos relacional están representadas de este modo.

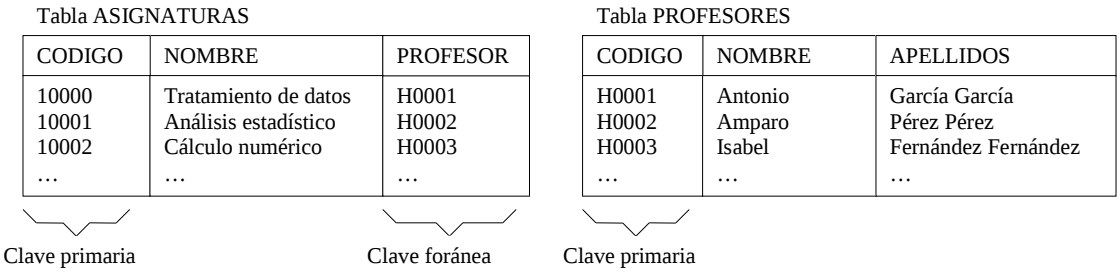

Figura 1.4.3.4: Ejemplo de clave foránea en una base de datos relacional.

1.4.4 Las doce reglas de Codd de definición de una DBMS relacional.

En un artículo de 1985 publicado en Computerworld, el Dr. Codd presentó doce reglas que una base de datos debe obedecer para que sea considerada relacional. Las doce reglas de Codd se han convertido en la definición teórica de una base de datos relacional. Las reglas se derivan del trabajo teórico de Codd sobre el modelo relacional y representan realmente más un objetivo ideal que una definición de una base de datos relacional. Estas doce reglas son:

1. *Regla de información.* Toda la información de una base de datos relacional está representada explícitamente a nivel lógico y exactamente de un modo: Mediante valores en tablas.

2. *Regla de acceso garantizado.* Todos y cada uno de los datos de una base de datos relacional se garantiza que sean lógicamente accesibles recurriendo a una combinación de nombre de tabla, valor de clave primaria y nombre de columna.

3. *Tratamiento sistemático de valores nulo.* Los valores nulos (distinto de la cadena de caracteres vacía o de una cadena de caracteres en blanco y distinta del cero o de cualquier otro número) se soportan en los DBMS completamente relaciones para representar la falta de información y la información inaplicable de un modo sistemático e independiente del tipo de datos.

4. *Catálogo en línea dinámico basado en el modelo relacional.* La descripción de la base de datos se representa a nivel lógico del mismo modo que los datos ordinarios, de modo que los usuarios autorizados puedan aplicar a su interrogación el mismo lenguaje relacional que aplican a los datos regulares.

5. *Regla de sub-lenguaje completo de datos.* Un sistema relacional puede soportar varios lenguajes y varios modos de uso terminal (por ejemplo, el modo de rellenar con blancos). Sin embargo, debe haber al menos un lenguaje cuyas sentencias sean expresables mediante alguna sintaxis bien definida, como cadenas de caracteres, y que sea completa en cuanto al soporte de todos los puntos siguientes:

    - Definición de datos.
    - Definición de vista.
    - Manipulación de datos (interactiva y por programa).
    - Restricciones de integridad.
    - Autorización.
    - Fronteras de transacciones (comienzo, cumplimiento y vuelta atrás).

6. *Regla de actualización de vista.* Todas las vistas que sean teóricas actualizables son también actualizables por el sistema.

7. *Inserción, actualización y supresión de alto nivel.* La capacidad de manejar una relación de base de datos o una relación derivada como un único operando se aplica no solamente a la recuperación de datos, sino también a la inserción, actualización y supresión de los datos.

8. *Independencia física de los datos.* Los programas de aplicación y las actividades terminales permanecen lógicamente inalterados cualquiera que sean los cambios efectuados ya sea a las representaciones de almacenamiento o a los métodos de acceso.

9. *Independencia lógica de los datos.* Los programas de aplicación y las actividades terminales permanecen lógicamente inalterados cuando se efectúen

sobres las tablas de base cambios preservadores de la información de cualquier tipo que teóricamente permita alteraciones.

10. *Independencia de integridad.* Las restricciones de integridad específicas para una base de datos relacional particular deben ser definibles en el sub-lenguaje de datos relacional y poder ser almacenadas en el catálogo, no en los programas de aplicación.

11. *Independencia de distribución.* Un DBMS relacional tiene independencia de distribución.

12. *Regla de no subversión.* Si un sistema relacional tiene un lenguaje de bajo nivel (un solo registro a la vez), ese bajo nivel no puede ser utilizado para subvertir o suprimir las reglas de integridad y las restricciones expresadas en el lenguaje relacional de nivel superior (múltiples registros a la vez).

La regla 1 es la definición informal de una base de datos relacional. La regla 2 refuerza la importancia de las claves primarias para localizar datos en la base de datos. El nombre de la tabla localiza la tabla correcta, el nombre de la columna encuentra la columna correcta y el valor de la clave primaria encuentra la fila que contiene un dato individual de interés. La regla 3 requiere soporte para falta de datos mediante el uso de valores NULL.

La regla 4 requiere que una base de datos relacional sea auto-descriptiva. En otras palabras, la base de datos debe contener ciertas *tablas de sistema* cuyas columnas describan la estructura de la propia base de datos.

La regla 5 ordena la utilización de un lenguaje de base de datos relacional. El lenguaje debe ser capaz de soportar todas las funciones básicas de un DBMS (creación de una base de datos, recuperación y entrada de datos, implementación de la seguridad de la base de datos, etc.).

La regla 6 trata de las vistas, que son tablas virtuales utilizadas para dar a diferentes usuarios de una base de datos diferentes vistas de su estructura. Es una de las reglas más difíciles de implementar en la práctica.

La regla 7 refuerza la naturaleza orientada a conjuntos de una base de datos relacional. Requiere que las filas sean tratadas como conjuntos en operaciones de inserción, supresión y actualización. La regla esta diseñada para impedir implementaciones que sólo soportan la modificación o recorrido fila a fila de la base de datos.

La regla 8 y la regla 9 aíslan al usuario o al programa de aplicación de la implementación de bajo nivel de la base de datos. Especifican que las técnicas específicas de acceso a almacenamiento utilizadas por el DBMS, e incluso los cambios a la estructura de las tablas en la base de datos, no deberían afectar a la capacidad del usuario de trabajar con los datos.

La regla 10 dice que el lenguaje de base de datos debería soportar las restricciones de integridad que restringen los datos que pueden ser introducidos en la base de datos y las modificaciones que pueden ser efectuadas en ésta.

La regla 11 dice que el lenguaje de base de datos debe ser capaz de manipular datos distribuidos localizados en otros sistemas informáticos. Por último, la regla 12 impide "otros caminos" en la base de datos que pudieran subvertir su estructura relacional y su integridad.

Ningún DBMS relacional actualmente disponible satisface totalmente las doce reglas de Codd. De hecho, se elaboran pruebas para productos DBMS comerciales, que muestran lo bien o mal que éstos satisfacen cada una de las reglas.

1.5 Ejercicios

1- Un archivo contiene almacenados, en formato de texto, la medición de la temperatura durante un día entero, en intervalos de una hora (24 valores). Realizar un programa en C que calcule el valor máximo y mínimo de la temperatura, así como la desviación media, cuya formula es: $D_m = \frac{0}{n}\sum_{i=0}^{n}\left|x_i - \bar{x}\right|$.

2- Un archivo, en formato de texto, contiene un número indeterminado de líneas, cada una de las cuales contiene un número entero de valor comprendido entre 0 y 99. Desarrollar un programa en C que cuente las apariciones de cada número en el archivo, mostrando el resultado final por pantalla.

3- Un archivo, en formato de texto, contiene un número indeterminado de líneas, cada una de las cuales contiene un número en coma flotante (float). Desarrollar un programa en C que cree un archivo en formato texto que almacene en cada línea la parte entera del número y a continuación separada por una coma, los tres primeros dígitos decimales.

4- Un archivo de texto contiene un número indeterminado de números enteros, cada uno en una línea. Escribir un programa en C que almacene en un archivo, en formato de texto, líneas que contengan el número entero y, separado por una coma, la suma de los números comprendidos entre 1 y el entero leído, cuya formula es: $(n+1)*n/2$.

5- Tenemos dos archivos de texto formados por el mismo número de líneas y deseamos comparar los archivos, línea a línea, mostrando las líneas en que difieren. Realizar un programa en C que indique, por pantalla, el número de línea/s en que ambos archivos difieren.