TEMA 4: ARCHIVOS Y BASES DE DATOS

PARTE I: ARCHIVOS O FICHEROS

Introducción. Concepto de Fichero

Hasta este momento hemos estado trabajando con información que estaba situada en memoria. Cuando arrancamos un programa, se genera la información (o se pide al usuario) y se procesa obteniendo unos resultados. Cuando el programa termina, la información guardada en memoria se pierde. Si necesitamos volver a procesar esta información tendremos que proporcionársela de nuevo al programa.

Por tanto, estas estructuras de datos que utilizan la memoria principal tienen dos limitaciones importantes:

1. Los datos desaparecen cuando el programa termina.
2. La cantidad de los datos no puede ser muy grande debido a la limitación de la memoria principal.

Una manera de evitar esto es guardando la información de alguna manera 'permanente' para poder acceder de nuevo a ella cuando sea necesario. Por eso existen también estructuras especiales que utilizan memoria secundaria: los ficheros.

Por tanto, un fichero va a ser una estructura donde podremos guardar información de forma permanente tal que se pueda recuperar en cualquier momento y además, se tratará de una estructura dinámica en el sentido de que su tamaño puede variar durante la ejecución del programa dependiendo de la cantidad de datos que contenga.

De esta forma podemos ver que serán necesarias unas operaciones básicas para poder escribir la información en el fichero y para poder recuperarla y ponerla en memoria, así como unas funciones para abrir el fichero y para cerrarlo.

Tipos de ficheros

En C/C++ existen básicamente dos tipos de ficheros, en función de la manera en que se guarda la información dentro de él: Los fichero de tipo texto y los ficheros binarios.

Ficheros de tipo texto

En los ficheros de tipo texto lo que guardaremos será la información en forma de caracteres, tal y como se mostraría por pantalla. Por ejemplo, el valor entero '**`25`**' se transformará en los caracteres '**`2`**' y '**`5`**'.

Por este motivo las operaciones de escritura y lectura de ficheros de este tipo, serán similares a las utilizadas para escribir en pantalla y leer de teclado.

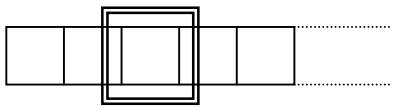
Ficheros de tipo binario

La información se guarda en el fichero tal y como está en memoria principal, es decir, compuesta por unos y ceros. Por ejemplo, el valor entero '**`25`**' se guardará como '**`00011001`**'.

Los ficheros binarios contienen secuencias de elementos de un tipo determinado de datos almacenados exactamente igual que en la memoria, por tanto sería similar a un vector. Para acceder a estos datos necesitaremos unas funciones específicas que serán diferentes a las de los ficheros de texto.

Tipos de acceso a ficheros

Al estar en memoria secundaria, no todos los elementos del fichero son accesibles de forma inmediata. Solamente se puede acceder cada vez a un único elemento del fichero, que se denomina *ventana del fichero*.



Dependiendo de cómo se desplaza la ventana por el fichero, podemos distinguir dos tipos de acceso:

- Acceso secuencial: La ventana del fichero sólo puede moverse hacia delante a partir del primer elemento y siempre de uno en uno.
- Acceso directo: La ventana del fichero se puede situar directamente en cualquier posición del fichero. Es un acceso similar al utilizado en los arrays.

El acceso directo suele ser más eficiente, ya que para leer un dato no hace falta leer antes todos los anteriores.

La razón por la que existe el acceso secuencial es que existen dispositivos de memoria secundaria que sólo admiten acceso secuencial (como por ejemplo las cintas). Además, el acceso secuencial se utiliza también en dispositivos que admiten acceso directo cuando queremos leer los elementos de forma secuencial, ya que este acceso es más sencillo.

Ficheros lógicos y ficheros físicos

En un lenguaje de programación, los ficheros son un tipo de dato más, y un fichero concreto se referencia utilizando una variable de tipo fichero. Es lo que denominamos *fichero lógico*.

En C++ existen dos tipos de datos básicos para declarar ficheros:

```
ifstream // Para declarar ficheros de entrada (in) (de donde se lee)
ofstream // Para declarar ficheros de salida (out) (donde se escribe)
```

Para utilizar estos tipos hay que incluir antes el fichero de cabecera `<fstream.h>`.

Ejemplo:

```
ofstream f;
```

Esta sentencia nos declara una variable (fichero lógico) de tipo fichero de salida.

Pero esta variable, para que nos sea de utilidad tiene que estar asociada con un fichero "real", es decir, por un fichero reconocido por el sistema operativo (por ej. "`datos.txt`") puesto que al final será el sistema operativo quien realice la escritura o lectura de ese fichero. Este fichero es lo que se denomina *fichero físico*.

Para relacionar el fichero lógico con el fichero físico necesitamos realizar una operación de *apertura del fichero*.

Operaciones sobre ficheros

Apertura del fichero:

Como ya se ha comentado, un fichero es, físicamente, un conjunto de bits guardados en un determinado soporte (generalmente magnético) y al que se accede mediante una referencia al soporte (unidad) y a la localización sobre el soporte (por ejemplo: Sector, cilindro, cara,... en el caso de soporte magnético sobre disco.)

La definición lógica del fichero es un conjunto de información útil para el usuario, guardada en una cierta carpeta o directorio y al que se puede acceder a través de un cierto camino o 'path' y un nombre asignado al fichero.

La manera de relacionar ambos conceptos es a través del sistema operativo, y en los programas realizados por los usuarios a través de la operación de *apertura de ficheros*.

La operación de apertura de fichero utiliza la información que conoce el usuario para relacionarla con la descripción física y preparar un descriptor que utilizará el programa para acceder a la información contenida en el fichero.

En C++ esta operación se realiza con la instrucción open:

```
nombre_fichero_logico.open (nombre_fichero_fisico);
```

Ejemplo:

```
f.open("datos.txt");
```

A partir de ese momento ya podemos utilizar el fichero.

Hay que tener en cuenta que, por defecto, los ficheros de entrada (`ifstream`) se abren sólo para lectura poniendo la ventana del fichero en el primer elemento del fichero. Además el fichero debe existir, si no se genera un error.

Por defecto, los ficheros de salida (`ofstream`) se abren sólo para escritura creando el fichero nuevo. Si el fichero ya existía es borrado.

Para modificar el modo de apertura se puede añadir un parámetro más a la instrucción open, que indica el modo de apertura. Sólo vamos a ver el modo ios::app que sirve para abrir un fichero de salida en modo añadir, de manera que no borra el fichero y la ventana del fichero se pone después del último elemento. Si el fichero no existe da error.

```
nombre_fichero_logico.open (nombre_fichero_fisico, ios::app);
```

Ejemplo:

```
f.open("datos.txt", ios::app);
```

Para saber si la apertura del fichero ha dado error se utiliza el operador ! aplicado al fichero:

Ejemplo:

```
if (!f)
    cout << "Error abriendo el fichero" << endl;
```

Esta condición se debe poner siempre que abramos un fichero, puesto que si ha habido un error, no se podrá realizar ninguna operación con él.

Cierre del fichero:

Cuando se han acabado de realizar todas las operaciones, SIEMPRE hay que **cerrar** el fichero. Esta operación destruye las estructuras que se han creado para abrirlo (tanto del programa como del sistema operativo). También actualiza totalmente el fichero, escribiendo toda la información que pudiera quedar en el buffer (ya que normalmente la información pasa a través de un buffer).

Para cerrar el fichero se utiliza la instrucción `close`:

```
nombre_fichero_logico.close ();
```

Ejemplo:

```
f.close();
```

Escritura en fichero:

Las funciones de escritura en fichero se utiliza para poner información que está en la memoria del ordenador, en un fichero, para poder utilizarla posteriormente. La manera de escribir en el fichero dependerá del tipo de fichero en el que queremos guardar la información: texto o binario.

En los siguientes puntos desarrollaremos los dos casos.

Lectura de fichero:

Las funciones de lectura en ficheros se utilizan para recuperar información guardada en fichero y ponerla en memoria para poder trabajar directamente con ella.

Al igual que en la escritura dependerá del tipo de fichero (si de texto o binario) las funciones son unas u otras.

En cualquier caso, para poder leer información de un fichero, hay que saber exactamente cómo ha sido guardada en él.

Otras instrucciones:

El carácter **`EOF`** es el carácter de final de fichero. Para poder averiguar si se ha llegado a leer este carácter, tenemos la función **`eof ()`** que detecta si se ha llegado al final de fichero.

eof () nos devuelve verdadero si estamos en el final de fichero. Falso en cualquier otro caso.

Procesamiento de un fichero

Siempre que trabajemos con ficheros, la primera tarea que tendremos que realizar será abrir el fichero y relacionarlo adecuadamente con un descriptor, que será, a partir de ese momento, la referencia que utilizaremos para trabajar con el fichero.

En ese momento ya se podrá realizar cualquier operación con los ficheros y finalmente lo cerraremos.

Resumiendo, para trabajar con ficheros deberemos seguir el siguiente esquema:

Apertura de Fichero → Operaciones de lectura o escritura → Cierre del fichero

Operaciones de lectura y de escritura en ficheros de texto

Las **operaciones** que se pueden realizar sobre un fichero de texto son exactamente las mismas que se pueden realizar sobre `cin` (para ficheros de entrada) y `cout` (para ficheros de salida). De hecho `cin` y `cout` son ficheros predefinidos, que están asociados con la entrada estándar y la salida estándar del sistema operativo que normalmente son el teclado y la pantalla respectivamente.

Ejemplo:

Para escribir el numero 10 en el fichero f:

```
f << 10;
```

Para escribir un string:

```
f << "Hola";
```

Escritura mediante <<

Es decir para poner el contenido de una variable x (de cualquier tipo simple) en un fichero:

```
f << x;
```

Lectura mediante >>

Mientras que para leer algo de un fichero y almacenarlo en un variable x:

```
f >> x;
```

Ejemplo: ***Programa para escribir los números del 1 al 10 en el fichero datos.txt***

```
#include<iostream.h>
#include<fstream.h>

int main()
{
    ofstream f;    //fichero de salida
    int i;

    f.open("datos.txt");  // Apertura del fichero
    if(!f)
        cout << "Error abriendo el fichero" << endl;
    else
    {  // Operaciones sobre el fichero
        for(i = 1; i <= 10; i++)
           f << i << endl; //escribe el contenido de i y salta 1 línea
        f.close();  // Cierre del fichero
    }
    return 0;
}
```

Ejemplo: ***Programa para leer los 10 números enteros del fichero y mostrarlos por pantalla.***

```
#include<iostream.h>
#include<fstream.h>

int main()
{
    ifstream f;
    int i, dato;

    f.open("datos.txt");
    if(!f)
        cout << "Error abriendo el fichero" << endl;
    else
    {
        for(i = 1; i <= 10; i++)
        {
            f >> dato;
            cout << dato << endl;
        }
        f.close();
    }
    return 0;
}
```

Sin embargo, lo normal es que no sepamos cuantos elementos vamos a leer, sino que queremos leer hasta que lleguemos al final del fichero. Para ello se puede utilizar un bucle while de la siguiente forma:

```
while (f >> dato)
    cout << dato << endl;
```

Cuando una instrucción para leer de fichero acaba con éxito, devuelve *cierto*, y cuando se produce algún tipo de error (entre los que se incluye llegar al final del fichero), devuelve *falso*. De esta forma, la instrucción anterior leerá, mientras sea posible, todos los números del fichero.

El programa anterior modificado de forma que incluya la instrucción while quedará de la siguiente forma:

Ejemplo: ***Programa para leer los números enteros de un fichero y mostrarlos por pantalla.***

```
#include<iostream.h>
#include<fstream.h>

int main()
{
    ifstream f;
    int dato;

    f.open("datos.txt");
    if(!f)
        cout << "Error abriendo el fichero" << endl;
    else
    {
        while(f >> dato)
            cout << dato << endl;
        f.close();
    }
    return 0;
}
```

Esta forma de leer del fichero se puede utilizar con cualquier tipo de lectura: con >> y también con `getline` si queremos leer hasta el salto de línea.

getline (variable_ifstream , variable_tipo_string);

Lectura mediante `get()`

Si queremos leer el fichero carácter a carácter, lo más normal será utilizar el método **get( )**, sin embargo éste no devuelve cierto o falso para saber si se ha podido leer con éxito, puesto que tiene que devolver el carácter que ha leído. La forma de leer un fichero con get() será por tanto ligeramente distinta a la que hemos visto.

Ejemplo: ***Programa para leer los caracteres de un fichero y mostrarlos por pantalla.***

```
#include<iostream.h>
#include<fstream.h>

int main()
{
    ifstream f;
    char dato;

    f.open("datos.txt");
    if(!f)
        cout << "Error abriendo el fichero" << endl;
    else
    {
        dato = f.get();
        while(! f.eof())
        {
            cout << dato << endl;
            dato = f.get();
        }
        f.close();
    }
    return 0;
}
```

El método eof() es cierto si el dato que hemos leído era un final de fichero y falso en caso contrario. Por esta razón hay que leer primero el carácter y después comprobar si hemos llegado a final de fichero.

Lectura de estructuras

Cuando queremos leer datos simples de un fichero, se puede realizar fácilmente introduciendo la lectura dentro de la condición de un bucle while. Sin embargo, para leer una estructura se deben leer primero cada uno de sus campos antes de considerar la lectura correcta, por lo que para poder efectuar la lectura en la condición del while, habrá que definir una función que realice dicha lectura y devuelva **true** si se ha podido realizar sin errores y **false** en caso contrario. Veamos primero como sería la lectura sin usar funciones y en el punto siguiente modificaremos el programa, incluyendo funciones, de manera que sea más correcta su implementación.

Ejemplo:

```
#include <iostream.h>
#include <fstream.h>
#include <string>

struct Telefono
{
    string nombre;
    int telefono;
};

int main(void)
{
    Telefono tel;
    ifstream f;

    f.open("guia.dat");
    if(!f)
        cout << "Error abriendo el fichero" << endl;
    else
    {
        getline(f, tel.nombre); //Lee del fichero una línea de texto
        f >> tel.telefono; //Lee el entero que corresponde al número
        f.ignore(); //Ignora un carácter del buffer que es el '\n'

        while(!f.eof())
        {
            cout << tel.nombre << endl << tel.telefono << endl;

            getline(f, tel.nombre); //Lee del fichero
            f >> tel.telefono; //Lee el entero
            f.ignore(); //Ignorar el '\n'
        }
        f.close();
    }
    return 0;
}
```

Paso como parámetro de un fichero a una función

Para terminar conviene observar que los ficheros se han de pasar siempre como parámetros por referencia, no importa si los vamos a modificar o no. Además para este caso hemos usado la función **getline**, que nos permite leer una línea de fichero completa. Es evidente que para que esto funciones el fichero de datos debe incluir en cada línea un dato distinto (en una línea un nombre, en la siguiente el teléfono asociado y así sucesivamente). Si modificamos el programa anterior incluyendo una función `F_IntroTel` que se encargará de leer la información del fichero y devolvernos un booleano indicando si hemos llegado al final de fichero o no.

Ejemplo:

```
#include <iostream.h>
#include <fstream.h>
#include <string>

struct Telefono
{
    string nombre;
    int telefono;
};

bool F_IntroTel(ifstream & f, Telefono & tel);  //prototipos
void EscribeTel(Telefono tel);

int main(void)
{
    Telefono tel;
    ifstream guia;

    guia.open("guia.dat");
    if(!guia)
        cout << "Error abriendo el fichero" << endl;
    else
    {
        while(F_IntroTel(guia, tel))
        {
            EscribeTel(tel);
            cout << endl;
        }
        guia.close();
    }
    return 0;
}

void EscribeTel(Telefono tel);
{
    cout << tel.nombre << endl << tel.telefono << endl << endl;
}

bool F_IntroTel(ifstream & f, Telefono & tel)
{
    getline(f, tel.nombre); //Lee del fichero una línea de texto
    f >> tel.telefono; //Lee el entero que corresponde al número
    f.ignore(); //Ignora un carácter del buffer que es el '\n'
    return (!f.eof());
}
```

Operaciones de lectura y de escritura en ficheros binarios

Antes de describir las operaciones de lectura y escritura en ficheros binarios recordemos que una característica importante de este tipo de ficheros es que podíamos acceder directamente a un dato sin tener que leer los anteriores. Para poder realizar este acceso directo, necesitaremos por tanto unas funciones específicas.

Método de acceso directo

El acceso directo consiste en poder mover la ventana del fichero a la posición del fichero que queramos, sin necesidad de leer todas las anteriores. En C++ la ventana del fichero corresponde únicamente a un carácter y se mueve carácter a carácter, por tanto la posición corresponderá al número de caracteres anteriores y NO al número de datos. La primera posición del fichero es siempre la posición 0.

Ejemplo: Supongamos el siguiente fichero f con su contenido

Fichero f			
HOLA\n	Carácter en la posición 0:	H	
PEPE\n	Carácter en la posición 3:	A	
	Carácter en la posición 6:	P	(en DOS o WINDOWS)[1]

Los métodos utilizados para el acceso directo son los siguientes:

Para `ifstream`:

```
nombre_fichero_logico.seekg(pos)
nombre_fichero_logico.tellg()
```

Para `ofstream`:

```
nombre_fichero_logico.seekp(pos)
nombre_fichero_logico.tellp()
```

`seekp(pos)` o `seekg(pos)` coloca la ventana del fichero en la posición `pos`. `tellg()` o `tellp()` devuelven un entero que indica la posición actual de la ventana del fichero.

Ejemplo:

```
// Colocar la ventana del fichero al principio del fichero
    f.seekg(0);

// Ir a la posición 6 (7º carácter) de f
    f.seekg(6);
    cout << f.get();        ->  P
    cout << f.tellg();      ->  7
```

El método tellg ha devuelto 7 porque al leer el carácter, la ventana se ha movido al siguiente carácter.

Lectura y escritura de ficheros binarios

En C++ la lectura en binario se realiza mediante el método `read` y la escritura mediante el método `write`. En ambos casos hay que pasar como primer parámetro un puntero de tipo char a la variable a leer o escribir, y como segundo dato el numero de bytes a leer o escribir.

[1] En el sistema operativo Windows, el salto de línea se escribe en fichero utilizando 2 caracteres, concretamente los correspondientes a los códigos 13 y 10.

Ejemplo: ***Lectura de enteros en binario.***

```
#include<fstream.h>
#include<iostream.h>

int main()
{
    ifstream f;
    int dato;

    f.open("datos.bin");
    if(!f)
        cout << "Error abriendo el fichero" << endl;
    else
    {
        while(f.read((char *)(& dato), sizeof(dato) ) )
            cout << dato << endl;
        f.close();
    }
    return 0;
}
```

Ejemplo: ***Escritura de 10 enteros en binario.***

```
#include<fstream.h>
#include<iostream.h>

int main()
{
    ofstream f;
    int i;

    f.open("datos.bin");
    if(!f)
        cout << "Error abriendo el fichero" << endl;
    else
    {
        for(i = 1; i <= 10; i++)
            f.write((char *)(& i), sizeof(i) );
        f.close();
    }
    return 0;
}
```

PARTE II: BASES DE DATOS

Hasta ahora hemos visto como almacenar la información en el ordenador de forma permanente mediante el uso de ficheros, pero también se ha visto que, trabajar con los ficheros, conlleva conocer las características de los mismos; tipo de fichero y tipo datos que almacenan, en que orden se ha guardado, etc., ya que solo así se podrá leer o modificar la información. Por lo tanto, para trabajar con la información de forma eficiente, esta debe estar almacenada de forma coherente y sencilla.

En este apartado del tema introduciremos el concepto de base de datos, y la diferencia que existe con los sistemas de archivos, partiendo de la evolución que durante las últimas décadas ha sufrido el procesamiento de los datos y la gestión de la información.

Introducción

En los años 50, se empezaron a utilizar los computadores en los negocios para las tareas administrativas con el fin de reducir el papeleo. Normalmente se trataban de procesar datos para la contabilidad y para ello se intentaba imitar los procedimientos de papel de tal forma que: los archivos del computador se correspondían con los archivos en papel y los registros en los archivos del computador contenían la información que podía almacenar una carpeta individual de un archivo manual. A estos sistemas se les llamaron *sistemas de procesamiento de datos*, debido a que ejecutaban las funciones habituales de tratamiento de registros.

En los años 60 el acceso a los datos era secuencial y, dado que el almacenamiento en disco era caro, se almacenaban en cinta magnética. Como los datos eran utilizados por diferentes aplicaciones, se debía reorganizar la información, reordenando los datos según un identificador y fusionando después la información. Normalmente se procesaban los datos en bloques, es decir, todos los registros de un archivo se procesaban al mismo tiempo, habitualmente por la noche al cerrar el negocio.

Los sistemas de gestión de archivos puramente secuenciales eran eficaces cuando se trataba de producir informes una o dos veces al mes, pero para muchas tareas rutinarias no era suficiente, se necesitaba un acceso directo a los datos, es decir, la capacidad de acceder y procesar directamente un registro dado sin ordenar primero el archivo o leer los registros en secuencia.

Los problemas de redundancia en la introducción de datos, y por tanto de mayor probabilidad de error, así como la necesidad de reorganizar la información (dado que el acceso era secuencial) fueron resueltos parcialmente con la introducción de los *archivos de acceso directo* y, particularmente, de los archivos secuenciales indexados (ISAM - Indexed Sequential Access Method) que se utilizaron ampliamente en los años 70.

A diferencia de los de acceso secuencial, los archivos de acceso directo permiten la recuperación de los registros aleatoriamente, por lo que pueden recuperarse inmediatamente. Los archivos ISAM son los archivos más utilizados en procesos de tipo comercial. Estos archivos permiten que uno o más campos de datos - llamados conjuntamente **clave** - se utilicen precisamente para indicar qué registro se recuperará. Este potente y práctico método dotó de gran flexibilidad a las aplicaciones comerciales pero esto sólo fue una solución parcial. Para lograr una solución más completa a estos problemas, fue necesario introducir los sistemas de gestión de bases de datos.

A finales de los años 60 y principios de los 70 se dio la transición de los sistemas computacionales comerciales que pasaron del procesamiento de los datos al procesamiento de la información. La información era mucho más que simples registros relacionados con el negocio. Esto condujo a una fuerte demanda de *sistemas de información para la gestión* donde los sistemas debían utilizar los datos ya existentes en el computador para dar respuesta a un amplio espectro de preguntas de gestión o administración. [(*)]

[(*)] En este contexto se hace una distinción entre datos e información. Los datos pueden considerarse como hechos aislados y la información corresponde a los datos procesados, es decir, organizados.

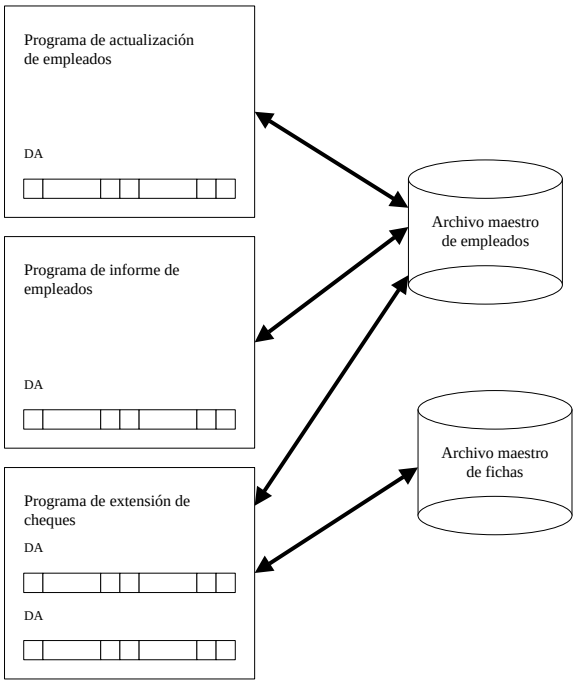
Por tanto, podríamos decir que una **base de datos** es una colección de elementos de datos interrelacionados que pueden utilizarse (o ser procesados) por uno o más programas de aplicación, y que un **sistema de bases de datos** es aquel sistema que está formado por una base de datos, por un sistema computacional de propósito general llamado sistema de gestión de bases de datos (SGBD) que manipula la base de datos, así como por el hardware y el personal apropiados. Un sistema de bases de datos bien diseñado, cuenta con funciones que facilitan la manipulación de la información (inserción, borrado, modificación de registros….) de tal forma que transforma los datos puros en información.

Sistemas de Gestión de Archivos

Los sistemas de información tradicionales almacenan la información en ficheros que son tratados por los sistemas de gestión de archivos, es decir, por un programa o conjunto de programas que se encargan de gestionar un conjunto de archivos de datos.

En una aplicación convencional con archivos, éstos se diseñan siguiendo las instrucciones de los correspondientes programas. Esto es, se decide si debe existir o 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 está 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; y se corre el riesgo de tener datos incongruentes entre los distintos archivos. Por ejemplo, para el caso de una empresa cuyo sistema contiene información de los empleados, podría darse el caso de que hubiera dos domicilios diferentes del mismo individuo en dos archivos distintos (por estar uno actualizado y el otro no).

En estas aplicaciones convencionales con archivos, el conocimiento a cerca del contenido de un archivo (qué datos contiene y cómo están organizados) está incorporado a los programas de aplicación que utilizan el archivo. Como ejemplo de utilización de un sistema de gestión de archivos se puede ver, en la figura de la derecha, 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.



Resumiendo, podemos decir que las deficiencias que sufren los sistemas de gestión de archivos son:

- *Redundancia e inconsistencia de datos*. Muchas aplicaciones utilizaban sus propios archivos, entonces, además de que podían existir campos repetidos lo cual obligaba a introducir varias veces los datos o a que no se actualizaran hasta tiempo después, estos podían tener distintas longitudes en

los diferentes ficheros, lo cual hacía inconsistentes los datos en las diferentes versiones. Es decir, esta redundancia conduce a un almacenamiento y coste de acceso a los datos más alto.

- *Pobre control de los datos*. En los sistemas de archivos no había un control centralizado a nivel de los datos, dado que el mismo elemento podía tener varios nombres dependiendo del archivo en que estuviera contenido. Esto daba lugar a confusiones debido a los homónimos (un mismo término que tiene diferentes significados en diferentes contextos) y sinónimos (términos diferentes que significan lo mismo), cosa que se evitaba, como iremos viendo a lo largo de este tema, con el sistema de bases de datos.
- *Capacidades inadecuadas de manipulación de los datos*. Los archivos secuenciales indexados (ISAM) permitieron que las aplicaciones tuvieran acceso a un registro particular mediante una clave, pero esto fue suficiente mientras solamente se quiso un registro único. El problema vino cuando se quería obtener relaciones más fuertes entre los datos contenidos en diferentes archivos, ya que estos sistemas son incapaces de proporcionarlas. Para ello se desarrollaron los sistemas de bases de datos, para facilitar la interrelación entre los datos en archivos diferentes.
- *Esfuerzo excesivo de programación*. Un nuevo programa de aplicación requería con frecuencia un conjunto completamente nuevo de definiciones de los archivos (aunque alguno ya existiera, seguro que habían muchos más por definir para que hubiera una consistencia en los datos). Entonces el programador tenía que recodificar todas las definiciones de los elementos de los datos necesarios ya existentes, así como codificar todos los elementos nuevos. Existía una interdependencia muy fuerte entre los programas y los datos que daba lugar a dos problemas; dificultad de acceso a los datos (cada vez que se quería obtener un conjunto de datos del sistema con unas características determinadas había que programar) y un aislamiento de los datos (dado que los formatos de los datos podían ser diferentes en cada archivo, dando problemas de integridad y además, cada vez que se introducían nuevas restricciones a estos datos había que reprogramar). Las bases de datos brindan una separación entre el programa y los datos, de modo que los programas pueden ser, en cierta medida, independientes de los detalles de definición de los datos. Al garantizar un acceso a un fondo común de datos compartidos y al soportar lenguajes poderosos para la manipulación de los datos, los sistemas de bases de datos eliminan una gran cantidad de programación inicial y de mantenimiento.

Por tanto, los problemas de mantener grandes sistemas basados en archivos condujeron, a finales de los sesenta, al desarrollo de los sistemas 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 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.

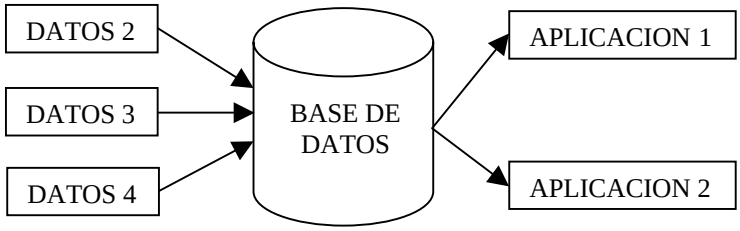
Sistemas de Bases de Datos. SGBD.

Como ya hemos dicho, los SBD surgen como alternativa a los sistemas de archivos, intentando eliminar o al menos reducir sus inconvenientes. Desde el punto de vista lógico (programas y usuarios), los datos y la definición de sus relaciones se almacenan en un único lugar, que es común. Físicamente, los datos se almacenan en uno o varios ficheros. El acceso de los datos se realiza, a través del sistema de gestión de bases de datos (SGBD o DBMS, Data Base Management System en inglés), mediante sentencias específicas que pueden incluirse dentro de lenguajes de alto nivel. Con esto, podemos definir un sistema de base de datos de la siguiente forma:

Un sistema de 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.

El esquema de funcionalidad de un SBD, se podría representar de la siguiente forma:

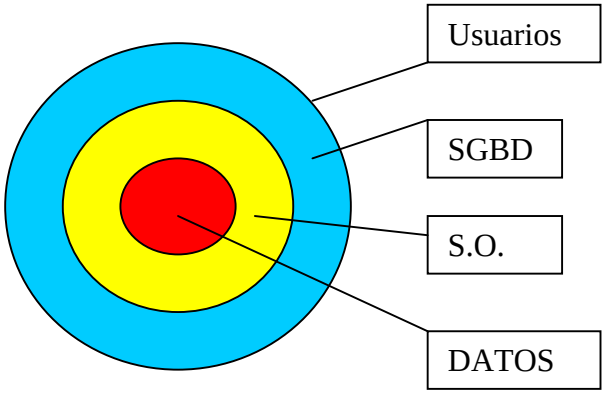
Los requisitos que debe cumplir un buen sistema de base de datos son:



- *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.

Resumiendo, tenemos que:

Las bases de datos permiten el almacenamiento, gestión y aprovechamiento de grandes volúmenes de información archivados durante periodos largos de tiempo en una computadora. Por ejemplo, en una empresa permiten crear, borrar, actualizar, recuperar relacionar y procesar nóminas pedidos, albaranes, facturas, etc. Y, el software que permite manejar la información almacenada en la base de datos es el Sistema de Gestión de Base de Datos (SGBD).



Clasificación de las bases de datos

En la actualidad estamos inmersos en varias décadas de largo esfuerzo por desarrollar sistemas de gestión de bases de datos cada vez más poderosos. Este proceso ha sido testigo del desarrollo evolutivo de los sistemas basados en los **modelos de datos**, y que no son más que métodos conceptuales para estructurar los datos, es decir, que nos permiten describir los datos y las relaciones entre ellos.

Se puede definir el *modelo de datos* como una colección de herramientas conceptuales para describir los datos, las relaciones de datos, la semántica de los datos y las ligaduras de consistencia entre los datos.

Antes de clasificar las bases de datos, debemos mencionar algunos de los modelos de datos más significativos que dieron lugar a determinadas Bases de Datos.

Modelo Jerárquico

En el modelo jerárquico, los datos se representan mediante colecciones de registros y las relaciones entre los datos se representan mediante enlaces, los cuales pueden verse como punteros. Los registros en la base de datos se organizan como colecciones de árboles.

Modelo de Red

El modelo de red es parecido al jerárquico ya que, los datos y las relaciones entre los datos también se representan mediante registros y enlaces respectivamente. La diferencia es que aquí los registros se organizan como colecciones de grafos dirigidos.

Modelo Relacional

En el modelo relacional se usa una colección de tablas para representar tanto los datos como las relaciones entre esos datos. Cada tabla tiene varias columnas y cada columna tiene un nombre único. A diferencia de los anteriores, no usa punteros sino que relaciona los registros a través de los valores que contiene.

Veamos un poco las características de las bases de datos a las que dieron lugar estos tres modelos:

Bases de datos jerárquicas

Los primeros SBD introducidos a mediados de los 60 estaban basados en el modelo jerárquico que resume que: *todas las interrelaciones entre los datos pueden estructurarse como jerarquías*. Con esto, los datos se representan mediante una estructura en árbol. En un sistema jerárquico de BD los archivos se conectan entre sí mediante punteros físicos o campos de datos añadidos a los registros individuales. Un **puntero** (apuntador) es una dirección física que identifica dónde puede encontrarse un registro sobre el disco. En una jerarquía, un **hijo** (un registro "subordinado" en una interrelación jerárquica) puede solamente tener un **padre** (un registro "propietario" en una interrelación jerárquica), pero un padre puede tener varios hijos. A este tipo de relación se le llama *relación 1-a-Muchos*.

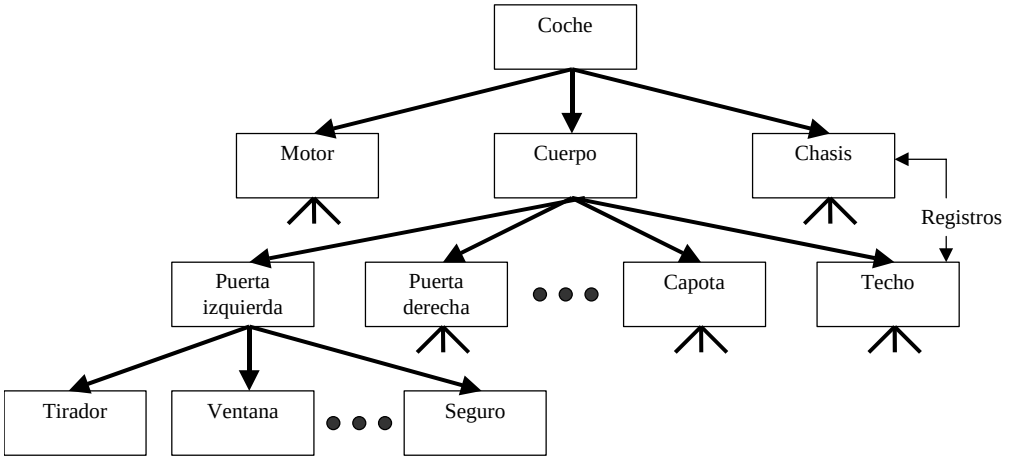
Veamos esto con una de las aplicaciones más importantes de los sistemas de gestión de base de datos primitivos como puede ser el planeamiento de la producción de empresas de facturación. Si un fabricante de automóviles decidía producir 10.000 unidades de un modelo de coche y 5.000 unidades de otro modelo, 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 subensamblajes (válvulas, cilindros, bujías, etc.) y luego en sub-subensamblajes, etc. El manejo de esta lista de piezas, conocido como una "cuenta de materiales", era un trabajo a medida para los ordenadores.

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



Ejemplo de base de datos jerárquica.

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. Y dado 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.

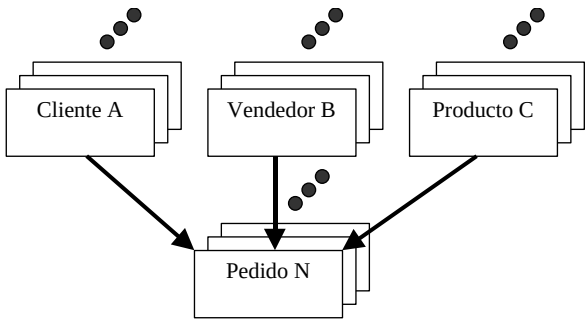
IMS sigue siendo el SGBD más ampliamente instalado en los maxicomputadores IBM.

Los problemas que presentan las BD jerárquicas son:

- Únicamente pueden representar relaciones de 1-a-Muchos.
- Las relaciones Muchos-a-Muchos requieren la redundancia de información dado que, si un registro tipo aparece como "hijo" en más de dos relaciones, se debe de replicar. Como consecuencia, esto puede producir problemas de integridad y consistencia de los datos.
- Los lenguajes de manipulación asociados son fuertemente navegacionales.

Bases de datos de red

Rápidamente se comprobó que el modelo jerárquico tenía algunas limitaciones importantes, ya que no todas las interrelaciones podían expresarse fácilmente en una estructura jerárquica, por lo que surgieron las redes. Estas redes se denominan diagramas. Una **red** es una interrelación de datos en la cual un registro puede estar subordinado a registros de más de un archivo. A causa de la necesidad obvia de manipular tales interrelaciones, a finales de los años 60 se desarrollaron los sistemas *de red*. Al igual que los sistemas de bases de datos jerárquicos, los sistemas de bases de datos de red emplearon punteros físicos para enlazar entre sí los registros de diferentes archivos.

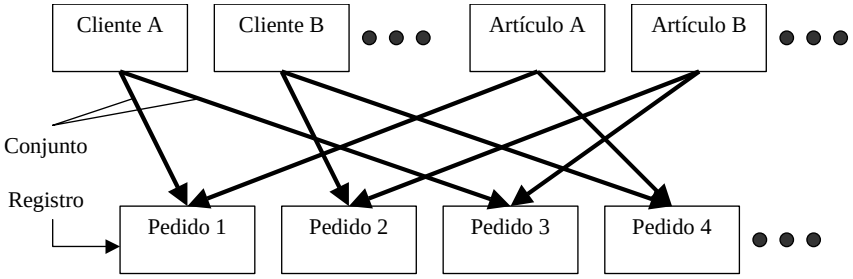


Ejemplo de múltiples relaciones hijo/padre

Resumiendo, la estructura sencilla de una base de datos jerárquica se convertía en una desventaja cuando los datos tenían estructuras más complejas. Por ejemplo, en una base de datos de procesamiento de pedidos, 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. La estructura de datos simplemente no se ajustaría a la jerarquía estricta de IMS.

Para manejar aplicaciones tales como el procesamiento de pedidos, se desarrolló un nuevo modelo de datos de red. El modelo de datos de red extendía el modelo jerárquico permitiendo que un registro participara en múltiples relaciones padre/hijo, reduciendo o eliminando de este modo las redundancias. Estas relaciones eran conocidas como *conjuntos* en el modelo de red. Entre mediados de los años 60 y principios de los 70 se desarrollaron y se comercializaron exitosamente varios SGBD en redes por lo que, en 1971, este modelo de datos se normalizó, es decir, se publicó un estándar oficial para bases de datos de red que se conoció como el modelo CODASYL.



Ejemplo de base de datos en red (CODASYL).

Hay que comentar que IBM nunca desarrolló un SGBD de 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 SGBD Adabas.

Para un programador, acceder a una base de datos de 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 de 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 minicomputadoras, como Digital Equipment Corporation y Data General, implementaron bases de datos de red.
- Rendimiento. A pesar de su superior complejidad, las bases de datos de 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.

Por otra parte, las bases de datos de red tenían también sus desventajas. Al igual que las bases de datos jerárquicas, 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 de red eran herramientas para programadores. Por ejemplo, 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. Con

frecuencia la anotación de las peticiones para informes a medida duraba 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.

Bases de datos relacionales

El uso de punteros físicos en las BD jerárquicas y de red era a la vez su punto fuerte y débil. Fuerte porque permitieron la recuperación rápida de los datos que tuvieran interrelaciones predeterminadas. Débil porque estas interrelaciones tenían que definirse antes de que el sistema fuera puesto en explotación. Era difícil, si no imposible, recuperar datos basados en otras interrelaciones, cosa que con el tiempo fue inaceptable.

En 1970, E. F. Codd publicó un artículo revolucionario que desafió fuertemente el juicio convencional de la "condición" de las bases de datos. Codd argumentó que los datos deberían relacionarse mediante interrelaciones naturales, lógicas, inherentes a los datos, más que mediante punteros físicos. Es decir, si la información lógica necesaria para hacer la combinación estaba presente en los datos, las personas deberían ser capaces de combinar los datos de fuentes diferentes. Esto abrió una nueva perspectiva para los sistemas de gestión de información ya que las interrogaciones a las bases de datos no necesitarían, en adelante, limitarse a las interrelaciones indicadas por los punteros físicos sino que se hace mediante interrelaciones lógicas, resolviendo así los problemas anteriormente planteados.

A grandes rasgo, podemos decir que las bases de datos relacionales se basan en el modelo de datos relacional dónde el elemento básico es *el concepto de relación* (que se representa como una tabla) y que hace referencia a las entidades del mundo real (datos) y a como se relacionan entre ellas.

Clasificación de las bases de datos

El hecho de ir implementando nuevas bases de datos de acuerdo con la evolución sufrida por los modelos de datos, hizo que tradicionalmente las bases de datos se clasificaran en tres grupos: *jerárquicas*, *de red* y *relacionales*. Las dos primeras se diferencian en los tipos de relaciones que permiten, pudiendo decirse que la estructura jerárquica es un caso particular de la estructura de red. Por otra parte, las bases de datos relacionales son conceptualmente distintas, pues en ellas las relaciones se almacenan y manipulan de forma completamente distinta.

Si tenemos en cuenta que en la actualidad han ido apareciendo nuevos y más potentes modos de manipular datos así como nuevas tecnologías, podríamos ampliar la anterior clasificación considerando que, los sistemas de bases de datos se pueden clasificar de forma conveniente atendiendo a las estructuras de datos que manejan y a los operadores presentados al usuario y que le permiten acceder a la información almacenada en ella.

Desde esta perspectiva, los sistemas más antiguos se han denominado *pre-relacionales*, y se clasifican en tres categorías. Luego aparecen los sistemas *relacionales*, que marcan la frontera y definen el "antes y el después". Y posteriormente, los sistemas *post-relacionales*, que están todavía en evolución y marcan la pauta de las nuevas tendencias y tecnologías. Veamos algunos ejemplos de estos SGBD:

Pre-relacionales:	• De lista invertida (CA-DATACOM/DB, etc.) • Jerárquicos (IMS de IBM, etc.) • De red (CA-IDMS/DB, etc.)
Relacionales:	• Relacionales (ORACLE, DB2, SQL/DS, Rdb/VMS, INGRES, INFORMIX, SQLSERVER, etc.)
Post-relacionales:	• Sistemas deductivos de administración de bases de datos • Sistemas semánticos de administración de bases de datos • SGBD de relación universal • SGBD orientados a objetos • Sistemas extensibles de administración de bases de datos • Sistemas expertos de administración de bases de datos