Práctica 3:

Gestor de base de datos

ESECUELE

Francisco Andreu Sanz y Jesús Barceló Putter
Pareja 10, Grupo 1273.

Índice

Introducción:

En ésta última práctica implementamos un **gestor de bases de datos.** Como veremos a lo largo de la práctica, contará con algunas limitaciones, pues su estructura es algo compleja. Lo principal será que dicho gestor cumpla una **serie de funciones:**

Ser capaz de crear **bases de datos.**
Crear **tablas**(de 4 tipos, INT, LNG, DBL y STR) las cuales por limitación no contendrán **restricciones.**
Crear índices que al fin y al cabo la función que tendrán será que las búsquedas en tablas se efectúen de forma eficiente.
La inserción de distintos datos importados desde un fichero de texto y almacenarlo de forma correcta en cualquier tabla de la base de datos.
Que sea posible la ejecución de **consultas,** tanto simples como complejas con distintos operadores(y más de uno en cada consulta).

Lo primero que implementaremos será el módulo **table.c**, el cual, después de database, es el principal "motor" del programa. En él implementaremos las funciones de crear tablas(en ficheros que serán guardados en la carpeta de la base de datos), abrirlas, escribir en ellas y leer tuplas(es decir, el contenido de las tablas lo guardamos en el mismo archivo que las tablas), para lo que se necesitará la implementación de otro módulo.

El módulo **record.c** es el más *simple* por así decirlo, guarda el contenido de las tuplas para posteriormente pasárselas a la función de table.c que corresponda(table_read_record).

Por su parte, **index.c** tiene una implementación bastante particular, cuyo funcionamiento explicaremos en la metodología. En el módulo, implementamos funciones de crearlos, de abrirlos, así como de insertar claves.

Objetivos:

Los objetivos principales los podemos resumir tanto en **comprender el funcionamiento del gestor de bases de datos como** ser capaces de implementarlo módulo por módulo.
Reforzaremos conocimientos con respecto a las consultas para ser capaces de **ejecutar** tanto las que te pide el enunciado de la práctica como las del ejemplo de la base de datos de **bank.**
Realizaremos la creación de una base de datos de Twitter similar a la implementada con PostgreSQL ó as librerías ODBC, e insertaremos los datos pertenecientes a cada tabla **basándonos en la estructura de la práctica 1.**

Metodología:

En éste apartado explicaremos paso por paso cada uno de los apartados que se valoran en la práctica. Cómo los hemos implementado, las dificultades que hayamos podido tener al realizarlos, las herramientas con las que hemos trabajado y, en definitiva, cómo y por qué hemos hecho cada uno de los apartados de esa manera.

Añadir los tipos de dato DBL y LNG

Sin duda el apartado más sencillo y que no tomó más de 5 minutos. Debíamos modificar el módulo **type.c** de tal manera que almacenase los tipos de dato **double y long. Lo primero fue cambiar la estructura,** añadiendo dos casos más en la enumeración que se nos daba. Luego procedimos a cambiar las funciones.

Fue tan simple como añadir dos casos más en todos los "**switch"** y los "**if"** para cada uno de los tipos de dato a añadir en las funciones ya implementadas. Esos casos, dependiendo de la función, consistían en:

La **reserva de memoria** de los dos tipos de dato, es decir, variando el argumento de la función *malloc* al **tamaño** del tipo de dato en cuestión.
Un **casting** al tipo de dato que hará que una variable de **tipo genérico (**void* value) pase a contener un valor de tipo **INT, LNG, DBL, o STR** según corresponda.
Un *return* del tamaño del tipo, o del valor numérico de la enumeración correspondiente al tipo.

Para éste apartado **no vimos conveniente realizar** ningún prueba, pues con **table.c y record.c** probaríamos también de forma indirecta el correcto funcionamiento de **type.c.**

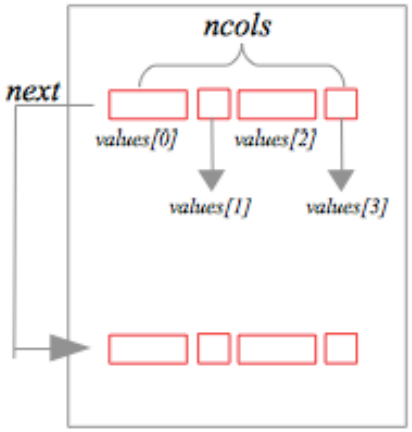
Implementación de table, record y sus main de prueba.

Comenzamos realizando el módulo **record.c,** pues para realizar alguna función de table se requería haber diseñado este módulo.

Record

Nuestra implementación del TAD **record** consta de únicamente tres campos.

```
struct record_ {
    int ncols;
    long next;
    void** values;
};
```



Adjuntamos un dibujo intentando explicar el funcionamiento del Tipo Abstracto de dato **record.**

> **ncols** guarda el número de columnas que existen en una tupla, para que al leer el archivo sepamos dónde cuántas columnas tiene una fila.
>
> El campo **next** almacena un número, que es **la suma de todos los sizeof("")** de una fila. Es decir, para *pasar* a la fila siguiente se recorrerá la tupla contando el tamaño de cada columna.
>
> El campo **values** fue algo que en un principio nos costó entender. Se comprende mejor si lo vemos así (*void*)**, esto es, un **array de tipo genérico.** Es decir, cada posición del array puede almacenar un tipo diferente de dato. Así pues, en el ejemplo de arriba, *values[0] y values[2]* podrían ser de tipo STR, *values[1]* de tipo INT *y values[3]* un LNG, por poner un ejemplo.

Una vez comprendida la estructura de record, veremos las cabeceras de cada una de las funciones que hemos implementado y comprendemos sus argumentos
(la explicación completa de la implementación de cada función hemos decidido incluirla en el código y no en la memoria).

record_t* record_create(void values, int ncols, long next);**
Reservamos memoria, e insertamos en la estructura *record* cada uno de los parámetros de la función. Devolvemos el record creado.

void* record_get(record_t* record, int n);

El parámetro n es la posición del elemento que vamos a devolver. Por tanto, se devuelve el tipo genérico que corresponde a la columna n del array de *values* del record pasado por primer parámetro.

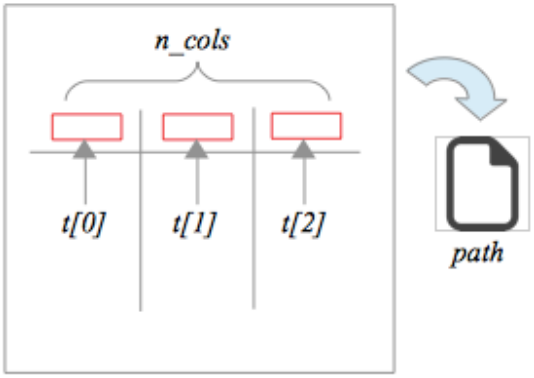
long record_next(record_t* record);
Devuelve el campo *next* del record pasado por parámetro.

void record_free(record_t* record);
Liberamos la memoria que habíamos reservado para el record pasado por parámetro.

Table

Nuestra implementación de **table** la queríamos hacer lo más simple posible y por tanto sólo utilizabamos tres campos en la estructura. Sin embargo, realizamos una última modificación incluyendo un campo para guardar la primera y última posición(para no tener que hacer uso de fseek todo el rato, lo que **ralentizaba excepcionalmente las consultas**).

```
struct table_ {
    int n_cols;
    type_t* t;
    FILE* path;
    long first_pos;
    long last_pos;
};
```



La implementación de table fue algo más compleja debido a que hubo dos funciones que nos dieron algún quebradero de cabeza probándola con el **main de pueba.** Sin embargo, como podemos ver, el TAD table_ es sencillo:

El campo **n_cols** al igual que en record nos guarda el número de columnas existentes.

El array de tipos **t** nos guarda el tipo de dato de cada columna. Habrá tantos como *n_cols* haya.

Por su parte, **path** nos guarda el nombre del fichero que se creará de la **tabla.**

First_pos y Last_pos guardan en un *long* la posición donde comienza el primer record, y donde acaba el último, respectivamente

Vamos a ver entonces la cabecera de cada una de las funciones del módulo.

Void table_create(char* path, int ncols, type_t* types);
A diferencia de *record_create,* esta función se limita a crear y escribir en el *path*, primer parámetro, el número de columnas y el array de tipos de la tabla. Por último cerramos el fichero.

table_t* table_open(char* path);
Esta función es más compleja de lo que parece en un principio. Se abre la tabla cuyo nombre se pasa por parámetro, y se va leyendo cada uno de los campos de la estructura, los cuales se asignarán a una tabla a la cual reservaremos memoria y devolveremos.

void table_close(table_t* table);
Cerramos el fichero y liberamos cada uno de los parámetros de la tabla del parámetro.

int table_ncols(table_t* table);
Devolvemos el campo *n_cols* de la tabla pasada por parámetro.

type_t* table_types(table_t* table);
Devolvemos el array de tipos de la tabla pasada por parámetro.

long table_first_pos(table_t* table);
Devolvemos la primera posición de la tabla pasada por parámetro. Para ello, leemos la cabecera de la tabla(**n_cols y types**). Quizá es engorroso el leer la cabecera, pero a cambio no tenemos la necesidad de depender de otro campo del TAD tabla.

long table_last_pos(table_t* table);
Similar a la anterior pero simplemente nos posicionamos al final del archivo.

record_t* table_read_record(table_t* table, long pos);
Nos colocamos en la posición que pasamos como segundo argumento, y a partir de ahí, creamos un record con la función ya implementada *record_create* y vamos leyendo de la tupla cada uno de los valores. El número de columnas será el mismo que el de la tabla, y el campo next lo iremos grabando y aumentando a medida que vayamos recogiendo los *values.* Fue quizá la más difícil de implementar de este módulo.

void table_insert_record(table_t* table, void values);**
Escribimos en el fichero tanto el tamaño como el valor, de cada uno de los valores de la tupla, tras habernos posicionado al final del fichero.

Para comprobar que funcionaban correctamente tanto table.c como record.c, realizamos un main de prueba(el cual adjuntamos en el .zip de la práctica) que ejecuta cada una de las funciones de los módulos.

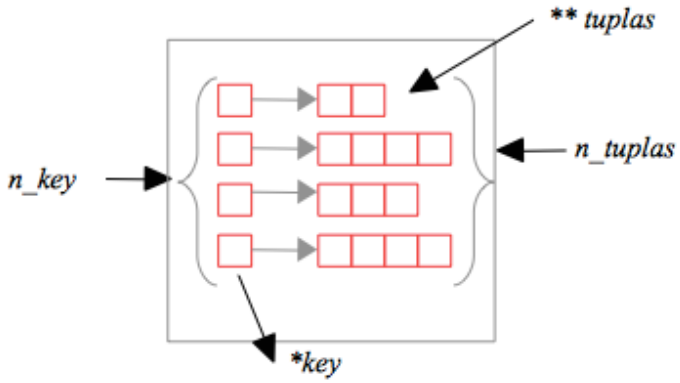
En el mismo, creamos un **array de tipo genérico y reservamos 4 posiciones,** una para cada tipo implmentado. Además, al haber una posición de cada tipo, creamos otro array de **type_t** cuyas posiciones corresponden al tipo de cada posición de **values.**

Tras esto, probamos cada una de las funciones de los dos módulos y vemos que los resultados son los esperados.

Index

La implementación de **index** por su parte tuvimos bastantes dudas de cómo realizarla en un principio. Pensamos distintas estructuras para el tipo abstracto, hasta que decidimos que la más lógica y eficiente según nuestro punto de vista era esta:

```
struct index_ {
    int n_key;
    int* key;
    int *n_tuplas;
    long** tuplas;
};
```



Aún con el dibujo puede resultar algo confuso, así que conviene explicar cómo funciona la estructura y qué es cada campo.
Un **índice** necesitará una ser referenciado a una columna de tipo INT de una tabla. Los diferentes valores de la columna aparecerán **de forma ordenada** en el array *int* key.*

El entero *n_key* guardará, por así decirlo, el número de claves diferentes que existen en el índice.

Cada clave aparecerá en un número determinado de **tuplas.** Por ejemplo, según el dibujo, la primera clave aparece únicamente en dos posiciones, luego la matriz *tuplas* en la *posición vertical* [0], contendrá los dos valores, los cuales son las posiciones donde aparece la clave key[0]. Además, el tamaño(en este caso 2), se guardará también en la posición [0] de *n_tuplas*.

Teniendo claro como funciona exactamente el TAD, es conveniente citar algunos requisitos a la hora de trabajar con **índices:**

Si fuese necesario insertar una clave, **esta debe hacerlo de forma ordenada**, es decir, la clave se colocará en una posición *i* tal que ***T[i-1]<T[i]*** **y** ***T[i]<T[i+1].*** El por qué es muy simple. Para que el índice busque claves de forma eficiente, recurrimos al Algoritmo de **Búsqueda Binaria.**
El motivo por el que la matriz tuplas es de tipo long es porque en una función nos pasan como argumento un tipo *long.* Esto puede deberse a que en una base de datos grande, el número de filas de una tabla pueden alcanzar números más grandes que el tipo entero.

Veamos las funciones a implementar:

int BBin(int* table, int tam, int clave);
La función de búsqueda binaria será una función auxiliar que nos buscará una *clave* que le pasamos por argumento, además del tamaño *tam* de la tabla ordenada *table.* Devuelve -1 si no lo encuentra, o la posición *i* del array donde se encuentra la clave.

index_t* index_form (int key, int* keys, int* value, long values);**
La hemos creado únicamente para asignar todos los campos de la estructura de índice para el main de prueba y no acceder a los campos de la estructura como tal.

int index_create(char* path);
Simplemente creamos un índice con nombre *path.* Para reconocerlo como un índice vacío, escribimos un 0 en ese mismo fichero.

int index_save(index_t* index, char* path);
Esta función nos envía un índice por argumento, cuyos campos escribiremos de forma ordenada en un fichero cuyo nombre nos viene también por argumento. Primero escribimos el número de claves, luego el array entero de las claves, y luego de forma ordenada, el número de posiciones y array entero de posiciones de cada clave. Por último cerramos el fichero.

index_t* index_open(char* path);
Al contrario que en la función anterior, creamos un índice y reservamos memoria para él y sus campos, y a continuación vamos leyendo**(de la misma forma que hemos esrito en el fichero *.index*)** cada uno de los valores de cada campo de la estructura. La memoria de los elementos de la matriz la vamos reservando **a medida que vamos leyendo.**

int index_put(index_t *index, int key, long pos);
Sin duda la función más compleja del módulo junto con la siguiente. Busca una clave e inserta una posición de una tupla para esa clave. Primero ejecutamos una búsqueda de la clave mediante el algoritmo de **Bbin.** Si lo encuentra bien, sino la crea **en la posición correcta y desplaza todas las claves que había delante, una posición más.** Una vez que conseguimos eso, simplemente añadimos un valor *pos* al campo *tuplas* de **esa posición en concreto.**

void index_get(index_t *index, int key, long poss, int* nposs);**
Esta función, pese a no ser la más complicada, nos ocasionó algunos problemas. Lo que hace es buscar una clave en un índice, y una vez que la encuentra devuelve**(por paso por referencia)** las tuplas donde aparece y el número de tuplas en las que aparece*(poss y nposs)*. El problema lo tuvimos a la hora de guardar el **array de posiciones mediante paso por referencia** pues en un principio lo hicimos de la forma ***poss[j],* es decir, accediamos al contenido en la posición j del array, pero ésto ocasionaba que la posición [j] fuera la posición **vertical** la que ibamos modificando, luego optamos por implementarlo así poss[0][j], es decir, la j varía simplemente en los parámetros *horizontales.*

void index_close(index_t *index);
Liberamos cada uno de los campos de *index* y lo dejamos apuntando a NULL.

También creamos un main de prueba para comprobar el correcto funcionamiento de éste módulo. Tuvimos bastantes problemas a la hora de leer **posiciones de memoria no reservadas,** los cuales resolvimos gracias a la herramienta **Valgrind** y en parte gracias a la herramienta de Debugging de **NetBeans**.
En el main de prueba dábamos valores a cada uno de los campos del índice, y posteriormente buscábamos las posiciones en las que aparecían determinadas claves, después de insertar una la cual desplazaba el resto correctamente a una posición superior. Los resultados fueron los esperados, lo hacía de forma correcta.

Operation.c

Todos los módulos en este apartado son similares pues parten de las mismas funciones. En este apartado me limitaré a explicar la característica principal de cada operador implementado**(COUNT, LIMIT, OFFSET, UNION)**

Operador COUNT.
Este operador nos proyecta una tabla de una única fila, y columna, que son **el número de tuplas (una columna de tipo INT).** ¿Cómo hemos implementado esto? Bien, en la estructura del operador COUNT tenemos un campo de la estructura que es un contador*(count).*
En la función **operation_count_create** igualamos el número de filas a éste campo *count.* ¿Cómo averiguamos el número de filas? Mediante sucesivas llamadas a la función **operation_next** mientras sea distinta de 0., incrementamos **counts.**

Operador LIMIT.
Lo que conseguimos con LIMIT es que nos proyecte por pantalla un númer que le especifiquemos de tuplas. Por ejemplo, al ejecutar la consulta users *SEQUENTIAL 1 LIMIT*

Unicamente nos mostrará la primera fila de la tabla *users.*Esto lo conseguimos gracias a que en ka función **int operation_limit_next** creamos un contador que se va incrementando siempre que llamemos a esta función. Cuando el contador sea mayor que el límite que establecemos, devolvemos 0 y paramos de imprimir tuplas.

Operador OFFSET.
Es la "inversa" de LIMIT. Es decir, imprimimos todas las tuplas excepto el número que le especificamos. El modo es similar al anterior, la función **operation_offset_next** tiene un contador llamado *first* que como su nombre indica, nos dirá cual es la primera tupla a imprimir(según el número con el que hagamos la consulta).

Operador UNION.
En un principio nos trajo algunos problemas, pero la implementación una vez que la comprendimos fue de las más sencillas. UNION simplemente te imprime dos tablas que le indiquemos al hacer la consulta(*t1 t2 UNION*). Esto es, imprimimos la primera y al terminar ésta, comienza a imprimirse la segunda. **Una particularidad es que las tablas a imprimir deben tener el mismo número de columnas, sino, mostrará campos sin datos.** La estructura de este operador tiene la particularidad de tener dos punteros a *operation_t,* al contrario que los anteriorres que contaban con uno solo.

Para la explicación paso por paso del código, hemos comentado los módulos de operation también.

Una vez diseñados y probados todos los módulos a implementar, toca trabajar con la gestión de la base de datos.
Para empezar, probamos la base de datos de bank, que nos viene en el archivo *bank.bash.*

```
d> INDEX i_accounts_clients_0 accounts_clients 0
INDEX i_accounts_clients_1 accounts_clients 1
d>

d>
```

Insertamos tanto las tablas como los indices, lo hace todo correctamente.

```
i> COPY clients ../bank_files/clients.txt
COPY accounts ../bank_files/accounts.txt
COPY accounts_clients ../bank_files/accounts_clients.txt(10 rows inserted)
i> (15 rows inserted)
i>
(18 rows inserted)
```

E insertamos los elementos en cada una de las tablas, como pone en el archivo *.bash* Vamos a probar a ejecutar las consultas del *.bash.* y veremos cómo las opera de forma correcta.

La primera:

```
q> accounts SEQUENTIAL 1 STR Brooklyn C_COLEQCTE SELECT INT 0 P_COL INT 2 P_COL 2 PROJECT
1       456
3       2411
10      6545
14      741
(4 rows retrieved)
```

La segunda:

```
q> accounts SEQUENTIAL 1 STR Queens C_COLEQCTE SELECT INT 0 P_COL INT 2 P_COL 2 PROJECT accounts_clients SEQUENTIAL PRODUCT 0 2 C_COLEQCOL SELECT INT 0
P_COL INT 1 P_COL INT 3 P_COL 3 PROJECT clients SEQUENTIAL PRODUCT 2 3 C_COLEQCOL SELECT INT 0 P_COL INT 1 P_COL STR 4 P_COL 3 PROJECT
4       245     Brandon
4       245     Lee
5       778     Lee
13      12      Lee
(4 rows retrieved)
```

Como vemos, la base de datos de *bank* opera de **forma correcta** en todos sus apartados. Además, Valgrind no encuentra Leaks ni errores de escritura para ellas(**ni para las consultas de la base de datos de Twitter**).

```
10      Simmons
(10 rows retrieved)
q> clients SEQUENTIAL 9 LIMIT
1       Smith
2       Jones
3       Simpson
4       Cooper
5       Johnson
6       Macdonald
7       White
8       Lee
9       Brandon
(9 rows retrieved)
q>
==13886==
==13886== HEAP SUMMARY:
==13886==     in use at exit: 0 bytes in 0 blocks
==13886==   total heap usage: 164 allocs, 164 frees, 41,848 bytes allocated
==13886==
==13886== All heap blocks were freed -- no leaks are possible
==13886==
==13886== For counts of detected and suppressed errors, rerun with: -v
==13886== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
jesusbarceloputter@ubuntu:~/Escritorio/esecuele-edat2014$
```

Vamos pues, a probar a insertar las tablas de Twitter. Para ello, crearemos en primer lugar las tablas.

La tabla **users,** la cual tendrá 4 columnas. La crearemos de la siguiente manera:

TABLE users 4 LNG STR STR STR

Como podemos observar, la columna de fecha la haremos ser una cadena de caracteres al no haber implementado el tipo **timestamp.**

La tabla tweets, que contiene 6 columnas. La crearemos de la siguiente forma:

TABLE tweets 6 STR STR STR STR STR STR

Aquí los ID(tanto Retweet_of, como Reply_to como tweet_id) son STR. Pues el tamaño de los números de los ID de los tweets **excede el tamaño máximo que almacena un LNG.**

La tabla follows, con 2 columnas.

TABLE follows 2 STR STR

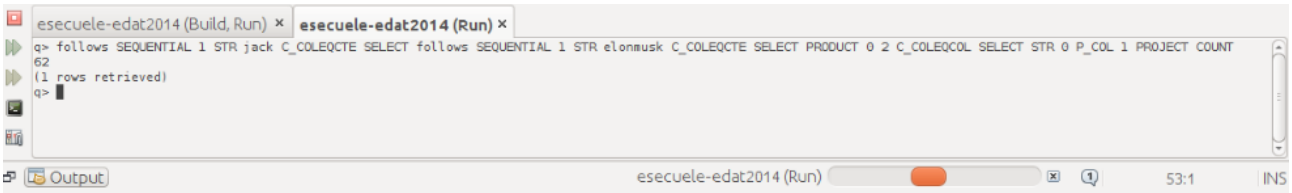
Hemos conseguido también insertar **todos los datos de los txt de prueba de la Práctica 1.** No hemos podido basarnos en la práctica 2 al no haber un fichero de texto con las tablas de la práctica 2.

Con respecto a las consultas, creemos que teóricamente están bien redactadas, pero no hemos podido compilar ninguna de ellas en nuestro programa en ESECUELE.

La **primera** sería así:

follows SEQUENTIAL 1 STR jack C_COLEQCTE SELECT follows SEQUENTIAL 1 STR elonmusk C_COLEQCTE SELECT PRODUCT 0 2 C_COLEQCOL SELECT STR 0 P_COL 1 PROJECT COUNT

El resultado de la consulta es **62.**



Las siguientes consultas no hemos sido capaces de que se realicen de forma correcta en ESECUELE, sin embargo hemos realizado las mismas con la base de datos de POSTGRE, es decir, en lenguaje **SQL**.

_____

Hemos intentado realizar la segunda consulta en ESECUELE, pero nos ha fallado numerosas veces. Esto es lo más cerca que hemos estado de conseguirla.

follows SEQUENTIAL 1 STR TheOnion C_COLEQCTE SELECT STR 0 P_COL 1 PROJECT tweets SEQUENTIAL STR 4 P_COL STR 1 P_COL 2 PROJECT PRODUCT 0 1 C_COLEQCOL SELECT STR 2 P_COL 1 PROJECT

_____

La **segunda** sería:

*SELECT tab1.author FROM (SELECT * FROM tweets as tab2 WHERE tab2.author = 'TheOnion') as tab, tweets as tab1 WHERE tab.tweet_id=tab1.retweet_of;*

La **tercera** sería:

*SELECT COUNT(*) FROM (SELECT * FROM tweets as tab, tweets as tab1 WHERE tab.reply_to=tab1.tweet_id AND tab1.tweet_id = 'jack') UNION ALL (SELECT * FROM tweets as tab2, tweets as tab3 WHERE tab3.tweet_id = 'jack' AND tab3.tweet_id = tab2.retweet_of));*

Conclusiones:

Hemos aprendido más **conocimientos sobre la base de datos** y, sobre todo, hemos sido capaces de desarrollar, tanto en teoría como llevar a la práctica, un gestor de bases de datos que además funciona con Queries.
Hemos tenido alguna dificultad con la implementación, fallos que hemos sabido corregir, y hemos conseguido hacer consultas muy funcionales. Además, hemos conseguido también aprender nuevos conocimientos sobre las **funciones relativas a ficheros, en este caso binarios.**