TP Bases de Datos II

Parte 3

(ENTREGA 3)
En esta segunda parte del trabajo práctico nos enfocaremos en la implementación de una nueva abstracción: el paginador, y en el guardado de los datos en disco.

Requerimientos para la entrega

1. Guardado y recuperación de los datos en disco
2. Paginador implementado e integrado

Introducción

Hasta el momento veníamos representando el contenido de la tabla como una lista o array de bytes de tamaño fijo (páginas) alojados en memoria, al cual accedemos directamente. Sin embargo esto tiene limitaciones:

- Ausencia de abstracciones: los componentes que necesiten acceder a una página necesitan conocer los detalles de representación e implementación.
- Pérdida de datos, cuando se cierra el programa la información se pierde.
- Toda la información se aloja en memoria, si el volúmen crece lo suficiente puede volverse sumamente ineficiente y consumir demasiados recursos.

Por lo tanto, en esta iteración vamos empezar a atacar estos problemas:

- Por un lado implementaremos un componente llamado Paginador (o Pager) que se ocupará de manejar el acceso a páginas, ocultando los detalles de representación y persistencia.
- Por otro lado nos ocuparemos de persistir los datos en un archivo en disco, y leerlos del mismo.

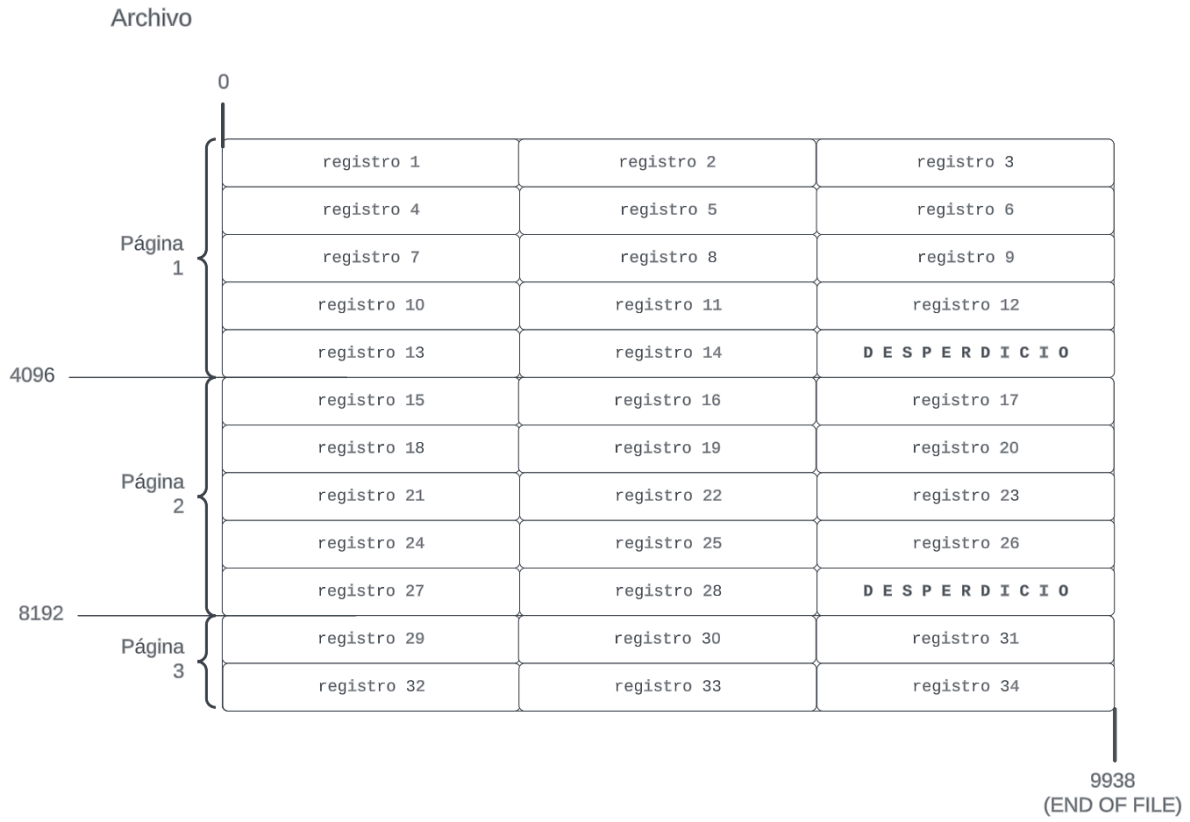
Persistencia

Al iniciar la conexión con la base de datos (inicio del programa) abriremos el archivo de la base de datos (que debe pasarse como argumento al ejecutar el programa desde la consola (por ejemplo: `python main.py archivo.db`), y calcularemos la metadata relacionada a la tabla (páginas, registros, etc).
Al cerrar la conexión con la base de datos (o sea, en la ejecución del metacomando `.exit`) guardaremos en el archivo las páginas nuevas o modificadas desde el inicio de la conexión, en la posición del archivo correspondiente. Es decir, si al iniciar la conexión existían 2 páginas, y en la segunda página agregamos registros, entonces sobre-escribiremos en la posición del archivo correspondiente con el nuevo contenido (a partir de la posición 4096 del archivo, que corresponde al comienzo de la página 2). Si además agregamos una nueva página, esta nueva página se escribirá en la posición 8192 (correspondiente al comienzo de la página 3).
Ya que los datos están serializados previamente en memoria, no necesitamos modificarlos para guardar en el archivo.

Tenemos que tener en cuenta que al estar trabajando a nivel bytes, y que no vamos a guardar ningún metadato en el archivo por ahora, vamos a tener que recalcular los metadatos a partir del tamaño del archivo, mediante las siguientes definiciones

```
Sean:
   CP = cantidad de páginas
   TA = tamaño de archivo
   CR = cantidad de registros
Tal que:
   CP:
     si (TA mod 4096 == 0) -> TA / 4096
     sino -> int(TA / 4096) + 1
   CR:
     si (TA mod 4096 == 0) -> CP * 14
     sino -> int(TA / 4096)*14 + (TA mod 4096)/291
```

Para entender la organización de los datos dentro del archivo veamos un ejemplo. Asumimos que tenemos 34 registros. Como dijimos que en una página entran 14 registros enteros, vamos a tener 3 páginas en total. Las páginas que ya completaron su máximo de registros (primera y segunda página) ocupan en el archivo 4096 bytes. Sin embargo, la tercer página que solo contiene 6 registros ocupará solo 291*6 = 1746 bytes en el archivo, y no se ocupan los restantes 2350 del total de página. Esto se debe a que al abrir el archivo para calcular los metadatos necesitamos saber dónde termina el contenido real. Si ocuparemos los 4096 completos, pero solo guardaremos 6 registros en esa página no tendríamos manera de saber que el resto de los bytes son basura de la memoria, y deberíamos guardar en el archivo metadata adicional a estos efectos, pero no haremos eso.



Paginador

La idea del paginador es abstraernos de los detalles de la tarea de buscar una página, es decir, debe ocultar qué es lo que hacemos "por detrás" para conseguir el bloque de datos correspondiente a la página solicitada, y proveernos de una interfaz para tal fin. A estos efectos, la interfaz del paginador expondrá idealmente un solo método que, dado un número de página, devuelve un puntero a la misma. Por ejemplo:

```
Pager:
    get_page(i: int) -> *Page
```

A su vez, el paginador mantendrá un caché propio de las páginas ya leídas del disco. Al solicitarle al paginador una determinada página, primero revisa la cache, si está ahí la devuelve, sino, lee del disco, la guarda en la caché, y luego la devuelve.

Comportamiento esperado de get_page:

```
Dado número de página i:
  Si la página i está en cache ->
    Devolver puntero a cache[i]
  Sino ->
    Si la página existe en el archivo ->
      Calcular posición pagina i
      Leer página de esa posición del archivo
      Guardar bloque en cache[i]
      Devolver puntero a cache[i]
    Sino ->
      Crear nueva página
      Guardar en cache[i]
      Devolver puntero a cache[i]
```

Decimos que get_page debe devolver un puntero a la página y no el bloque de datos en si mismo ya que queremos modificar el mismo espacio de memoria al que apunta la caché, cada vez que insertemos un nuevo registro en la página.