Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC3413 — Implementación de Sistemas de Base de Datos — 1' 2024

LABORATORIO 1

Laboratorio:	Almacenamiento de datos.
Publicación:	Lunes 1 de abril.
Ayudantía:	Viernes 5 de abril.
Entrega:	**Viernes 12 de abril hasta las 23:59 horas**.

Descripción del laboratorio

El objetivo de este primer laboratorio es conocer y familiarizarse con el almacenamiento de IIC3413DB [1], un sistema relacional que iremos desarrollando durante el curso. Para conocer el sistema de almacenamiento e identificar las componentes claves del mismo, para este laboratorio usted deberá completar la implementación de la clase `HeapFilePage` que nos permite manejar tuplas de una relación con un estructura de HeapFile.

Contexto en IIC3413DB

Buffer manager. IIC3413DB ya cuenta con un sistema de buffer manager y páginas. Esta pieza de software se encuentra implementado en el módulo de `storage`, en los archivos `buffer_manager.h` y `page.h` respectivamente. La clase `Page` implementa el acceso "plano" a una página de datos. Cada página tiene tamaño de 4096 bytes. Para acceder estos datos, `Page` tiene el método `data()` cual entrega la variable `bytes` (atributo privado de la clase `Page`). Aquí `bytes` actual cómo un puntero a donde empieza la secuencia de bytes con el contenido de la página. La clase `BufferManager` implementa el buffer manager del sistema con los métodos pin/unpin. Para solicitar una página a buffer manager, este provee el método:

```
Page& get_page(FileId file_id, uint64_t page_number)
```

que dado el identificador de un archivo (implementado por la clase `FileId`) y la dirección de una página (almacenado como un `uint64_t`), entrega la referencia a la página correspondiente.

Es importante notar que `BufferManager` no provee un método pin. En cambio, cuando uno solicita una página con el método `get_page`, es el buffer manager el encargado de hacer pin de la página. En otras palabras, `get_page` es lo mismo que el método pin.

Para hacer unpin, la clase `Page` provee el método:

```
void unpin()
```

para hacer unpin de la página. Para marcar la página como "sucia", la clase `Page` cuenta con el método `void make_dirty()`. Notar que, si un proceso desea hacer unpin de la página y la página fue modifica, primero debe llamar al método `make_dirty()` para después llamar al método `unpin()`. El flujo de pins y unpins relevante para su proyecto se explicará más adelante en el enunciado.

Representación de tuplas. En IIC3413DB, que es una base de datos relacional, el objeto atómico de almacenamiento es una *tupla* (también llamada *record*). El módulo `relational_model` contiene la implementación de tuplas y del esquema. Para manipular las tuplas, IIC3413DB provee una interfaz en `record.h`.

Dado que en este laboratorio queremos explorar métodos eficientes para almacenar tuplas en una página, para nosotros tuplas serán simplemente una secuencia de bytes. Para esto necesitamos métodos para convertir las tuplas desde su representación interna en una clase de C++ a bytes y vice versa. Para esto, el módulo `relational_model` contiene un `record_serializer` que provee siguientes dos métodos:

```
void serialize(const Record& record, char* out)
void deserialize(const char* in, Record& record)
```

cuales nos permiten transformar un record a bytes que se guardarán en una página del disco, o procesar bytes traidos desde el disco a buffer cómo una tupla de nuestra relación. Es importante notar qué la serialización de los records en IIC3413DB incluye su largo y el esquema. Por lo tanto, el método `deserialize(const char* in, Record& record)` lee una secuencia de bytes cual empieza en la posición apuntada por `in`, interpreta correctamente el largo de la tupla codificada en estos bytes, y escribe el resultado en `record`. Esto incluso significa que los bytes en `record` que recibe `deserialize` pueden contener bytes extra (esto será importante para interpretar los bytes de una página más abajo). Para manejar relaciones, IIC3413DB almacena tablas y tuplas a través de un heapfile.

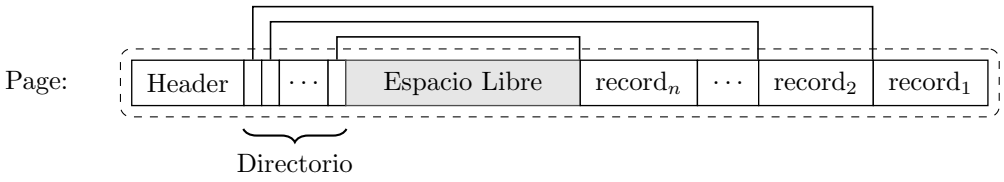
HeapFile en IIC3413DB

A continuación describiremos la estructura del heapfile y sus páginas ocupadas por IIC3413DB.

Formato y manejo del HeapFile.

HeapFile (implementado en la clase `HeapFile` en el módulo `heap_file`) tiene un formato muy sencillo, y consta de una secuencia de páginas identificadas por su posición en el archivo (e.g. la página 0 está en la posición 0, la página 1 en la posición 4096, etc.). Para identificar los records de una relación, nuestra clase `HeapFile` usa la estructura `RID` (record id), que se define cómo un par $(\texttt{pageno}, \texttt{dir\_pos})$, dónde `pageno` significa el número de la página, y `dir_pos` la posición del record en la página, identificado por la posición en el directorio de dicha página. Noten que `RID` es definido de manera implícita, y no será guardado en ninguna parte en nuestra base de datos. A continuación explicaremos el formato de las páginas.

Formato y directorio de cada página. Cada página del HeapFile debe tener la estructura de directorio similar que para manejo de "records de tamaño variable". Cada página (page) tiene el siguiente formato:



El header debe ser de tamaño fijo (8 bytes) y esta compuesto de dos campos: `dir_count` y `free_space`. El orden de aparición de estos campos en el header es el mismo (de izquierda a derecha – primero `dir_count` y después `free_space`). Específicamente, los campos contienen lo siguiente:

- `dir_count`: esta compuesto por 4 bytes y codifica el número de entradas en el directorio.
- `free_space`: esta compuesto por 4 bytes y codifica el tamaño de espacio libre, número de bytes libres en la página. El espacio libre se define como la cantidad de bytes sin usar entre la última entrada del directorio y el último record (`record`$_n$), esto es, la zona gris del diagrama de arriba.

El directorio consta de una secuencia de punteros [$\texttt{pointer}_1, \ldots, \texttt{pointer}_n$] donde:

- $\texttt{pointer}_i$: son 4 bytes que representan la posición donde se almacena el $\texttt{record}_i$. Esto es, la distancia desde el byte 0 en la página hasta donde empieza el string.

Por ejemplo, si el directorio tiene 3 entradas, entonces el directorio tendrá largo de 12 bytes, 4 bytes por cada entrada. Por último viene la sección de la página donde se almacena el contenido de los records, que se almacenan de derecha a izquierda para dejar espacio para insertar nuevos records y entradas al directorio. Recuerden que una página contiene precisamente 4096 bytes. Por lo tanto toda la información descrita arriba debe estar almacenada en un bloque de 4096 bytes.

Es importante notar que $\texttt{pointer}_i$ no necesita guardar el largo de un record, dado que la serialización de los records en IIC3413DB ya incuye esta información. Quiere decir, para leer el $\texttt{record}_i$ correctamente, basta con llamar `RecordSerializer::deserialize(const char* in, Record& record)`, dónde $\texttt{in} = \texttt{pointer}_i$.

En el módulo `heap_file` existe la clase `HeapFilePage` cual define los métodos para manipular las páginas del heapfile. Su trabajo en este laboratorio será completar algunos de estos métodos para respetar la lógica que se explica abajo.

Operaciones del HeapFile

Las operaciones fundamentales para el funcionamiento del storage es poder insertar tuplas nuevas, o eliminar tuplas existentes. El método más sencillo que provee la clase `HeapFile` es:

```
void delete_record(RID rid)
```

Aquí `RID = (pageno,dir_slot)`, y el `HeapFile` consigue la página `pageno` y elimina el record en la posición del directorio `dir_slot`. Para esto, *el único* cambio que se hace a la página es cambiar la entrada del directorio en la posición `dir_slot` a -1. No se cambia ningún otro byte. En particular, eliminación de una tupla no aumenta el espacio libre (sino lo pierde temporalmente), y mantiene el mismo tamaño del directorio de la página.

El segundo método crucial nos permite insertar records en una página. Para esto, `HeapFile` provee el siguiente método:

```
RID insert_record(Record& record)
```

Aquí el método toma un record y lo inserta en la primera página con el espacio suficiente pare este record. En particular, esto significa que el método empieza iterar sobre las páginas del `HeapFile` partiendo desde la primera, revisa si hay espacio suficiente para colocar el record, y si hay, lo coloca en el espacio libre de la página y actualiza su directorio. Recuerden que el campo `free_space` del header de la página contiene la información necesaria para calcular el espacio libre y hasta dónde llega el espacio libre. Dependiendo del estado de nuestro directorio, existen dos opciones para colocar la nueva tupla en la página *cuando haya espacio suficiente para realizar la operación*:

1. Si hay alguna entrada del directorio con el valor -1 (un record eliminado), la tupla debe ser colocada en el *primer directorio* con el valor -1. Esto significa que este directorio ahora contendrá la posición del byte dónde empieza nuestro record. (Acuerden también que en el caso que el último record fue eliminado, esto no significa que su espacio pasa a ser espacio libre. Espacio libre siempre termina dónde nos señala el valor del campo `free_space`). También se actualiza el valor del campo `free_space` con la nueva información.

2. Si no hay ningun directorio con el valor -1, se crea un nuevo directorio y la tupla se ingresa al espacio libre, con el valor del nuevo directorio actualizado a la posición de nueva tupla. Es importante mencionar que en este caso el tamaño necesario para poder almacenar la tupla nueva también incluye el tamaño de un directorio nuevo (i.e. la condición es que tamaño de un directorio + tamaño de la tupla sea menor que la capacidad de `free_space`). Posteriormente se actualiza el tamaño del directorio.

En los dos caso, el método devuelve el RID de la nueva tupla. Es importante mencionar que para escribir la tupla en la página, es primero necesario serializarla con los métodos explicados arriba (el código tiene un hint también).

La última operación importante del `HeapFile` es:

```
void vacuum()
```

En este método `HeapFile` itera página por página, y elimina todo el espacio cual se perdió por eliminación de tuplas, y todas las entradas del directorio que ya no se usan. Para esto, el algoritmo que se usa para modificar una página es el siguiente:

- Primero se detectan las entrada activas del directorio (las entradas con un valor distinto al -1).
- Se arma una página nueva cual contiene solo estos records en el mismo orden en el cual aparecen *en el directorio* de la página actual.
- La página actual se reemplaza con la página nueva.

Noten que la página nueva probablemente tendrá distinto número de directorios y distinto free space. Cómo un ejemplo, supongan que la página actual contiene tres entradas en el directorio, correspondiendo a los records r_1, r_2 y r_3. Si el record r_2 fue eliminado, se perdió su espacio, y la entrada dos del directorio quedó en -1. Esto significa que el `vacuum()` debería producir una página cual contiene solo dos directorios, y almacena solo r_1 y r_3. La página nueva también tendrá el mismo orden de las tuplas cual indica el directorio de la página original. Es importante mencionar que en el caso de que insert usa un directorio cual fue eliminado antes, esto puede hacer que el orden de las tuplas no será el mismo cómo en la página original, sino que solo sigue el orden de las entradas en el directorio. De nuevo, el algoritmo es simple: iterar por el directorio, y armar una página nueva con las tuplas dadas por este orden.

Los tres métodos están implementados en la clase `HeapFile`, pero cada manipulación de los datos en una página se hace con un método de la clase `HeapFilePage`. Algunos métodos de esta clase no fueron implementadas en IIC3413DB, y es su tarea completar su implementación.

Tareas para hacer en este laboratorio

Problema 1: Eliminación de tuplas [1 punto]. En la clase `HeapFilePage` falta implementar el método:

```
void delete_record(uint32_t dir_pos)
```

Este método es llamado por `delete_record(RID rid)` de la clase `HeapFile` para eliminar la tupla identificada por el `RID`, y debería implementar la lógica explicada arriba (actualizando el valor del directorio correspondiente). Cómo un hint, se les recomienda revisar cómo en el constructor de la clase `HeapFilePage` uno actualiza la variable `dirs`, la cual apunta a la posición dónde empiezan los directorios. Por la magia de C++ ahora pueden ocupar la variable `dirs` (dereferenciada) cómo si fuese el arreglo con las entradas del directorio en sus entradas, y aceder directamente a sus posiciones.

Problema 2: Inserción de tuplas [3 puntos]. En la clase `HeapFilePage` falta implementar el método:

```
bool try_insert_record(Record& record, RID* out_record_id))
```

Este método es llamado por `RID insert_record(Record& record)` de la clase `HeapFile` para insertar la tupla `record`, y debería validar si la tupla cabe en la página (devolviendo true en este caso), y en el caso positivo, insertarla (en un directorio existente, o uno nuevo cual se agrega al final) y en el espacio libre, actualizando la información de la página de manera adecuada.

Problema 3: Vacuum [2 puntos]. En la clase `HeapFilePage` falta implementar el método:

```
void vacuum(const Schema& schema)
```

Este método hace el vacuum al nivel de la página cómo fue explicado arriba. Cómo una sugerencia, para actualizar el contenido de la página al cual se le aplica `vacuum()`, se recomienda reservar un espacio del tamaño de la página `char [SIZE]` y armar los bytes de la página nueva en este espacio, para posteriormente copiarlo a los bytes de la página que se está procesando. Armar la modificación in-place puede ser bastante complicado en este caso. Para trabajar con una tupla, siempre pueden instanciar el objeto `record` usando la información del `schema`.

En términos de su implementación, hay que notar que `HeapFilePage` ya maneja los pins y unpins para ustedes (pins se hacen en el constructor y unpins con el destructor), entonces en un flujo normal no deberían tener problemas con esto. Lo qué si deben manejar bien es el `make_dirty()`, cual se debería marcar cada ves que una página fue modificada.

Es importante que su implementación siga las instrucciones anteriores, esto es, que siga el formato de la página entregado. Para la corrección se harán varios test de inserción y lectura, y las páginas del `HeapFile` deben estar exactamente como se describen anteriormente. Durante la primera semana del laboratorio se publicarán algunos test para que puedan comprobar que el formato implementado es correcto.

Código

El código base está disponible en [1]. Asegúrense que están usando la última versión del sistema! Los tests se publicarán en el mismo repositorio.

Evaluación y entrega

El día límite para la entrega de esta tarea será el viernes 12 de abril a las 23:59 horas. Para ello se utilizará el formulario de canvas. *Su entrega debe contener todo el código de la carpeta* `src` *y nada más, y debe compilar cuando se les agregan otras carpetas en el repositorio. El código de la carpeta* `src` *debe ser entregado cómo un archivo* `.zip`. Por último, la evaluación será en base a varios test que su solución debe responder.

Ayudantía y preguntas

El día viernes 5 de abril se realizará una ayudantía donde se darán más detalles sobre IIC3413DB y el laboratorio. Para preguntas se pide usar el foro del curso.

Referencias

[1] Carlos Rojas, Diego Bustamante, Vicente Calisto, Tristan Heuer. PUCDB2024. https://github.com/DiegoEmilio01/IIC3413.