lab1 write up

1. Tuples and Schemas

The first step of this lab is to implement the basic elements of the database, tuples. In our lab, tuples have fixed length, which means, every non-empty tuple has the same bytes.

For `TupleDesc`, each field has the information of the type of data that should be stored in this field and the name of this field; these two attributes are encapsulated in a class called `TDItem`. In this class, I used a `List<TDItem>` to store all of the `TDItem`s, called `fieldList`. Therefore, for the constructor, the `fieldList` can be initialized. If the field name is not specified, it will use the default empty string as its field name. One important method is `getSize()`, from which we can get the size (in bytes) of each tuple, and this size is calculated by adding up all the size of each field.

`Tuple` has the attribute of `TupleDesc`, from which we can know the metadata of this tuple. Another important attribute that tuple has is the values of each field. Similar to `TDItems`, I use an `ArrayList` to store all of the values of each field. `Tuple` also has the attribute of `RecordId`, from which we can identify the address of this tuple, i.e., in which page at which slot.

2 Catalog (new class: Table)

`Catalog` consists of a list of all the tables and schemas that are currently in the database. To implement this, I use two `HashMap`s to store the map between the table name to the table id (`talbeName2Id`), and also the map between the id to the file (`id2Table`). For convenience, I create a class named `Table` with attributes (`DbFile`, `name`, and `pkeyField`), which represent the database file (in our lab, heap file), the table name and the primary key field. With `Table` class, the method `addTable` can be easily implemented by adding a table instance to the `id2Table` map. Meanwhile, the `tableName2Id` can be updated as well. With these two Hash maps, the schema, heap file, table name, and primary key can be accessed if the table id is provided.

3. Buffer Pool

The buffer pool can cache "active" pages in memory that have been read from disk. One import attribute of `BufferPool` is the maximum pages it can hold at the same time, named `maxPages`. For the easy access to the page with a provided page id, a hash map is used to map from the id of page (`PageId`) to the page (`Page`) that is currently in the buffer bool, named `pageMap`.

The logic of the `getPage` method is, for a given page id, if this page is in the buffer pool, that is, to check if `pageMap` contains this page id. If not and the current buffer pool is not full, it will ask heap file to get this page from the disk. More specifically, the `DataBase` will get the heap file that the requested page resides in, and the heap file will read the desired page from the disk (the heap file access method will implement later). Once the page is read to the buffer pool, the `pageMap` will update and this page will be returned.

4. HeapPage and HeapFile

From the process flow of query execution in `SimpleDB` that Ryan presented in class, we know that we must get to a certain tuple or page efficiently. With this understanding, I think that's why we have the `RecordId`, which helps to identify a tuple, and the `PageId`, to identify a page. Second, I have a clearer picture of how these heap files, heap pages, and tuples are stored, how the buffer pool operates.

4.1 RecordId and PageId

`PageId` is an interface and since we are using `HeapPage`, we actually implement the `HeapPageId`. `HeapPageId` is to identify the requested heap page. Since the location of each page, that is, in which table and at which page, is unique, it is reasonable to get the hash code with the combination of its table id and the page number. Also, the heap file access method needs these two attributes to get the desired page in disk with the given table id and the page number.

`RecordId` is to identify the requested tuple. Similarly, each tuple's location at which page and which slot is identical, and can be used to get the hash code.

4.2 HeapPage

In `HeapPage`, it's all about the bitmap to mark the usage of each tuple and the fixed-length records. Based on these, the number of records for each page can be calculated. To check if a slot is used, we can calculate where its indicator bit and then check whether it is 1 or 0. With the bitmap, we can implement the `iterator()` by collecting all the valid tuples (bit indicator = 1) as a list and then return the iterator, which can be leveraged in the `HeapFileIteror`.

4.3 HeapFile (new class HeapFileIterator)

In `HeapFile`, there first important method is the `readPage`, to read a page from disk. Since those pages are stored consecutively, the offset of a page can be calculated by the page number times the bytes for each page. As the guide indicated, the `RandomAccessFile` can be used to read all the bytes for any page with the provided offset and number of bytes to read. The total number of pages in the file can be calculated by `ceil(file.length() / bytes per page)`.

The second is the `iterator`, to implement it, I created a `HeapFileIterator` class which implements the `DbFileIterator`. In this class, the `open` method is to get all the tuples on the first page and initialize the iterator as the list iterator. So there is a help method `getTuplesFromPage` to return all tuples on a certain page. With the `next` calls, the iterator will keep query the next tuple in the tuple list or get tuples from the next page and update the iterator.

5. `SeqScan` operator

The `SeqScan` has an attribute of the `HeapFileIterator`, named `itt`. The `open` method will initialize the `itt` as the iterator the heap file and then call the `open` method of `itt` itself. The `hasNext` and `next` methods actually call the `hasNext` and `next` methods of `itt`.

6. Test

About the unit test, I assume that a test for the `HeapPage` can be helpful, since, during my coding process, it took plenty of time to understand how it works, for example, the `getPageData` method.