Description of Lab 1:

Lab 1 was implementing SimpleDB, a relational database that stores tables with tuples as each row. These were represented with the `Tuple` class which stores each `Field` as a list whose types are determined by the `TupleDesc`. It stores the metadata for each `Tuple` including the names and types of each `Field`.

At the heart of SimpleDB is the `Database` class that is a singleton class with references to various other components (e.g. the `Catalog` and `BufferPool`). The tables are stored in an aptly named `Catalog` which keeps track of the relationship between the tables, their names, their ids, and their primary keys (if they are present). While interacting with SimpleDB, the `Database` is the appropriate method for interconnecting the different components, as they are not stand alone.

One component is the `BufferPool`, which keeps track of the pages that have recently been accessed (i.e. it functions as a cache of pages). This component cannot stand alone because it does not have its own method for reading the pages from disk. This is where the `HeapFile` comes in. It has a method for reading pages from disk into itself (since it is a representation of a file within the database). Once it reads a page, it can provide it to the `BufferPool` that can cache the page for quicker retrieval later.

The `HeapPage` is the random collection of `Tuples` that is used by the `BufferPool` and `HeapFile` as an intermediary to access the `Tuples`. Similarly, the `HeapPageId` just keeps track of which table and page number a specific `HeapPage` is tied to.

Finally, SimpleDB has operations which are handled by various types of operators. In this lab we worked through `SeqScan` which in most cases functions as a wrapper of the `iterator` returned by `HeapFile#iterator()`. This shows another interconnection between the components and demonstrates the higher level of abstraction that the clients would be more likely to interact with.

Design decisions:

- `Tuple` utilizes an underlying `List<Field>` for quick `get/set` of the data using indices.
- `TupleDesc` utilizes `List<TDItem>` for quick `get/set` of field names/types.
- `Catalog` utilizes two `Maps` with `id→name` and `name→File/PrimaryKey`. This is important because it allows the `files` to be accessed from both `name` and `id`. This also allows for quick lookup of the `name` from the `id`.
- `BufferPool` utilizes a `Map` with `PageId→Page` in conjunction with a `maxCapacity` value. This is important because it allows the `BufferPool` to cache a certain number of `Pages` that have been seen with quick lookup using their `PageId`.
- `HeapFile` utilizes a `RandomAccessFile` to deserialize the bytes from disk back into a `HeapPage` that can be passed to other components like the `BufferPool`. `RandomAccessFile` is important because it enables reading from an offset (i.e. the `PageId` that dictates the location of the serialized page).
- `HeapFile#iterator()` iterates `Tuple` by `Tuple` loading one `Page` in at a time (to prevent overloading the memory with larger tables). Additionally it stores the state of the iterator in the given `HeapFile`, so that when referenced by the `Database` singleton it's consistent.
- `SeqScan#getTupleDesc()` breaks the current `TupleDesc` stored for the given `tableid` into subparts (i.e. names and types) so that the alias can be prepended to each `Field`.

Example of a unit test that could be added to improve the set:

One unit test that could be added to improve the testing suite would be to verify that the `BufferPool` worked properly. For example, a unit test that asserts that the behavior of the permissions of the `Page` returned from `BufferPool#getPage()` is correct.

Changes you made to the API: None.

Describe any missing or incomplete elements of your code: None.

Additional Feedback: None.