Lab 2 Report

Group Members:

Yee Yi Xian 1004601

Rebecca Ling 1004413

Design Decisions and Code Explanation

Below we explain our design decisions and the ~~important~~ non-trivial part of our code, as split into the classes we were required to implement for Lab 2.

Predicate and JoinPredicate

- Predicate encapsulates the operands and the comparative operator to the field(s) that operations call to make specific comparisons. Predicate is for comparing tuples to a specified field value and join predicate is for binary comparison for join operation.

Filter

- Filter iteratively fetches the next tuple from single child Operator, and tests the predicate condition to a specified field value

Join

- The iterator checks whether child1 has a tuple first. If it does not, it will return null. If it has, it would look for the tuple in child2 as well and returns a marged tuple found from both children.

IntegerAggregator and String Aggregator

- In order to merge a new tuple into the aggregate, we first needed to check if the field existed in the hash, gbHash. If it does not exist, insert it into the hash. Based on the sql operand function being passed in (MAX, SUM, COUNT, MIN, AVG), we will then update the aggregate value for the relation. Groupby is the operation that defines the observation group of each aggregate calculation.

Aggregate

- In the constructor, we first check whether the gfield has any groupings. If it does, we check the field type of afield. If it does not, we get the field type from the child itself. From there, we create a new IntegerAggregator or StringAggregator, depending on the field type, and assign it as an aggregator.
- This makes it easier for subsequent implementations of the rest of the skeleton code in this class, as there is less need to worry about the field type.

HeapPage

- We implemented deleteTuple(), insertTuple(), markDirty(), isDirty() and markSlotUsed().
- In deleteTuple(), we retrieve the tupleNo from and check whether the slot is used, using isSlotUsed(). If it is, it will replace the tuple with a null and use markSlotUsed() to mark the slot as not used. If it is not used, it will throw an exception, saying slow is empty.
- In insertTuple(), if getNumEmptySlots() returns 0, it will throw the DbException, saying that it is full. Otherwise, it will iterate through the entire page and look for an empty slot to change the tuple from null to the desired tuple and use the markSlotUsed() to mark the slot as used. It also sets a new RecordId to the tuple.

HeapFile

- We implemented insertTuple() and deleteTuple() methods.
- For insertTuple(), we looped through existing pages to find an empty slot in a page to insert the tuple. If there are no existing pages (ie. pageArr is empty), we create a new page to add in a tuple.
- For deleteTuple(), we retrieved the pageId and heap page from the Tuple and the database respectively, then deleted the tuple and returned the updated page.

BufferPool

- We implemented insertTuple(), deleteTuple, flushAllPages(), discardPage(), flushPage() and evictPage()
- For insertTuple(), we inserted the tuple into the heapFile, file, and looped through the pageArr to check for any changed pages to mark dirty and assign the id to the page. We also evicted any pages that were larger than the current buffer pool.
- For deleteTuple(), we also marked any pages that were dirtied by the operation , and assigned the id to the page
- flushPage() checks if the page is dirty. If it is dirty, it will flush the page to the disk using writePage(), and mark the page as not dirty. flushAllPages() calls the flushPage method to flush all dirty pages to the disk
- discardPage() removes a specific page id from the buffer pool
- evictPage() discards a page from the bufferpool, and flushes the page to disk to ensure dirty pages are updated on the disk. We implemented a **randomised page eviction policy** as it is a safer option (more consistent) to prevent situations where things don't work out according to the best case assumptions.

Insert and Delete

- It inserts/deletes tuples through the BufferPool, which calls the insert/delete functions in the HeapFile, which calls the insert/delete functions in the HeapPage. These implementations were all explained earlier.

Missing or incomplete elements

We managed to successfully implement everything as required in Lab 2. Yay!