CS 143 Lab 2 Report

Aanish Patel Sikora: 804028077
Tushar Shrimali: 804070047

FILTER/JOIN:

Predicate: Has methods to get the field, op and operand. Only important function is filter which uses Fields' compare method to compare a particular field of the two tuples.

Join Predicate: Has methods to get the two fields and operator. Only important function is filter which takes two tuples and applies the Field's compare method to specified fields from both tuples.

Filter:This is the equivalent of a relational select. The crucial method is fetchNext. This basically iterates through all of the tuples and applies the Predicate.filter method to them.

Join: This is the equivalent of a relational join. The constructor takes two Dbiterators pointing to two sets of tuples and a Join predicate. The crucial method is fetchnext. This method returns the next tuple that the join creates, or else if there is none. I implemented it using a nested loops. Tuple temp starts off as null and is assigned to the first tuple of the first child pointer. Then, for each tuple pointed to by the first child, the function goes through each tuple in the second child pointer. If any tuple from the first child and any tuple from the second child pass the filter, another function called CrossProduct is called. This function takes in the two tuples and a tupleDesc which has all fields from both tuples. The function returns a new tuple which is a combination of the two and this is returned by the fetchnext method.

AGGREGATES:

Integer Aggregator: This calculates, over a set of IntFields, any aggreate: max, min, count, sum or average. Either grouping or no grouping is specified and depending on which, an appropriate TupleDesc object is initialized. The crucial function here is mergeTupleIntoGroup which takes a tuple as a parameter. There are four possible scenarios. Either grouping or no grouping and either this is the first or non-first tuple being grouped. If it is the first tuple, there are no other tuples to compare against and the aggregate operators become simple. However, if it is not the first tuple, comparisons must be made against other tuples to perform the aggregation operator (this depends on the specific operator in question). If there is grouping specified, the specific field on which to group by must be identified and additional comparisons must be made. Once again, either this tuple is the first to group or not. A helper function called tupleAggregator handles the 5 different aggregate operations. It is called in each of the four above cases, with parameters set to distinguish them. TupleAggregator handles the special case of the tuple being the first one and if grouping is or isn't specified. If the operation is minimum, the current value is tested against the comparison value and either replaces the comparison value or it doesn't (depending on current < comparison). If there is no comparison value, current is by default the minimum. Maximum is

analogous. Sum is similar as well. Here, instead of comparing the values, we simply sum them. Similarly, for Count, instead of storing the value, we either store 1 (if its the first tuple) or increment the current value by 1. Average is the most difficult case. Unlike the previous four operators where there are only two distinct scenarios, average treats all 4 different scenarios differently. Special member variables are kept; two arraylists called sum and count and an integer called avgIndex. In the first case of no grouping and the first value, simply add 1 to count and the value to sum. The corresponding tuple is simply set to the value. Next, for no grouping and not the first value, we add the 1 to the count of the first group (this is the only group) and the value to the sum of the first group. The tuple is set to the integer division of sum/count. Now, for grouping and the first value. The count and sum of this group, is set to 1 and the current value. Again, the value is set as the average. For grouping and not the first value, for this group, count is incremented and the value is added to the sum. The integer division yields the average. ArrayLists are needed to store count and sum as there may be many groups all of which have a corresponding average which needs to be independently calculated.
String Aggregator: Analogous at a high level to integerAggregator, except the only operation supported is count.
Aggregate: This computes aggregation over a single column. In the constructor, either a integer or string aggregator is created as specified. An iterator is created that will iterator over all the tuples in the column and call the mergeTupleIntoGroup method.

INSERT:This inserts tuples into a specific table. The fetchnext method goes through each of the tuples and uses the BufferPools insert tuple method to insert tuples into a particular table.
DELETE:This deletes tuples from a specific table. The fetchnext method goes through each of the tuples and uses the BufferPools delete tuple method to delete tuples from a particular table.

HEAPPAGE:
Deletetuple - Set SlotUsed to false with markSlotUsed(RecordId().tupleno(), false) and update the tuples array to null (delete value).

Insert tuple -For all slots, iterate through them starting from the first and insert tuple into first empty slot. Update appropriate bitmap slot by calling markSlotUsed on slotindex with true.

isDirty/markDirty - Straightforward functions. Set and return tid according to passed parameters.

markSlotUsed -
Calculate byte in the header(a bitmap) that contains the bit for the tuple.
To fill slot - The above is ORed with the 0s containing 1 at the bit posiition. The resultin value will contain 1 at the position and hence slot will be filled

To clear slot - The above is ANDed with the 1s containing 0 at the bit posiition. The resulting value will contain 0 at the position and hence slot will be unmarked.

HEAPFILE:

writePage - Open new RandomAccessFile and navigate to end of file with seek(). Write the data from new page and close file.

insertTuple -
1) Check if any of the existing pages in the bufferpool have any slots empty. If so, insert Tuple t and page p to the return ArrayList. We need to return pagesarr since return type of function is ArrayList<Page>
2) Else, create a new HeapPage and insert tuple in page into first slot. Write page from bufferpool (Note: We can also call writepage here but I've added the implementation within the function) and return ArrayList containing the new page.

deleteTuple - Locate page with parameters tid and t.getRecordId().getPageId(). Call Page.deleteTuple implemented from HeapPage previously and return page.

BUFFERPOOL:

insertTuple - Call insertTuple (tid and tuple t passed as parameters) on a heapfile (using tabled). Mark page dirty too.

NOTE: insertTuple in Bufferpool uses insertTuple from HeapFile and insertTuple in HeapFile uses the same from HeapPage. This is the added layer of abstraction as specified in the spec.

deleteTuple - Analogous to insert

flushAllPages - Call flushPage on all Pages

flushPage - If page is dirty, write page to disk.

evictPage -
EVICTION POLICY: Since we are not required to use any particular eviction policy or care about speed constraints, the policy used here is very straightforward although not so feasible. We iterate through the list of pages and discard the first page that has returns null for isDirty(). The page is then flushed to be certain information (if any) is updated on disk.