Time spent on the project: 12hours.
Group task division: coding and report done 50/50 by the 2 group members.

Heap File Class

- InsertTuple: we must first find a page with enough space for the new tuple we want to insert. If a free page is available, we insert the tuple directly into the page and return. If no empty pages are found, we first create a new page, insert the tuple, then add the new page to the corresponding file for that table (we access the file with the "raf" method). This method returns an array list containing the modified / new page.
- deleteTuple: given a tuple t, the method retrieves the page id and via the buffer pool method deletes the tuple from that page. The method returns an array list of the modified page.

Heap Page Class

A bug in the iterator class was present from previous PAs, blocking the delete test. For this reason we have created the iterator class outside the Heap page class to have better control over it. The same for the HeapFile iterator.

- InsertTuple: first finds an available slot in the current page header. Once found, the method sets the record id of the tuple pointing to the pid of the current page, then adds the tuple to the page's tuple list and marks the corresponding slot as used.
- deleteTuple: given a tuple t, the method first retrieves the record id, then set the corresponding slot of the record to empty and delete the relative index in the tuple list (we also set to false the used slot flag).

Integer Aggregator Class (same logic for string aggregator class)

- mergeTupleIntoGroup: this method add a given tuple field into the group following an operator initialized in the constructor. First we check if the index of the group field is already initialized by checking the "gbfield" data structure. We used 2 different arrays to memorize the grupped values: "countGroupBy" and the "ValuesGroupBy". The goal of this method is to add the right values to these 2 data structures. We used a switch case structure to select the right operator case and manipulate the values to aggregate them in the correct way.
- iterator: First we check if the index of the group field is already initialized by checking the "gbfield" data structure; depending on the result we follow one of the 2 different branches of the method. In the first case where "gbfield" has not been initialized, a tuple is created and initialized with the correct value depending on the operator. In the second case we must go through all the already present values of the Group By and manipulate the values depending on the operator we have.

Join Class

- fetchNext: multiple concatenated loops : 2 external loops iterating until there are no more elements to merge between the 2 children component given in input; 2 internal loops, activated only when the tuples are matching a predicate, iterating over all the fields that need to be joined from 2 matching tuples.

Hash Equi Join Class

fetchNext: When the 2 tuples evaluated match, we merge them (for this a new support tuple is created). The logic is similar to the Join Class, where 1 child stay still and another child is analyzed in a loop over all the tuples. Once all the elements of one child are analyzed, we move to the second element of the first child, rewind the second child and repeat the process.

Aggregate Class

- fetchNext: if the iterator has already been initialized we simply return the next element. Otherwise we need to merge the tuples present in the child using an integer or string aggregator depending on the case, and then open the iterator.

Insert Class

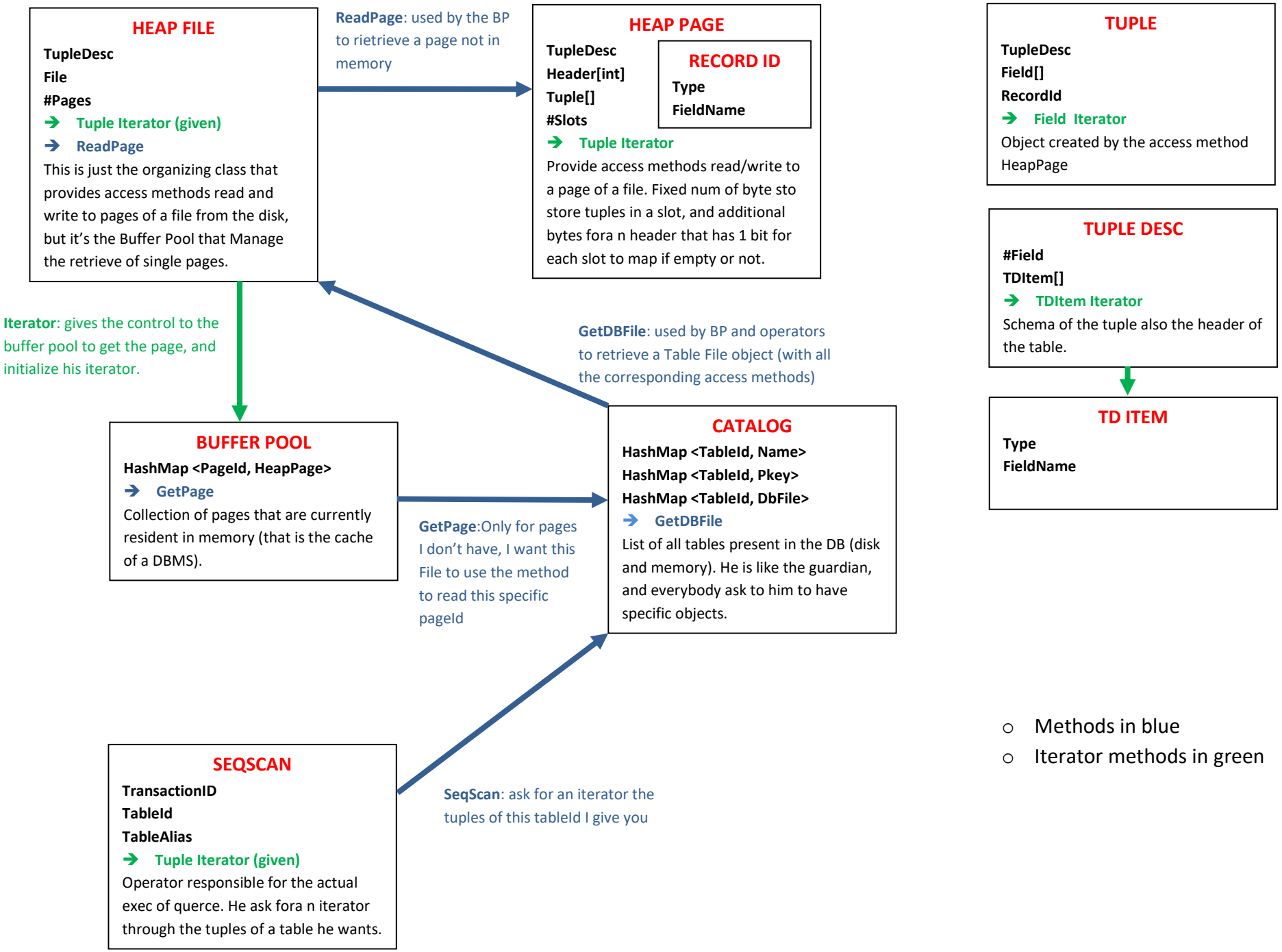
- fetchNext: iterate through all the children and insert the tuples in the referenced table using the buffer pool method “insertTuple”. This method also return a tuple with one field indicating the number of tuples inserted.

Delete Class

- fetchNext: iterate through all the children and delete the tuples in the referenced table using the buffer pool method “deleteTuple”. This method also return a tuple with one field indicating the number of tuples deleteed.

In the next page the high level schema of the simple db project 1, showing the connections between classes and methods needed also for projects 2 and 3. This schema helped to keep track of all the basic connections between classes.

Sources of informations: Javadoc, stackoverflow, github


HEAP FILE
TupleDesc
File
#Pages
➔ Tuple Iterator (given)
➔ ReadPage
This is just the organizing class that provides access methods read and write to pages of a file from the disk, but it's the Buffer Pool that Manage the retrieve of single pages.
ReadPage: used by the BP to rietrieve a page not in memory
HEAP PAGE
TupleDesc
Header[int]
Tuple[]
#Slots
RECORD ID
Type
FieldName
➔ Tuple Iterator
Provide access methods read/write to a page of a file. Fixed num of byte sto store tuples in a slot, and additional bytes fora n header that has 1 bit for each slot to map if empty or not.
TUPLE
TupleDesc
Field[]
RecordId
➔ Field Iterator
Object created by the access method HeapPage
TUPLE DESC
#Field
TDItem[]
➔ TDItem Iterator
Schema of the tuple also the header of the table.
TD ITEM
Type
FieldName
Iterator: gives the control to the buffer pool to get the page, and initialize his iterator.
GetDBFile: used by BP and operators to retrieve a Table File object (with all the corresponding access methods)
BUFFER POOL
HashMap <PageId, HeapPage>
➔ GetPage
Collection of pages that are currently resident in memory (that is the cache of a DBMS).
GetPage:Only for pages I don't have, I want this File to use the method to read this specific pageId
CATALOG
HashMap <TableId, Name>
HashMap <TableId, Pkey>
HashMap <TableId, DbFile>
➔ GetDBFile
List of all tables present in the DB (disk and memory). He is like the guardian, and everybody ask to him to have specific objects.
SEQSCAN
TransactionID
TableId
TableAlias
➔ Tuple Iterator (given)
Operator responsible for the actual exec of querce. He ask fora n iterator through the tuples of a table he wants.
SeqScan: ask for an iterator the tuples of this tableId I give you
Methods in blue
Iterator methods in green