Buffer Pool Class

- getPage: fixed the method for the retrieve of a page (from disk) not present in the BufferPool. From first assignment the method was built to manage only HeapPages, now it must manage any kind of page including heap and betree pages (bug of my first assignment code).
- flushPage: from a given pid it takes the corresponding page in the BufferPool check if it's dirty and update the disk if dirty. Finally after updating the page in the disk it set the dirty bit to false.
- evictPage: the policy I built takes the first page in the bufferpool that has not been used. I extract the first page that is not dirty and then discard it. If in some rare case I have no pages in that state in the buffer Pool, I flush all the pages and delete the first of the list after the flush.
- discardPage: remove the page with the given pid.
- deleteTuple/insertTuple: I add or delete a tuple from a table. Both the method insertTuple() and deleteTuple() of the DBFile class returns the list of modified pages after the action of deleting or inserting. I need to update the buffer Pool for each modified page and mark the page as dirty. If the buffer pool is full I first call the evictPage method to make place for the new page.

BTreeFile Class

- findLeafPage: steps of the process
    - If the given page is a leaf page: I found the page I wanted; I can return it.
    - If the given page is an internal page: I still have to check the rules in order to go down to find the correct leaf page. I first check the special rule for field given == null and I go down to the left child. In all the other cases I pass through all the entries of the internal page and check: if the key is less than the page I want I go down to the left child; if there are no more entries next I go down to the right child. Every time I go down to the right or left child I recursively call the findleafpage method to check if I am still in another internal page or if I have found the leaf page.
- splitLeafPage/splitInternalPage: A first step is to move half of the elements (tuples for leafpage BTree entries for internal page) to a new page of same kind. Once done that process I need to update (only for leaf pages) the siblings pointers (in my logic I'm mooving the elements from the first left part of the page so I have to change the pointers of the left sibling). For the internal page instead I have to delete the middle entry from the current page as in internal page we do not keep the element after moving it to the parent (another difference with splitting leaf page). Then I have to update the pointers and entries in the parent page. (comments in the code for more details).
    - ➔ Main difficulties: understand the correct pointer situation and how to update it.

Only 2 classes for the leaf pages for extra points

- stealFromLeafPage: I first check the number of pages to steal from the victim in order to maintain the correct balanced number of element in each page. Then with a left or right iterator, depending on the flag passed, I delete and add elements in order. Finally I update the parent element key that points to the 2 leaf page used, with the correct value that represent the new leaf page situation.
- mergeLeafPages: I delete all the elements from a victim page to move them to another page. I set the correct sibling (as I have deleted a page), mark the page as empty and delete the parent entry that was used to point to the 2 child pages now merged.

In the next page the high level schema of the simple db project 1 and 2. Drawing this schema helped me a lot to understand the connections between classes and build the methods for both projects.

Sources of informations: Javadoc, stackoverflow, github

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.

Iterator

TD ITEM

Type
FieldName

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.

BTREE LEAF PAGE

Header[int]
Tuple[]
Left/right sibling
#Slots
➔ **Tuple Iterator**
Add the pointer management methods for sibling leaf pages. In the Btree structure the leaf page is the only to contain the actual content o f a table (tuples)

BTREE PAGE ID

PageIDNum
TableID
PageCatalog (Internal, Leaf, Root etc.)
➔ **GetPageNum()**

GetChild: can be another internal page or a leaf page

BTREE ENTRY

Key
LeftChild
RightChild
➔ **GetRight or Left child**

BTREE INTERNAL PAGE

Header[int]
Children[PageIDNum]
Child category (leaf or internal
Keys[BTreeEntry.Key]
➔ **BTreeEntry Iterator**
Contain elements that allows the tree search based on the key. Doesn't contain content, only index references of keys that are in the leaf page child.

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.

BTREE FILE

TupleDesc
File
TableId
➔ **Tuple Iterator (given)**
➔ **BTreeEntry Iterator (given)**
➔ **ReadPage**
➔ **FindLeafPage**
➔ **Split**
➔ **Steal / Merge**
Allow the DBMS to manage a balanced data structure. Use Iterator to navigate the structure.

ReadPage: used by the BP to rietrieve a page not in memory. Returns only page with tuples not structure pages

Iterator: gives the control to the buffer pool to get the page, and initialize his iterator on that page retrieved.

BUFFER POOL

HashMap <PageId, ….Page>
➔ **GetPage (add a Page from disk)**
➔ **Flush (update page to disk)**
➔ **Evict (delete page)**
➔ **Insert/Delete tuple**
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.

GetDBFile: used by BP and operators to retrieve a Table File object (with all the corresponding access methods)

SeqScan: ask for an iterator the tuples of this tableId I give you

SEQSCAN (operator)

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.

Btree Indexing: an index is a structure that improves query optimization to be faster. There are some data structures without data, where the only purpose is to help the navigation through the multitude of keys representing real data. In particular BTree structure avoid imbalance problem related to most of other tree structure.

BTREE ENTRY (1)

BT PageID left pointer	*BT Entry Key*	*BT Page ID right pointer*

BT entry iterator

BTREE ENTRY (2)