**Team Members**

Preyang Shah (Leader) - A20385823 - pshah115@hawk.iit.edu

Riten Chhatrala - A20385102 - rchhatrala@hawk.iit.edu

Meet Patel - A20379333 - mpatel104@hawk.iit.edu

Niket Patel - A20384264 - npatel102@hawk.iit.edu

**File List**

Makefile

README.txt

buffer\_mgr.c

buffer\_mgr.h

buffer\_mgr\_stat.c

buffer\_mgr\_stat.h

dberror.c

dberror.h

dt.h

expr.c

expr.h

record\_mgr.c

record\_mgr.h

rm\_serializer.c

storage\_mgr.c

storage\_mgr.h

tables.h

test\_assign3\_1.c

test\_expr.c

test\_helper.h

|  |  |  |
| --- | --- | --- |
|  |  |  |

**Problem Statement**

* The main aim of this assignment is to implement a record manager. Tables with fixed schema is handled by the record manager. Insert, delete, update, scan can be done through the records in a table.
* When scan is done, it is associated with a search condition and returns records that match search condition. In a separate page file, each table should be stored. Also, page of the file should be accessed through the buffer manager implemented.

**Data Structures**

|  |
| --- |
| In Record Manager, few attributes are defined. We have implemented a structure named tableInfo. Attributes of the structure are as follows:   * scLen- Length of the schema is indicated |
| * recoBeginPage - It indicates the Page number of the first record in a table |
| * recoEndPage - It indicates the Page number of the last record in a table. |
| * tupleSize - It indicates the number of tuples in a table |
| * slotWindow - It indicates the size of a slot holding a record. |
| * maxSlots - It indicates the maximum number of slots. |
| * tstone\_head - It indicates the head node of the tstone list.   Few attributes are required to perform the scan to retrieve all tuples from a table which satisfies some condition. We have implemented the structure of such attributes known as recordInfo. Attributes of the structure are as follows: |
|  |
| * condition - It indicates the condition defined by the client. |
| * recentSlot - It indicates the current slot of a record. |
| * recentPage - It indicates the current page of a record. |
| * totalPages - It indicates the total number of pages. |
| * totalAvailSlot - It indicates the total number of slots.   Few attributes are required to track a deleted record. We have implemented the structure of such attributes known as recordInfo tableNode. Attributes of the structure are as follows: |
|  |
|  |
| * id - page id, slot id and tombstone id of a record is stored in it. |
| * next - next node in the linked list is stored in it. |
|  |

**Record Manager & Table Functions**

|  |
| --- |
| **initRecordManager:**   * Record Manager gets started by this function and the set up values are initialized. * Value 1 is passed as argument in case of primary key constraint. |
|  |
| **shutdownRecordManager:** |
|  |
|  |
| * Resources assigned to the record manager are freed by this function.   **createTable:** |
| * Table with a given name is created by this function. * The function first of all checks whether the table file exists or not. If yes, it throws an error. * Function writes the tableInfo on the page 0 and schema on page 1, if a new file is created. Also, the recoBeginPage is set to page 2 |
| **openTable:** |
|  |
| * File with given table name is opened by this function. * The function first of all checks whether the file exists or not. If not, it throws an error. * Buffer manger with given file name is initialized. * Page 0 and Page 1 is read; tableInfo and schema is loaded into memory.   **closeTable:** |
| * File and Buffer manager of a given file is closed by this function. * The function first of all checks whether the file exists or not. If not, it throws an error. * All the resources assigned with table are freed. |
| **deleteTable:** |
| * Table file is deleted by this function. * The function first of all checks whether the table exists or not. If not, it throws an error. * File and the buffer manager associated with that file is deleted. |
| **getTupleSize :** |
|  |
|  |

* As a part of tableInfo, tupleSize is stored in the memory.
* When written to file, it is stored on page 0 and loaded to memory.
* Each and every time a successful insert and delete is called, the value is updated.

|  |
| --- |
| **Record Functions** |
|  |
|  |
| **insertRecord:** |
|  |
| * This function first of all checks whether there are any RID’s in the tstone list or not. If yes, one of the RID is used and the attributes of the record are updated accordingly. * A new slot’s value must be computed since empty slots are unavailable. A new page is created if the new slot’s location is equal to the maximum number of slots for a page. Otherwise, current page is used.Record’s value for page and slot are updated accordingly. * To get the record in a proper format, serializeRecord is used. * To update the buffer pool, pinPage, markDirty, unpinPage and forcePage functions are used. The number of tuples are increased and the table is written to file.   **deleteRecord:** |
|  |
| * This function first of all checks whether the tstone list is empty or not. If yes, a new tableNode is created, its contents with the values from RID are updated and adds it to the tstone list. * tstone\_iter is used to go to the end of the list and a new tableNode with RID contents is added there, if the list is not empty. * Number of tuples are reduced and the table is written to the file.   **updateRecord:** |
|  |
| * To get the record in a proper format, serializeRecord is used. * To update the buffer pool, pinPage, markDirty, unpinPage and forcePage functions are used and the table is written to file.   **getRecord:** |
|  |

* Given RID value is used by this function to return a record to the user.
* This function first of all checks that the RID is not in the tstone list. Other check is used to see if the tuple number is greater than the number of tuples in the table (RC\_RM\_NO\_MORE\_TUPLES is returned as it is an error), if it is a valid record.
* To update the buffer pool, pinPage and unpinPage is used.
* To obtain a valid record from the record string which was retrieved, deserializeRecord is used. record -> data is updated accordingly.

**Scan Functions**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | | **startScan:** | | * RM\_ScanHandle data structure is initialized by this function which is passed as an argument. * A node storing the information about the record to be searched and the condition to be evaluated is also initialized by this function. * The node initialized in the previous step i.e step 2 is assigned to scan->mgmtData | | **next:** | | * A record is fetched as per the page and slot id by this function. * For a deleted record, it checks the tombstone id. * It checks for the slot number of the record to see if it is the last record, if the bit is set and the record is a deleted one. * The slot id is set to be 0 to start next scan from the beginning of the next page in case of the last record. * The slot number is increased by one to proceed to the next record in case where it is not a last record. * Next function is called after the updated record id parameters are assigned to the scan mgmtData. * The given condition is evaluated to check if the record is the one required by the client after verifying the tombstone parameters of the record. * Next function is called again with the updated record id parameters, if the record fetched is not the required one. * Once the scan is complete, it returns RC\_RM\_NO\_MORE\_TUPLES. * If no error occurs, it returns RC\_OK. * Function completesand the calls to this function returns the next tuple that satisfies the scan condition. | |  | |  | | **closeScan:** | | In this function, the scan handler is set to be free indicating the record manager that all associated resources are cleaned up. |   **Schema Functions**  **getRecordSize:**   |  | | --- | | * The size of the record is returned by the function. * The size of the record is counted by this function based on the schema. For the calculation, datatype of each attribute is considered.   **createSchema:**   * The schema object is created by this function and the memory is assigned. * Number of attributes, their datatypes and size is stored. | | **freeSchema:**   * All the memory assigned to schema object is freed by this function.   1) DataType  2) AttributeNames  3) AttributeSize  **Attribute Functions**   |  | | --- | | **createRecord:**   * Memory allocation is done for record and record data for creating a new record in this function. Memory allocation occurs as per the schema.   **freeRecord:**   * The memory space allocated to record and its data is freed by this function.   **getAttr:**   * The space is allocated to the value data structure by this function where the attribute values are to be fetched. * To get the offset value of different attributes as per the attribute numbers, attrOffset function is called. * Attribute data is assigned to the value pointer as per different data types. Hence, fetching the attribute values of a record.   **setAttr:**   * attroffset function is called by this function to get the offset value of different attributes as per the attribute numbers. * Attribute values are set with the values provided by the client as per the attributes datatype. Hence, setting the attribute values of a record. | |  | |  | |  |  |  | | --- | | **Helper Functions:**  **storeTableInfoIntoFile :** | | * tableInfo is written to the file by this function. * It is written on page 0 and keyInfo is written on page 2. | |  | |  | | **deserializeRecord :** | | * Record is read by the function from the table file, then it gets parsed, and a record data object is returned.   **deserializeSchema :**   * Schema is read by the function from the table file, then it gets parsed, and a schema object is returned. | |  | |  | |  | |  | | **getSlotSize :**   * The slot size required to write one record on the file is calculated by this function based on the serializeRecord function. | |  | | **convertTableInfoToString and convertStringToTableInfo :**   * The tableInfo is converted to a string by this function to write on the file. Also, the read data is converted from the file to tableInfo object by this function. | |  | | **keyInfoToStr and strToKeyInfo :** | | * The keyAttributeInfo is converted to a string by this function to write on the file. Also, the read data is converted from the file to keyList by this function. | |  | | | **Additional Features:**  **myscans:**   * It basically scans the records with the input matrix records and if it finds the same record, then it will print “found match record”. * It uses MAKE\_CONS, MAKE\_ATTRREF, MAKE\_BINOP\_EXPR for scanning the records.   **Tombstones:**   * Tombstones are stored in a linked list tableInfo->tstone\_head. The list contains tableNodes and each tableNode contains RID and pointer to the next tableNode. In record functions, this list is used. * RID has another attribute tstone (boolean). It is true if the RID is a tombstone. It helps the scan functions to check which values need to be skipped while traversing records. RID struct is altered in tables.h.   **Primary Key Constraint:**   * Support for primary key constraint is implemented, which applies to two situations:   1) A record with duplicate key attribute value is inserted to a table.  2) A record is updated to change the key attribute to one which already exists.  Both these violations are captured in our implementation   * Argument mgmtData = 1 is passed to initRecordManager to use this feature.   **Additional Error Checks:**  Below are the error cases that are checked and tested :   * To create a table that already exists. * To open or close a table that does not exist. * To close a table that is not open. * To delete or update a record that does not exist.   **Test Cases:**  We have implemented below listed test cases:  **test\_assign3\_1.c**  Many memory leaks were shown in Valgrind, some because of our script and other because of test cases. We solved many issues including ones with the test cases. In test\_assign3\_1.c , freeRecord(r) and free(rids) were missing in some cases. Some memory leaks still exist. We tried to solve but once freed, the script did not work outside valgrind.  **test\_expr.c**  Because of the values that were not being freed (stringToVal, serializeValue, etc), there were many memory leaks in this file. Hence, we added extra variables to solve this issue and many memory leaks were resolved.  **Additional Test Cases:**  Test cases for primary key constraint and tombstone has been implemented in the test\_assign3\_2.c.The instructions to run these test cases are provided below.  **Running instruction:**   * We added 3 test case files. You can simply follow this command to run it.  1. Test\_assign3\_1   You can run this file by follow below instruction.   * make clean * Make assign3\_1 * ./assign3\_1  1. Test\_expr   You can run this file by follow below instruction.   * make clean * Make expr * ./expr  1. Test\_assign3\_2   You can run this file by follow below instruction.   * make clean * Make assign3\_2 * ./assign3\_2 | |  | |  | |  | |  | |  |  |  | | --- | |  | |  | |
|  |
|  |
|  |