Report for Project 1 – Implementing a Record Manager

Assumptions (besides those given on the project page):

1. We do not actually support NULL value. If a string field is not set, its default value is an empty string, i.e., "".
2. We do not support buffer mechanism for DML, but directory is always cached and maintained in memory till all the data are updated to its page. In addition, a predefined number of page load (it is 10 in current implementation) is set to buffer a certain number of pages when re-organizing table.
3. The function *deleteTuples* marks all records as deleted, but all slots are preserved.
4. Users cannot add duplicated attribute (i.e., with same name) to a table; otherwise *addAttribute* function will return error code.
5. The **Attribute** passed to *readAttribute* function is always valid.

General Ideas of Design:

1. **Catalog Cache:**
   Catalog information is cached in a static vector of **AttributeCache**, which is a **struct** consisting of *tableName* and a vector of corresponding **Attribute**s *attrs*. It is implemented with lazy-load mode. Each time when **Attribute**s for specific table are needed, the code will try to find the entry with given *tableName* in the cache, and
   a. If found, return the corresponding *attrs*;
   b. If not found, read **Attribute**s from catalog table and store the **Attribute**s in cache with corresponding *tableName*.
   - **Update:** when one **Attribute** is added or removed from catalog table, the cache will be updated correspondingly (not fully supported—we have not implemented *dropAttribute* function).
   - **Delete:** when one table is being deleted, its corresponding entry in the cache will be also removed.

2. **Design of Scan:**
   The *scan* function in **RM** will initialize the **RM_ScanIterator** passed in, setting its reading status to true by invoking *startReading* function. **RM_ScanIterator** tracks page number and slot number, which indicate current reading slot. When **RM_ScanIterator** is initialized, scan criteria are also passed to **RM_ScanIterator**, and page number is set to 0 and slot number is set to -1. When *getNextTuple* function is invoked, it looks up database to locate the next qualified tuple starting from the very next slot following current page number and slot number.

Record Format: Starts with the count of fields in current record it stores, followed by data just as the input of insert function. When reading data from database, the count of fields will be stripped before the data are returned. To be consistent with the data format in record content, we use 4 bytes to store count of fields.

Count of fields	Record content as input for insert function, etc.
4 bytes	Variable length

Page Format: Slotted page format with 20% buffer space. The buffer space can be used during update, but is invisible to insertion, i.e., insertion cannot take this space.

1. Directory: it stores information in reverse order, appended at the end of each page with format as below (slot directory count starts from 0):

<---- Node (N - 1) ----->			<---- Nodes --->	<------- Node 0 -------->			<---- Control ----->
Slot flag	Slot offset	Slot length	…………	Slot flag	Slot offset	Slot length	Offset of the last slot directory info
2 bytes	2 bytes	2 bytes	6 * (N - 2) bytes	2 bytes	2 bytes	2 bytes	2 bytes

   a) Control info: stores the offset of the last slot in directory, relative to the beginning of the page.
   b) Slot flag: 0 - free space, 1 - normal data, 2 - tombstone, 3 - abandoned after re-organize page.
   c) Slot offset: stores the offset where the corresponding slot starts on this page, relative to the beginning of the page.
   d) Slot length: stores the length of the corresponding slot, including count of fields and record content.

   Since PF_PAGE_SIZE is 4096, which can be stored with **unsigned short** type, we use 2 bytes to store each piece of data info mentioned above. In the case shown above, there are N slot nodes. The offset stored in *Control* is [PF_PAGE_SIZE - 2 - 6 * N], and the length of directory is (2 + 6 * N). We define several macros for them for convenience and consistency.

2. Data: start from the very beginning of each page with the structure introduced above in Record Format.

3. Handling of overflow:
   a. Look up space overflowed data can fit in starting from the beginning of the table file, and
      i. If a slot for prior data is empty and enough to the data, put the data in and get corresponding RID;
      ii. If no such slot can be found, add a new page to put the data and return corresponding RID.
   b. Update prior slot, which stores the original data, to store the new RID, with 4 bytes storing page number and 4 bytes storing slot number. Since every slot stores 4-byte-length count of fields and at least 4-byte-length record, there must be enough space to store the RID of overflow slot.

Classes and Functions:

A. Record Manager (RM)

1. Design:
    a. Catalog File (named as "catalog.dat"):
    It is created while RM is initialized, if it does not exist; otherwise, RM uses existing catalog file. The structure is: TABLE_NAME (TypeVarChar, 20), COLUMN_NAME (TypeVarChar, 20), TYPE (TypeInt, 4), LENGTH (TypeInt, 4), POSITION (TypeInt, 4). The POSITION field starts from 1.
    b. Public methods mainly initialize a PF_FileHanle for given table and pass it to corresponding private methods, which perform complex logic by leveraging several helper methods, such as *prepareAttributes* and *locateData*, as explained below. In this way, for example, we re-use *insert* method not only in *insertTuple* function, but *updateTuple* and *insertTableAttributes* functions also utilize it. If any errors occur during the calling process, error codes are returned and corresponding error messages are shown.
    c. A **struct** named **Slot** is composed to store information of each slot, namely slot flag, slot offset and slot length. It also represents a node in directory. When the directory is buffered in memory as a **vector**, an extra slot is appended. It stores the offset of free space left at the end of all data and keeps tracking the size of this free space. This slot is calculated while reading directory information to memory and dropped when the directory information is written to page.
2. Public Member Functions:
    a. `RM::RM()`
    Invokes private method *initCatalog* to initialize catalog file, namely "catalog.dat". If the catalog file exists, do nothing.
    b. `RC createTable(const string tableName, const vector<Attribute> &attrs)`
    Creates a table with given name and attributes. A data file named "<tableName>.dat" is created, and attributes are inserted into catalog table. If a table with the given name exists, return error code.
    c. `RC deleteTable(const string tableName)`
    Removes the table with given name, and deletes corresponding attributes from catalog table. The data file is deleted from disk. Corresponding entry in catalog cache is also removed if exists. If failed to delete the data file, return error code.
    d. `RC getAttributes(const string tableName, vector<Attribute> &attrs)`
    Tries to find attributes in the catalog cache. If found, return the vector of attributes; otherwise, reads catalog file to load attributes of given table. Attributes are arranged according to its position fields.
    e. `RC insertTuple(const string tableName, const void *data, RID &rid)`
    Calculates input record size with attributes and invokes *insert* method as explained below. Directory is also updated accordingly by *updateDirectory* method.
    f. `RC deleteTuples(const string tableName)`
    Changes the slot flag of all the tuples to 0.
    g. `RC deleteTuple(const string tableName, const RID &rid)`

Locates the ultimate slot storing actual data by invoking *locateData* method in case the given RID points to a tombstone, and then marks the slot flag as 0. Therefore, it is still traceable from the original RID afterward.

h. `RC updateTuple(const string tableName, const void *data, const RID &rid)`

Locates the ultimate slot storing actual data by invoking *locateData* method in case the given RID points to a tombstone, and then calls check if the given data fit in the space original data are taking:

i. If the length of given data is the same as the one of original data, just put the data in the slot.
ii. If the length of given data is less than the one of original data, put the date in the slot starting from its offset; a new slot is added to the directory and points to the left free space.
iii. If the length of given data is greater than the one of original data, check if there is still enough free space left at the end of current page (<u>20% initial free space is available on this occasion</u>):
    1. If yes, moves all the following data afterward to offer new data enough space; all the offset values of following data are updated accordingly.
    2. If no, invokes *insert* method to put the new data in a new slot, and update the original slot as a tombstone (slot flag is set as 2) pointing to the slot new data reside (see item 3 in Page Format for more details).

If a tombstone is created, the slot length shrinks to 8 bytes as mentioned above. Under all circumstances, *updateDirectory* method is invoked if necessary.

i. `RC readTuple(const string tableName, const RID &rid, void *data)`

Invokes *locateData* method to find ultimate slot storing the actual data, and returns the data after stripping the count of fields information.

j. `RC readAttribute(const string tableName, const RID &rid, const string attributeName, void *data)`

Invokes *read* method, which is also called by *readTuple* function, to load the tuple at given RID, and leverages attribute list to find the attribute value specified by given attribute name. Returned data format is the same as that of input data for *insertTuple* function, i.e., a 4-byte length field followed by the actual data if attribute type is string (*TypeVarChar*).

k. `RC reorganizePage(const string tableName, const unsigned pageNumber)`

Since the order of slots on one page is not consistent with that of the nodes in directory storing its corresponding information, it copies valid data (slot flag is 1 and 2) to another page space in memory according to the directory, marking slots with free space (slot flag is 0) as abandoned (slot flag is 3).

l. `RC scan(const string tableName, const string conditionAttribute, const CompOp compOp, const void *value, const vector<string> &attributeNames, RM_ScanIterator &rm_ScanIterator)`

See more details in the "Design of Scan" section above.

3. Primary Private and Helper Methods:

a. `RC initCatalog()`

Initializes catalog and writes to file "catalog.dat" if it does not exist. For now it consists of only one table, namely "catalog", with the structure introduced in Design section above. The table structure information is populated in this table upon it is created with the same record format as that of other tables.

b. `RC insert(PF_FileHandle &handle, const void *data, const unsigned short size, RID &rid)`
   Starts searching available slot in which given data can fit and update RID to point to the slot. If no enough free space can be found on current existing pages, one new page is created to put the data and corresponding directory is updated.
   i. Parameter *handle*: is used to operate current table file.
   ii. Parameter *data*: is the data used to insert.
   iii. Parameter *size*: is the size of the data.
   iv. Parameter *rid*: is intended to return location information of the data after insertion.

c. `RC locateData(PF_FileHandle &handle, RID &rid, void *page, Slot *directory, unsigned short &count)`
   Follows the path indicated in tombstone to find ultimate slot storing the actual data, and RID is updated accordingly to point at the ultimate slot. If a slot flag is found as 3, it indicates this slot is abandoned after *reorganizePage* and the given RID is invalid; an error code is returned. If an empty slot is found, RID is also updated accordingly, since this method may be utilized to check if the given RID is free space.
   i. Parameter *handle*: is used to operate current table file.
   ii. Parameter *rid*: passed in the original position of the data, and stores the position of the ultimate slot storing the actual data.
   iii. Parameter *page*: is the data of current page *rid* points to.
   iv. Parameter *directory*: is the directory of current page.
   v. Parameter *count*: is the count of slots for which the directory stores information.

d. `RC updateDirectory(Slot *directory, unsigned short &count, const unsigned short slotNum, const unsigned short size)`
   Similar to *update* method, instead of operating data on a page, it manipulates information stored in directory. One key point is to manage the node pointing to the free space at the end of all the data. When new data are added, its size shrinks according to the size of data and minus the length of a node (6 bytes) when a slot is added.
   i. Parameter *directory*: is used to operate current table file.
   ii. Parameter *count*: is the count of slots for which the directory stores information. It can be updated if new slot is created, e.g., a new slot points to the extra free space.
   iii. Parameter *slotNum*: is the position to be updated in given directory.
   iv. Parameter *size*: is the length of new data intended to fit in the position of *slotNum*.

B. Scan Iterator (RM_ScanIterator)

This class is mainly used to load records satisfying given criteria passed in through *RM::scan* function, where initial status of the class is set: 1) criteria, including tableName, contitionAttribute, compOp, value, and attributeNames, are all set to private fields of this class; 2) reading status, which is stored via a bool private field *_reading*, is set to true—this prevents criteria change until the reading status is set to false by *close* function. Private fields are:

int _pageNum	The page number of current page
int _slotNum	The slot number of current record
bool _reading	Indicator of reading status
string _tableName	The name of table which is being scanned
string _conditionAttribute	The attribute as a filter
CompOp _compOp	The comparing operator
void *_value	The comparing value
vector<string> _attributeNames	To be projected attributes

Public Member Functions:

1. `RC getNextTuple(RID &rid, void *data)`
   Similar to the ++ operation of the usual smart pointer, which could get the next record of the table in the ascending ordering of RID. However, this function only gets the next valid record according to the compOp operator and value. When it finds the next valid record, it will return the RID of that record and the selected attributes value according to the projected attributes list. When this function reaches the end of the table, it will return -1.
2. `RC close()`
   Frees the memory in the private fields and sets the reading status to false to make the scan iterator invalid. It can't be used until Record Manager sets to its reading status false.
3. `RC startReading()`
   Sets reading status to true. Therefore, criteria cannot be changed while projecting records according to given criteria. The status is set to false when invoking *close* function.

C. Extra Points

We have implemented two of the three functions in the part of extra credit – *addAttribute* and *reorganizeTable*.

1. `RC RM::addAttribute(const string tableName, const Attribute &attr)`
   Adds a new attribute for some table. If the name of the new attribute already exists in the catalog table of our system, it returns a nonzero number. Otherwise, it inserts a new tuple in the catalog table.
2. `RC RM::reorganizeTable(const string tableName)`
   Reorganizes a table in a more complete way than the function reorganizePage. After reorganization, all records in the table will be moved to the head of the table and placed in a compact way—all slots are adjacent to each other without any free space. There is one only free slot in the end of each page. Moreover, all tombstones will be deleted which means the flag of each data slot will be 1.
   NOTE: After reorganization of the table, the RID of each slot could be changed.

More details:

When loading attributes from catalog, the Attribute vector will be populated according the position field of each Attribute stored in catalog.