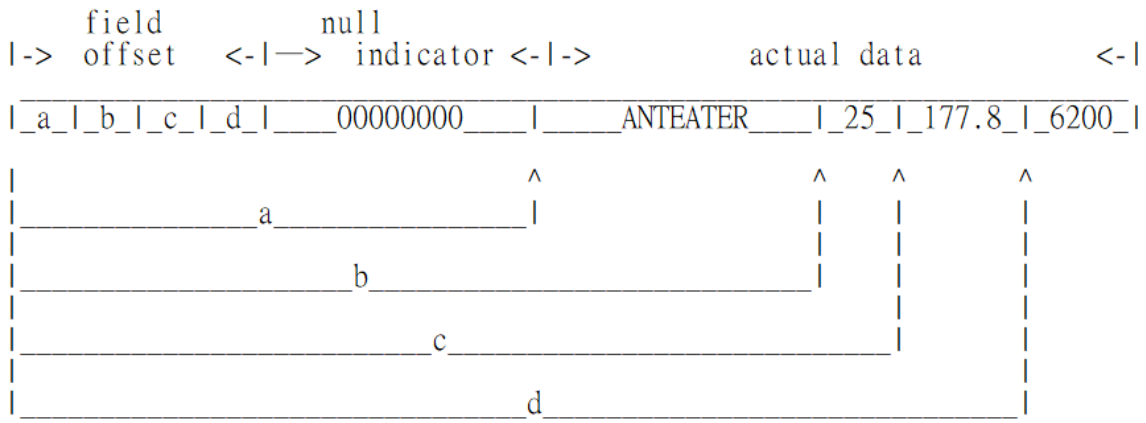
Project2 Report

1. Meta-data
- Show your meta-data design (Tables and Columns table) and information about each column.
Tables table contains four columns below:
- table-id: start from 1 and is the primary key for tables.
- table-name and file-name: by default, they are the same.
- user-flag: there are two values for this column, namely System and User.
  If the table is Tables or Columns, the column is System. Otherwise, it is User.
  A user cannot delete tables with user-flag of System.

Columns table contains five columns below:
- table-id: corresponds to the column "table-id" in Tables table.
- column-name: the name of the field.
- column-type: the data type of the field. There are three data types: TypeVarChar, TypeInt, TypeReal.
- column-length: the length of the field.
- column-position: the order of the field.

Before any table is created, Tables table and Columns table should be established first. When a new table is created, its table-wise information is inserted into Tables table with a unique table-id, and its field information is also inserted into Columns table, one entry for each field.

2. Internal Record Format
- Show your record format design and describe how your design satisfies O(1) field access. If not, just mention that your team hasn't implemented this feature.
A whole record consists of three elements as the following figure shows:
- Field Directory: offset of each field away from the start of the whole record
- null indicator: If the corresponding bit to each field is set to 1, then the actual data does not contain any value for this field.
- actual data



The length of FIELD OFFSET = recordDescriptor.size() * sizeof(short)

FIELD OFFSET stores the offset of every field. Each of the offset is a short integer which occupies sizeof(short), that is 2KBs. By reading the value in field offset, we can get information (offset, length) of a field and directly access the field we want with O(1).

e.g. get the 2nd field of data
offset = b
length = c - b

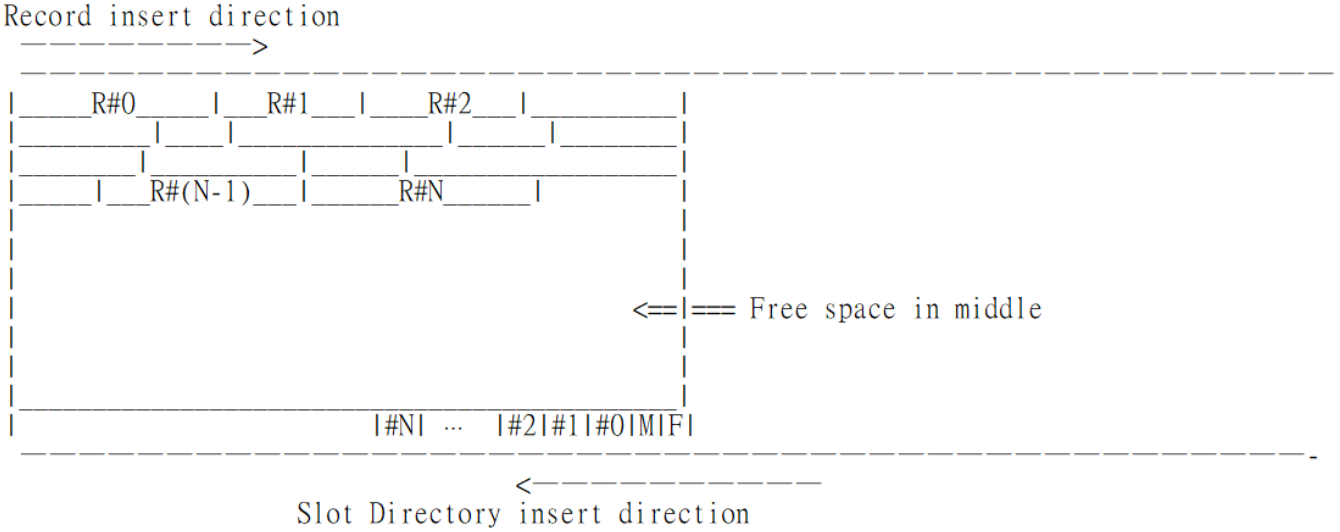
We can read the value by starting from offset(b) with length(c-b).

Each record corresponds to an RID, which consists of Page Number and Slot Number. We can use the RID to locate the record by first reading the page indicated by Page Number. And then look into the Slot Directory in this page by Slot Number to get the record's Offset and Length. And then use the Offset and Length information to locate and read the record in this page.

- Describe how you store a VarChar field.
VarChar field has two components: string length and the string. First allocate memory of sizeof(int) to store string length. And then allocate memory of sizeof(char)*string length to store the string.

- Describe how you deal with an update and delete.
Please refer to "4. Page Format - Describe how you deal with an update and delete."

3. Page Format
- Show your page format design.
It looks like below:



R#n: the record with Slot Number #n
F: KBs that are used in this page (occupies 2KBs)
M: number of slots in this page (occupies 2KBs)
#n: information of the record with Slot Number #n, including Offset (2KBs) and Length (2KBs)

Note that F starts from 4(KBs) since the initial state (M,F) = (0,4) needs to be written in this page even without any insertion of records. M and F both occupy 2KBs, hence the initial F is 4.

Slot Directory contains each record's Offset and Length, both of which are short type. RID's Slot Number starts from 0, and its Offset and Length information is inserted into Slot Directory backwards.

- Describe how you deal with an update and delete.

Update

There are four cases for update:

Case 1:

If the new record is larger than the old record and unfortunately the page storing the record has no sufficient space to contain the new record, first we insert the new record to another page with enough free space. Then we delete the old record, move subsequent records rightwards or leftwards based on the size difference between the old record and the new RID (size: 2*sizeof(short)), and insert the new RID, which points to the new location of the record, to the original location. Then we put a TOMBSTONE mark (-1) into the old RID's Length in the Slot Directory to indicate that the record has migrated to another page. Slot Directory (Offset) is updated for the subsequent records. The page storing the old record maintains its M (slot number), and F (used bytes) is updated.

When a user wants to read the old RID, our system reads the original location first. After finding out the TOMBSTONE mark, it knows that the original location stores the new RID and uses the new RID to read the record.

Case 2:

If the new record is larger than the old record and the page has enough space to contain the extra bytes, we delete the old record, move subsequent records rightwards by the size difference, and insert the new record to the original location. Slot Directory (Offset) is updated for the subsequent records. The page storing the old record maintains its M (slot number) and increases its F (used bytes).

Case 3:

If the new record is as large as the old record, we delete the old record and insert the new record to the original location. Slot Directory remains the same. The page storing the old record maintains both its M (slot number) and F (used bytes).

Case 4:

If the new record is smaller than the old record, we delete the old record, move subsequent records leftwards by the size difference, and insert the new record to the original location. Slot Directory (Offset) is updated for the subsequent records. The page storing the old record maintains its M (slot number) and decreases its F (used bytes).

The four cases are summarized in the following table:

Case	1	2	3	4
Situation	new record is larger than the old record and the page storing the record has no sufficient space	new record is larger than the old record	new record is as large as the old record	new record is smaller than the old record
Action	put a TOMBSTONE mark (-1) into the old RID's Length in the Slot Directory to indicate that the record has migrated to another page	delete the old record, move subsequent records rightwards by the size difference, and insert the new record to the original location	delete the old record and insert the new record to the original location	delete the old record, move subsequent records leftwards by the size difference, and insert the new record to the original location
Slot Directory	updated	updated	unchanged	updated
M (slot number)	unchanged	unchanged	unchanged	unchanged
F (used bytes)	updated	increased	unchanged	decreased

Delete

Similar to Update's Case 4, we delete the record and move subsequent records leftwards to fill the hole. Then we put a DELETION mark (-2) into the RID's Length in the Slot Directory to indicate that the record has been deleted. Slot Directory (Offset) is updated for the subsequent records. The page storing the record maintains its M (slot number) and decreases its F (used bytes).

When we insert a new record into a page, first we check if there is any DELETION mark in this page. If yes, then we replace that Slot in the Slot Directory with the new record's Offset and Length and let the new record own the previous deleted record's RID. In this way, we can recycle RID and use the page space more efficiently.

4. File Format

- Show your file format design

A table one-on-one corresponds to a file which consists of pages of 4096 bytes. The first page is a hidden page, which stores readPageCount, writePageCount, and appendPageCount. Real pages storing records start from Page Number 0 (actually the second page of the file).

5. Implementation Detail

- Other implementation details goes here.

RecordBasedFileManager (RBFM)

RBFM_ScanIterator

RBFM_ScanIterator has class members including fileHandle, recordDescriptor, conditionAttribute, compOp, value, attributeNames, rid. All these members will be set in RecordBasedFileManager::scan().

RecordBasedFileManager::scan()

This method sets class members for RBFM_ScanIterator. Note that rid.pageNum and rid.slotNum are set to be 0 initially.

RBFM_ScanIterator::getNextRecord()

This method goes through a file and filters out data matching the conditionAttribute. The procedure of scan can be divided as:

a) read a record with rid

b) read the field based on conditionAttribute, and compare the field with input value.
If the record does not match the condition, increment rid, then go to step a)
If the record matches the condition, go to step c).

c) construct the data to be returned based on attributeNames

d) rid of the matched record will also be returned to RID &rid. When getNextRecord() is called again, it resumes scan from the previous position.

RelationManager (RM)

RM_ScanIterator::getNextTuple()

A wrapper of RBFM_ScanIterator::getNextRecord().

RelationManager::scan()

This method initializes fileHandle, which opens file with tableName, and the fileHandle will be passed into RBFM_ScanIterator. Attributes are retrieved based on tableName using getAttributes(). The remaining parameters conditionAttribute, compOp, value, and attributeNames will also be passed into RBFM_ScanIterator.