CSE 510 - Database Management System Implementation
Spring 2018
Phase II
Due Date: Midnight, March 25th

1 Goal

The version of the MiniBase I have distributed to you implements various modules of a relational database management system. Our goal this semester is to use these modules of MiniBase as building blocks for implementing a *column-oriented DBMS*.

2 Project Description

The following is the list of modifications that you need to perform for this phase of the project. Note that getting these working may involve other changes to various modules not described below.

- Create a new class called `ValueClass` with the following specifications; note that this is an abstract class to be extended as "integer" or "string" values as needed:

```
public abstract class ValueClass
extends java.lang.Object
- Constructor Summary:
ValueClass()
```

- Create a new tuple ID (`TID`) class with the following specifications (see the `RID` class):

```
class TID
---
global.TID
extends java.lang.Object
---
Field Summary
int     numRIDs;
int     position;
RID[]   recordIDs;
---
Constructor Summary
TID(int numRIDs)
```

```
          default constructor of class
TID(int numRIDs, int position)
          constructor of class
TID(int numRIDs, int position, RID[] recordIDs)
          constructor of class
---
Method Summary
 void  copyTid(TID tid)
          make a copy of the given tid
 boolean equals(TID tid)
          Compares two TID objects
 void  writeToByteArray(byte[] array, int offset)
          Write the tid into a byte array at offset
 void   setPosition(int position)
          set the position attribute with the given value
 void   setRID(int column, RID recordID)
          set the RID of the given column
```

- Create a new package called `columnar` with the following specifications (see the `heap` package for an analogy):
  - Create a new class called `Columnarfile` (see the `Heapfile` class for analogy:)

```
class Columnarfile
---
columnar.Columnarfile
---
Field Summary
static int  numColumns
AttrType[]      type
---
Constructor summary
Columnarfile(java.lang.String name, int numColumns, AttrType[] type)
          Initialize: if columnar  file does not exits, create one
          heapfile (``name.columnid'') per column; also create a
          ``name.hdr'' file that contains relevant metadata.
---
Method Summary
 void  deleteColumnarFile()
          Delete all relevant files from the database.
 TID  insertTuple(byte[] tuplePtr)
          Insert tuple into file, return its tid
 Tuple  getTuple(TID tid)
```

```
          Read the tuple with the given tid from the columnar file
 ValueClass  getValue(TID tid, column)
          Read the value with the given column and  tid from the
          columnar file
 int    getTupleCnt()
          Return the number of tuples in the columnar file.
 TupleScan openTupleScan()
          Initiate a sequential scan of tuples.
 Scan    openColumnScan(int columnNo)
          Initiate a sequential scan along a given column.
 boolean  updateTuple(TID tid, Tuple newtuple)
          Updates the specified record in the columnar file.
 boolean  updateColumnofTuple(TID tid, Tuple newtuple, int column)
          Updates the specified column of the specified record in the
          columnar file
 boolean  createBTreeIndex(int column)
          if it doesn't exist, create a BTree index for the given column
 boolean  createBitMapIndex(int columnNo, valueClass value)
          if it doesn't exist, create a bitmap index for the given column
          and value
 boolean  markTupleDeleted(TID tid)
          add the tuple to a heapfile tracking the deleted tuples from
          the columnar file
 boolean  purgeAllDeletedTuples()
          merge all deleted tuples from the file as well as all from all
          index files.
```

– Create a new class called `TupleScan`; this scans all columns simultaneously to return complete tuples (see the `Scan` class for analogy:)

```
class TupleScan
---
Constructor Summary
TupleScan(Columnarfile cf)
---
Method Summary
 void  closetuplescan()
        Closes the TupleScan object
 Tuple  getNext(TID tid)
        Retrieve the next tuple in a sequential scan
 boolean position(TID tid)
        Position all scan cursors to the records with the given rids
```

- Create a new package called `bitmap` with the following specifications (see the `btree` package for an analogy):
  - Create a class for bitmaps called `BM` with the following specifications (see `BT` for analogy):

```
Class BM
---
Constructor Summary
BM()
---
printBitMap(bitmap.BitMapHeaderPage header)
          For debug.
```

  - Create a class called `BitMapFile` with the following specifications (see `BTreeFile` for analogy):

```
Class BitMapFile
---
Constructor Summary
BitMapFile(java.lang.String filename)
          BitMapFile class; an index file with given filename
          should already exist, then this opens it.
BitMapFile(java.lang.String filename, Columnarfile columnfile,
           int ColumnNo, valueClass value)
          BitMapFile class; an index file with given filename
          should not already exist; this creates the BitMap file
          from scratch.


---
MethodSummary
 void  close()
         Close the BitMap file.
 void  destroyBitMapFile()
         Destroy the entire BitMap file.
 bitmap.BitMapHeaderPage getHeaderPage()
         Access method to data member.
 boolean Delete(int position)
         set the entry at the given position to 0.
 boolean Insert(int position)
         set the entry at the given position to 1.
```

  - Extend the class `HFPage` with the following method.

```
 void  setCurPage_forGivenPosition(int Position)
          sets the value of curPage to the page which contains
  the entry at the given position
```

- Create a class called `BMPage` with the following specifications (see `heap.HFPage` for analogy):

```
Class BMPage
---
Constructor Summary
BMPage()
          Default constructor
BMPage(Page page)
          Constructor of class BMPage open a BMPage and
          make this BMpage point to the given page
---
 int   available_space()
         returns the amount of available space on the page.
 void   dumpPage()
         Dump contents of a page
 boolean empty()
         Determining if the page is empty
 void   init(PageId pageNo, Page apage)
         Constructor of class BMPage initialize a new page
 void   openBMpage(Page apage)
          Constructor of class BMPage open a existed BMPage
 PageId  getCurPage()
 PageId  getNextPage()
 PageId  getPrevPage()
 void   setCurPage(PageId pageNo)
         sets value of curPage to pageNo
 void   setNextPage(PageId pageNo)
         sets value of nextPage to pageNo
 void   setPrevPage(PageId pageNo)
         sets value of prevPage to pageNo
 byte[]  getBMpageArray()
 void    writeBMPageArray(byte[])
```

- Create a new class called `ColumnDB` under `diskmgr` with the following specification (see `DB`):

```
Class ColumnDB
---
Constructor Summary
ColumnDB()
           default constructor.
---
Method Summary
```

```
same as the existing DB class (though implementations
may differ)
```

- Create a class called `iterator.ColumnarFileScan` by modifying the class `iterator.FileScan`.

```
Class ColumnarFileScan
---
Constructor Summary
ColumnarFileScan(java.lang.String file_name,
                 AttrType[] in1,
                 short[] s1_sizes,
                 short len_in1,
                 int n_out_flds,
                 FldSpec[] proj_list,
                 CondExpr[] outFilter)
```

  Methods are similar to `iterator.FileScan` though implementations may be different

- Extend the class `global.IndexType` with *BitMapIndex* type.

- Create a class called `index.ColumnIndexScan` by modifying the class `index.IndexScan`.

```
Class ColumnIndexScan
---
Constructor Summary
ColumnIndexScan(IndexType index,
          java.lang.String relName,
          java.lang.String indName,
          AttrType type,
          short str_sizes,
          CondExpr[] selects,
          boolean indexOnly)
```

  Methods are similar to `index.IndexScan` though implementations may be different

- Modify Minibase disk manager in such a way that it counts the number of reads and writes. One way to do this is as follows:

  - First add `pcounter.java`, where

```
package diskmgr;
public class PCounter {
  public static int rcounter;
  public static int wcounter;
```

```
  public static void initialize() {
      rcounter =0;
      wcounter =0;
}
  public static void readIncrement() {
      rcounter++;
    }
  public static void writeIncrement() {
      wcounter++;
    }
}
```

into your code.

– Then, modify the `read_page()` and `write_page()` methods of the `diskmgr` to increment the appropriate counter upon a disk read and write request.

- Implement a program `batchinsert`. Given the command line invocation

  `batchinsert DATAFILENAME COLUMNDBNAME COLUMNARFILENAME NUMCOLUMNS`

  where `DATAFILENAME`, `COLUMNDBNAME`, and `COLUMNARFILENAME` are strings while `NUMCOLUMNS` is integer The format of the data file will be as follows:

```
atr1name:atrtype atr2name:atrtype ....
value11 value12 ...
value21 value22 ...
.....
```

  If the database/columnarfile already exists in the database, the tuples will be inserted into the existing table.

  At the end of the batch insertion process, the program should also output the number of disk pages that were read and the number of disk pages that were written.

- Implement a program `index`. Given the command line invocation

  `index COLUMNDBNAME COLUMNARFILENAME COLUMNNAME INDEXTYPE`

  where `COLUMNNAME`, `COLUMNDBNAME`, `COLUMNARFILENAME`, and `INDEXTYPE` are all strings.`INDEXTYPE` is either `BTREE` or `BITMAP`.

  At the end of the process, the program should also output the number of disk pages that were read and the number of disk pages that were written.

- Implement a program `query`. Given the command line invocation

  `query COLUMNDBNAME COLUMNARFILENAME [TARGETCOLUMNNAMES] VALUECONSTRAINT NUMBUF ACCESSTYPE`

the program will access the database and printout the matching results. The value constraint is of the form "{COLUMNNAME OPERATOR VALUE}". "ACCESSTYPE" is "FILESCAN", "COLUMNSCAN", "BTREE", or "BITMAP". The query will run in a way that leverages the specified access type. Minibase will use at most NUMBUF buffer pages to run the query (see the Class BufMgr).

At the end of the query, the program should also output the number of disk pages that were read and the number of disk pages that were written (if any).

- Implement a program delete_query which works like query, but eliminates all matching tuples from the database. In addition to having all the inputs of query, the input to delete_query also specifies whether the deleted tuples will be purged from the database or not.

  At the end of the query, the program should also output the number of disk pages that were read and the number of disk pages that were written (if any).

IMPORTANT: If you need to process large amounts of data (for example to sort a file), do not use the memory. Do everything on the disk using the tools and methods provided by minibase.

3 Deliverables

You have to return the following before the deadline:

- Your source code properly **commented**, tared and ziped.
- The output of your program with the provided test data
- A report following the given report document structure. As part of your work, I expect that you will develop 5 different storage (clustering and indexing) schemes. The report should experimentally analyze the read and write performance of different clustering and indexing alternatives for batch insertions and for different query types.

  The report should also describe *who did what*. This will be taken very seriously! So, be honest. Be prepared to explain on demand (not only your part) but the entire set of modifications. See the report specifications.
- A confidential document (individually submitted by each group member) which rates group members' contributions. Please provide a brief explanation for each group member.