CSE 510 - Database Management System Implementation
Spring 2020
Phase II
Due Date: Midnight, March 18th

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 *Bigtable-like DBMS*.

Please consider the following papers as a starting point of your reading in the area:

> `Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data.7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), {USENIX} (2006), pp. 205-218.`

2 Project Description

The following is the list of tasks 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 package called `BigT`, with three classes
  - `BigT.bigt`, which creates and maintains all the relevant heapfiles (and index files of your choice to organize the data),
  - `BigT.Map` Minibase stores data in the form of tuples. Bigtable on the other hand stores data in the form of maps

    $$(row : string, column : string, time : int) \rightarrow string.$$

    Therefore, the first task to extend Minibase with a new map construct. The map construct will be similar to the tuple, but with a fixed structure; a tuple can have any arbitrary length (as long as it is bounded by max_size) and any arbitrary fields, but a map will have 4 fixed fields:
    * `RowLabel:attrString`
    * `ColumnLabel:attrString`
    * `TimeStamp:attrInteger`
    * `Value:attrString`

- `BigT.Stream` This supports the `getNext()` interface which will retrieve maps from the big table in some specified order.

These classes will provide the following constructors and methods:

- `BigT.bigt`:
  - `bigt(java.lang.String name, int type)`: Initialize the big table. `type` is an integer between 1 and 5 and the different types will correspond to different clustering and indexing strategies you will use for the bigtable.
  - `void deleteBigt()`: Delete the bigtable from the database.
  - `int getMapCnt()`: Return number of maps in the bigtable.
  - `int getRowCnt()`: Return number of distinct row labels in the bigtable.
  - `int getColumnCnt()`: Return number of distinct column labels in the bigtable.
  - `MID insertMap(byte[] mapPtr)` Insert map into the big table, return its Mid. The `insertMap()` method ensures that there are at most three maps with the same row and column labels, but different timestamps, in the bigtable. When a fourth is inserted, the one with the oldest label is dropped from the big table.
  - `Stream openStream(int orderType, java.lang.String rowFilter, java.lang.String columnFilter, java.lang.String valueFilter)`: Initialize a stream of maps where row label matching `rowFilter`, column label matching `columnFilter`, and value label matching `valueFilter`. If any of the filter are null strings, then that filter is not considered (e.g., if `rowFilter` is null, then all row labels are OK). If `orderType` is
    - 1, then results are first ordered in row label, then column label, then time stamp
    - 2, then results are first ordered in column label, then row label, then time stamp
    - 3, then results are first ordered in row label, then time stamp
    - 4, then results are first ordered in column label, then time stamp
    - 6, then results are ordered in time stamp
- `BigT.Map`: You will need to create a class `BigT.Map`, similar to `heap.Tuple` but having a fixed structure (and thus a fixed header) as described above. Thus, the constructor and get/set methods associated with the `BigT.Map` should be adapted as appropriate:
  - `Map()`: Class constructor create a new map with the appropriate size.
  - `Map(byte[] amap, int offset)`: Construct a map from a byte array.
  - `Map(Map fromMap)`: Construct a map from another map through copy.
  - `java.lang.String getRowLabel()`: Returns the row label.
  - `java.lang.String getColumnLabel()`: Returns the column label.
  - `int getTimeStamp()`: Returns the timestamp.
  - `java.lang.String getValue()`: Returns the value.
  - `Map setRowLabel(java.lang.String val)`: Set the row label.

* `Map setColumnLabel(java.lang.String val)`: Set the column label.
* `Map setTimeStamp(int val)`: Set the timestamp.
* `Map setValue(java.lang.String val)`: Set the value.
* `byte[] getMapByteArray()`: Copy the map to byte array out.
* `void print()`: Print out the map.
* `size()`: Get the length of the tuple
* `mapCopy(Map fromMap)`: Copy the given map
* `mapInit(byte[] amap, int offset)`: This is used when you don't want to use the constructor
* `mapSet(byte[] frommap, int offset)`: Set a map with the given byte array and offset.

After creating this map construct, remove the `heap.Tuple` completely from the minibase and modify all related code to use `BigT.Map` instead; that is, your version of minibase will not store tuples anymore.

– `BigT.Stream`: This class will be similar to `heap.Scan`, however, will provide different types of accesses to the bigtable:

* `Stream(bigt bigtable, int orderType, java.lang.String rowFilter, java.lang.String columnFilter, java.lang.String valueFilter)`: Initialize a stream of maps on `bigtable`.
* `void closestream()`: Closes the stream object.
* `Map getNext(MID mid)`: Retrieve the next map in the stream.

Note that the above methods use a map id class, `MID`, that needs to be declared in a `global.MID`, similar to `global.RID`.

- Minibase stores tuples in tables which are encoded in *Heapfiles* (`heap.Heapfile`). Each heapfile corresponds to a data table in a relational database. Tuples are inserted into and deleted from the heap file using *insertRecord()* and *deleteRecord()* methods. In the BigTable, each record will correspond to a map instead of a tuple. Modify the `heap` package, and the classes within, accordingly. For example, the HeapFile class is modified as follows:

  – `Heapfile(java.lang.String name)`: Initialize.
  – `void deleteFile()`: Delete the file from the database.
  – `boolean deleteMap(MID mid)`: Delete map from file with given mid.
  – `int getMapCnt()`: Return number of maps in the file.
  – `Map getMap(MID mid)`: Read the map from file.
  – `MID insertMap(byte[] mapPtr)` Insert map into file, return its Mid.
  – `Scan openScan()`: Initiate a sequential scan.
  – `boolean updateMap(MID mid, Map newmap)`: Updates the specified map in the heapfile.

  Note that the above methods replace `global.RID` with `global.MID`; therefore, `global.RID` is redundant and will be eliminated. Therefore, all classes that refer to `RID`, will need to be modified to use `MID` instead.

- Create the class `iterator.MapUtils` by modifying the class `iterator.TupleUtils`. For example,

- `static int CompareMapWithMap(Map m1, Map m2, int map_fld_no)`: This function compares a map with another map in respective field, and returns: 0 if the two are equal, 1 if `m1` is greater, -1 if `m2` is smaller
- `static boolean Equal(Map m1, Map m2)`: This function Compares two maps in all fields

Other methods of `iterator.MapUtils` and classes (e.g., global.TupleOrder or `iterator.Sort`) referring to tuples are also adapted accordingly.

- Under the `diskmgr` package, create a new class called `bigDB` by modifying `diskmgr.DB`. This class creates and maintains all the relevant files and btree based index files to organize the data. In addition to the existing methods of the `diskmgr.DB`, the `diskmgr.bigDB` also contains the following classes and methods:
  - `bigDB(int type)`: Constructor for the big table database. `type` is an integer denoting the different clustering and indexing strategies you will use for the graph database. Note that each big table database may contain
    - Type 1: No index
    - Type 2:
      - one btree to index row labels
    - Type 3:
      - one btree to index column labels
    - Type 4:
      - one btree to index column label and row label (combined key) and
      - one btree to index timestamps
    - Type 5:
      - one btree to index row label and value (combined key) and
      - one btree to index timestamps

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

  where `DATAFILENAME` and `BIGTABLENAME` are strings and `TYPE` is an integer (between 1 and 5), the program will store all the maps in a bigtable.

  The format of the data file will be as follows:

  ```
  rowlabel1 columnlabel1 timestamp1 value1
  rowlabel2 columnlabel2 timestamp2 value2
  .....
  ```

  Given these maps, the name of the bigtable that will be created in the database will be `BIGTABLENAME_TYPE`. If this bigtable already exists in the database, the tuples will be inserted into the existing bigtable.

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

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

  `query BIGTABLENAME TYPE ORDERTYPE ROWFILTER COLUMNFILTER VALUEFILTER NUMBUF`

  the program will access the database and printout the matching maps in the requested order.

  Each filter canbe

  - "*": meaning unspecified (need to be returned),
  - a single value, or
  - a range specified by two column seperated values in square brackets (e.g. "[56, 78]")

  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 written (separately).

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**, `tar`ed and `zip`ed.
- The output of your program with the provided test data and the driver.
- 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 out of 10 (10 best; 0 worst). Please provide a brief explanation for each group member.