CSE 510 - Database Management System Implementation
Spring 2017
Phase II
Due Date: Midnight, March 14th

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 *graph DBMS*.

2 Project Description

Minibase stores data in the form of tuples. The graph database on the other hand will store

- *nodes* of the form
  $$(node\_id, node\_label, node\_descriptor),$$
  where each node descriptor is a 5-dimensional vector, and
- *edges* of the form
  $$(edge\_id, source\_node, destination\_node, edge\_label, weight).$$

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.

2.1 Task 1: Create new Attribute Type

- Define a new data type `Descriptor` which consists of 5 integers, where each integer takes values from 0 to 10000 (i.e., 2-byte representation is sufficient).

```
filename: Descriptortype.java
 =========================================================
package global;

public class Descriptor {
  double value [];

  value = new int[5];

  value[0] = 0;
```

```
    value[1] = 0;
    value[2] = 0;
    value[3] = 0;
    value[4] = 0;

    void set(int value0, int value1, int value2, int value3, int value4) {
       value[0] = value0;
       value[1] = value1;
       value[2] = value2;
       value[3] = value3;
       value[4] = value4;
    }
    int get(int idx) {
       return value[idx];
    }
    double equal (Descriptor desc) {
        //return 1 if equal; 0 if not
    }
    double distance (Descriptor desc) {
        //return the Euclidean distance between the descriptors
    }
  }
```

- Modify attribute type definitions in `Class AttrType` to include a new attribute type called `attrDesc` in addition to the integer, real, and string attribute types already defined in MiniBase.
- Modify tuple field get and set methods to accomodate the new attribute type:

```
getDescFls(int fldNo)
  convert this field into Descriptor type
setDescFld(int fldNo, Descriptor val)
   set this field to Descriptor value
```

- Modify page get and set methods to accomodate the new attribute type:

```
getDescValue(int position, byte[] data)
  read from given byte array at the specified position convert it to Descriptor
setDescValue(Vector100Dtype value, int position, byte[] data)
  update a Descriptor in the given byte array at the specified position
```

- Modify operand definitions to include operands of type `attrDesc`.

- Modify tuple comparison methods, `CompareTupleWithTuple` and `CompareTupleWithValue`, such that they return the distance (which is of type double) between the two inputs, if the fields that are compared are of type `attrDesc`.

2.2 Task 2: Modify Condition Expression and Evaluation

- Modify condition expressions, `CondExpr`, to include an extra field `distance` to be used with `attrDesc` values. If the operands are of type `attrDesc`, then the value of *distance* will be set to a non-negative integer.
- Modify `Eval` (which evaluates condition expressions) to work with values of type `attrDesc`. For example,
  - the operator `aopGE` should return true, if the distance between the two operands is greater than or equal to the value of the distance parameter.
  - the operator `aopEQ` should return true, if the two operands are exactly the same distance from each other as the distance parameter.

  Operators `aopGT`, `aopLE`, `aopLT`, `aopNE` are to be similarly defined.

Make sure that the relevant iterators (such as `NestedLoopsJoins` and `SortMerge`), which take `CondExpr` type input parameters, work with the new definition of `CondExpr`.

2.3 Task 3: Create new Node File Type

- Create a new node ID (`NID`) class by extending the `RID` class.
- We extend Minibase with a new node construct. The node 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 node will have 2 fixed fields, named "Label" and "Descriptor":
  - `Label:attrString`
  - `Descriptor:attrDesc`

  Minibase stores tuples in tables which are organized 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 graph database, nodes will be stored in node heap files. To achieve this, we create a new `nodeheap` package by modifying the `heap` package, and the classes within, accordingly.

  - `HeapFile` class is modified into `NodeHeapFile` as follows:
    - `NodeHeapfile(java.lang.String name)`: Initialize.
    - `void deleteFile()`: Delete the file from the database.
    - `boolean deleteNode(NID nid)`: Delete node with given nid from the file.
    - `int getNodeCnt()`: Return the number of nodes in the file.
    - `Node getNode(NID nid)`: Read the node from file.

        * `NID insertNode(byte[] nodePtr)` Insert node into file, return its NID.
        * `NScan openScan()`: Initiate a sequential scan.
        * `boolean updateNode(NID nid, Node newNode)`: Updates the specified node in the node-heapfile.
    - `Tuple` is extended into `Node`.
        * `Node()`: Class constructor creates a new node with the appropriate size.
        * `Node(byte[] anode, int offset)`: Construct a node from a byte array.
        * `Node(Node fromNode)`: Construct a node from another node through copy.
        * `attrString getLabel()`: Returns the label.
        * `attrDesc getDesc()`: Returns the descriptor.
        * `Node setLabel(attrString Label)`: Set the label.
        * `Node setDesc(attrDesc Desc)`: Set the descriptor.
        * `byte[] getNodeByteArray()`: Copy the node to byte array out.
        * `void print()`: Print out the node.
        * `size()`: Get the length of the node
        * `nodeCopy(Node fromNode)`: Copy the given node
        * `nodeInit(byte[] anode, int offset)`: This is used when you don't want to use the constructor
        * `nodeSet(byte[] fromnode, int offset)`: Set a node with the given byte array and offset.
    - `HFPage` is modified into `NHFPage` appropriately
    - `Scan` is modified into `NScan` appropriately

2.4 Task 4: Create new Edge File Type

- Create a new edge ID (`EID`) class by extending the `RID` class:
- We extend Minibase with a new edge construct. The edge 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 an edge will have 4 fixed fields:
    - `Source:NID`
    - `Destination:NID`
    - `Label:attrString`
    - `Weight:attrInteger`

  Minibase stores tuples in tables which are organized 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 graph database, edges will be stored in edge heap files. To achieve this, we create a new `edgeheap` package by modifying the `heap` package, and the classes within, accordingly.

- `HeapFile` class is modified into `EdgeHeapFile` as follows:
  - `EdheHeapfile(java.lang.String name)`: Initialize.
  - `void deleteFile()`: Delete the file from the database.
  - `boolean deleteEdge(EID eid)`: Delete edge with given eid from the file.
  - `int getEdgeCnt()`: Return the number of edges in the file.
  - `Edge getEdge(EID eid)`: Read the edge from file.
  - `EID insertEdge(byte[] edgePtr)` Insert node into file, return its EID.
  - `EScan openScan()`: Initiate a sequential scan.
  - `boolean updateEdge(EID eid, Edge newEdge)`: Updates the specified edge in the EdgeHeapfile.
- `Tuple` is extended into `Edge`.
  - `Edge()`: Class constructor creates a new node with the appropriate size.
  - `Edge(byte[] aedge, int offset)`: Construct a node from a byte array.
  - `Edge(Edge fromEdge)`: Construct an edge from another edge through copy.
  - `attrString getLabel()`: Returns the label.
  - `attrInteger getWeight()`: Returns the weight.
  - `NID getSource()`: Returns the node ID of the source node.
  - `NID getDestination()`: Returns the node ID of the destination node.
  - `Edge setLabel(attrString Label)`: Set the label.
  - `Edge setWeight(attrInteger Weight)`: Set the weight.
  - `Edge setSource(NID sourceID)`: Set the source.
  - `Edge setDestination(NID destID )`: Set the destination.
  - `byte[] getNodeByteArray()`: Copy the edge to byte array out.
  - `void print()`: Print out the edge.
  - `size()`: Get the length of the edge
  - `edgeCopy(Edge fromEdge)`: Copy the given edge
  - `edgeInit(byte[] aedge, int offset)`: This is used when you don't want to use the constructor
  - `edgeSet(byte[] fromedge, int offset)`: Set an edge with the given byte array and offset.
- `HFPage` is modified into `EHFPage` appropriately
- `Scan` is modified into `EScan` appropriately

2.5 Task 5: Modify Tuple Comparisons

- Extend the `CompareTupleWithTuple`, `CompareTupleWithValue`, and `Equal` methods of the class `iterator.TupleUtils` to take two extra parameters
  - `double distance`

– `Descriptor target`

If the attribute being compared is of type `attrDesc` and the `target` is given as [-1,-1,-1,-1,-1] then

– `CompareTupleWithTuple` and `CompareTupleWithValue` returns: 0 if the distance between the descriptor is less than or equal to `distance`, 1 otherwise

– `Equal` returns 1 if their distance is less than or equal to `distance` and 0 otherwise

If the attribute being compared is of type `attrDesc` and the `target` is different from [-1,-1,-1,-1,-1]

– `CompareTupleWithTuple` and `CompareTupleWithValue` returns: 0 if the distances of the both descriptors from `target` are equal, 1 if the distance of the first descriptor is smaller, -1 if it is larger

– `Equal` returns 1 if the distances of the both descriptors from `target` are equal, 0 otherwise

2.6 Task 6: Modify the Iterators

Methods of `iterator.TupleUtils` and classes (e.g., `iterator.Sort`) referring to tuples are adapted accordingly by adding a distance and/or target parameter as appropriate.

- Modify the `Sort` iterator such that, if the sort attribute is the descriptor, it sorts all tuples taking into account the given distance and target parameters.

2.7 Task 7: New Z-tree Index File and Associated Access Methods

This task involves creation of a Z-tree index file and associated access methods to support insertion, deletion, region search, and nearest neighbor search operations. The data structures and relevant algorithms are described at

`http://http://www.cs.cmu.edu/˜christos/courses/721.S03/LECTURES-PDF/0230-z-ordering.PDF`

MiniBase uses the classes in the `btree` package to create btree data structures that index the records in the heapfiles of the database through the `RIDs` of the records in the data `HeapFiles`. In this task, you will use the `btree` package as a template to create a new un-clustered index to support range searches leveraging Z-curves.

- Create a new index type `ZIndex`

  ```
  IndexType.ZIndex
  ```

  that indexes data of type "`attrDesc`".

- Create a new package `ZIndex` that supports indexing 5D points based on their Z-values and the associated access methods. The new index type needs to support the following access methods:

  ```
  ZFileScan()
  ZFileRangeScan(KeyClass key, int distance)
  ```

where the KeyClass is extended to accomodate keys of type `Descriptor` (similar to existing `IntegerKey` and `StringKey`).

Note that to implement these operations, you may need to convert the spatial range scan operations to range queries on B-trees using z-values.

- Modify the `index` package to leverage the new `ZIndex` to create scans.

2.8 Task 8: Modify Database Definitions

- Under the `diskmgr` package, create a new class called `graphDB` by modifying `diskmgr.DB`. This class creates and maintains all the relevant files (node heap file, edge heap file, and btree based index files of your choice to organize the data). In addition to the existing methods of the `diskmgr.DB`, the `diskmgr.graphDB` also contains the following classes and methods:
  - `graphDB(int type)`: Constructor for the graph database. `type` is an integer denoting the different clustering and indexing strategies you will use for the graph database. Note that each graph database contains
    - one `NodeHeapFile` to
    - store the nodes, one `EdgeHeapfile` to store edges,
    - one btree to index node labels,
    - one ztree to index node descriptors,
    - one btree to index edge labels, and
    - one btree to index edge weights.
  - In addition to the default methods, the graphDB will also provide the following methods
    - `int getNodeCnt()`: Returns the number of nodes in the database.
    - `int getEdgeCnt()`: Returns the number of edges in the database.
    - `int getSourceCnt()`: Returns the number of distinct source nodes in the database.
    - `int getDestinationCnt()`: Returns the number of distinct destination nodes in the database.
    - `int getLabelCnt()`: Returns the number of distinct labels in the database.

2.9 Task 9: Create a Page Counter

- 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 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.

2.10 Task 10: Batch Node Insert

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

  ```
  batchnodeinsert NODEFILENAME GRAPHDBNAME
  ```

  where `NODEFILENAME` and `GRAPHDNAME` are strings.

  The format of the node data file will be as follows:

  ```
  nodelabel1 nodedesc11 nodedesc12 nodedesc13 nodedesc14 nodedesc15
  nodelabel2 nodedesc21 nodedesc22 nodedesc23 nodedesc24 nodedesc25
  .....
  ```

  Note that the node labels will be unique.

  Given these nodes, the name of the graph database that will be created in the database will be `GRAPHDBNAME`. If this graph database already exists in the database, the nodes will be inserted into the existing graph database.

  At the end of the batch insertion process, the program should also output the relevant database statistics (node and edge counts) and the number of disk pages that were read and written (separately) during the operation.

2.11 Task 11: Batch Edge Insert

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

  ```
  batchedgeinsert EDGEFILENAME GRAPHDBNAME
  ```

  where `EDGEFILENAME` and `GRAPHDNAME` are strings.

  The format of the node data file will be as follows:

```
sourcelabel1 destlabel1 edgelabel1 edgeweight1
sourcelabel2 destlabel2 edgelabel2 edgeweight2
.....
```

If this graph database already contains edges, the new edges will be inserted into the existing graph database.

At the end of the batch insertion process, the program should also output the relevant database statistics (node and edge counts) and the number of disk pages that were read and written (separately) during the operation.

2.12 Task 12: Batch Node Delete

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

  ```
  batchnodedelete NODEFILENAME GRAPHDBNAME
  ```

  where `NODEFILENAME` and `GRAPHDNAME` are strings.

  The format of the node data file will be as follows:

  ```
  nodelabel1
  nodelabel2
  .....
  ```

  Note that the node labels will be unique.

  Given these nodes, the nodes listed in the input file (and any edges connected to them) will be removed from the graph database.

  At the end of the batch deletion process, the program should also output the relevant database statistics (node and edge counts) and the number of disk pages that were read and written (separately) during the operation.

2.13 Task 13: Batch Edge Delete

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

  ```
  batchedgedelete NODEFILENAME GRAPHDBNAME
  ```

  where `NODEFILENAME` and `GRAPHDNAME` are strings.

  The format of the node data file will be as follows:

  ```
  sourcelabel1 destinationlabel1 nodelabel1
  sourcelabel2 destinationlabel2 nodelabel2
  .....
  ```

  The edges listed in the input file will be removed from the graph database.

  At the end of the batch deletion process, the program should also output the relevant database statistics (node and edge counts) and number of disk pages that were read and written (separately) during the operation.

2.14 Task 14: Simple Node Query

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

  ```
  nodequery GRAPHDBNAME NUMBUF QTYPE INDEX [QUERYOPTIONS]
  ```

  the program will access the database and printout the matching nodes in the requested order. More specifically,

  - if `QTYPE` = 0, then the query will print the node data in the order it occurs in the node heap.
  - if `QTYPE` = 1, then the query will print the node data in increasing alphanumerical order of labels.
  - if `QTYPE` = 2, then the query will print the node data in increasing order of distance from a given 5D target descriptor.
  - if `QTYPE` = 3, then the query will take a target descriptor and a distance and return the labels of nodes with the given distance from the target descriptor.
  - if `QTYPE` = 4, then the query will take a label and return all relevant information (including outgoing and incoming edges) about the node with the matching label (if any).
  - if `QTYPE` = 5, then the query will take a target descriptor and a distance and return all relevant information (including outgoing and incoming edges) about the nodes with the given distance from the target descriptor.

  If `INDEX` is 1 then, the query will be processed using an index, otherwise only the relevant heap files will be used.

  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).

2.15 Task 15: Simple Edge Query

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

  ```
  edgequery GRAPHDBNAME NUMBUF QTYPE INDEX [QUERYOPTIONS]
  ```

  the program will access the database and printout the matching edges in the requested order. More specifically,

  - if `QTYPE` = 0, then the query will print the edge data in the order it occurs in the node heap.
  - if `QTYPE` = 1, then the query will print the edge data in increasing alphanumerical order of source labels.
  - if `QTYPE` = 2, then the query will print the edge data in increasing alphanumerical order of destination labels.
  - if `QTYPE` = 3, then the query will print the edge data in increasing alphanumerical order of edge labels.
  - if `QTYPE` = 4, then the query will print the edge data in increasing order of weights.
  - if `QTYPE` = 5, then the query will take a lower and upper bound on edge weights, and will return the matching edge data.
  - if `QTYPE` = 6, then the query will return pairs of incident graph edges.

If `INDEX` is 1 then, the query will be processed using an index, otherwise only the relevant heap files will be used.

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).

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.
- A report following the given report document structure. The report should detail how the various tasks are implemented.

  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.