CSE 510 - Database Management System Implementation

Spring 2022

Phase II

Due Date: Midnight, March 20th

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 an *RDF DBMS*.

2 Project Description

Minibase stores data in the form of tuples. The RDF databases on the other hand store data in the form of triples

$$(subject, predicate, object).$$

In this project, we will extend the definition of the triple, with a concept of "confidence", stating how confident we are regarding the fact asserted by the given quadruple:

$$(subject, predicate, object, confidence).$$

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 quadruple ID (`QID`) class with the following specifications (see the `RID` class):

```
class QID
---
global.QID
extends java.lang.Object
---
Field Summary
PageID  pageNo;
int     slotNo;

---
Constructor Summary
QID()
          default constructor of class
```

```
QID(PageId pageno, int slotno)
          constructor of class


---
Method Summary
 void  copyQid(QID qid)
          make a copy of the given qid
 boolean equals(QID qid)
          Compares two QID objects
 void  writeToByteArray(byte[] array, int offset)
          Write the qid into a byte array at offset
```

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

```
class LID
---
global.LID
extends java.lang.Object
---
Field Summary
PageID  pageNo;
int     slotNo;

---
Constructor Summary
LID()
          default constructor of class
LID(PageId pageno, int slotno)
          constructor of class



---
Method Summary
 void  copyLid(LID lid)
          make a copy of the given lid
 boolean equals(LID lid)
          Compares two LID objects
 EID     returnEID()
          Returns the corresponding EID
 PID     returnPID()
          Returns the corresponding PID
 void  writeToByteArray(byte[] array, int offset)
          Write the lid into a byte array at offset
```

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

```
class EID
---
global.EID
extends java.lang.Object
---
Field Summary
PageID  pageNo;
int     slotNo;

---
Constructor Summary
EID()
          default constructor of class
EID(LID lid)
          constructor of class

---
Method Summary
 void  copyEid(EID eid)
          make a copy of the given eid
 boolean equals(EID eid)
          Compares two EID objects
 LID     returnLID()
          Returns the corresponding LID
 void  writeToByteArray(byte[] array, int offset)
          Write the eid into a byte array at offset
```

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

```
class PID
---
global.PID
extends java.lang.Object
---
Field Summary
PageID  pageNo;
```

```
int      slotNo;

---
Constructor Summary
PID()
          default constructor of class
PID(LID lid)
          constructor of class

---
Method Summary
 void  copyPid(PID pid)
          make a copy of the given pid
 boolean equals(PID pid)
          Compares two PID objects
 LID     returnLID()
          Returns the corresponding LID
 void  writeToByteArray(byte[] array, int offset)
          Write the pid into a byte array at offset
```

- 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 RDF database, each record will correspond to a quadruple instead of a tuple and the quadruples will be stored in quadruple heap files. In addition, we will also store entities and predicates in seperate label heap files.
  - Create a new `quadrupleheap` package by modifying the `heap` package, and the classes within, accordingly.
    - `HeapFile` class is modified into `QuadrupleHeapFile` as follows:
      - `QuadrupleHeapfile(java.lang.String name)`: Initialize.
      - `void deleteFile()`: Delete the file from the database.
      - `boolean deleteQuadruple(QID qid)`: Delete quadruple with given qid from the file.
      - `int getQuadrupleCnt()`: Return number of quadruples in the file.
      - `Quadruple getQuadruple(QID qid)`: Read the quadruple from file.
      - `QID insertQuadruple(byte[] quadruplePtr)` Insert quadruple into file, return its QID.
      - `TScan openScan()`: Initiate a sequential scan.
      - `boolean updateQuadruple(QID qid, Quadruple newQuadruple)`: Updates the specified quadruple in the quadrupleheapfile.
    - We extend Minibase with a new *extended* quadruple construct. The extended quadruple 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 extended quadruple (which we will refer to simply as a "quadruple") will have 4 fixed fields:

  · `Subject:  EID`
  · `Predicate:PID`
  · `Object:EID`
  · `Value:attrReal`

`Tuple` is modified into `Quadruple`.

  · `Quadruple()`: Class constructor creates a new quadruple with the appropriate size.
  · `Quadruple(byte[] aquadruple, int offset)`: Construct a quadruple from a byte array.
  · `Quadruple(Quadruple fromQuadruple)`: Construct a quadruple from another quadruple through copy.
  · `EID getSubjecqid()`: Returns the subject ID.
  · `PID getPredicateID()`: Returns the predicate ID.
  · `EID getObjecqid()`: Returns the object ID.
  · `double getConfidence()`: Returns the confidence.
  · `Quadruple setSubjecqid(EID subjecqid)`: Set the subject ID.
  · `Quadruple setPredicateID(PID predicateID)`: Set the predicate ID.
  · `Quadruple setobjecqid(EID objecqid)`: Set the object ID.
  · `Quadruple setConfidence(double confidence)`: Set the confidence
  · `byte[] getQuadrupleByteArray()`: Copy the quadruple to byte array out.
  · `void print()`: Print out the quadruple.
  · `size()`: Get the length of the quadruple
  · `quadrupleCopy(Quadruple fromQuadruple)`: Copy the given quadruple
  · `quadrupleInit(byte[] aquadruple, int offset)`: This is used when you don't want to use the constructor
  · `quadrupleSet(byte[] fromquadruple, int offset)`: Set a quadruple with the given byte array and offset.

 ∗ `HFPage` is modified into `THFPage` appropriately
 ∗ `Scan` is modified into `TScan` appropriately

– Create a new `labelheap` package by modifying the `heap` package, and the classes within, accordingly.

 ∗ `HeapFile` class is modified into `LabelHeapFile` as follows:

  · `LabelHeapfile(java.lang.String name)`: Initialize.
  · `void deleteFile()`: Delete the file from the database.
  · `boolean deleteLabel(LID lid)`: Delete label with given lid from the file.
  · `int getLabelCnt()`: Return number of labels in the file.
  · `java.lang.String getLabel(LID lid)`: Read the label with the given lid from file.

    · `LID insertLabel(java.lang.String Label)` Insert label into file, return its LID. If the label alreday exists, then simply return the existing LID.
    · `LScan openScan()`: Initiate a sequential scan.
    · `boolean updateLabel(LID lid, java.lang.String newLabel)`: Updates the specified label
  ∗ `Tuple` is modified into `Label` as follows:
    · `Label()`: Class constructor creates a new label with the appropriate size.
    · `java.lang.String getLabel()`: Returns the label
    · `Label setLabel(java.lang.String label)`: Sets the label.
    · `void print()`: Print out the label.
  ∗ `HFPage` is modified into `LHFPage` appropriately
  ∗ `Scan` is modified into `LScan` appropriately

- 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`. Modify the classes (and the methods) in the `btree` package in such a way that instead of indexing records in `HeapFiles` (using RIDs), it can be used for indexing quadruples in the `QuadrupleHeapFiles` (using QIDs) or for indexing labels in `LabelHeapFiles` (using LIDs).

- Under the `diskmgr` package, create a new class called `rdfDB` by modifying `diskmgr.DB`. This class creates and maintains all the relevant files (entitylabel heap file, predicate label heap file, quadruple 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.rdfDB` also contains the following classes and methods:

  – `rdfDB(int type)`: Constructor for the RDF database. `type` is an integer denoting the different clustering and indexing strategies you will use for the rdf database. Note that each RDF database contains one `QuadrupleHeapFile` to store the quadruples, one `LabelHeapfile` to store entity labels, and another `LabelHeapfile` to store subject labels. You can create as many btree index files as you want over these quadruple and label heap files.
  – `int getQuadrupleCnt()`: Returns the number of quadruples in the database.
  – `int getEntityCnt()`: Returns thenumber of entities in the database.
  – `int getPredicateCnt()`: Returns the number of predicates in the database
  – `int getSubjectCnt()`: Returns the number of distinct subjects in the database.
  – `int getObjectCnt()`: Returns the number of distinct objects in the database.
  – `EID insertEntity(java.lang.String EntityLabel)` Inserts the given entity label into the database and return its entity id. The `insertEntity()` method ensures that there is only one entity with a given entity label in the database.
  – `boolean deleteEntity(java.lang.String EntityLabel)` Removes the given entity label from the database.

  - `PID insertPredicate(java.lang.String PredicateLabel)` Inserts the given predicate label into the database and return its predicate id. The `insertPredicate()` method ensures that there is only one predicate with a given entity label in the database.
  - `boolean deletePredicate(java.lang.String PredicateLabel)` Removes the given predicate label from the database.
  - `QID insertQuadruple(byte[] quadruplePtr)` Inserts the given quadruple into the database and return its quadruple id. The `insertQuadruple()` method ensures that there is only one quadruple with the same subject, predicate, and object. If a second one is attempted to be inserted, only the one with the **largest** confidence is maintained in the database.
  - `boolean deleteQuadruple(byte[] quadruplePtr)` Removes the given quadruple from the database.
  - `Stream openStream(int orderType, java.lang.String subjectFilter, java.lang.String predicateFilter, java.lang.String objectFilter, double confidenceFilter)`: Initialize a stream of quadruples, where the subject label matches `subjectFilter`, predicate label matches `predicateFilter`, object label matches `objectFilter`, and confidence is greater than or equal to the `confidenceFilter`. If any of the filters are null strings or 0, then that filter is not considered (e.g., if `subjectFilter` is null, then all subject labels are OK). If `orderType` is
    * 1, then results are first ordered in subject label, then predicate label, then object label, and then confidence,
    * 2, then results are first ordered in predicate label, then subject label, then object label, and then confidence,
    * 3, then results are first ordered in subject label, then confidence,
    * 4, then results are first ordered in predicate label, then confidence,
    * 5, then results are first ordered in object label, then confidence, and
    * 6, then results are ordered in confidence.

- `diskmgr.Stream`: This class will be similar to `heap.Scan`, however, will provide different types of accesses to the quadruples in the RDF database:
  - `Stream(rdfDB rdfdatabase, int orderType, java.lang.String ubjectFilter, java.lang.String predicateFilter, java.lang.String objectFilter, double confidenceFilter)`: Initialize a stream of quadruples on `rdfdbable`.
  - `void closestream()`: Closes the stream object.
  - `Quadruple getNext(QID qid)` : Retrieve the next quadruple in the stream.

- Create the class `iterator.QuadrupleUtils` by modifying the class `iterator.TupleUtils`. For example,
  - `static int CompareQuadrupleWithQuadruple(Quadruple q1, Quadruple q2, int quadruple_fld_no)`: This function compares a quadruple with another quadruple in respective field, and returns: 0 if the two are equal, 1 if `q1` is greater, -1 if `q2` is smaller

  - `static boolean Equal(Quadruple q1, Quadruple q2)`: This function compares two quadruples in all fields

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

- Create the class `iterator.LabelUtils` by modifying the class `iterator.TupleUtils`.
- 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 `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 INDEXOPTION RDFDBNAME
  ```

  where `DATAFILENAME` and `RDFDNAME` are strings and `INDEXOPTION` is an integer, the program will store all the quadruples in an RDF database indexed according to `INDEXOPTION`.

  IMPORTANT: As part of your work, I expect that you will develop at least 5 different indexing schemes.

  The format of the data file will be as follows:

  ```
  subjectlabel1 predicatelabel1 objectlabel1 confidence1
  subjectlabel2 predicatelabel2 objectlabel2 confidence2
  .....
  ```

Given these quadruples, the name of the rdf database that will be created in the database will be `RDFDBNAME_INDEXOPTION`. If this rdf database already exists in the database, the tuples will be inserted into the existing rdf database.

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 RDFDBNAME INDEXOPTION ORDER SUBJECTFILTER PREDICATEFILTER OBJECTFILTER CONFIDENCEFILTER NUMBU
  ```

  the program will access the database and printout the matching maps in the requested order ("*" indicates a null filter).

  Minibase will use **<u>at most</u>** `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).

- Implement a command line program `report` which outputs the various statistics for the RDF database.

  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), **<u>do not</u>** 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.
- A report following the given report document structure. The report should experimentally analyze the read and write performance of different 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.