Mickey Kim (pe18297)
Java Database Assignment Report

Preface:

I wrote the following as text files to document my progress through the assignment. Each text file also had some copy-pasted code as the snapshot of current progress. I made sure to do appropriate unit testing for each stage of development, as is reflected in the code. Any refactoring benefited from this as tests had to either pass or also be refactored to ensure code operation. Writing and performing testing probably took longer than the actual code, but it was invaluable in both ensuring that the actual code did what I intended and also gave me some reassurance that everything was proceeding well.

I. Record.java

For the first version of Record.java, I implemented the strings as a list. I chose to use a list rather than an array because databases have fields that can be changed often. Thus, a list makes more sense as items can be more efficiently added or deleted. Records are thus also accessed by index like an array. This seems reasonable for now. This first version has a simple length query and get/set methods along with their respective helper methods (add and clear lists) with tests.

The first version of Record.java is as follows:

```
import java.util.ArrayList;
import java.util.List;
import java.io.*;

class Record {
   private List<String> row;

   Record() {
      this.row = new ArrayList<>();
   }

   Record(String... data) {
      this.row = new ArrayList<>();
      for (int i = 0; i < data.length; i++) {
         this.row.add(data[i]);
      }
   }

   void add(String data) {
      this.row.add(data);
   }

   void clear() {
      this.row.clear();
   }

   int size() {
      return this.row.size();
   }

   String getField(int idx) {
      checkRecordExists(idx);
      return this.row.get(idx);
   }

   void setField(int idx, String data) {
      checkRecordExists(idx);
      this.row.set(idx, data);
   }
```

```
   private void checkRecordExists(int idx) {
      if (idx >= this.row.size() || idx < 0) {
         System.out.println("No such field in record.");
         throw new IndexOutOfBoundsException();
      }
   }

   // ---  testing ---

   // tests add(string), length(), and clear() methods
   void basicTests() {
      Record test = new Record();
      test.add("test1");
      test.add("test2");
      assert(test.size() == 2);
      test.clear();
      Record testArr = new Record("test1", "test2");
      assert(testArr.size() == 2);
   }

   // tests getField(int) setField(int, String)
   void getsetTests() {
      //redirect System.out
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      PrintStream out = new PrintStream(baos);
      PrintStream console = System.out;
      System.setOut(out);
      //Begin tests
      Record test = new Record();
      boolean caught = false;
      test.add("test1");
      test.add("test2");
      assert(test.getField(0).equals("test1"));
      assert(test.getField(1).equals("test2"));
      //out of bounds call
      try { test.getField(2); }
      catch (IndexOutOfBoundsException e) { caught = true; }
      assert(caught == true);
      caught = false;
      try { test.getField(-1); }
      catch (IndexOutOfBoundsException e) { caught = true; }
      assert(caught == true);
      caught = false;
      test.setField(0, "update1");
      assert(test.getField(0).equals("update1"));
      //check nothing is incorrectly pushed down in the list
      assert(test.getField(1).equals("test2"));
      try { test.getField(2); }
      catch (IndexOutOfBoundsException e) { caught = true; }
      assert(caught == true);
      caught = false;
      test.setField(1, "update2");
      assert(test.getField(0).equals("update1"));
      assert(test.getField(1).equals("update2"));
      try { test.getField(2); }
      catch (IndexOutOfBoundsException e) { caught = true; }
      assert(caught == true);
      caught = false;
      Record testArr = new Record("test1", "test2");
      assert(testArr.size() == 2);
      assert(testArr.getField(0).equals("test1"));
      assert(testArr.getField(1).equals("test2"));
      //reset System.out
      System.out.flush();
      System.setOut(console);
   }

   void runTests(String[] args) {
      basicTests();
```

```
        getsetTests();
    }

    public static void main(String[] args) {
        Record program = new Record();
        program.runTests(args);
    }
}
```

II. Table.java

For Table.java, I created a class with a variety of different methods to handle record aggregation. There are three fields at the moment which contain the table's name, column headers, and records. The column headers are a list of strings, and records are a list of records. Records are accessed from the list via index starting from [0]. There are constructors which, depending on arguments, set the table's name and columns from arguments passed as strings. Additionally, there are methods to get and set column names, get column and record sizes, and methods to select, add, insert, delete, and update records. Relevant methods are validated by checking if the record has the same number of items as the table's columns. Columns and records are both currently accessed through indices.

The initial Table.java code is as follows:

```
import java.util.ArrayList;
import java.util.List;
import java.io.*;

class Table {
    private String name;
    private List<String> columns;
    private List<Record> records;

    enum DataType{
        RecordData, ColumnData
    }

    Table() {
        this.name = "untitled";
        this.columns = new ArrayList<String>();
        this.records = new ArrayList<Record>();
    }

    Table(String name) {
        this.name = name;
        this.columns = new ArrayList<String>();
        this.records = new ArrayList<Record>();
    }

    Table(String name, String... columns) {
        this.name = name;
        this.columns = new ArrayList<String>();
        this.records = new ArrayList<Record>();
        setColumnNames(columns);
    }

    void setColumnNames(String... names) {
        this.columns.clear();
        for (String entry : names) {
            this.columns.add(entry);
        }
    }

    int getColumnSize() {
        return this.columns.size();
    }
```

```
int getRecordSize() {
   return this.records.size();
}

String getColumnName(int idx) {
   checkDataExists(Table.DataType.ColumnData, idx);
   return this.columns.get(idx);
}

String getName() {
   return this.name;
}

void setName(String name) {
   this.name = name;
}

Record select(int idx) {
   checkDataExists(Table.DataType.RecordData, idx);
   return this.records.get(idx);
}

void add(Record data) {
   checkRecordInputSize(data);
   this.records.add(data);
}

void insert(int idx, Record data) {
   checkDataExists(Table.DataType.RecordData, idx);
   checkRecordInputSize(data);
   this.records.add(idx, data);
}

void delete(int idx) {
   checkDataExists(Table.DataType.RecordData, idx);
   this.records.remove(idx);
}

void update(int idxRecord, int idxField, String data) {
   select(idxRecord).setField(idxField, data);
}

private void checkDataExists(Table.DataType dataType, int idx) {
   int datasz = 0;
   if (dataType == Table.DataType.RecordData) {
      datasz = this.records.size();
   } else if (dataType == Table.DataType.ColumnData) {
      datasz = this.columns.size();
   }
   if (idx >= datasz || idx < 0) {
      System.out.println("No such data in table.");
      throw new IndexOutOfBoundsException();
   }
}

private void checkRecordInputSize(Record data) {
   if (data.size() != this.columns.size()) {
      System.out.println("Input data does not match table columns.");
      throw new IllegalArgumentException();
   }
}

//--- testing ---

void testTableCreation() {
   //redirect System.out
   ByteArrayOutputStream baos = new ByteArrayOutputStream();
   PrintStream out = new PrintStream(baos);
   PrintStream console = System.out;
```

```
        System.setOut(out);
        //Begin tests
        Table test = new Table();
        assert(test.getName().equals("untitled"));
        String testName = "test";
        test.setName(testName);
        assert(test.getName().equals(testName));
        Table test0 = new Table(testName);
        assert(test0.getName().equals(testName));
        //checking column validity
        test0.setColumnNames("1", "2", "3", "4");
        assert(test0.getColumnName(0).equals("1"));
        assert(test0.getColumnName(1).equals("2"));
        assert(test0.getColumnName(2).equals("3"));
        assert(test0.getColumnName(3).equals("4"));
        Table test1 = new Table("title", "X", "Y", "Z");
        assert(test1.getColumnName(0).equals("X"));
        assert(test1.getColumnName(1).equals("Y"));
        assert(test1.getColumnName(2).equals("Z"));
        //reset System.out
        System.out.flush();
        System.setOut(console);
    }

    void testTableManipulation() {
        //redirect System.out
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        PrintStream out = new PrintStream(baos);
        PrintStream console = System.out;
        System.setOut(out);
        //Begin tests
        //checking add records
        Table test1 = new Table();
        test1.setColumnNames("1", "2", "3", "4");
        Record testR1 = new Record("a", "b", "c", "d");
        test1.add(testR1);
        assert(test1.select(0).getField(0).equals("a"));
        assert(test1.select(0).getField(1).equals("b"));
        assert(test1.select(0).getField(2).equals("c"));
        assert(test1.select(0).getField(3).equals("d"));
        Record testR2 = new Record("a", "b", "c");
        boolean caught = false;
        try { test1.add(testR2); }
        catch (IllegalArgumentException e) { caught = true; }
        assert(caught == true);
        caught = false;
        //checking delete records
        test1.delete(0);
        Record testR3 = new Record("q", "w", "e", "r");
        test1.add(testR3);
        assert(test1.select(0).getField(0).equals("q"));
        assert(test1.select(0).getField(1).equals("w"));
        assert(test1.select(0).getField(2).equals("e"));
        assert(test1.select(0).getField(3).equals("r"));
        test1.delete(0);
        try { test1.delete(0); }
        catch (IndexOutOfBoundsException e) { caught = true; }
        assert(caught == true);
        caught = false;
        //checking insert records
        test1.add(testR1);
        test1.add(testR1);
        test1.insert(1, testR3);
        assert(test1.select(1).getField(0).equals("q"));
        assert(test1.select(1).getField(1).equals("w"));
        assert(test1.select(1).getField(2).equals("e"));
        assert(test1.select(1).getField(3).equals("r"));
        try { test1.insert(4, testR3); }
        catch (IndexOutOfBoundsException e) { caught = true; }
```

```
      assert(caught == true);
      caught = false;
      //checking update records
      test1.delete(0);
      test1.delete(0);
      test1.add(testR1);
      test1.update(0, 0, "x");
      assert(test1.select(0).getField(0).equals("x"));
      try { test1.update(2, 0, "x"); }
      catch (IndexOutOfBoundsException e) { caught = true; }
      assert(caught == true);
      caught = false;
      try { test1.update(0, 4, "x"); }
      catch (IndexOutOfBoundsException e) { caught = true; }
      assert(caught == true);
      caught = false;
      //reset System.out
      System.out.flush();
      System.setOut(console);
   }

   void runTests() {
      testTableCreation();
      testTableManipulation();
   }

   public static void main(String[] args) {
      Table program = new Table();
      program.runTests();
   }
}
```

III. File.java

For the File.java class, I first decided on what my delimiters would be. I thought that the files would be good if they were in a readable text format. Thus, I chose to use tabs as unit delimiters rather than the suggested commas, as I think tabs would be less common to find in strings than commas. I only wrote four methods for this first iteration: getName, writeTableToString, writeStringToFile, and readFileToTable. The methods for the latter three are a bit long, but they are rather straightforward, using methods I have written in the previous classes. If I need to refactor them later, they are also generally split in a logical order of sub-processes.

The initial File.java code is as follows:

```
import java.io.*;

class File {
   private static final String UNITDELIM = "\t";
   private static final String RCRDDELIM = "\n";
   private static final String EXTENSION = ".mdb";

   private String filename;
   private int lineCnt;

   File() {
      this.filename = "untitled" + EXTENSION;
   }

   File(String file) {
      this.filename = file + EXTENSION;
   }

   String getName() {
      return this.filename;
   }
```

```
String writeTableToString(Table table) {
   StringBuilder output = new StringBuilder();
   int colsz = table.getColumnSize();
   int recsz = table.getRecordSize();
   // table name
   output.append(table.getName() + RCRDDELIM);
   // columns
   for (int i = 0; i < colsz; i++) {
      output.append(table.getColumnName(i));
      if (i < colsz - 1) {
         output.append(UNITDELIM);
      }
   }
   output.append(RCRDDELIM);
   // records
   for (int i = 0; i < recsz; i++) {
      for (int j = 0; j < colsz; j++) {
         output.append(table.select(i).getField(j));
         if (j < colsz - 1) {
            output.append(UNITDELIM);
         }
      }
      output.append(RCRDDELIM);
   }
   return output.toString();
}

void writeStringToFile(String input) {
   try (PrintStream ps = new PrintStream(filename)) {
      ps.print(input);
   } catch (Exception e) {
      System.out.println(e.getMessage());
   }
}

Table readFileToTable(String filepath) throws Exception {
   BufferedReader bReader = new BufferedReader(new FileReader(filepath));
   String line;
   String tableName = new String();
   Table outputTable = new Table();
   this.lineCnt = 0;
   while ((line = bReader.readLine()) != null) {
      if (this.lineCnt == 0) {
         tableName = line;
         this.lineCnt++;
      } else if (this.lineCnt == 1) {
         String[] headers = line.split(UNITDELIM);
         outputTable = new Table(tableName, headers);
         this.lineCnt++;
      } else {
         Record newRecord = new Record();
         String[] fields = line.split(UNITDELIM);
         for (String entry : fields) {
            newRecord.add(entry);
         }
         outputTable.add(newRecord);
         this.lineCnt++;
      }
   }
   return outputTable;
}

void testFileCreation() {
   // make file objects
   File test0 = new File();
   assert(test0.getName().equals("untitled" + EXTENSION));
   String testStr = "test";
   File testFile = new File(testStr);
```

```
      assert(testFile.getName().equals(testStr + EXTENSION));
      // make tables
      Table testTable = new Table(testStr);
      testTable.setColumnNames("1", "2", "3");
      Record testR1 = new Record("a", "b", "c");
      testTable.add(testR1);
      Record testR2 = new Record("x", "y", "z");
      testTable.add(testR2);
      String testOutputStr = testFile.writeTableToString(testTable);
      assert(testOutputStr.equals(
         testStr + RCRDDELIM +
         "1" + UNITDELIM + "2" + UNITDELIM + "3" + RCRDDELIM +
         "a" + UNITDELIM + "b" + UNITDELIM + "c" + RCRDDELIM +
         "x" + UNITDELIM + "y" + UNITDELIM + "z" + RCRDDELIM
      ));
      testFile.writeStringToFile(testOutputStr);
   }

   void testFileParsing() {
      String testStr = "test";
      File testFile = new File(testStr);
      Table testOut = new Table();
      boolean caught = false;
      try { testOut = testFile.readFileToTable(testStr + EXTENSION); }
      catch (Exception e) { caught = true; }
      assert(caught == false);
      assert(testFile.getName().equals(testStr + EXTENSION));
      String testInputFile = testFile.writeTableToString(testOut);
      assert(testInputFile.equals(
         testStr + RCRDDELIM +
         "1" + UNITDELIM + "2" + UNITDELIM + "3" + RCRDDELIM +
         "a" + UNITDELIM + "b" + UNITDELIM + "c" + RCRDDELIM +
         "x" + UNITDELIM + "y" + UNITDELIM + "z" + RCRDDELIM
      ));
   }

   void runTests() {
      testFileCreation();
      testFileParsing();
   }

   public static void main(String[] args) {
      File program = new File();
      program.runTests();
   }
}
```

IV. Printing

For printing, I wanted the output to be similar to MariaDB. The major challenge in accomplishing this, at a basic level, is autosizing the width of columns based on the longest field in each column. To solve this, I created a colWidth field in the Print class as an array of integers which would hold the maximum size of each column. I then created some methods which would traverse along a Table's columns and records to find and set max values for colWidth based on the length of each field's string. Next, I decided that the Print methods would generate strings, formatted to create an ASCII grid like MariaDB outputs. The non-data characters of the grid were set as constants (for flexibility in changing them later). I also set some constants to determine the amount of empty space padding on the x-axis to the left and right of each field. I then created methods to generate the horizontal divider along with formatting the column names and record data to insert vertical bars between the fields. I initially had a separate method for both column names and record fields, but they were almost identical, so I refactored them into the same method. If the column data is required, the method is sent an argument of "-1" for int idxData. If the record data is required, integers of 0 or greater are sent (corresponding to their index).

The current Print.java code is as follows:

```
class Print {
   private int[] colWidth;

   private static final String X_DIV = "+";
   private static final String H_DIV = "-";
   private static final String V_DIV = "|";
   private static final String EMPTY = " ";
   private static final String NEWLN = "\n";
   private static final int XPADD = 2;
   private static final int HPADD = XPADD/2;

   Print() {
      this.colWidth = new int[0];
   }

   // return a copy of the selected width
   int getColWidth(int i) {
      int ret = this.colWidth[i];
      return ret;
   }

   // set initial column widths from column names
   void setInitialWidths(Table inputTable, int colNum) {
      this.colWidth = new int[colNum];
      for (int i = 0; i < colNum; i++) {
         this.colWidth[i] = inputTable.getColumnName(i).length();
      }
   }

   // get max column widths from record fields
   void setMaxWidths(Table inputTable) {
      int colNum = inputTable.getColumnSize();
      setInitialWidths(inputTable, colNum);
      for (int i = 0; i < inputTable.getRecordSize(); i++) {
         for (int j = 0; j < colNum; j++) {
            this.colWidth[j] =
               Math.max(this.colWidth[j], inputTable.select(i).getField(j).length());
         }
      }
   }

   // Returns string like "+---+---+---+\n"
   String generateHorizontalDivider(Table inputTable) {
      int colsz = inputTable.getColumnSize();
      StringBuilder horDivBuilder = new StringBuilder();
      for (int i = 0; i < colsz; i++) {
         horDivBuilder.append(X_DIV);
         for (int j = 0; j < getColWidth(i) + XPADD; j++) {
            horDivBuilder.append(H_DIV);
         }
      }
      horDivBuilder.append(X_DIV);
      horDivBuilder.append(NEWLN);
      return horDivBuilder.toString();
   }

   // Returns string like "| data 1 | data 2 | data 3 |\n"
   // idxData = -1 for column headers; idxData >= 0 for records
   String generateDataString(Table inputTable, int idxData) {
      int colsz = inputTable.getColumnSize();
      if (idxData > colsz || idxData < -1) {
         System.out.println("No such field in table.");
         throw new IndexOutOfBoundsException();
      }
      StringBuilder dataBuilder = new StringBuilder();
      String currField;
      for (int i = 0; i < colsz; i++) {
         if (idxData == -1) {
```

```
            currField = inputTable.getColumnName(i);
        } else {
            currField = inputTable.select(idxData).getField(i);
        }
        dataBuilder.append(V_DIV);
        dataBuilder.append(EMPTY);
        dataBuilder.append(currField);
        for (int j = 0; j < getColWidth(i) - currField.length() + HPADD; j++) {
            dataBuilder.append(EMPTY);
        }
    }
    dataBuilder.append(V_DIV);
    dataBuilder.append(NEWLN);
    return dataBuilder.toString();
}

// Returns string   +---+---+---+
// in the format:   | 1 | 2 | 3 |
//                  +---+---+---+
//                  | a | b | c |
//                  +---+---+---+
String printTableToString(Table inputTable) {
    int colsz = inputTable.getColumnSize();
    int recsz = inputTable.getRecordSize();
    String horDiv = generateHorizontalDivider(inputTable);
    StringBuilder tableStringBuilder = new StringBuilder();
    tableStringBuilder.append(horDiv);
    // i = -1 for column headers; i = 0..recsz for records
    for (int i = -1; i < recsz; i++) {
        tableStringBuilder.append(generateDataString(inputTable, i));
        if (i == -1) {
            tableStringBuilder.append(horDiv);
        }
    }
    tableStringBuilder.append(horDiv);
    return tableStringBuilder.toString();
}

// --- testing ---

void testPrintingMethods() {
    Print testPrint = new Print();
    String testStr = "test_table";
    // make tables
    Table testTable = new Table(testStr);
    testTable.setColumnNames("1", "2", "3");
    Record testR1 = new Record("a", "b", "c");
    testTable.add(testR1);
    Record testR2 = new Record("dd", "eee", "fff");
    testTable.add(testR2);
    Record testR3 = new Record("gg", "hh", "iiii");
    testTable.add(testR3);
    testPrint.setInitialWidths(testTable, testTable.getColumnSize());
    assert(testPrint.getColWidth(0)==1);
    assert(testPrint.getColWidth(1)==1);
    assert(testPrint.getColWidth(2)==1);
    testPrint.setMaxWidths(testTable);
    assert(testPrint.getColWidth(0)==2);
    assert(testPrint.getColWidth(1)==3);
    assert(testPrint.getColWidth(2)==4);
    assert(testPrint.generateHorizontalDivider(testTable).equals(
        "+----+-----+------+\n"
    ));
    assert(testPrint.generateDataString(testTable, -1).equals(
        "| 1  | 2   | 3    |\n"
    ));
    assert(testPrint.generateDataString(testTable, 0).equals(
        "| a  | b   | c    |\n"
    ));
```

```
      assert(testPrint.generateDataString(testTable, 1).equals(
         "| dd | eee | fff  |\n"
      ));
      assert(testPrint.generateDataString(testTable, 2).equals(
         "| gg | hh  | iiii |\n"
      ));
      assert(testPrint.printTableToString(testTable).equals(
         "+----+-----+------+\n" +
         "| 1  | 2   | 3    |\n" +
         "+----+-----+------+\n" +
         "| a  | b   | c    |\n" +
         "| dd | eee | fff  |\n" +
         "| gg | hh  | iiii |\n" +
         "+----+-----+------+\n"
      ));
   }

   void runTests() {
      testPrintingMethods();
   }

   public static void main(String[] args) {
      Print program = new Print();
      program.runTests();
   }
}
```

V. Keys

Adding keys to records was an interesting challenge. At first, I thought that perhaps the Record class should be responsible for its own key. But the requirement that keys be unique to a table suggested that the key values should instead be handled by the Table class. I thought then that perhaps I should create a list of keys, but then how should the keys and the records be linked? Perhaps, instead, I could use a Linked Hash Map, with hash keys being record keys and hash values being records themselves. I could then have the keys be specified by the user, with the Table class handling validation. Unfortunately, hash maps don't extend collections, so the generic functions such as add() broke. Also, if Records are identified via keys instead of index, inserting records doesn't seem to make sense (and also not possible using a Linked Hash Map).

Just how the keys should be specified was another challenge. It didn't seem very elegant for the user to have to specify what the key should be every time they want to add or update a record. In MariaDB, tables are first created with attributes and then populated with records that adhere to those attributes. I thought then that the columns in a table could hold data about the table's attributes, including which column holds the record keys. Thus, I decided to create a new class called ColumnID.java which would just have some basic information about what the column name is and whether or not it contains keys. I also considered the possibility of adding attribute types later, which would fit nicely somewhere in this class.

Currently, ColumnID.java is as follows:

```
class ColumnID {
   private String name;
   private boolean containsKeys;

   ColumnID() {
   }

   ColumnID(String name) {
      this.name = name;
      this.containsKeys = false;
   }

   ColumnID(String name, boolean containsKeys) {
```

```
      this.name = name;
      this.containsKeys = containsKeys;
   }

   String getName() {
      return this.name;
   }

   void setName(String name) {
      this.name = name;
   }

   boolean containsKeys() {
      boolean status = this.containsKeys;
      return status;
   }

   void setKeyStatus(boolean status) {
      this.containsKeys = status;
   }

   // --- testing ---

   void testColumnCreation() {
      String testStr = "test";
      String newTest = "testkey";
      ColumnID test0 = new ColumnID(testStr);
      assert(test0.getName().equals(testStr));
      assert(test0.containsKeys() == false);
      ColumnID test1 = new ColumnID(newTest, true);
      assert(test1.getName().equals(newTest));
      assert(test1.containsKeys() == true);
      test1.setKeyStatus(false);
      assert(test1.containsKeys() == false);
   }

   void runTests() {
      testColumnCreation();
   }

   public static void main(String[] args) {
      ColumnID program = new ColumnID();
      program.runTests();
   }
}
```

After refactoring, Table.java currently looks like this:

```
import java.util.ArrayList;
import java.util.List;
import java.util.LinkedHashMap;
import java.io.*;

class Table {
   private String name;
   private List<ColumnID> columns;
   private LinkedHashMap<String,Record> records;
   private int keyColumn = -1;

   private static final String noSuchRecord = "No such record exists in table.";
   private static final String noSuchColumn = "No such column exists in table.";
   private static final String duplicateKey = "Duplicate key exists in table.";
   private static final String noKeySpecified = "No key specified in table.";

   Table() {
      this.name = "untitled";
      this.columns = new ArrayList<ColumnID>();
      this.records = new LinkedHashMap<String,Record>();
```

```
}

Table(String name) {
   this.name = name;
   this.columns = new ArrayList<ColumnID>();
   this.records = new LinkedHashMap<String,Record>();
}

Table(String name, ColumnID... columns) {
   this.name = name;
   this.columns = new ArrayList<ColumnID>();
   this.records = new LinkedHashMap<String,Record>();
   setColumnIDs(columns);
}

void setColumnIDs(ColumnID... columns) {
   checkIfUniqueKeyColumnExists(columns);
   for (ColumnID entry : columns) {
      this.columns.add(entry);
   }
}

int getColumnSize() {
   return this.columns.size();
}

int getRecordSize() {
   return this.records.size();
}

String getColumnName(int idx) {
   checkIfColumnExists(idx);
   return this.columns.get(idx).getName();
}

String getName() {
   return this.name;
}

void setName(String name) {
   this.name = name;
}

Record select(String key) {
   checkIfRecordExists(key);
   return this.records.get(key);
}

void add(Record data) {
   String key = data.getField(this.keyColumn);
   checkIfRecordsMatchColumns(data);
   checkIfDuplicateKey(key);
   this.records.put(key, data);
}

//TO DO: check if trying to update key but key already exists
void update(String key, int idx, String input) {
   checkIfRecordExists(key);
   select(key).setField(idx, input);
}

void delete(String key) {
   checkIfRecordExists(key);
   this.records.remove(key);
}

// --- helper methods ---

private void checkIfUniqueKeyColumnExists(ColumnID... values) {
```

```
    int colCnt = 0;
    this.columns.clear();
    for (ColumnID entry : values) {
       if (entry.containsKeys() == true && this.keyColumn == -1) {
          this.keyColumn = colCnt;
       } else if (entry.containsKeys() == true && this.keyColumn != -1) {
          this.columns.clear();
          System.out.println(duplicateKey);
          throw new IllegalArgumentException();
       }
       colCnt++;
    }
    checkIfKeyColumnExists();
 }

 private void checkIfKeyColumnExists() {
    if (this.keyColumn == -1) {
       this.columns.clear();
       System.out.println(noKeySpecified);
       throw new IllegalArgumentException();
    }
 }

 private void checkIfColumnExists(int idx) {
    if (idx >= this.columns.size() || idx < 0) {
       System.out.println(noSuchColumn);
       throw new IndexOutOfBoundsException();
    }
 }

 private void checkIfRecordExists(String key) {
    if (!this.records.containsKey(key)) {
       System.out.println(noSuchRecord);
       throw new IllegalArgumentException();
    }
 }

 private void checkIfDuplicateKey(String key) {
    if (this.records.containsKey(key)) {
       System.out.println(duplicateKey);
       throw new IllegalArgumentException();
    }
 }

 private void checkIfRecordsMatchColumns(Record data) {
    if (data.size() != this.columns.size()) {
       System.out.println("Input data does not match table columns.");
       throw new IllegalArgumentException();
    }
 }

 List<String> getKeyList(){
    ArrayList<String> keys = new ArrayList<>(records.keySet());
    return keys;
 }

 int getKeyColumn() {
    int keyColumn = this.keyColumn;
    return keyColumn;
 }

 // --- testing ---

 private void testTableCreation() {
    //redirect System.out
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    PrintStream out = new PrintStream(baos);
    PrintStream console = System.out;
    System.setOut(out);
```

```
    //Begin tests
    Table test = new Table();
    assert(test.getName().equals("untitled"));
    String testName = "test";
    test.setName(testName);
    assert(test.getName().equals(testName));
    Table test0 = new Table(testName);
    assert(test0.getName().equals(testName));
    //checking column validity
    test0.setColumnIDs(
       new ColumnID("key", true),
       new ColumnID("2", false),
       new ColumnID("3"),
       new ColumnID("4")
    );
    assert(test0.getColumnName(0).equals("key"));
    assert(test0.getColumnName(1).equals("2"));
    assert(test0.getColumnName(2).equals("3"));
    assert(test0.getColumnName(3).equals("4"));
    Table test1 = new Table(
       "title",
       new ColumnID("X", true),
       new ColumnID("Y", false),
       new ColumnID("Z")
    );
    assert(test1.getColumnName(0).equals("X"));
    assert(test1.getColumnName(1).equals("Y"));
    assert(test1.getColumnName(2).equals("Z"));
    //reset System.out
    System.out.flush();
    System.setOut(console);
}

private void testTableManipulation() {
    //redirect System.out
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    PrintStream out = new PrintStream(baos);
    PrintStream console = System.out;
    System.setOut(out);
    //Begin tests
    //checking add records
    Table test1 = new Table();
    test1.setColumnIDs(
       new ColumnID("key", true),
       new ColumnID("2", false),
       new ColumnID("3"),
       new ColumnID("4")
    );
    Record testR1 = new Record("key1", "b", "c", "d");
    test1.add(testR1);
    assert(test1.select("key1").getField(0).equals("key1"));
    assert(test1.select("key1").getField(1).equals("b"));
    assert(test1.select("key1").getField(2).equals("c"));
    assert(test1.select("key1").getField(3).equals("d"));
    Record testR2 = new Record("a", "b", "c");
    boolean caught = false;
    try { test1.add(testR2); }
    catch (IllegalArgumentException e) { caught = true; }
    assert(caught == true);
    caught = false;
    //checking delete records
    test1.delete("key1");
    Record testR3 = new Record("key2", "w", "e", "r");
    test1.add(testR3);
    assert(test1.select("key2").getField(0).equals("key2"));
    assert(test1.select("key2").getField(1).equals("w"));
    assert(test1.select("key2").getField(2).equals("e"));
    assert(test1.select("key2").getField(3).equals("r"));
    test1.delete("key2");
```

```
      try { test1.delete("key2"); }
      catch (IllegalArgumentException e) { caught = true; }
      assert(caught == true);
      caught = false;
      //checking update records
      test1.add(testR1);
      test1.update("key1", 0, "x");
      assert(test1.select("key1").getField(0).equals("x"));
      try { test1.update("b", 0, "x"); }
      catch (IllegalArgumentException e) { caught = true; }
      assert(caught == true);
      caught = false;
      try { test1.update("key1", 4, "x"); }
      catch (IndexOutOfBoundsException e) { caught = true; }
      assert(caught == true);
      caught = false;
      //reset System.out
      System.out.flush();
      System.setOut(console);
   }

   void runTests() {
      testTableCreation();
      testTableManipulation();
   }

   public static void main(String[] args) {
      Table program = new Table();
      program.runTests();
   }
}
```

I also had to refactor File.java to incorporate keys. For saving keys, I decided to put an asterisk in front of the column name that held the record keys. Likewise, I converted the files to tables using that asterisk to denote key columns.

Additionally, I also had to refactor Print.java to incorporate the changes to Table methods. As records are accessed by keys instead of indices, I had to change the for loops to iterations through the records hash map (which was possible as they were converted to a linked hash map).

VI. Database.java

For Database organization, I created a file called Database.java which serves as a collection of tables along with information about the database's name and folder. I took a lot of concepts from Table.java as it had similarities in aggregating objects. I decided that accessing tables by index didn't really make sense, so I put them in a LinkedHashMap with their keys being generated by table names (thus tables should have unique names, which is reasonable).

The current version of Database.java is as follows:

```
import java.util.ArrayList;
import java.util.List;
import java.util.LinkedHashMap;
import java.io.*;

class Database {
   private String name;
   private String folder;
   private LinkedHashMap<String,Table> tables;

   private static final String EXTENSION = ".dbf";
   private static final String noSuchTable = "No such table exists in database.";
   private static final String duplicateKey = "Duplicate table names in database.";
   private static final String sizeMismatch = "Records are not the same size.";
```

```
Database() {
   this.name = "untitled";
   this.folder = "databases";
   this.tables = new LinkedHashMap<String,Table>();
}

Database(String name) {
   this.name = name;
   this.folder = "databases";
   this.tables = new LinkedHashMap<String,Table>();
}

Database(String name, String folder) {
   this.name = name;
   this.folder = folder;
   this.tables = new LinkedHashMap<String,Table>();
}

Database(String name, String folder, Table... tables) {
   this.name = name;
   this.folder = folder;
   this.tables = new LinkedHashMap<String,Table>();
   setTables(tables);
}

void setTables(Table... tables) {
   this.tables.clear();
   for (Table entry : tables) {
      checkIfDuplicateKeyAndClear(entry.getName(), true);
      this.tables.put(entry.getName(), entry);
   }
}

String getName() {
   return this.name;
}

void setName(String name) {
   this.name = name;
}

String getFolder() {
   return this.folder;
}

void setFolder(String folder) {
   this.folder = folder;
}

void add(Table table) {
   String key = table.getName();
   checkIfDuplicateKeyAndClear(table.getName(), false);
   this.tables.put(key, table);
}

Table select(String key) {
   checkIfTableExists(key);
   return this.tables.get(key);
}

void delete(String key) {
   checkIfTableExists(key);
   this.tables.remove(key);
}

void update(String tableName, String recordName, Record newRecord) {
   checkIfTableExists(tableName);
   Record currRecord = select(tableName).select(recordName);
   checkValidRecordUpdate(currRecord, newRecord);
```

```
        for (int i = 0; i < currRecord.size(); i++) {
            select(tableName).update(recordName, i, newRecord.getField(i));
        }
    }

    List<String> getKeyList() {
        ArrayList<String> keys = new ArrayList<>(tables.keySet());
        return keys;
    }

    // --- helper methods ---

    private void checkValidRecordUpdate(Record currRecord, Record newRecord) {
        if (currRecord.size() != newRecord.size()) {
            System.out.println(sizeMismatch);
            throw new IndexOutOfBoundsException();
        }
    }

    private void checkIfTableExists(String key) {
        if (!this.tables.containsKey(key)) {
            System.out.println(noSuchTable);
            throw new IllegalArgumentException();
        }
    }

    private void checkIfDuplicateKeyAndClear(String key, boolean clear) {
        if (this.tables.containsKey(key)) {
            if (clear == true) this.tables.clear();
            System.out.println(duplicateKey);
            throw new IllegalArgumentException();
        }
    }

    // --- testing ---

    private void testDatabaseCreation() {
        // redirect System.out
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        PrintStream out = new PrintStream(baos);
        PrintStream console = System.out;
        System.setOut(out);
        // Begin tests
        // create database
        Database testDB = new Database();
        assert(testDB.getName().equals("untitled"));
        String testNameDB = "test_database";
        testDB.setName(testNameDB);
        assert(testDB.getName().equals(testNameDB));
        String testFolderDB = "test_folder";
        testDB.setFolder(testFolderDB);
        assert(testDB.getFolder().equals(testFolderDB));
        // create table1
        String testNameT1 = "test_table1";
        Table testTable1 = new Table(
            testNameT1,
            new ColumnID("key", true),
            new ColumnID("2", false),
            new ColumnID("3")
        );
        Record testT1R1 = new Record("key1", "1", "1");
        Record testT1R2 = new Record("key2", "1", "2");
        testTable1.add(testT1R1);
        testTable1.add(testT1R2);
        // create table2
        String testNameT2 = "test_table2";
        Table testTable2 = new Table(
            testNameT2,
            new ColumnID("key", true),
```

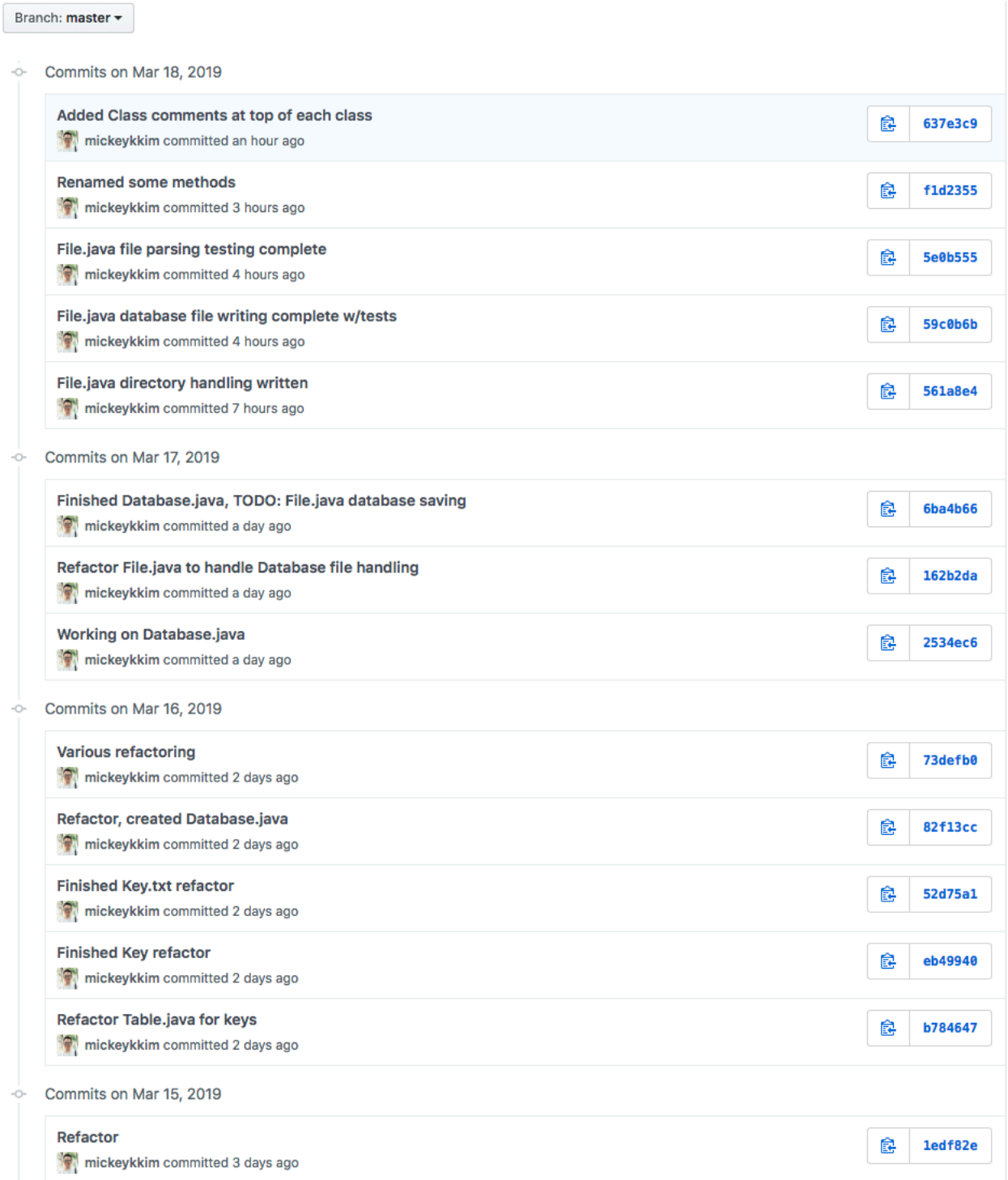
```
            new ColumnID("2", false),
            new ColumnID("3")
        );
        Record testT2R1 = new Record("key1", "1", "1");
        Record testT2R2 = new Record("key2", "1", "2");
        testTable2.add(testT2R1);
        testTable2.add(testT2R2);
        // add tables to database
        testDB.add(testTable1);
        assert(testDB.select(testTable1.getName()) == testTable1);
        testDB.add(testTable2);
        assert(testDB.select(testTable2.getName()) == testTable2);
        List<String> testDBKeys = testDB.getKeyList();
        assert(testDBKeys.get(0) == testNameT1);
        assert(testDBKeys.get(1) == testNameT2);
        // update tables
        Record testUpdateR = new Record("key1", "3", "3");
        testDB.update(testTable1.getName(), "key1", testUpdateR);
        assert(testDB.select(testTable1.getName()).select("key1").getField(1).equals("3"));
        assert(testDB.select(testTable1.getName()).select("key1").getField(2).equals("3"));
        // delete tables
        testDB.delete(testTable2.getName());
        boolean caught = false;
        try { testDB.select(testTable2.getName()); }
        catch (IllegalArgumentException e) { caught = true; }
        assert(caught == true);
        caught = false;
        try { testDB.delete(testTable2.getName()); }
        catch (IllegalArgumentException e) { caught = true; }
        assert(caught == true);
        caught = false;
        //reset System.out
        System.out.flush();
        System.setOut(console);
    }

    private void runTests() {
        testDatabaseCreation();
    }

    public static void main(String[] args) {
        Database program = new Database();
        program.runTests();
    }
}
```

At this point, I had to refactor File.java to handle saving and loading databases as files. This required thinking about how databases would be saved. I decided to have a database info file that consisted of the database's name, directory, and table file names. This would be saved with the tables as individual files using the methods I had initially written to the database's directory. When loading the files from the database info file, the database and tables as listed in the file would all be read in as objects. The testing for File.java involved creating databases and tables with records from hand-written strings and saving them to files in a user-specified subfolder. These same files were then read in and compared to a sampling of the objects (database, tables, and records) used to create them.

As it is Monday and everything up until the Database task is completed to my satisfaction, I decided to stop here and write the report. The final versions of each class are uploaded on SAFE along with a makefile to compile the code ($ make all will compile and run unit tests on everything). The last thing I did was to add class comments to the top of each file to describe the classes and what their responsibilities are in the database.

Epilogue:

Throughout the process, I used a private GitHub repository to track my progress and backup the files. The following is a screenshot of the latest commits (39 commits in total).



In retrospect, there is a significant amount of code repeated in Table.java and Database.java. Since these classes are similar in many ways, I thought that perhaps as future work I could implement an interface to share methods between the two. But for now, the code all behaves as expected.