Database System Implementation CSCI 421 Group Project Phase 1

v2.0

1 Phase Description

This is the first phase of the semester long group project.

In this phase you will be implementing a Storage Manager and the basics of a Catalog.

There are a few basic rules when implementing this phase:

- Data must be stored to file system as bytes, not text. Opening the data files in a text editor should result in mostly unreadable text. Storing data as text will result in a maximum grade of 50%.
- You can assume the database location path exists and is accessible.
- You must implement and use a page buffer.

2 Phase Layout

All code related to the storage manager, both created and provided, must be in a class called `StorageManager`; additional files are fine, but the required functions must me in this Java class. A abstract class called `AStorageManager` has been provided. You must use and implement the functionality in this class. DO NOT change anything in this class.

When submitting you will only submit your created code in zip file, `phase1.zip`;. Maintain any file structure.

You may make additional files as needed by your implementation to help with organization/clarity.

Failure to follow this structure will result in a zero on this phase. As well as changing an provided files that state **DO NOT CHANGE**.

3 Basic Project Details

This section will outline some basic terms about the project and this phase.

- **Database Location**: the absolute path to the directory to store the database in; including the trailing slash. This includes any page files and database stores.
  Example: `/home/scj/myDB`
- **Table**: a collection of records. The Java data structure of a table in the storage manager is an ArrayList of Records. The user of the table must convert to the proper data types.
- **Record**: individual entry, tuple, in a table. The Java data structure of a ArrayList of Objects. The user of the record must convert to the proper data types.

- **Page**: a limited size data store on file system. Data will be stored to files on the file system. These are called pages. Pages have a fixed size. A page can contain multiple data rows. A table can span multiple pages. A page stored on hardware should not be readable in a text editor; it should be binary data.
  Pages can have different number if records based on the size of each of the records. Not all records will be the same size; records with Varchars cause this scenario.
  Record can contain any combination of the following data types:
  - Integer (4 bytes)
  - Double (8 bytes)
  - Boolean (1 byte)
  - Char(N) (N times 2 bytes)
  - Varchar(N) (up to N times 2 bytes plus 2 bytes)

  More details below.
  The layout of the pages on hardware should give no indication of what is stored in them and which pages belong to which table/index.
  Page files will be referenced by an integer value. They will be places in a directory call `pages` in the database location.
  You do not have to reuse page numbers.
  Records cannot span multiple pages. Assume a page size will be provided that will allow at least one complete record to be stored.
- **Page Buffer**: an in-memory buffer to store recently used pages. This has a size limit. As you read in pages they will be stored in this buffer. When the buffer is full, it will write out the least recently used page to the file system to make room for the new page. Pages in the buffer will be modified in-memory only and written to the file system when making room for new pages or the buffer purge command is called. If the system is closed before either event happens the changes to a page in the buffer are lost. Pages can only be accessed if they are first loaded into the buffer.
  The goal of the buffer is to reduce the reads/writes to the file system.
- **Primary Key**: Primary keys will be limited to one attribute. The storage manager will store the data in primary key order. The primary key can be any of the attributes in the row; not just the first.
- **Data Types**: The storage manager only needs to know the data types for comparing primary key values and reading/writing data to hardware. The data types cannot be stored in the pages; they must be stored in the catalog and only in the catalog.
  There are 5 data types in this project:
  - `Integer`: the standard Java integer.
  - `Double`: the standard Java double.
  - `Boolean`: the standard Java boolean.
  - `Char(x)`: the standard Java String. This is a fixed-size array of length x. It must be padded to maintain the length when stored. Padding must be removed when returned from storage.
  - `Varchar(x)`: the standard Java string. This is a variable-size array of maximum length x. This is not padded when stored.

Below will outline the functionality of each function. Additional details can be found in the

documentation in the provided class files.

You can assume one instance of your database will be active during a single execution of the program.

4 `ACatalog` Class

This section will outline the details of the functions in the `ACatalog` abstract class. You must implement each of the functions as outlined in this document. The implementation will be placed in `Catalog` class.

DO NOT CHANGE ANYTHING IN THIS FILE

Not all functionality in this class will need to be implemented in this phase. Stub out any unneeded functions when creating your `Catalog` class.

4.1 `catalog` static instance variable

This will store the active/current catalog the database is using. Only one catalog can be active at a time.

4.2 `getCatalog` function

This will return the active/current instance of the catalog that the database is using.

4.3 `createCatalog` function

This will create a new `Catalog` instance.

Params:

- `location`: the location of the database to create a catalog for.
- `pageSize`: the page size for the database.
- `pageBufferSize`: the size of the page buffer for the database. This is the maximum number of pages the buffer can hold before it must write pages to physical storage.

This function will call the `Catalog` constructor and set this new catalog to the active catalog.

The `Catalog` constructor will do one of two things:

- If there is a database at the provided location, it will read in the saved metadata for that database's catalog and return that catalog. It will ignore the provided page size and use the stored page size.
- If there is no database at the provided location, it will create a completely new catalog using the provided information.

4.4 `getX` functions

These are getters for the referenced variables in the catalog.

4.5 `containsTable` function

This will return a true/false response; true if a table with the provided name exists in the catalog, false otherwise.

4.6 `addTable` function

This will create. store, and return an instance of the `ITable` interface (details below).

Params:

- `tableName`: the name of the table to add. Table names are unique, adding a table with a duplicate name will result in an error.
- `attributes`: An array list of Attributes (details below) for this table. Attribute names are unique within a table, duplicate attribute names will result in an error.
- `primaryKey`: The Attribute instance that is to be used as the primary key for this table. This attribute must be in the array list of attributes, if not this will result in an error.

You must create and return an instance of an `ITable`; which will be an instance of a `Table` (detail below). How this data is stored in the catalog is up to your design, but it must be stored and persist between runs.

If an error occurs it will print a useful error message to `System.err` and return null.

4.7 `dropTable` function

This function will delete the table with the provided name. This will remove all pages and information stored about the table.

Will return true upon successful drop. If an error occurs it will print a useful error message to `System.err` and return false.

4.8 `alterTable` function

This function will be implemented in a later phase.

4.9 `clearTable(String tableName)` function

This function will clear all of the data stored in the table with the provided name. This includes clearing any indices related to this table (later phase). This does not touch any metadata related to this table.

It must notify the Storage Manager to clear the data as well.

4.10 `addIndex` and `dropIndex` functions

These functions will be implemented in a later phase.

4.11 `saveToDisk` function

This function will save the catalog to disk. How the catalog is saved to disk is up to your team, but it cannot be saved as text.

The purpose of this save is to allow the database to be restarted at a later date and still maintain its prior schema.

This will include storing this such as index metadata, table metadata, page size, etc.

Will return true upon successful save. If an error occurs it will print a useful error message to `System.err` and return false.

5 `Attribute` Record

This section will outline the details of the variables and functions of the `Attribute` record. This will used to store metadata about table attributes.

This is a provided file. **DO NOT CHANGE ANYTHING IN THIS FILE**

Variables:

- `attributeName`: the name given to this attribute.
- `attributeType`: the type of the attribute. It must be one of the 5 types outlined above.

Functions:

- `getX()`: getters for the above variables.
- `toString()`: used to pretty print the record.
- `equals(Object obj)`: overridden Object equals. Two attributes are considered equal if their names are the same.

6 `ForeignKey` Record

This section will outline the details of the variables and functions of the `ForeignKey` record. This will used to store metadata about table foreign keys.

This is a provided file. **DO NOT CHANGE ANYTHING IN THIS FILE**

Variables:

- `attrName`: the name of the referring attribute.
- `refTableName`: the name referenced table.
- `refAttribute`: the name of the referenced attribute in the referenced table.

Functions:

- `getX()`: getters for the above variables.
- `toString()`: used to pretty print the record
- `equals(Object obj)`: overridden Object equals. Two foreign keys are considered equal if variables are all equal.

This will be used in a later phase.

7 `ITable` Interface

This section will outline the details of the variables and functions of the `ITable` interface. This will used to store metadata about tables. You must implement each of the functions as outlined in this document. The implementation will be placed in `Table` class.

This is a provided file. **DO NOT CHANGE ANYTHING IN THIS FILE**

7.1 `getX()` and `setX(Y)` functions

Getters/Setters for various class variables.

7.2 `getAttrByName(String name)` function

This will return the `Attribute` instance from this table with the provided name. If not attribute with that name exists it will return null.

7.3 `addAttribute(String name, String type)` function

This will create and add an `Attribute` with the provided name and type to the table. Attributes will be added to the end of the array of attributes. This will also tell the storage manager to add the attribute to the data stored in the table.

An error will be reported if an attribute with the provided name already exists in the table.

Will return true if successful, false otherwise.

This will be implemented in a later phase.

7.4 `dropAttribute(String name)` function

This will drop the `Attribute` with the provided name from the table. Attributes remaining will be shifted down after the drop. This will also tell the storage manager to drop the attribute from the data stored in the table.

An error will be reported if an attribute with the provided name does not exist in the table.

Will return true if successful, false otherwise.

This will be implemented in a later phase.

7.5 `addForeignKey(ForeignKey fk)` function

This will add the provided foreign key to the table.

An error will be reported if the same foreign key already exists in the table.

Will return true if successful, false otherwise.

This will be implemented in a later phase.

8 `AStorageManager` Class

This section will outline the details of the functions in the `AStorageManager` abstract class. You must implement each of the functions as outlined in this document. The implementation will be placed in `StorageManager` class.

This is a provided file. **DO NOT CHANGE ANYTHING IN THIS FILE**

8.1 `sm` static instance variable

This will store the active/current storage manager the database is using. Only one storage manager can be active at a time.

8.2 `getStorageManager` function

This will return the active/current instance of the storage manager that the database is using.

8.3 `createStorageManager` function

This will create a new `StorageManager` instance.

This function will call the `StorageManager` constructor and set this new storage manger to the active storage manager. It will create a new page buffer.

8.4 `clearTableData(ITable table)` function

This function will clear all of the underlying data stored in this table. Includes all pages; on physical disk and in the buffer.

8.5 `getRecord(ITable table, Object pkValue` function

This function will get a single record from the provided table's data that have the provided primary key value; only one such record should exist.

It will return the record with the matching primary key value; null otherwise.

The caller is responsible for converting the attribute values to the correct data types.

8.6 `getRecords(ITable table)` function

This function will return all of the records for the provided table. The data will be returned in primary key order (the order that the data is stored on hardware).

The caller is responsible for converting the attribute values to the correct data types.

Note: There is a difference between no records and errors. No records will return an empty array list. Error will return null.

Note: This can be a very large array depending on the amount of data stored.

8.7 `insertRecord(ITable table, ArrayList<Object> record)` function

This function will insert the provided record into the provided table.

Params:

- `table`: an instance of an `ITable`. This is the metadata about the table that the record is to be inserted in.
- `record`: this in an ArrayList of Objects. These object make up the data items of this record.

This is the process for inserting a record (this will change in a later phase when indexing occurs):

```
if there are no pages for this table:
    make a new page for the table
    add this entry to the page
    end

Read each table page in order:
    iterate the records in page:
        if the record belong before the current record:
            insert the record before it

    if the page becomes overfull:
        split the page
        end

If the record does not get inserted:
    insert it into the last page
    if page becomes overfull:
        split the page
    end

splitting a page:
    make a new page
    remove half the items from the first page
    add the items to the second page
```

Some how you will have to track the ordering of the pages. The will NOT be in number order. For instance:

- We insert 5 items into page 1
- When inserting item 6 a page split occurs
  - page 1 gets 3 items
  - page 2 gets 3 items
  - the page order becomes 1, 2
- We insert 3 more items that all belong in page 1, split occurs
  - page 1 gets 3 items
  - page 3 gets 3 items

- the page order becomes 1, 3, 2

This example above assumes a fixed size record for clarity purposes; this will not always be the case.

The number of records in a page will be determine by the database page size and the size of each record. Not all records are the same size.

Example:

A table was created with integer, double, char(3), varchar(10). We insert the following 2 records:

```
(12,13.5,"foo", "h")
(1,100.1, "hi", "hello foo")
```

Each integer is 4 bytes, double is 8 bytes. These are fixed sizes.

The char string is also a fixed size of 6 bytes (3 chars time 2 bytes per char). Even though "hi" is two characters it will need to be padded to three with a nul-character when stored to hardware.

The varchar string is where the variability comes in. In the first record it will be stored in 4 bytes (1 char times 2 bytes plus 2 bytes). The second record will be 20 bytes (9 char times 2 bytes plus 2 bytes).

What is the extra 2 bytes? It is an integer used to store the number of characters stored to hardware for this varchar.

Example table page structure:

```
<numRecords>
<row1>
<row2>
....
<rowN>
```

Note: everything will be stored as binary and there will be no newlines. They were added here for clarity.

`numRecords` is the number of records stored in this page. This is important because each page will have a different number.

Rows will be just a combination of data items for that row. Data types cannot be stored in the page; the catalog is used for this.

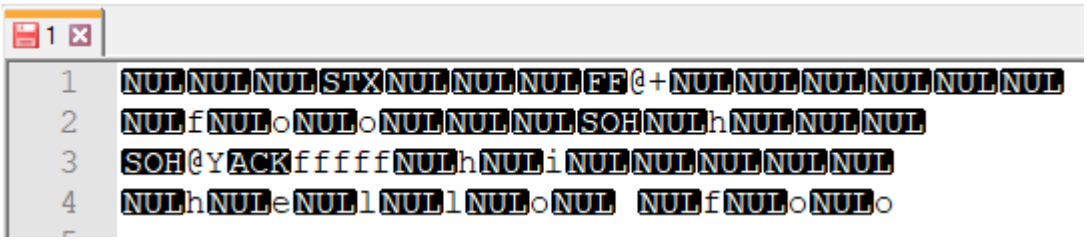
Example row using the two rows from the prior example:

```
Integer(2) Integer(12) Double(13.5) Char(f) Char(o) Char(o) Integer(1)
Char(h) Integer(1) Double(100.1) Char(h) Char(i) Char(null) Integer(9)
Char(h) Char(e) Char(l) Char(l) Char(o) Char( ) Char(f) Char(o) Char(o)
```

Spaces and newlines added for clarity.

Without any additional info there is not way to know where one record ends and another begins. This becomes even more difficult then they are written as binary.

When opened in a text editor you will see the following:



Newlines were added for clarity. This was actually all on one line.

Things to note:

- There are many "unreadable" characters
- You can read some of the character data in it true form. This is due to how Java writes character as binary. This is okay.
- all other non-character data is under-readable.

Storing items such as integer, booleans, and doubles as text is not allowed and will result in heavy penalties.

You must store the attribute values in a row in the order they are given. No moving the primary key to the front, strings to the end, etc.

Note: these inserts to do get stored to hardware until they are written out of the page buffer.

Above is just an example of how to do this. You can uses your own page format if you wish, but you are not allowed to store Java Objects to the pages (i.e. using serializable classes).

For example this is not allowed:

```
import java.io.Serializable

public class Page implements Serializable{
    ....
}

Page p = new Page(....);
FileOutputStream fileOut = new FileOutputStream(filepath);
ObjectOutputStream objectOut = new ObjectOutputStream(fileOut);
objectOut.writeObject(p);
```

There will be scenarios where some attribute values will be `null`. You will have to be creative on how to store this data in the page.You can add additional data to the page to help with this, but that data must count towards the page size.

8.8 `deleteRecord(ITable table, Object primaryKey)` function

This function will delete the record with the provided primary key value from the table's data.

Params:

- `table`: an instance of an `ITable`. This is the metadata about the table that the record is to be deleted from.]
- `primaryKey`: the value of the primary key for the record to be deleted. The metadata for this key will be obtained from the catalog's metadata for this table.

The process for deleting a record (this will change in a later phase when indexing occurs):

```
read each table page in order:
    iterate the records in the page
        if the current record's pk equals the provided pk:
            delete the record
            move all other records up to cover the empty space
            update the record count in the page

            if the page become empty:
                remove its reference from the table's metadata

        if the current record's pk is greater than the provided pk:
            stop the record does not exist
```

Deleting of empty pages from hardware should not occur until the buffer writes the page to hardware.

8.9 `updateRecord(ITable table, ... )` function

This function will update the provided record with the data of the new record. This can cause the record to move to a new page if:

- the primary key changes
- the size increases causing a page split. This can also cause other records to move as well.

Params:

- `table`: an instance of an `ITable`. This is the metadata about the table that the record is to be updated in.
- `oldRecord`: the data values of the record before the changes.
- `newRecord`: the data values of the record after the changes

There are many ways to do this; some will be easier than others. This will be left to your design choice.

8.10 `purgePageBuffer()` function

This function will force a purge of the page buffer. Purging the page buffer will force all of the pages in the buffer to be written to hardware. This will basically empty the buffer.

This will need to be called to properly and safely shutdown the database. Failing to do so will result in inconsistencies in your database.

9 Other Classes Not Mentioned

In the provided code there are other class files not discussed here. They are either testing classes for each phase, or needed for later phases.

Testers are provided for your convenience. Provided testers may not test all possible cases; passing the provided testers does not guarantee a good score. You must conduct your own through testing.

Your project MUST work with the provided testers. Failure to do so will result in heavy penalties.

10 Project Constraints

This section outlines details about any project constraints or limitations.

Constraints/Limitations:

- Everything, except for the values in Strings (char and varchar), is case-insensitive; like SQL. Anything in double quotes is to be considered a String.
- You must use a Java and the requirements provided.
- Your project must run and compile on the CS Linux machines.
- Submit only your source files in the required file structure. Do not submit and IDE directories or projects.
- Your code must run with any provided tests cases/code. Failure to do so will result heavy penalties.
- Any Java errors, such as file reading/writing errors, should be handled with a useful error message printed to `System.err`. The user can determine how to handle the error based on the return of the functions.

11 Grading

Your implementation will be grading according to the following:

- (25%) Catalog functions
- (25%) Table functions
- (50%) StorageManager functions

Penalties:

- (-50% of points earned) Data is written as text not binary data.
- (up to -50% of points earned) Does not work with any provided testing files.
- (up to -100% of points earned) Does not compile on CS machines.

Penalties will be based on severity and fix-ability. The longer it takes the grader to fix the issues the higher the penalty.

Examples:

- Code does not compile due to missing semicolon: -5%

- Code does not compile/run with testers due to missing functions or un-stubbed functions: -10%
- Multiple syntax errors causing issues compiling issues and takes over 30 mins to fix: -100%
- Multiple Crashes with provided testers that take an extended time to fix: -50%

These are just examples.The basic idea is "Longer to fix, more points lost. Eventually give up, assign a zero."

12 Submission

Zip any code that your group wrote in a file called `phase1.zip`. Maintain any required package structure. Do NOT submit any provided code. The grader will use their own versions.

Submit the zip file to the Phase 1 Assignment box on myCourses. No emailed submissions will be accepted. If it does not make it in the proper box it will not be graded.

The last submission will be graded.

There will be a 48 hour late window. During this late window no questions will be answered by the instructor. No submissions will be accepted after this late window.

13 Tips and Tricks

Here are a few tips and tricks to help you:

- START EARLY. Some of this can be tricky and you will have questions.
- Come to the in class project sessions. Instructor will be there to help.
- Design, Design, Design. "Every hour in design can save 8 hours of coding." is a common saying.
- look into Java's `DataOutputStream` and `DataInputStream`. They will make writing binary so much easier.
- Things like `Integer.bytes` might be useful.
- If you follow the interfaces and use them as intended, the work can be divided up.
- Design smaller helper functions to make life easier.
- You can add additional classes.
- Read the entire document and the provided code commenting.
- Looking ahead at future phases might be helpful.