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 some basic SQL features.

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

The structure of the Storage Manager and SQL parser is up to your group.

The project will execute in one of three ways based on the language you choose:

- Python: `python main.py <db_loc> <page_size> <buffer_size>`
- Java: `java Main <db_loc> <page_size> <buffer_size>`
- C: `./main <db_loc> <page_size> <buffer_size>`

Please include a `README` outlining the structure and building instructions for your project.

3 Basic Project Details

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

- **Database Location** `<db_loc>`: 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. A table will be stored on hardware as a single file. This file will be stored as binary collection of pages.
- **Catalog**: A data structure used to store the schema of the database.
- **Record**: individual entry, tuple, in a table.
- **Page**: a limited size (`<page_size>`, data store on file system. Tables will be stored as a file on the file system. These files will be made up of pages. Pages have a fixed size. A page can contain multiple data rows. A table can span multiple pages. A table 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
  - Double
  - Boolean
  - Char(N)
  - Varchar(N) (up to N chars)

  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.

  Table files will be referenced by an integer value. They will be places in a directory call `tables` in the database location.

  You do not have to reuse table 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. `<buffer_size>`. 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 a single 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 integer.
  - `Double`: the standard double.
  - `Boolean`: the standard boolean.
  - `Char(x)`: the standard 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 string. This is a variable-size array of maximum length x. This is not padded when stored.

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

4 Project Requirements

This section will outline the details of the project functionality. Your project must implement the functionality as outlined here.

4.1 Main Program

Your main program will be the entry point of the system. Upon starting of the system it will:

- Look at the provided database location and determine if there is a database at that location.
  - If there is a database present it will restart that database; ignoring the provided page size. It will use the new buffer size.
  - If there is no database present it will create a new database at that location using the provided page and buffer size.
- Once the database has been created/restarted, it will go into a loop asking for commands from the user.
- Once the quit command has been executed the system will safely shutdown the system.

Commands for each phase will be outlined.

4.2 Catalog

Your project is required to make use of a catalog to store the schema of the database. You can store this catalog in what every data structure you wish, but it must be stored as binary on hardware, and must be saved between runs.

4.3 Storage Manager

You project must make use of a storage manager to communicate with hardware. The SQL parser cannot directly access hardware to obtain data. You will need functionality for:

- getting a record by primary key
- getting a page by table and page number
- getting all records for a given table number
- inserting a record into a given table
- deleting a record by primary key from a given table
- updating a record by primary key in a given table

Pieces of this functionality will be implemented in different phases.

4.3.1 Inserting a record

This section will outline how the process of inserting a record in a table should occur.

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 file for the table
    add this entry to a new page
    insert the page into the table file
    end
```

```
Read each table page in order from the table file:
    iterate the records in page:
        if the record belongs before the current record:
            insert the record before it

    if the current page becomes overfull:
        split the page
        end

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

splitting a page:
    make a new page
    remove half the items from the current page
    add the items to the new page
    insert the new page after the current page in the table file
```

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.

There are many ways to structure the table file, pages, and records. How you store the data in each structure is up to your implementation but they must have this basic structure:

Table file structure:

```
<numPages>
<page1>
<page2>
....
<pageN>
```

Table page structure:

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

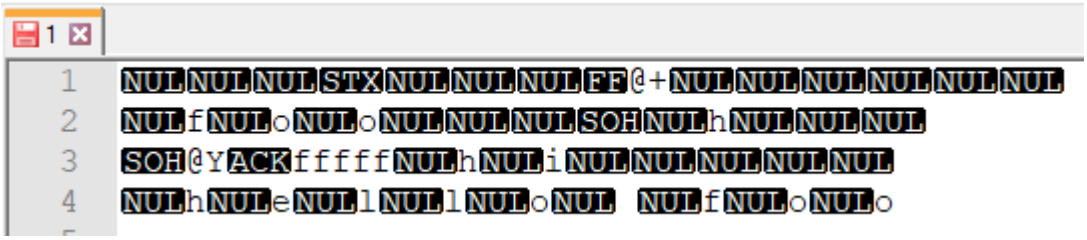
Note: everything will be stored as binary and there will be no newlines. They were added here for clarity.

`numRecords/pages` is the number of records/pages stored. This is important because each table/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.

Without any additional info there is no way to know where one record/page 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 some languages 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/Python 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.

No two records can have the same primary key. If this occurs, the insert should report a duplicate key error and not insert the record.

4.4 Page Buffer

Your storage manager must implement a page buffer. The page buffer will have a fixed size (number of pages it can hold).

A page must be read into the buffer from hardware before it can be accessed/modified.

A page will only be written back to hardware when either the system is shut down or more room is needed in the buffer for more pages. The buffer will use a LRU (least recently used) method for determine the page to write to hardware first from the buffer.

All pages in the buffer will be written to hardware when the system is shutdown.

A page with no changes does not have to be rewritten to hardware.

5 Query Processor

The query processor will be used to parse SQL-style commands entered by the user. This section will outline the commands required for this phase. Additional commands and changes to these commands will happen in later phases.

5.1 Create table command

This command will be used to create the schema for a table. This schema will be added to the catalog. This schema will be used by the system to store/access/update/delete data in the created table.

The structure of the command for this phase:

```
create table <name>(
    <attr_name1> <attr_type1> primarykey,
    <attr_name2> <attr_type2>,
    ....
    <attr_nameN> <attr_typeN>
);
```

- `<name>` is the name of the table. Table names are unique in the system.
- `<attr_name>` is the name of the attribute. Attribute names are unique within a table.
- `<attr_type>` is the type of the attribute. These types are outlined above.
- `primarykey` is the attribute that is the primary key of the table. The table can have only one primary key and it does not have to be the first attribute.

Newline and spacing added for clarity but are not required.

Examples:

```
create table foo( num integer primarykey );
create table foo( age char(10),
    num integer primarykey );
```

The command should report "SUCCESS" if the table is successfully created. "ERROR" and a reason for the error if the table is not created.

5.2 Select command

This command will be used to access data in tables. In this phase it will be a very basic command. In later phases, more in-depth functionality will be added.

Phase 1 select:

```
select *
from <name>;
```

- `<name>` is the name of the table. Table names are unique in the system.

This will display all of the data in the table in an easy to read format. This includes column names.

An error will be reported if the table does not exist.

5.3 Insert command

These statements will look very similar to SQL, but the format is going to be changed to help reduce parsing complexity.

Be aware just like in SQL, insert will insert a new tuple and not update an existing one. If it tries to insert a tuple with the same primary key values as one that exists it will report an error and stop adding tuples. Any tuples already added will remain. Any tuple remaining to be added will not be added.

The typical format:

```
insert into <name> values <tuples>;
```

Lets look at each part:

- `insert into`: All DML statements that start with this will be considered to be trying to insert data into a table. Both are considered keywords.
- <name>: is the name of the table to insert into. All table names are unique.
- `values` is considered a keyword.
- <tuples>: A space separated list of tuples. A tuple is in the form:

  ```
  ( v1 ... vN )
  ```

  Tuple values will be inserted in the order that that table attributes were created. The spaces/newlines after the commas are not required and added for clarity/readability.

Examples:

```
insert into foo values (1 "foo" true 2.1);
insert into foo values (1 "foo bar" true 2.1),
                       (3"baz" true 4.14),
                       (2"bar" false 5.2),
                       (5 "true" true null);
```

All primary key, data types, and not null constraints must be validated. Primary key values can never be null.

Upon error the insertion process will stop. Any items inserted before the error will still be inserted.

Note: `null` is a special value to represent there is no value for that attribute. Also note that spaces are allowed in strings.

5.4 Display schema command

This command will display the catalog of the database in an easy to read format. For this phase it will just display:

- database location
- page size
- buffer size
- table schema

The command will be `display schema;`.

5.5 Display info command

This command will display the information about a table in an easy to read format. Tt will display:

- table name
- table schema
- number of pages
- number of records

The command will be `display info <name>;`.

6 Testing

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

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

7 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/Python/C 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 errors, such as file reading/writing errors, should be handled with a useful error message printed to the language's error stream. The user can determine how to handle the error based on the return of the functions.

8 Grading

Your implementation will be grading according to the following:

- (25%) Inserting records functionality
- (50%) Storage Manager functionality
- (25%) Create table/display functionality

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."

9 Submission

Zip any code that your group wrote in a file called `phase1.zip`. Maintain any required package structure.

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

10 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.
- Looking ahead at future phases might be helpful.