Database System Implementation CSCI 421 Group Project Phase 2

v1.0

1 Phase Description

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

In this phase you will be implementing a DDL parser. The tool needed to create the database schema.

There are a few basic rules when implementing this phase:

- You must use the storage manager you created in phase 1. You will not use all of its features in this phase.
- You can assume the database location path exists and is accessible.
- Recall database schema is stored in a catalog. You must use your implemented catalog. Schema created in this phase must be stored in the catalog.

2 Phase Layout

You will be implementing the `DDLParser`, partial `DMLParser` and partial `Database` classes. These files have been provided.

When submitting you will only submit your created code in a zip file, `phase2.zip`. This includes all code from phase 1; as it will be needed for this phase.

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

3 `Database` Class

This section will outline the details of the functions in the `Database` class. You must implement each of these functions as outlined in this document. The implementation will be placed in the provided `Database` class file.

3.1 `executeStatement` function

This function will be used when executing database statements that do not return anything. For instance schema creation/modification, insert, delete, and update statements. It will need to determine the different types and send them to the proper parser; DDL or DML.

3.2 `executeQuery` function

This function will be used when executing database queries that return tables of data.

In this phase you will not be implementing this function. Just leave it stubbed out.

3.3 `terminateDatabase` function

This function will be used to safely shutdown the database. It will store any needed data needed to restart the database to physical hardware; this includes safely shutting down the storage manager.

3.4 `main` function

This function will be used to start/restart a database. It will:

- create/restore the catalog.
- create/restore the storage manger.

It will then go into a loop asking for SQL statements/queries. It will call the proper parsing function based on statement or query. It will process multiple input lines as a single statement/query until it gets a semi-colon; your parsers should consider a newlines and tabs as the same as a space. To exit the program the command `quit;` will be entered. After each non-quit statement it will report `ERROR` or `SUCCESS` based on the results of the statement.

Queries will display the results of the query in an easily readable/understandable form. (Later phase)

The program will be ran as:

```
java Database <db_loc> <page_size> <buffer_size>
```

4 `DDLParser` Class

This section will outline the details of the functions in the `DDLParser`class file. You must implement each of the functions as outlined in this document.

4.1 `parseDDLStatement` function

This function will handle the parsing of ddl statements. It will return true on success and false on error. This function is responsible for outputting a reasonable error message to `System.err` upon error.

5 DDL Statements

In this phase there are only three types of DDL statments:

- `create table`: used to create a table. This will be very similar to SQL syntax but with reduced complexity.
- `drop table`: used to drop a table from the database; including its data.
- `alter table`: this will be used to add/remove columns from a table.

Each of these will be outlined below.

Each statement can be one line or multiple lines. The new line character is to be considered the same as a space. All statements will end with a semi-colon. Multiple spaces are to be considered a single space; but where spaces are shown below at least one space will exist there.

5.1 `create table` statements

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

The typical format:

```
create table <name>(
    <a_name> <a_type> <constraint_1>,
    primarykey( <a_1> ),
    foreignkey( <a_1> ) references <r_name>( <r_1> )
);
```

Lets look at each part:

- `create table`: All DDL statements that start with this will be considered to be trying to creating a table. Both to be considered keywords.
- <name>: is the name of the table to create. All table names are unique.
- <a_name> <a_type> <constraint_1>:
  defines a new attribute with provided name, type, and constraints.
- Attribute types can only be integer, double, boolean, char(x), and varchar(x); as outlined in phase 1.
- Constraints: There are two types of constraints that can be added to a single variable:
  - `notnull`: The value of this attribute cannot be null. Keyword.
  - `primarykey`: This attribute becomes the single attribute primary key for the table. A table can only have one primary key. Any attempt to make another will result in an error. Keyword.
- `primarykey( <a_1> )`: This will make a primary key for the table using a single attribute. A table can only have one primary key. Any attempt to make another will result in an error.
- `foreignkey( <a_1> ) references <r_name>( <r_1> )`: this will make a foreign key to the table with the `r_name`. <a_1> is the name of an attribute in the table being created and will match to <r_1> in the referenced table. The referenced table must exist and the data types of the attributes must match. `foreignkey` and `references` are key words.

Names can start with a alpha-character and contain alphanumeric characters.

Primary keys are assumed to be automatically not null.

Examples:

```
CREATE TABLE BAZZLE( baz double PRIMARYKEY );
```

```
create table foo(
    baz integer,
    bar Double notnull,
    primarykey( bar ),
    foreignkey( bar ) references bazzle( baz )
);
```

Note: case does not matter, except for in string literals.

5.2 `drop table` statements

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

The typical format:

`drop table <name>;`

Lets look at each part:

- `drop table`: All DDL statements that start with this will be considered to be trying to drop a table. Both are considered to be keywords.
- `<name>`: is the name of the table to drop. All table names are unique.

Example:

```
drop table foo;
```

5.3 `alter table` statements

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

The typical formats:

```
alter table <name> drop <a_name>;
alter table <name> add <a_name> <a_type>;
alter table <name> add <a_name> <a_type> default <value>;
```

Lets look at each part:

- `alter table`: All DDL statements that start with this will be considered to be trying to alter a table. Both are considered to be keys words.
- `<name>`: is the name of the table to alter. All table names are unique.
- `drop <a_name>` version: will remove the attribute with the given name from the table; including its data. `drop` is a keyword.
- `<name> add <a_name> <a_type>` version: will add an attribute with the given name and data type to the table; as long as an attribute with that name does not exist already. It will then will add a null value for that attribute to all existing tuples in the database. `add` is a keyword.
- `<name> add <a_name> <a_type> default <value>`: version: will add an attribute with the given name and data type to the table; as long as an attribute with that name does not exist already. It will then will add the default value for that attribute to all

existing tuples in the database. The data type of the value must match that of the attribute, or its an error. `default` is a keyword.

Any attribute being dropped cannot be the primary key. If the attribute being drop is part of any foreign key constraint, that constraint must be dropped as well.

Examples:

```
alter table foo drop bar;
alter table foo add gar double;
alter table foo add far double default 10.1;
alter table foo add zar varchar(20) default "hello world";
```

Note: altering a table is not just as easy as removing/adding an attribute. For example, things like number of records per page need to be modified.

6 DML Statements

In this phase there are three types of DML statements:

- insert,
- update,
- and delete.

Each of these will be outlined below.

- Each statement can be one line or multiple lines.
- The newline character is to be considered the same as a space.
- Multiple spaces are to be considered a single space; but where spaces are shown below at least one space will exist there.
- All statements will end with a semi-colon.

In this phase you will be implementing the `parseDMLStatement` function in the `DMLParser` class. It will handle the statements outlined below.

6.1 `insert` statements

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 comma 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 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", true, 2.1),
                       (3,"baz", true, 4.14),
                       (2,"bar", false,5.2),
                       (5, "true", true, null);
```

All primary key, foreign 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.

6.2 `delete` statements

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

The typical format:

```
delete from <name> where <condition>
```

Lets look at each part:

- delete from: All DML statements that start with this will be considered to be trying to delete data from a table. They both are to be considered keywords.
- <name>: is the name of the table to delete from. All table names are unique.
- where <condition>: A condition where a tuple should deleted. If this evaluates to true the tuple is remove; otherwise it remains. See below for evaluating conditionals. If there is no where clause it is considered to be a where true and all tuples get deleted. where is considered a keyword.

Example:

```
delete from foo;
delete from foo where bar = 10;
delete from foo where bar > 10 and foo = "baz";
delete from foo where bar != bazzle;
```

If a value being deleted is referred to by another table via a foreign key the delete will not happen and an error will be reported.

Upon error the deletion process will stop. Any items deleted before the error will still be deleted.

6.3 `update` statements

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

The typical format:

```
update <name>
set <column_1> = <value>
where <condition>;
```

Lets look at each part:

- `update`: All DML statements that start with this will be considered to be trying to update data in a table. Keyword.
- <`name`>: is the name of the table to update in. All table names are unique.
- `set` <`column_1`> `=` <`value`> Sets the column to the provided values. `set` is a keyword.
- <`value`>: attribute name, constant value, or a mathematical operation.
- `where` <`condition`>: A condition where a tuple should updated. If this evaluates to true the tuple is updated; otherwise it remains the same. See below for evaluating conditionals. If there is no `where` clause it is considered to be a `where true` and all tuples get updated.

Example:

```
update foo set bar = 5 where baz < 3.2;
update foo set bar = 1.1 where a = "foo" and bar > 2;
```

Records should be changed one at a time. If an error occurs with a tuple update then the update stops. All changes prior to the error are still valid.

7 Conditionals

This section will outline the process of evaluating conditionals in the `where` clause.

Conditionals can be a single relational operation or a list of relational operators separated by `and` / `or` operators. `and` / `or` follow standard computer science definitions:

- <`a`> `and` <`b`>: only true if both `a` and `b` are true.
- <`a`> `or` <`b`>: only true if either `a` or `b` are true.

`and` has a higher precedence than `or`. Items of the same precedence will be evaluated from left to right.

Example:

```
x > 0 and y < 3 or b = 5 and c = 2
```

You would first evaluate `x > 0 and y < 3`, then `b = 5 and c = 2`. Then `or` their results.

This project will only support a subset of the relational operators of SQL:

- $=$: if the two values are equal.

- `>` : greater than.
- `<` : less than.
- `>=` : greater than or equal to.
- `<=` : less than or equal to.
- `!=` : if the two values are not equal.

Relational operators will return true / false values. You can use the Java built in comparison functionality to compare values. The left side of a relational operator must be an attribute name; it will be replaced with its actual value at evaluation. The right side must be an attribute name or a constant value; no mathematics. The data types must be the same on both sides on the comparison.

Examples:

```
foo > 123
foo < "foo"
foo > 123 and baz < "foo"
foo > 123 or baz < "foo bar"
foo > 123 or baz < "foo" and bar = 2.1
foo = true
foo = baz
```

Strings, unless quoted or true/false, are to be considered column names.

8 Mathematics

This section will outline the basic mathematics allowed. The operators will work how Java handles similar operations. Assume one mathematical operator per expression. Mathematics are only allowed in a `set` operation of an `update` statement.

Strings, unless quoted, are to be considered column names.

- `+` : basic addition.
- `-` : basic subtraction.
- `*` : basic multiplication.
- `/` : basic division.

When assigning a mathematical result into a column data types of integer can be converted in doubles and vice versa (truncates). Mathematics are only allowed on integer and double values.

Examples:

```
foo + 5
foo * 2.1
bar - foo
1 - bar
```

9 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. String literals will be quoted and can contain spaces. Example: `"foo bar"` is a string literal. `foo` is a name; not a string literal. The quotes do not get stored in the database. They must be added when printing the data.
- 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.
- Words such as `Integer`, `create`, `table`, `drop`, etc are considered key words and cannot be used in any attribute or table name.
- Data must be checked for validity. Examples:
  - type matching
  - not nulls
  - primary keys
  - foreign keys
- Any changes to data in the database must keep the database consistent. This means that all not null, foreign keys, and primary keys must be observed.

10 Grading

Your implementation will be graded according to the following:

- (20%) Database functionality
  - (7.5%) `execute statement` functionality
  - (7.5%) `terminate database` functionality
  - (5%) `main` functionality
- (35%) DDL Parser functionality
  - (20%) `create table` functionality
  - (5%) `drop table` functionality
  - (10%) `alter table` functionality
  - (45%) DML Parser statements functionality
    - (10%) `insert` functionality
    - (15%) `delete` functionality
    - (15%) `update` functionality

Penalties:

- (-50% of points earned) Data is written as text not binary data.
- (-25% of points earned) Does not use storage manager to handle hardware access. Catalog reading/writing to hardware does not need to be handled by the storage manager.
- (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."

11 Submission

Zip any code that your group wrote in a file called `phase2.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.

12 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.
- 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.
- Use a tree structure for `where` clauses.