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 more of the Query Parser.

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

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.

2.1 `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>,
    <constraint>
);
```

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.
  - `unique`: The values for this attribute must be unique. Keyword.

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

Primary keys are assumed to be automatically not null and unique.

Examples:

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

create table foo(
    baz integer primarykey,
    bar Double notnull,
    bazzle char(10) unique notnull
);
```

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

You will need to update the functionality of the insert command from phase one to handle nulls, uniques, etc.

You will also need to update the `display` commands to show the new schema items.

2.2 `drop table` statements

These statement will look very similar to SQL, but format is going to be changed to help reduce parsing complexity. This will remove the table from the system. This includes the data and schema.

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;
```

2.3 `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.

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.

3 DML Statements

In this phase there are two asdditional types of DML statements:

- 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.

3.1 `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.

How to delete a record in the storage manager:

```
read each table page in order from the table file:
    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 file
                move all other pages up in the file
                update the page count

        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.

3.2 `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>`: a constant value.
- `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.

Updates are not as simple as just changing values. Updates can cause a record to move a page (if the primary key changes), or make page splits happen (the size of the record increases).

4 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. 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.

5 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 errors, such as file reading/writing errors, should be handled with a useful error message printed to the language error stream. 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
- Any changes to data in the database must keep the database consistent. This means that all not null, uniques, and primary keys must be observed.

6 Grading

Your implementation will be graded according to the following:

- (40%) DDL Parsing functionality
  - (10%) updated `create table` functionality
  - (10%) `drop table` functionality
  - (20%) `alter table` functionality
- (60%) DML Parser statements functionality
  - (10%) updated `insert` functionality
  - (25%) `delete` functionality
  - (25%) `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.”

7 Submission

Zip any code that your group wrote in a file called `phase2.zip`. Maintain any required code structure.

Submit the zip file to the Phase 2 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 as outlined in the syllabus. During this late window no questions will be answered by the instructor. No submissions will be accepted after this late window.

8 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.
- Read the entire document and the provided code commenting.
- Looking ahead at future phases might be helpful.
- Use a tree structure for `where` clauses. You will need this functionality for update, delete, and selection (phase 3).