CS448
Designing and Implementing a Mini Relational DBMS

Credit: 20 points

Due Date: Midnight of April 2, 2014 without any penalties.

The last day to submit the program is April 9, 2014 with 1 point penalty per class period the program is late (maximum loss of 3 points).

Project Description:

In this project, you are going to design and implement a relational database management system (DBMS). The mini relational database management system that you are going to design and implement should provide the following functionality:

- *Persistent data management:*
  Persistent data management requires that data as well as schemas in databases be retained in disks after the program terminates. Thus, any changes to data and databases schemas must eventually be committed to disks.
- *Access Control:*
  Access control involves specification and enforcement of data sharing or monopoly (exclusive access).
- *A SQL interpreter/Commands:*
  The SQL interpreter should be able to parse and execute basic SQL commands (a subset of the SQL command set), provide integrity checking, and print out appropriate error messages in case errors are involved in the SQL commands.

Specifically you are asked to:

1. Allow the creation and the deletion of schemas for the database relations. The schema will contain the name of the relation, the attributes of the relation, and the allowable range of domains for attributes (constraints) of the relation, etc.
2. Allow the insertion and the deletion of tuples.
3. Allow the update of values of the attributes of tuples.
4. Allow the retrieval of data that may involve one or more relations.
5. Deny or abort any operations that would cause the schema integrity constraints to be violated.
6. Report lexical, syntactic, or semantic errors involved in any (SQL) commands.

Project Details

A database, for this project, is defined as a collection of relations or tables and their corresponding schemas. Some commercial DBMSs store the whole database as a single huge file, which involve complex physical data management. In this project, you can store each relation in a UNIX file for easy maintenance.

A schema of a relation contains information about this relation, such as the name of the relation, names and types of attributes belonging to this relation, as described earlier. Besides, a schema can contain more dynamic information such as number of tuples in the relation, making the implementation easier. Schemas themselves in a database form a relation and can also be stored in a separate UNIX file. Although this relation can be handled (queried) as any other relations, care has to be taken for doing so. Users are not allowed to update this relation, nor can they access it under certain circumstances. So the system has to provide only limited access to this relation or simply prevent users from accessing it for security and privacy purposes. You can assume three types of users (database administrator who has access to whole schema, and user-A and user-B with limited subschemas).

Since relations are stored in UNIX files, you must commit every transaction to disk, except "read-only'' transactions. It is also required that each transaction be committed separately. For example, after every transaction that alters the state of the database, you shouldn't simply write out the entire database on the disk since this would be grossly inefficient on the existing hardware. For certain operations, you are encouraged to find a more efficient way to implement them. A smarter way of dealing with tuple insertion, for example, is to reuse the space of previously deleted tuples.

For simplicity, the DBMS needs to handle only one user at a time and you can assume that you will never have more data than you can store in main memory. Also, this project does not require the DBMS to maintain a log, and you need not worry about recovery in case of hardware or system software errors.

Schemas, Subschemas, and Access Control

A schema (relation table definition) will have a fixed structure with the following information: relation name, attribute name, attribute type, and domain constraints. An example of the schema could be:

Table 1. Domain constraints

Relation	Attribute Name	Attribute Type	Domain Constraints
test	student_name	char(15)	student_name != “”
test	student_id	int	student_id !=0
test	student_age	int	(student_age > 10) AND (student_age < 70)

Table 2. Possible types for an attribute

int	Integer
char(strlen)	Character string of length str_len
decimal	Real Number

An *attribute name* must be alphanumeric (starting with an alphabetic character), have a length of at most 256 characters, and must accept the underscore character (_).

The field *constraint* is a Boolean predicate consisting of valid attribute names, the logical operators (AND, OR), the relational operators (=, !=, >, <. >=, <=), the arithmetic operators (*, /, +, –), and constants of equivalent types. It is used mainly for integrity checking.
For example, the definitions in table 3 below are valid:

Table 3. Specification of domain constraints

Relation	Attribute Name	Attribute Type	Domain Constraints
test	s1	char(15)	s1 != “”
test	s2	decimal	(s2 * 1.0) > 10.2
test	s3	int	(s3 > 10) OR (s3 = 0 AND s3 < 5)
test	s4	char(20)	(s4 != “TST”) AND (s4 != “”) AND (s4 != “xyzzy”)
test	s5	char(3)	(s5 = “T1”) OR (s5 = “T2”) OR (s5 = “T3”)

Your DBMS should be able to handle the concept of users and subschemas. For example, if the following relation is defined:

Table 4. Example relation schema

Relation	Attribute Name	Attribute Type	Domain Constraints
students	name	char(20)	Name != “”
students	ssn	char(9)	(ssn != “”) AND (ssn != “000000000”)
students	phone	char(7)	
students	gpa	decimal	(gpa >= 0.0) AND (gpa <= 4.0)

then, an employee of the registrar’s office (or database administrator) should be able to access all fields. However a student using the database would have the following view of the relation:

Table 5. Subschema for the relation in table 4

Relation	Attribute Name	Attribute Type	Domain Constraints
students	name	char(20)	Name != “”
students	phone	char(7)	

Hence, only limited access to the database is granted to that user. Also, for this project, you are not required to handle passwords for the users.

Data Definition and Data Manipulation Commands

You are required to process the following data definition and data manipulation commands as well as the help commands listed at the end of the table below:

Table 6. List of commands

Command	**Command structure**
Create Table	CREATE TABLE *table_name* (*attr_name1 attr_type1, attr_name2 attr_type2*, …)
Drop Table	DROP TABLE *table_name*
Select	SELECT *attr_list* FROM *table_list* WHERE *condition_list*
Insert	INSERT INTO *table_name* VALUES(*attr_value1*, *attr_value2*, …) Note that NULL is not permitted for any attribute. Also, your program need not support the nested SQL statement.
Delete	DELETE FROM *table_name* WHERE *condition_list*
Update	UPDATE *table_name* SET *attr1_name* = *attr1_value*, *attr2_name* = *attr2_value*… WHERE *condition_list*
HELP TABLES	Prints out the list of tables defined.
HELP DESCRIBE T_NAME	Describes the schema of table T_NAME.
HELP CMD	Describes the built-in command CMD, where CMD is one of the data definition and manipulation commands listed above.
QUIT	Quits the program.

Development Guideline

Language

You can use Java, C or C++ to implement the DBMS. Your command parser can either use the native features of your language of implementation or use a helper language such as Lex & Yacc. Note that you are not allowed to use any database related libraries (e.g. JDBC for Java) of the language of your choice or an existing DBMS for data manipulation. You should store all persistent data in Unix files.

Phase 1: Command parsing

Your DBMS should have a data definition and manipulation language, which implements a subset of SQL. Your command parser should accept input from the command line and parse it to check if the input complies with the rules of the language. Each command in the language should be terminated with a semicolon (i.e. the main unit of execution in your language is a command followed by a semicolon, just like in Java or Sqlplus). You can assume that a command will never have a newline character in it (the whole command will be written as a single line with a semicolon at the end). Upon parsing a command, you should output an error message if the input is not acceptable in the grammar of your language and do the necessary processing if it is acceptable.

One way to parse commands is to define your grammar using Lex & Yacc and include the actions corresponding to each of the commands in your language in your parser. Another way (for the purposes of this project) is to use an input scanner, such as the Scanner class in Java, to accept input and include a method for parsing the input into individual tokens to determine its compliance with the language rules as well as the corresponding action to be taken by the DBMS.

Detailed description of the structure of commands:

Note that the commands you will implement for this project have a more restricted structure than regular SQL commands (for easy implementation). The command forms specified here are a subset of the forms acceptable by SQL, therefore even if you implement the whole SQL language, it will still cover the command forms in this project. You can also assume that the reserved keywords will be input in all uppercase, but you can feel free to implement case-insensitive keywords if you prefer. Note that you are **not** asked to implement **nested** statements in any of the commands.

1. CREATE TABLE:

This command has the form:

CREATE TABLE table_name **(** attribute_1 attribute1_type **CHECK (**constraint1**),** attribute_2 attribute2_type**,** …**, PRIMARY KEY (** attribute_1, attribute_2 **)**, **FOREIGN KEY (** attribute_y **) REFERENCES** table_x **(** attribute_t **), FOREIGN KEY (** attribute_w **) REFERENCES** table_y **(** attribute_z **)…);**

The "CREATE TABLE" token is followed by any number of attribute name – attribute type pairs separated by commas. Each attribute name – attribute type pair can **optionally** be followed by a constraint specified using the keyword "CHECK" followed by a domain constraint in one of the forms specified in Table 3, enclosed in parentheses (Note that **optional** means that the input by the user is optional, not the implementation). This is followed by the token "PRIMARY KEY" and a list of attribute names separated by commas, enclosed in parentheses. Note that the specification of the primary key constraint is mandatory in this project and will always follow the listing of attributes. After the primary key constraint, the command should accept an **optional** list of foreign key constraints specified with the token "FOREIGN KEY" followed by an attribute name enclosed in parentheses, followed by the keyword "REFERENCES", a table name and an attribute name enclosed in parentheses. Multiple foreign key constraints are separated by commas.

The output should be "Table created successfully" if table creation succeeds, and a descriptive error message if it fails.

2. DROP TABLE:

This command has the form:

DROP TABLE table_name**;**

The "DROP TABLE" token is followed by a table name.

The output should be "Table dropped successfully" if table dropping succeeds, and a descriptive error message if it fails.

3. SELECT:

This command has the form:

SELECT attribute_list **FROM** table_list **WHERE** condition_list**;**

The token “SELECT” is followed by an attribute list, followed by the token “FROM” and a table name list. This is followed by an **optional** “WHERE” keyword and condition list. For simplicity, you are only asked to implement an attribute list consisting of attribute names separated by commas and not using the dot notation, in addition to “*”, which stands for all attributes. You can also assume that no attributes of different tables will have the same name. The table list will also be a simple list of table names separated by commas. The condition list has the following format:

attribute1 operator value1

OR

attribute1 operator value1 AND/OR attribute2 operator value2 AND/OR attribute3 operator value3…

The operator can be any of “=”, “!=”, “<”, “>”, “<=”, “>=”.

For simplicity, you can assume that if there are multiple conjunction/disjunction operators in the predicate, they will all be the same operator (i.e. there will not be AND and OR operators mixed in the same condition). Hence, the conditions do not need to be enclosed in parentheses. The values in the conditions can either be a constant value or the name of another attribute.

An example command is as follows:

SELECT num FROM Student, Enrolled WHERE num = snum AND age > 18;

assuming *num* and *age* are attributes of the Student table and *snum* is an attribute of the Enrolled table.

The output of this command should be the list of matching tuples if there is no error. Otherwise, a descriptive error message should be printed. The first line of the result should be the names of the attributes separated by tab characters (as you would get from Sqlplus). Then you should print the tuples, one line per record, with different attribute values separated by tab characters.

4. INSERT:

This command has the form:

INSERT INTO table_name **VALUES (** val1, val2, … **);**

The “INSERT INTO” token is followed by a table name, followed by the token “VALUES” and a list of values separated by commas enclosed in parentheses. Each value should be either a number (integer or decimal) or a string enclosed in single quotes. Note that you are asked to implement only one form of this command, where the values listed are inserted into the table in the same order that they are specified, i.e. the first value corresponds to the value of the first attribute, the second value corresponds to the value of the second attribute etc. Note that to satisfy this requirement, you should store the ordering of attributes when a table is created.

The output should be the message “Tuple inserted successfully” if the insertion succeeds, and a descriptive error message if it fails. Note that you need to check for any constraints specified on the table to make sure they are not violated before proceeding with the insertion. Specifically, you should check that:

a) A record with the same primary key value does not already exist in the database
b) The values inserted comply with any domain constraints defined on the table
c) If any foreign key constraints are defined, there should be matching values in the tables referred to.

5. DELETE:

This command has the form:

DELETE FROM table_name **WHERE** condition_list**;**

The “DELETE FROM” token is followed by a table name, followed by the **optional** “WHERE” keyword and a condition list. The condition list has the following format:

attribute1 operator value1

OR

attribute1 operator value1 AND/OR attribute2 operator value2 AND/OR attribute3 operator value3…

The operator can be any of “=”, “!=”, “<”, “>”, “<=”, “>=”.

For simplicity, you can assume that if there are multiple conjunction/disjunction operators in the predicate, they will all be the same operator (i.e. there will not be AND and OR operators mixed in the same condition). Hence, the conditions do not need to be enclosed in parentheses.

The output should be the message “X rows affected”, where X is the number of tuples deleted if there are no errors. Otherwise a descriptive error message should be printed.

6. UPDATE:

This command has the form:

UPDATE table_name **SET** attr1 = val1, attr2 = val2… **WHERE** condition_list**;**

The “UPDATE” token is followed by a table name, which is followed by the token “SET” and a list of attribute name=attribute value pairs separated by commas. This is followed by an **optional** “WHERE” token and a condition list in the same form as the condition list in the DELETE command.

The output should be the message “X rows affected”, where X is the number of tuples updated if there are no errors. Otherwise a descriptive error message should be printed.

7. HELP TABLES:

This command has the form:

HELP TABLES;

The output should be the list of tables in the database, with one row per table name. If there are no tables in the database, you should print the message “No tables found”.

8. **HELP DESCRIBE** T_NAME:

This command has the form:

HELP DESCRIBE table_name**;**

The token “HELP DESCRIBE” is followed by a table name.

The output should be the list of attribute names and types in the table and a list of any constraints (primary key, foreign key, domain), one row for each attribute. If there are any constraints for an attribute, you should print the primary key constraint first, foreign key constraint next and domain constraints last. If the table does not exist, you should print an error message.

An example output is as follows:

```
snum -- int -- primary key -- snum>0
sname -- char(30)
age -- int -- age > 0 AND age < 100
deptid -- int -- foreign key references Department(deptid)
```

9. **HELP** CMD:

This command has the form:

HELP command**;**

The token “HELP” is followed by a command, where command is one of the following:

CREATE TABLE
DROP TABLE
SELECT
INSERT
DELETE
UPDATE

The output should be a short description of the corresponding command and its expected format.

10. QUIT:

This command has the form:

QUIT;

Upon issuance of this command, the program should commit any pending changes to the disk and terminate.

Phase 2: Persistent data management

Your DBMS should persist data across different runs, i.e. if there are any transactions changing the state of the database in one run of your program (i.e. table creation/dropping, record insertions/deletions/updates), the changes should persist after the program is ended. To achieve this, you should commit transactions to the disk by saving the changes in the files that store the data for your DBMS. For simplicity, you can assume that your DBMS will exit properly after each run (i.e. it will not crash), therefore you can keep track of changes to the database in memory during runtime and commit to the disk before exiting the program.

For easy data persistence, it is advisable to store all relation schemas in one file, and have a separate file for the actual data (tuples) in each relation. You can take advantage of object serialization to save data to/retrieve data from a file if Java is your language of implementation. Vector/ArrayList classes in Java also come in handy for keeping a list of tuples.

There are three main classes of operations that would result in changes to the state of the database:

1. Create Table: This operation results in the addition of a relation schema to the list of all schemas in the database. When the user issues this command, you should make sure there is no table with the same name in the database before adding the table to the list of schemas. If a table with the same name exists, the command should result in an error. Note that you also need to take care of any foreign key constraints, making sure that the tables and attributes referred to when creating this table already exist in the list of schemas as well.

2. Drop Table: This operation results in the removal of a relation schema from the list of all schemas in the database. When the user issues this command, you should make sure the specified table already exists in the database. Otherwise the command should result in an error. Upon issuance of this command, the specified table should be removed from the list of schemas and all tuples of the table should be deleted too. Note that you also need to check for foreign key constraints before deleting a table. If any other tables in the database have attributes referring to any attributes of this table, the command should result in an error.

3. Insert/Delete/Update a record: These operations result in changes to the list of records/attributes of records in individual relation files. When inserting a record into the database, you should check that the primary key value of the record to be inserted does not match the primary key value of any existing record in the same table. Note that for simplicity, you do not have to check for referential integrity when deleting records. You can assume that no record with attributes referred to by another record in the database will be deleted.

Data storage structure

For this project, you can use any data structure to store the tuples of your database. The naïve approach would be to store all data in an unstructured list such as an array, which would require searching through the whole list to find a record matching a specific condition. Alternatively you could use a structure such as a B-tree that keeps the tuples sorted on a specific attribute value or use a hash table indexed by a specific attribute of each table. You can also create multiple indices on the tables, one for each (some) attribute. To implement these data structures, you can either make use of existing data structure libraries in your implementation language or code the data structure yourself. While each tuple of a relation can be represented as a single entity (object) with multiple attributes in your storage structure, you can also use an alternative structure such as a multi-dimensional array to facilitate relational operations like selection, projection and join.

Tips:
The following abstractions are useful for retrieval of records from the database:
TableScanner: An iterator over the table retrieving one tuple at a time (You might want to embed identification of deleted tuples in such abstraction).
TupleReader: Given a representation of a tuple (be it binary or textual) and a schema, it returns record's value for a given attribute name.

Phase 3: Relational Operations

The project involves implementation of the following relational algebra operations as part of the SELECT, DELETE and UPDATE commands:

a) **Select**: This operation is involved in the following commands:
- SELECT: Evaluation of the predicate in the WHERE clause determines which tuples of the relation(s) should be included in the result set.
- DELETE: Evaluation of the predicate in the WHERE clause determines which tuples of the relation should be deleted.
- UPDATE: Evaluation of the predicate in the WHERE clause determines which tuples of the relation should be updated.

b) **Project:** This operation is used for selecting the attributes to display in a SELECT command.

c) **Join:** This operation is involved only in the SELECT command, if multiple tables are listed in the FROM clause and the WHERE clause includes conditions that require the equality of attributes from different tables. Note that in this project your language is not required to recognize the “join” keyword in SQL. Therefore, join operations will be implicit (through equating attributes of different tables).

d) **Cartesian Product:** This operation is involved only in the SELECT command, if multiple tables are listed in the FROM clause and the WHERE clause **does not** include **any** equality operators between attributes of different tables. The Cartesian product simply creates a result set which consists of each record of one table joined with each record of another table. If there are *n* records in one table, and *m* records in the other, their Cartesian product will have *m x n* records. Just like the join operation, the Cartesian product operation will be implicit too, as no special keywords in your language will be reserved for this operation.

Phase 4: Access Control

This phase of the project adds functionality on top of what is implemented in the previous phases. Your program should allow three types of users: Admin, User-A, User-B. Below is a list of operations to be supported for each user type.

Admin: The admin is the user with the highest privileges. When the admin user runs the program, he should have access to the following functionality:

1. **Add/delete users:** Your program should maintain a list of valid users of the DBMS in a UNIX file.

The admin can create a new user using the following command:
CREATE USER *username user-type;*
Here, the username is any string and the user-type is either User-A or User-B.
Upon issuance of this command, you should make sure a user with the same name does not already exist, and print an error message if it exists. If the user is created successfully you should print the message “User created successfully”, and update the file storing usernames and types with the new user before the program is quit.

The admin can delete an existing user using the following command:
DELETE USER *username;*
Upon issuance of this command, you should check that the specified user already exists. If not, you should print an error message. If the deletion is successful, you should print the message “User deleted successfully”. As a result of this command, you should delete the specified username from the file storing usernames and types before the program is quit.

2. **Create/delete subschemas:** The admin can provide some users with a restricted view of the database by creating subschemas on some of the tables. For the sake of this project, we assume that whenever the admin creates a subschema for a database table, that table is presented in its restricted view to all users who are of type User-B. The implementation of subschema storage can be performed in different ways. One method would be to have a separate file to store all relations in their restricted view, just like the file you would have to store all relations.

   The following command will be used by the admin to create a subschema:
   CREATE SUBSCHEMA *table-name attribute-list;*

   Here, table-name is the name of the table on which to create the subschema, and attribute-list is a comma-separated list of attributes of that table to be included in the subschema. Upon issuance of this command, you should check that the specified table exists and the list of attributes is valid for that table. If not, you should print a descriptive error message. If the subschema creation succeeds, you should print the message “Subschema created successfully” and update the subschemas file accordingly before the program is quit. Note that if the creation of a new subschema is attempted for a table for which there is already a subschema, you should replace the existing subschema with the new one.

3. **Run SQL commands:** The admin user should be able to run any of the commands in Table 6.

In order to provide the first two functions, you will need to extend the valid command set in your language to include the commands listed above.

User-A: This user type is a regular user of the DBMS who is capable of running all commands in Table 6. This user type has access to a full view of the database.

User-B: This user type is entitled to a restricted view of the database as determined by the DBMS admin. This type of user is only allowed to perform read-only operations, which include the SELECT, HELP (all types of this command) and QUIT commands in Table 6. Whenever this type of user issues a query or runs the HELP DESCRIBE T_NAME command, care should be taken to present to him only data he is entitled to see.

Phase 5: Driver

Your program should be run using the following command:

./minidbms *username*

where username is either *admin* or one of the usernames created by the system administrator. Throughout a run of the program, you should make sure to limit the operations to be performed by the user to those the user is entitled to, based on his user type. If a user tries to perform an operation not supported for him, you should print the error message: “Invalid operation”.

You can submit in two execution modes: interactive mode and batch mode. When running in the interactive mode, your program should always generate prompt for the next command; when running in batch mode, your program will read sequences of SQL commands from a UNIX file. To achieve this, your SQL interpreter must allow inputs only from the standard input so that UNIX file redirection can be used to process SQL command sequences from files.

Appropriate error messages should be displayed at all times in a human readable form (meaning if you can’t tell immediately why you got the error, it’s not human readable). Points will be deducted for errors that are not handled correctly.

Time Savers:

1. Reuse of Basic table operations:
   Metadata (Schemas, Subschemas) can be accessed as typical relations.
2. Reuse of predicate evaluator:
   Predicate expression evaluation is needed
   a. When enforcing integrity constraints
   b. Relational operator that needs a predicate (see summary of those below).
   One way to achieve that is to add any form of constraints to the "where" clause.
3. You can think of all relational operators as derivatives of table scan. (To allow composition).
4. Type checking for compatible attribute types is needed whether it's in a query predicate, domain constraint, or referential integrity constraint.

Submission Instructions

Please create a README file containing identifying information. For example:

CS448 - Project 2
Author: John Doe
Login: jdoe
Email: jdoe@cs.purdue.edu

Include here anything you might want us to know when grading your project.

To turn in your project, ssh to lore.cs.purdue.edu, create a folder named project2 in your home directory and copy your project files and your README file to that folder.

After copying your files in the folder project2, execute the following command in your home directory:

turnin –c cs448 –p proj2 project2

To verify the contents of your submission, execute the following command right after submission:

turnin –c cs448 –p proj2 -v