CSCE 315: Programming Studio (Spring 2015)

Project 2: Database Management System

Due Dates and Updates

Here are the various due dates. See near the end for details on what you need to submit for each submission window.

Date	Due
02/06	Design documents
02/13	DB core function code (DB engine)
02/20	Parser code
02/27	Integrated Parser + DB engine
03/06	Final project code + DB app + report

Any updated info about the project will also be posted here.

Team Assignment

This project is a team project. The target team size is four. If the number of students is not divisible by four, we can have a few teams with size close to four. The teams will be assigned by the instructor.

In a Nutshell

This project consists of two parts:

1. Implement a simple *database management system* (DBMS) and provide your binaries (not the source code) to another designated team.
2. Write a DB application using the DBMS binaries supplied to you by another designated team.

PART I: Specification of the DBMS

Database management systems are very complex pieces of software. They support concurrent use of a database, transactions, permission handling, query optimizations, logging, you name it. To be efficient, they utilize highly tuned algorithms developed over the course of decades. So obviously, for a two-week long project, we must simplify things a bit. We thus base our DBMS on *relational algebra*.

Relational algebra is a formal system for manipulating relations. It consists of six primitive operations. Each of the operations take *relations* as arguments, and produce a *relation* as a result. Thus, the operations compose freely.

The upside of using relational algebra is that the implementation effort of the DBMS stays manageable. The downside is that queries tend to be more verbose and maybe a bit harder to construct than with SQL.

Terminology

Database
: a collection of relations

Relation
: a table with columns and rows

Attribute
: a named column of a relation

Domain
: the set of admissible values for one or more attributes

Tuple
: a row of a relation (sequence of values, one for each attribute of a relation)

Relational Algebra

The six operations of (the core of) relational algebra are:

1. *Selection:* select the tuples in a relation that satisfy a particular condition.
2. *Projection:* select a subset of the attributes in a relation.
3. *Renaming:* rename the attributes in a relation.
4. *Set Union:* compute the union of two relations; the relations must be *union-compatible*.
5. *Set Difference:* compute the set difference of two relations; the relations must be *union-compatible*.
6. *Cross Product:* compute the Cartesian product of two relations.

Grammar

The communication with the DBMS takes place using a domain-specific language. The grammar of *queries* in this language is as follows.

query := *relation-name* `<-` *expr* `;`

relation-name := *identifier*

identifier := *alpha* { (*alpha* | *digit*) }

alpha := `a` | … | `z` | `A` | … | `Z` | `_`

digit := `0` | … | `9`

expr := *atomic-expr* | *selection* | *projection* | *renaming* | *union* | *difference* | *product*

atomic-expr := *relation-name* | `(` *expr* `)`

selection := `select (` *condition* `)` *atomic-expr*

condition := *conjunction* { `||` *conjunction* }

conjunction := *comparison* { `&&` *comparison* }

comparison := *operand op operand* | `(` *condition* `)`

op := `==` | `!=` | `<` | `>` | `<=` | `>=`

operand := *attribute-name* | *literal*

attribute-name := *identifier*

literal := intentionally left unspecified (strings, numbers, etc.)

projection := `project (` *attribute-list* `)` *atomic-expr*

attribute-list := *attribute-name* { , *attribute-name* }

renaming := `rename (` *attribute-list* `)` *atomic-expr*

union := *atomic-expr* + *atomic-expr*

difference := *atomic-expr* - *atomic-expr*

product := *atomic-expr* * *atomic-expr*

Queries generated from the above grammar compute new relations based on existing relations. Queries can also name those new relations. We need, however, some ways to create some initial relations (constituting a database), update the relations within the database, store the results of queries back to the database, and delete tuples from relations. We specify a data manipulation language with the following commands for these purposes:

command := (*open-cmd* | *close-cmd* | *write-cmd* | *exit-cmd* | *show-cmd* | *create-cmd* | *update-cmd* | *insert-cmd* | *delete-cmd*) `;`

open-cmd := `OPEN` *relation-name*

close-cmd := `CLOSE` *relation-name*

write-cmd := `WRITE` *relation-name*

exit-cmd := `EXIT`

show-cmd := `SHOW` *atomic-expr*

create-cmd := `CREATE TABLE` *relation-name* `(` *typed-attribute-list* `) PRIMARY KEY (` *attribute-list* `)`

update-cmd := `UPDATE` *relation-name* `SET` *attribute-name* = *literal* { , *attribute-name* = *literal* } `WHERE` *condition*

insert-cmd := `INSERT INTO` *relation-name* `VALUES FROM (` *literal* { , *literal* } `) |`
`INSERT INTO` *relation-name* `VALUES FROM RELATION` *expr*

delete-cmd := `DELETE FROM` *relation-name* `WHERE` *condition*

typed-attribute-list := *attribute-name type* { , *attribute-name type* }

type := `VARCHAR (` *integer* `) | INTEGER`

integer := *digit* { *digit* }

A program in our data manipulation language (DML) is then defined as:

program := { (*query* | *command*) }

Example:

```
CREATE TABLE animals (name VARCHAR(20), kind VARCHAR(8), years INTEGER) PRIMARY KEY (name, kind);

INSERT INTO animals VALUES FROM ("Joe", "cat", 4);
INSERT INTO animals VALUES FROM ("Spot", "dog", 10);
INSERT INTO animals VALUES FROM ("Snoopy", "dog", 3);
INSERT INTO animals VALUES FROM ("Tweety", "bird", 1);
INSERT INTO animals VALUES FROM ("Joe", "bird", 2);

SHOW animals;

dogs <- select (kind == "dog") animals;
old_dogs <- select (age > 10) dogs;

cats_or_dogs <- dogs + (select (kind == "cat") animals);

CREATE TABLE species (kind VARCHAR(10)) PRIMARY KEY (kind);

INSERT INTO species VALUES FROM RELATION project (kind) animals;

a <- rename (aname, akind) (project (name, kind) animals);
common_names <- project (name) (select (aname == name && akind != kind) (a * animals));
answer <- common_names;

SHOW answer;

WRITE animals;
CLOSE animals;

EXIT;
```

Note that we made a distinction between queries and commands in the grammar of the DML. The result of a query is a *view*. A view is not stored in the database. Rather, it is a temporary relation whose lifetime ends when a DML program finishes. Only the updates caused by the commands persist from one DML program execution to another.

The relations themselves should be saved in a file in plain ASCII text (e.g. each tuple on its own line, with attribute values separated by commas). To make it simple, we assume that each database file can only store one relation and the filename is the same as the relation name with the suffix ".db". To load a relation from a database file, use the `OPEN` command. Opening a non-existing file will result in nothing. To add a new relation to a file, use the `WRITE` command (the filename will be "relationname.db" by default). If you have a new view that you want to save to a file, you can use this `WRITE` command. To save all changes to the relation in a database file and close, use the `CLOSE` command.
NOTE: You have to determine the specific behavior of `OPEN`, etc. For example, if you opened one db file, changed something, and opened the same db file again, does the db on file overwrite what's in memory?

To exit from the DML interpreter, use the `EXIT` command.

To print a certain relation or a view, use the `SHOW` command.

PART II: DB Application

Interfacing your DB and your Host Programming Language

Since the basic DB language you developed in PART I does not include:

- control flow (conditional statements, loops, etc.)
- I/O (keyboard input, arbitrary output)
- etc.

it is not possible to write a DB application using just the DML from PART I.

To overcome this shortcoming, you will have to write a hybrid code where the main language is C++ or Java (the host language). The host program will provide most of the user interface: displaying menus, taking user input, and showing results. Based on these user inputs, a custom query or command string can be generated and passed on to the DBMS to be parsed and executed.

Optional (i.e., as needed): You may also need to retrieve the results of the queries to feed into the host language's control flow. The DBMS object can contain a member function to access the relations, views, and the attributes by their name (as a string).

This is what an example interaction might look like in C++:

```
string name;
cin << name;

// create DB command on the fly
string query = string("") +
               "answer <- project (age) ( select (kind == \"dog\" && name == " + name + ") animals )";

// pass on the query
rdbms.execute(query);

if (rdbms.relation(relation_name).int_field(field_name) == 10) {
        …
}
```

The Application: A Blog (a.k.a The Social Electronic Diary)

Way, way back in the 1990s, there was Dear Diary. The ultimate secret keeper, this electronic diary was to some children of the '90s what the TalkBoy was for other children. Siblings could no longer easily find your handwritten diary under the mattress and thumb through it for their own amusement. This little handheld device could record secrets when you whispered them into its speaker, and it could store your innermost thoughts on its hard drive, by way of a mini-keyboard. And, with a hot pink case and purple keys… **oh man, was it cool!**

The basic function of an electronic diary is to store the user's writing in digital form and to make it available for reading later. This is the same basic functionality provided by a web log (later called a weblog, and then simply a blog). A main difference between a diary and a blog, however, is that a diary is usually meant to remain private (only one person should ever read it) but a blog is usually meant to be world-readable (content is published for all to see). Blog posts may be tagged with short labels to make it easier to find certain kinds of posts (e.g. funny, political, cats, etc.). A post can have many tags and a tag can be applied to many posts. Like diary entires, blog posts typically have a single author. Additionally, blogs may allow readers to comment on posts. Blogs which seek discussion amongst readers may even allow comments to have comments. Some blog

posts don't allow any comments at all. In a sense, a blog is a social electronic diary. Your task is to implement the functions of a multi-user blog (social electronic diary) using another team's DBMS from PART I.

Required functionality includes the following (note: the user interface can be simple ASCII text with text input):

- Make a post
- Edit a post's
  - Title
  - Author
  - Content
  - Tags
  - Commenting (turn on/off)
- Comment on a post
- Delete a post and all associated comments
- Search for a post by:
  - Author name
  - Title
  - Tag(s)
  - Date
- Exit the application

The edit function can be really rudimentary. You do not need to program a command-line editor or any kind of editor. Just display current content, and ask user to *replace* it with something new.

Furthermore, the entire use interface can be really rudimentary, based on a scrolling menu based system. An example is shown below (note: **BOLD** means user input).

```
[Main Menu]

1. Make a new post
2. Search for a post
3. Exit

* Enter command: 2

[Search Menu]

Search by:
1. Author
2. Title
3. Tag(s)
4. Date
5. Return to Main Menu

* Enter command: 1

* Enter author: Dr. Ritchey

[Dr. Ritchey's Posts]

1. Syllabus (01/21/2015)
2. Project 2: Database Management System (01/30/2015)
...
N. Return to Main Menu

* Enter ID: 2

[Project 2: Database Management System]

1. View
2. Edit
3. Delete
4. Comment
```

```
5. Return to Main Menu

* Enter command: 4

[Commenting on Project 2: Database Management System]

1. Comment on post
2. Comment on comment

* Enter command: 2

[Comments on Project 2: Database Management System]
1. On 01/30/2015, Dr. Ritchey said:
   Hope y'all like the project.

* Enter ID: 1

* Enter name: Namey McNamerson

* Enter comment: This project is the best!

Comment added.

[Project 2: Database Management System]

1. View
2. Edit
3. Delete
4. Comment
5. Return to Main Menu

* Enter command: 1

--------------------------------------------------------------------------------
Project 2: Database Management System
By: Dr. Ritchey
Date: 01/30/2015

Due Dates and Updates
Here are the various due dates. See near the end for details on what you need to
submit for each submission window.
...

Tags: Project, Database, CSCE 315

Comments:
1.      On 01/30/2015, Dr. Ritchey said:
        Hope y'all like the project.

        1.1     On 01/30/2015, Namey McNamerson said:
                This project is the best!

--------------------------------------------------------------------------------

[Project 2: Database Management System]

1. View
2. Edit
3. Delete
4. Comment
5. Return to Main Menu

* Enter command: 5

[Main Menu]

1. Make a new post
2. Search for a post
3. Exit

* Enter command: 3
```

```
Goodbye.
```

Deliverables and Requirements

- All teams must use `unix.cse.tamu.edu` so that your PART 1 binaries can be used by another team in PART 2.
- All teams must use github.tamu.edu. Make your project **private** and give access to the TAs and Instructors.
- Each team must maintain a development log (wiki page in github.tamu.edu titled "Development Log") updated by the team members. This log will be graded. There is no designated format, except that you need to time stamp, write down the name, and write a brief description of the activity. We will check your daily progress.
- Major routines should include unit testing.
- Demo in the lab may be required.

1. Design documents: Follow the guidelines in Scott' Hackett's "How to Write an Effective Design Document" (Writing for a Peer Developer). Include all four sections described in the guide.
    - Set up your design document ("Design document") as a wiki page in github.tamu.edu.
    - The design document should cover both phase 1 (DB engine) and phase 2 (DB app testing).
    - Documents for phase 2 should include ER diagram and corresponding relation schema, besides other things.
    - Grading:
        1. 20%: all four sections included.
        2. 50%: Part I DBMS - parser, DB engine
        3. 30%: Part II DB app testing - ER diagram, relation schema, testing workflow (`create`, `insert`, …)
2. DB core function code: Upload the core functions. These are host-language classes and functions that implement the core DB functionality. For example, function calls for creating a relation table data structure, etc. You don't need to link this with the parser just yet. You should be able to write a test program in the host-language that directly calls the core DB functions `create(…)`, `insert(…)`, etc.
    1. 10%: layout, style, comments
    2. 30%: commands (`open`, `close`, …, `delete`)
    3. 40%: queries (`select`, `project`, …, `product`)
    4. 10%: condition (conjunction, comparison, operators, etc.)
    5. 10%: development log
3. Parser code: Upload your parser code. It should be able to accept or reject an arbitrary command or query: Accept syntactically correct command or query, and reject anything that violates the syntax.
    1. 10%: layout, style, comments
    2. 30%: commands (`open`, `close`, …, `delete`)
    3. 40%: queries (`select`, `project`, …, `product`)
    4. 10%: condition (conjunction, comparison, operators, etc.)
    5. 10%: development log
4. Parser+DB engine integrated code: Upload your integrated parser + DB engine. The DBMS engine should compile into a stand-alone executable application so that when you run the application, it works as a DBMS command shell, where you can type in any command or query. Full DB functionality is expected (including file I/O). A full test of the DB shell, based on manually entered commands, should be conducted and the results included in the submission.
    1. 10%: layout, style, comments
    2. 20%: commands (`open`, `close`, …, `delete`)
    3. 30%: queries (`select`, `project`, …, `product`)
    4. 10%: condition (conjunction, comparison, operators, etc.)
    5. 20%: full test of all DB commands -- input and output log
    6. 10%: development log

5. Final project code + DB app + DB app demo + report:
    - DB app should compile into a stand-alone executable.
    - DB app demo should include a detailed operation of the DB app: input and output log.
    - post production notes (changes you had to make to your design and why, difficulties, solutions, lessons learned). Make it a wiki page "Post Production Notes".
    - individual work load distribution (percentage, must add up to 100%). Include this in the "Post production notes".
        - Formula for individual score calculation is as follows:

          ```
          individual score = min(sqrt(your percentage/25)*team_score,110)
          ```

          For example, if your contribution was 20% and your team score was 85, your individual score is `min(sqrt(20/25)*85,110) = 76`. Note that 25% is the baseline (equal contribution by all four members). If your contribution was 30% and your team score was 85, your individual score is `min(sqrt(30/25)*85,110) = 93`.
    - Development log (wiki page).
    - Final Grading:
        1. 5%: Layout, style, comments
        2. 40%: DBMS engine: completeness, functionality
        3. 10%: Test session log: completeness, accuracy
        4. 5%: Post production notes
        5. 10%: Development log
        6. 30%: Weighted grades from earlier submissions (design doc, parser, DB core function): this gives you some chance to make up for previous blunders.

Submission

- All submissions should be through CSNet.
- Design doc submission should be a single PDF file uploaded to CSNet. This will be a printout of your wiki page.
- First, fork your latest project into an archival branch named: Submission 1, Submission 2, and Submission 3, etc. for the code submissions, respectively.
- Use the "Download ZIP" feature in github.tamu.edu and upload the resulting zip file.
- As for the documents (development log, etc.), we will check the github.tamu.edu project.
- Late penalty is 1% per 1 hour.

Original concept/design/most of the DBMS text by Jaakko Järvi and Yoonsuck Choe. Modifications by Philip Ritchey and Walter Daugherity.