CSCE-315: Programming Studio (Summer 2017)

Project 3: Database Management System (*Updated Specification*)

Update

As foretold, here are the dreaded “updates in the middle of the project”: 😱

1. The “real data” you must now use is the real Fall 2017 class schedule on Howdy, which is available in 3 forms on build.tamu.edu:
   `"/pub/faculty_share/daugher/datafiles/Class Schedule Fall 2017.csv"`
   `"/pub/faculty_share/daugher/datafiles/Class Schedule Fall 2017.pdf"`
   `"/pub/faculty_share/daugher/datafiles/Class Schedule Fall 2017.xlsx"`

Rewrite your database ER diagrams, relations, and schemata (yes, that’s one plural of schema) to accommodate all columns of this class schedule except Cmp (which is always CS, since this is the College Station CaMPus’ schedule). For example, you may need to split CourseNumber into Subject, Course, and Section. Load this data into your database relations.

Note: These three files have not been sanitized, so, for example, you should discard the miscellaneous header lines and the hyperlinks (including the “Restrictions/Details” items that didn’t get merged with the text above them). Also beware of varying line-ending character sequences—I have not checked these files, but different applications may use different end-of-line sequences, *e.g.*, \n, \r, \n\r, or \r\n. This may produce unexpected results, for example if you print "ABC\rDEF".

2. Come up with 2 queries your DBMS loaded with this real data *can* answer but howdy.tamu.edu *cannot*, put these in your report, and demonstrate them July 3rd.

Due dates

Here are the various due dates. See near the end for details (i.e., what you need to submit for each submission window).

1. Design documents (due 6/23 Friday)
2. Parser code (due 6/27 Tuesday)
3. DB engine function code (due 6/30 Friday)
4. Final project code + report + demo (due 7/3 Monday)

Team configuration

This project is a team project, with four students per team, assigned by the instructor. Select as team leader someone who was not a team leader on project 2. Choose a catchy team name or acronym (but keep it clean ☺).

In a nutshell

1. This project consists of two parts. The task of the first part is to implement the core functions of a simple *database management system* (DBMS). In the second part, you will manually test the DBMS's basic functionality with real data. In both parts use Test-Driven Design and the Agile methodology.

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 have to simplify a lot. We thus base our DBMS on *relational algebra*, and only implement the parser and the DB engine (which responds to queries). For a complete DB server we would have to add network sockets, have secure login and queries (*e.g.*, using a TLS library), and so on, which is beyond the timeframe of this project.

Relational algebra is a formal system for manipulating relations.

It consists of only six primitive operations. Each of the operations takes *relations* as arguments, and produces a relation as a result. The operations thus 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, say, with "real" 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 (that is, 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.

For extra credit, also implement
7. *Natural join*: compute the combination of all tuples in two relations, say, R & S, which are equal on their common attribute

names. The common attributes only appear once in the result. Implement & separately, not as a computationally expensive Cartesian product filtered by conditions.

Grammar

The communication with the DBMS takes place using a domain-specific language. The grammar of *queries* in this language is as follows. (The | symbol means "or" and {} mean zero or more repetitions. Thus an *identifier* is a letter or underscore followed by zero or more letters, underscores, or digits.)

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*
| *natural-join*

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

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*

natural-join ::= *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 use 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

Note: You can experiment with databases by setting up a MySQL account (see https://wiki.cse.tamu.edu/index.php/How_to_Use_MySQL) and converting the following example statements into "real" SQL.

```
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. So 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, using the same DML described above (e.g., CREATE ... INSERT ... INSERT ....). To make it simple, let us 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 nonexisting file will result in nothing.

To add a new relation to a file, use the WRITE command (the filename will be by default "relationname.db").

To save all changes to the relation in a database file and close, use the CLOSE command.

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 Engine and Testing

The second phase of this project is to test your DBMS's functionality by manually typing in the commands in the DML, within the context of a specific DB application domain. The DB application domain will be the TAMU class schedule (like that on Howdy) reorganized into at least six relations like Instructor, Classroom, Department, ClassTimes, etc., to do selects and joins on.

For example, the Instructor relation might contain Name and CourseNumber, and the ClassTimes relation might contain CourseNumber, Day, StartTime, EndTime, etc. Then a query for what times Dr. Daugherity is in class would combine these two relations to find the answers.

Assignment in detail: What should the project teams do

Follow the Agile methodolgy as described in the slides, including:

- a master burn-down list (project backlog) of remaining tasks and estimated time for each task
- at least 2 "weekly" meetings to assign tasks from the burn-down list to the next sprint backlog
- at least 10 "daily" scrums (15-minute meetings) to report progress and
  problems and update the burn-down list items and time still needed
- a graph of projected and actual burn-down rate

Phase 1:
Implement a DBMS that can support the DML specified above and parse commands and queries into an abstract syntax tree (AST).
(Hint: Use a parser generator like ANTLR 4.)
Design the interplay between the DBMS and the host language of your choice, and implement this functionality using Test-Driven
Design. For example, a test harness could send command or query strings to the DBMS parser and compare the resulting AST
to the expected AST by doing an in-order traversal of both trees.
Design and thoroughly document the API of your DBMS library.
Write a concise design document that describes your design and implementation choices, and give short rationale for them.
For testing the parser you can output messages that will be executed in phase 2.

Phase 2:
Implement the DB engine to actually carry out the commands and queries by traversing the abstract syntax tree sent to it from the parser and evaluating the nodes. We will provide a set of test commands and expected outputs, in addition to the "animals" example above, to use in a Test-Driven Design like that of Phase 1. Your DBMS should process all of these commands correctly. Note that & will be tested for extra credit.

Deliverables and Requirements

- All teams must use github.tamu.edu. Also give access to the TA and the instructor.
- 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 parser) and phase 2 (DB engine and testing).
- ◦ Documents for phase 2 should include ER diagram and corresponding relation schema, besides other things.
- ◦ Grading:
  - 20%: all four sections included.
  - 30%: Part I DBMS - parser
  - 50%: Part II DB engine, DB testing - ER diagram, relation schema, testing workflow (create, insert, ...)

2 Parser code (Submission 1): Upload your parser code. It should be able to accept or reject an arbitrary command or query: Accept a syntactically correct command or query, and reject anything that violates the syntax.

- ◦ 10%: layout, style, comments
- ◦ 30%: commands (open, close, ..., delete)
- ◦ 40%: queries (select, project, ..., product)
- ◦ 10%: condition (conjunction, comparison, operators, etc.)
- ◦ 10%: development log

3 DB core function code (Submission 2): Upload the core functions. These are classes and methods that implement

the core DB functionality. For example, method 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 that directly calls the core DB functions create(....), insert(...), etc.

- 10%: layout, style, comments
- 30%: commands (open, close, ..., delete)
- 40%: queries (select, project, ..., product)
- 10%: condition (conjunction, comparison, operators, etc.)
- 10%: development log

4 Final project code (Submission 3) + DB app test session log + report: The DBMS should be a stand-alone executable that integrates the parser and the DB core functions.

- 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.
- DB test session log should include a detailed manual test of the DBMS, starting with the creation, insertion, etc. and the output of those commands. Set up a wiki page called "Test session 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".

    1 Formula for individual score calculation is as follows: individual score = min(sqrt(# on team * your percentage / 100) * team_score,110). For example, if your contribution was 20% and your 4-person team score was 85, your individual score is min(sqrt(4*20/100)*85,110) = 76. Note that the baseline is equal contribution by all team members. If your contribution was 30% and your 4-person team score was 85, your individual score is min(sqrt(4*30/100)*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)
    7. 5%: Extra credit for natural join.

Submission

- All submissions should be through csnet.cse.tamu.edu.
- 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, for the three code submissions listed above.
- Use the "Download ZIP" feature in github and upload the resulting zip file.
- As for the documents (development log, etc.), we will check the github project.
- Late penalty is 1% per 2 hours. So, if you're late 1 day, you lose 12%.

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