CSC173: Project 4
The Relational Data Model

In this project you will gain experience with databases and relational algebra by implementing your own database system. We will build on the presentation (and code) in the textbook.

Note: Your writeup for this project will probably be more extensive than previous projects. In addition to the usual instructions for building and running your project, please be sure to describe what your program is doing and what we're seeing. Of course, that should also be clear from the output of the program by itself.

Part 1

1. Implement a database containing the relations (tables) shown in FOCS Figure 8.1 and 8.2 (also seen in class). Use the method described in Section 8.2 in the section "Representing Relations" and elaborated in Section 8.4 "Primary Storage Structures for Relations." Use hashtables. Describe your implementation briefly but clearly in your writeup.

2. Implement the basic single-relation *insert*, *delete*, and *lookup* operations as functions. If you use the implementation described in the textbook, you will need separate functions for each relation (*e.g.*, `insert_CSG`). You should support leaving some attributes unspecified for *delete* and *lookup* (denoted with "*" in the textbook, perhaps something else in your code). Describe your implementation briefly but clearly in your writeup.

3. Use your *insert* method to populate the tables with (at least) the data given in the figures. Then demonstrate all three operations by performing the operations shown in Example 8.2 (p. 409). Be sure that your program explains itself when run (using informative printed messages).

a) lookup[?]("CS101", 12345, $*$),
CSG finds the grade of student with ID 12345 in CS101.
b) lookup[?]("CS205", "CS120"),
CP[?] asks whether CS120 is a prerequisite of CS205. produces either single tuple ("CS205", "CS120") if tuple is in the relation, or the empty set if not.
c) delete[?]("CS101",$*$),
CR[?] delete tuple with "CS101"
d) insert[?]("CS205", "CS120"),
CP[?] makes CS120 a prerequisite of CS205.
e) insert[?]("CS205", "CS101"),
CP[?] has no effect on the relation of Fig. 8.2(b), because the inserted tuple is already there.

Attributes

Course	StudentId	Grade
CS101	12345	A
CS101	67890	B
EE200	12345	C
EE200	22222	B+
CS101	33333	A−
PH100	67890	C+

σ_Course="CS101" (CSG)

Fig. 8.1. A table of information.

π_StudentId(σ_Course="CS101"(CSG))

CR ▷◁ CDH
course=course

StudentId	Name	Address	Phone
12345	C. Brown	12 Apple St.	555-1234
67890	L. Van Pelt	34 Pear Ave.	555-5678
22222	P. Patty	56 Grape Blvd.	555-9999

(a) StudentId-Name-Address-Phone

Course	Prerequisite
CS101	CS100
EE200	EE005
EE200	CS100
CS120	CS101
CS121	CS120
CS205	CS101
CS206	CS121
CS206	CS205

(b) Course-Prerequisite

Course	Day	Hour
CS101	M	9AM
CS101	W	9AM
CS101	F	9AM
EE200	Tu	10AM
EE200	W	1PM
EE200	Th	10AM

(c) Course-Day-Hour

Course	Room
CS101	Turing Aud.
EE200	25 Ohm Hall
PH100	Newton Lab.

(d) Course-Room

Fig. 8.2. Sample relations.

3. *lookup*(X, R). The result of this operation is the set of tuples in R that match the specification X; the latter is a symbolic tuple as described in the preceding item (2). For example, if we wanted to know for what courses CS101 is a prerequisite, we could ask

$$\textit{lookup}\big((*, \text{“CS101”}),\ \text{Course-Prerequisite}\big)$$

The result would be the set of two matching tuples

```
(1)  for each tuple t in StudentId-Name-Address-Phone do
(2)      if t has "C. Brown" in its Name component then begin
(3)          let i be the StudentId component of tuple t;
(4)          for each tuple s in Course-StudentId-Grade do
(5)              if s has Course component "CS101" and
                     StudentId component i then
(6)                  print the Grade component of tuple s;
         end
```

Fig. 8.8. Finding the grade of C. Brown in CS101.

Part 2

For this part, use the database with all the tuples from Figures 8.1 and 8.2 (that is, before any deletions).

name-(SNAP)->id-(CSG & course)->grade

1. Write a function to answer the query "What grade did *StudentName* get in *CourseName*?" as described in Section 8.6 "Navigation Among Relations." The code for your function MUST look like the pseudocode in Fig. 8.8. (I recommend using the pseudocode as comments in your code.) Demonstrate this functionality by asking the user for the query parameters, performing the query, and printing the results informatively (that is, a kind of REPL).

name-(SNAP)->id-(CSG)->course-(CR)->room

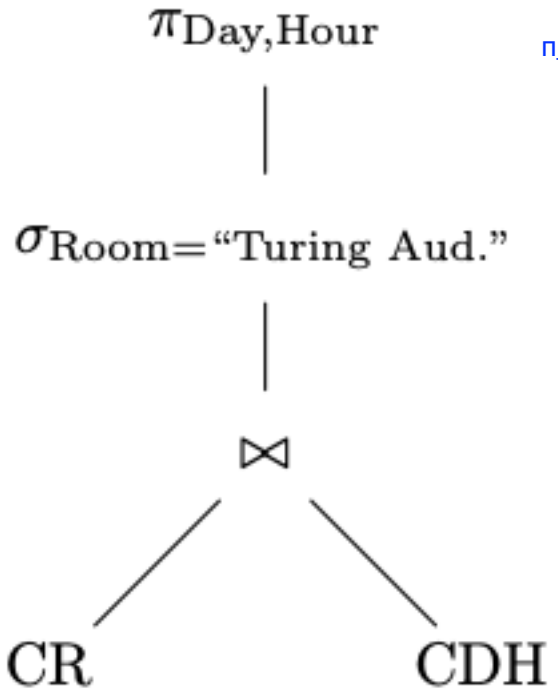
2. Write a function to answer the query "Where is *StudentName* at *Time* on *Day*?" (assuming they are in some course). Demonstrate this functionality also.

Part 3

1. Implement the Relational Algebra operations as described in Section 8.8 and demonstrate this by doing the operations on the "registrar" database described in Examples 8.12 (Selection), 8.13 (Projection), 8.14 (Join), and 8.15 (all three). Describe your implementation briefly but clearly in your writeup.

    - If you follow the implementation in the textbook, you will probably need a different function for each different set of arguments to the operator (*e.g.*, `select_CSG`, `join_CSG_SNAP`). Only implement the ones that you need for the examples listed above. You may also throw in any additional examples that you feel illustrate relevant aspects of your implementation.
    - Note that some of the Relational Algebra operations create a relation with a new schema from that of their operands. (You should know which operations do this...) But if tuples are implemented using structs, how do you create instances of these new "on-the-fly" relations?

      The answer is that you should solve the problem yourself by hand so that you know the schemas needed by your examples. Then add appropriate structure definitions to your program to allow you to do the examples. This is one of many differences between what you need to do for this project and a real database framework.



π_Day,Hour( )

σ_Room="Turing Aud." (CRDH)

CR ⊳⊲ CDH
course=course

Fig. 8.15. Expression tree in relational algebra.

Extra Credit (20% max total)

1. Implement functions for saving your database to one or more files, and loading from the same. Demonstrate this functionality in your program(s) and explain in your writeup. Note: you can do all the other parts of the project even without this functionality, so don't let it stop you or slow you down. [max 10% extra]

2. The code you wrote for the "registrar" database is obviously specific to it. A true database system, like SQLite or MySQL, allows you to represent any database schema. Generalize your code to represent arbitrary databases consisting of arbitrary relations. Note that you do not need to understand SQL for this. What you need to do is create "generic" representations of tuples and tables and then use them to write code for some specific example.

   If you choose to do this, and are confident in your implementation, you may use it for the required parts of the project. Explain what you've done in your writeup and we will give you the extra points if it all works. You may also choose to demonstrate this separately, for example with another program. Again, be clear in your writeup so that we can give you the points. [max 10% extra]

3. Identify a subset of SQL that you can support with your implementation. You should focus on the query parts of the language, not the data definition parts. Write a simple parser for that language, and use it to query (and perhaps also create) the "registrar" database. Demonstrate this by also allowing the user to enter queries and have your system parse and execute them and print the results. Explain what you've done and how it will be demonstrated by the program(s) in your writeup. [max 10% extra]

Project Submission

Your project submission MUST include the following:

1. A `README.txt` file or `README.pdf` document describing:
   (a) Any collaborators (see below)
   (b) How to build your project
   (c) How to run your project's program(s) to demonstrate that it/they meet the requirements

2. All source code for your project. Eclipse projects must include the project settings from the project folder (`.project`, `.cproject`, and `.settings`). Non-Eclipse

projects must include a `Makefile` or shell script that will build the program per your instructions, or at least have those instructions in your README.txt.

3. A completed copy of the submission form posted with the project description. Projects without this will receive a grade of 0. If you cannot complete and save a PDF form, submit a text file containing the questions and your (brief) answers.

We must be able to cut-and-paste from your documentation in order to build and run your code. **The easier you make this for us, the better grade your will be.** It is your job to make both the building and the running of programs easy and informative for your users.

Programming Policies

You must write your programs using the "C99" dialect of C. This means using the "`-std=c99`" option with `gcc` or `clang`. For more information, see Wikipedia.

You must also use the options "`-Wall -Werror`". These cause the compiler to report all warnings, and to make any warnings into errors that prevent your program from compiling. You should be able to write code without warnings in this course.

With these settings, your program should compile and run consistently on any platform. We will deal with any platform-specific discrepancies as they arise.

If you submit an Eclipse project, it must have these settings associated with the project. Projects with that compile with warnings will be considered incomplete.

Furthermore, your program should pass `valgrind` with no error messages. If you don't know what this means or why it is A Good Thing, look at the C for Java Programmers document which has a short section about it. Programs that do not receive a clean report from `valgrind` have problems that **should be fixed** whether or not they run properly. If you are developing on Windows, you will need to look for alternative memory-checking tools.

Late Policy

Late projects will **not** be accepted. Submit what you have by the deadline. If there are extenuating circumstances, submit what you have before the deadline and then explain

yourself via email.

If you have a medical excuse (see the course syllabus), submit what you have and explain yourself as soon as you are able.

Collaboration Policy

You will learn the most if you do the projects YOURSELF.

That said, collaboration on projects is permitted, subject to the following requirements:

- Groups of no more than 3 students, all currently taking CSC173.
- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the group should submit on the group's behalf and the grade will be shared with other members of the group. Other group members should submit a short comment naming the other collaborators.
- All members of a collaborative group will get the same grade on the project.

Academic Honesty

Do not copy code from other students or from the Internet.

Avoid Github and StackOverflow completely for the duration of this course.

There is code out there for all these projects. You know it. We know it.

Posting homework and project solutions to public repositories on sites like GitHub is a violation of the University's Academic Honesty Policy, Section V.B.2 "Giving Unauthorized Aid." Honestly, no prospective employer wants to see your coursework. Make a great project outside of class and share that instead to show off your chops.

Frequently Asked Questions

Q: In C, how can I make a hashtable that can hold any kind of tuple?

A: How would you do it in Java? Think about it. . .

Answer: Generics. And indeed `java.util.Hashtable` is a generic class.

Now, does C have generics? Answer: no. So this is an example of the kind of thing added to Java as it evolved from C, in order to capture common programming patterns more effectively. But if all we have is C, what can we do?

For some dynamic data structures, you can just manage "generic" objects of type `void*` and use casts as needed (I STRONGLY suggest that you encapsulate the casts in helper functions).

The problem is that unlike a linked list, a hashtable needs to know something about the objects it is managing. Why? Because it has to hash them, which means accessing some of their components. So you need to be able to tell the "generic" hashtable code which hash function to use for each type of struct it will be managing. There are ways (*e.g.*, function pointer in tuple or hashtable), and they all amount to reinventing (in fact, pre-inventing) the idea of true classes.

You could just cut and paste the code for different flavors of hashtable. So the routines use the right type of struct for arguments and they know what hash function to use. Is this ugly? Yes. Is it bad software engineering practice? Yes. Is it ok for this project? Yes. This is similar to the business about the schemas of tuples produced by JOIN operations, as described in the project description Part 3.

If you were ambitious you might observe that the cut-and-paste method is purely mechanical, other than writing the hash function itself. That is, you basically go through and change all occurrences of the element type name and that's really all you need to do. Given that, you could define a macro or macros that expanded into the code. Something like:

```
#define RELATION(ETYPE,HFUNC) ...
```

for element type `ETYPE` and name of hash function `HFUNC`. Using macros is a bit of a power C programmer technique. But it is how we did "generic" code back in the day, and as you may know, C++ grew out of just this kind of macro-powered code.

You could also implement tuples using something other than structs, as described in one of the Extra Credit problems. There are many possibilities. . .