Abraham Meza

CS 457/657 – 1001

Dongfang Zhao

November 10, 2021

Project 3: Table Joins

Implementation (Inner Join and Outer Join):

innerJoin(): fetches intersecting tuple data between two file(s)/table(s) by dissecting each file based on attributeNames and attributeValues. The function conducts validity checks, verifying that the input's parameterized data exists (i.e., query options, file/table names, matching acronyms, etc.). If an unknown element is introduced, then the function will prompt an error. In addition, the function prevents users from fetching information outside a database. Furthermore, the function prevents users from fetching information from a table/file that doesn't exist. Using a given set of parameterized data, the function uses "on" comparison conditions to filter out rows of data that contain matching values. Tuples that display matching values (intersection) will be included in the query, while those with no matching values (no intersection), will be omitted. Algorithm is very similar to queryTable(); however, uses an additional for-loop to fetch information for the additional file.

outerJoin(): fetches intersecting tuple data between file(s)/table(s) by dissecting each file based on attributeNames and attributeValues. However, unlike innerJoin(), outerJoin() will return every row from one specified table, even if the join condition fails. The specified table is based on a directional keyword (e.g., LEFT, RIGHT, and FULL). The function conducts validity checks, verifying that the input's parameterized data exists (i.e., query options, file/table names, matching acronyms, etc.). If an unknown element is introduced, then the function will prompt an error. In addition, the function prevents users from fetching information outside a database. Furthermore, the function prevents users from fetching information from a table/file that doesn't exist. Using a given set of parameterized data, the function uses "on" comparison conditions to filter out rows of data that contain matching values. Tuples that display matching values (intersection) will be included in the query. If the tuple does not contain matching values no intersection), then the program will print the tuple information of the specified directional table. The program will then initialize the empty cells of failed join tuples to NULL. Algorithm is very similar to queryTable(); however, uses an additional for-loop to fetch information for the additional file.

Compile Instructions:

IMPORTANT NOTES:

- SAME AS PROJECT 1 & 2 (Refer to PA1 Documentation, Appendix A)
    - The test script provided behaves weirdly, where each comment read by the program includes the comment content, followed by the next SQL command (i.e., CREATE DATABASE CS457_PA3;, etc.). Since my program ignores all SQL comment lines (any string that begins with "--"), some commands are completely ignored since it's considered as part of the same line. As a result, please use the script provided in the submission for proper functionality. The only difference is that the file excludes section comments, containing commands only.
- Added some additional commands at the end of PA3_test.sql for debugging purposes. Additional commands have been commented out and should not affect the original script.

Using the provided Makefile and PA3_test.sql script, type "make"

To run the program, type

- ./db (NO SCRIPT)
- ./db < PA3_test.sql (PIPELINE SCRIPT)

To clean up, type "make clean"

Documentation:

As recommended in the project specifications, the program's design utilizes nested for-loops to extract, analyze, and output tuple data among multiple files. Using the nested for-loop structure, each file has their own set of vectors used to record information, promoting separation of data to prevent potential data collisions. Once the information has been extracted, the program utilizes nested for-loops to conduct vector comparisons, filtering out combinations of indexes that satisfy the given "on" comparison. In other words, the program uses index mapping to illustrate a relationship between both files, generating a single vector to be referenced during the fetching process. The program then uses the successful pairs of indexes to access each file vector, and outputs the information based on query order. The program's design slightly changes based on the join operation (e.g., inner join vs. outer join). Inner Join combines records from two tables, looking for matching/intersecting tuples to print. While Outer Join prints out intersecting tuple data, including every row from one specified table based on direction.

Using the same structure as Project 1, the program takes in a user-input until it reaches the delimiter (';'), then parses the input into a vector to record each argument. Using a series of tree-like if-statements, the program compares each argument to identify the most appropriate function, checking that each parameter from the user-input is valid before continuing. Like Project 1, the program ensures that each structure's command can only be executed in its corresponding environment. In this case, a user cannot execute a "table" command (e.g., select, insert, delete, and update) unless they are in within a database directory. Should the program detect that the user is executing a "table" command outside a directory, the program will prompt an error message informing the user to utilize a "use" command to choose a directory to modify. Please refer to Appendix [A] for the list of new table commands and their formatting.

Using innerJoin(), the program takes in a user's query command, parses the information, and conducts a verification check to evaluate the validity of the user's query, file, and acronym options. Before conducting a query, the function checks for acronym consistency to ensure proper table references. If the user's table acronyms are not consisten within the given attributes and/or the "on" comparison statement, then the program will prompt an error message to indicate ambiguity or irrelevant acronyms. From there, the function checks for existing tables that match given file strings. Should either table not exist, the program prompts an error message indicating that the user cannot conduct a query for a non-existing instance. If the table exists, the function proceeds to verify the availability of query options. Using getline(), the function reads the first line of each file to determine each table's attribute categories (column-based). After determining a list of attributes for each file (vector for each file), the function conducts a series of comparisons to check the existence of each query option. If any of the query options is not a valid attribute in either table, then the program prompts an error message indicating an invalid user-input. To preserve Project 1's "SELECT *" implementation, the function checks for a * argument within the list of queries. If found, then the program replaces the user's query with all available attributes for both files. In addition to checking the existence of query attributes, the program also conducts a search for each operand attribute involved in the comparison statement. The same error checking applies. For each table, if the operand attributes exist, the program will record the corresponding attribute index, using it to fetch each tuple's corresponding attribute value for future comparison.

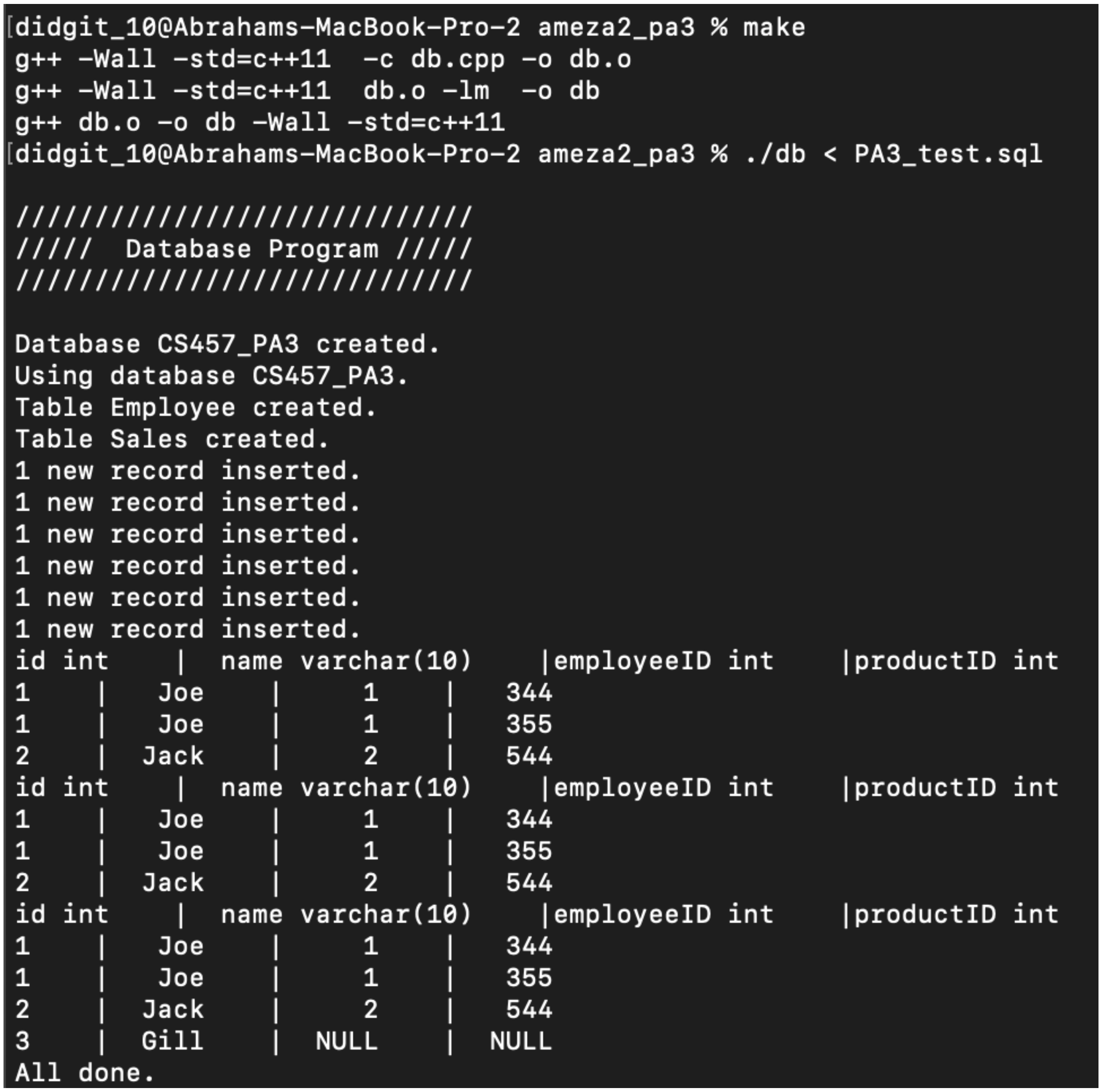
After verifying the validity of the user's input, for each file (vector for each file), the program will record each attribute's index, and use it to fetch each file's corresponding column values. After reading and fetching all relevant attribute value(s), the program uses the "on" condition argument to test each file's operand attributes against each other. Using Project 2's comparison algorithm, and nested for-loop to iterate over both operand vectors, the function generates a list of indexed tuple pairs that satisfy the given conditional statement, indicating which tuples need to be fetched (targetTuple vector for each file) (row-based). After determining which tuple pairs satisfy the "on" comparison, the program creates a temporary file called "copyTemp" to export the relevant table data. Using a nested for loop, the function then generates a combined table header, reorganizing each attribute based on the query order. Using the query order as a guideline to dictate the order in which data is represented, the program will export all intersecting tuples that satisfied the "on" comparison. innerJoin() has been implemented to use the left-side file (File 1) as the root reference to indicate which tuple lines need to be displayed. Based on the current lineIndex of the file, the program will reference targetTuple_1 to determine when a tuple should be outputted. If the current lineIndex matches a target tuple in File 1, then the program will fetch File 1's tuple data, then use the current index to find the matching tuple in File 2. After finding the matching tuple, the program will combine File 1 and File 2 tuple data, then print the combined values in query order (e.g., $targetTuple = (0,1) \rightarrow$ fetch tuple 1 from File 1, then fetch tuple 2 from File 2, combine, and print). Compared to queryTable(), innerJoin() outputs duplicate versions of tuple data if there exists a reoccurring matching value. As a result, innerJoin() introduces a numIterations counter that outputs duplicate tuples if there exists more than one matching value. If the current lineIndex does not match any of File 1's target tuples (no intersection), then current tuple in File 1 is omitted, exporting nothing. Once the relevant table has been exported, the program reads the updated temp file/table (getline()), and displays the contents to the user. After the fetching the relevant table information, the program deletes the temp file to promote data integrity (i.e., reading information, not writing information). It is important note that the program will automatically re-structure a default join command into an inner join operation through string manipulation.

Given that Outer Join and Inner Join operations conduct the same procedure to print out intersecting tuple data, outerJoin() uses the same algorithm as innerJoin(). However, outerJoin() must also print every row from a specified table. To determine the specified table, the function checks for a directional keyword in the user's join statement. Per project specification, outerJoin() has implemented a Left Outer Join operation, printing intersecting tuple data and all rows within File 1. If the user attempts to conduct a Right or Full Outer Join operation, then the program will prompt an error message, informing the user that the program does not support the given Outer Join type. To conduct a successful Left Outer Join, the program will output all intersecting tuple data using the same algorithm as innerJoin(). However, if the current lineIndex does not match any of File 1's target tuples, then the program will fetch the tuple data from File 1 and combine it with a set of initialized NULL values for each relevant attribute in File 2. The program will then combine the tuple values and print them in query order.

After testing the program against multiple test scripts and a few edge cases, I noticed that I had to implement some minor modifications to ensure that all project functionalities worked as intended. In Project 1, createTable() did not account for the table name and parameterized attributes to be attached to the same argument entry, creating an error as the program believed there existed an insufficient number parameters. To address the issue, a string modification algorithm was implemented to always insert a space character (' ') before the first occurrence of a starting parenthesis ('(') to indicate the starting point of parameterized data. The aforementioned solution will not affect commands that insert a space before parameterized data entries as the string is parsed based on words using a space character (' ') as the delimiter, disregarding empty words when identified. In addition to createTable()'s behavior, in Project 2, insertTuple() did not account for parameterized entries to be attached to the same argument entry. In other words, insertTuple() relied on the spaces after comma characters (',') to separate parameterized data. To address the issue, a string modification algorithm was implemented to always insert a space character (' ') after every comma character (',') occurrence, separating argument entries. The aforementioned solution will not affect commands that insert a space after each parameterized entry as the string is parsed based on words using a space character (' ') as the delimiter, disregarding empty words when identified. It is important to note that case insensitivity has been implemented into the project, accepting different cases for all user inputs except attribute names.

All in all, the program utilizes a variety of library functions (refer to Appendix [B]) to simulate an SQLite database. Using a tree-like structure, and user-input fail-safes, the program is well-rounded as it properly executes basic data manipulation, promoting database organization and efficiency.

Program Output:

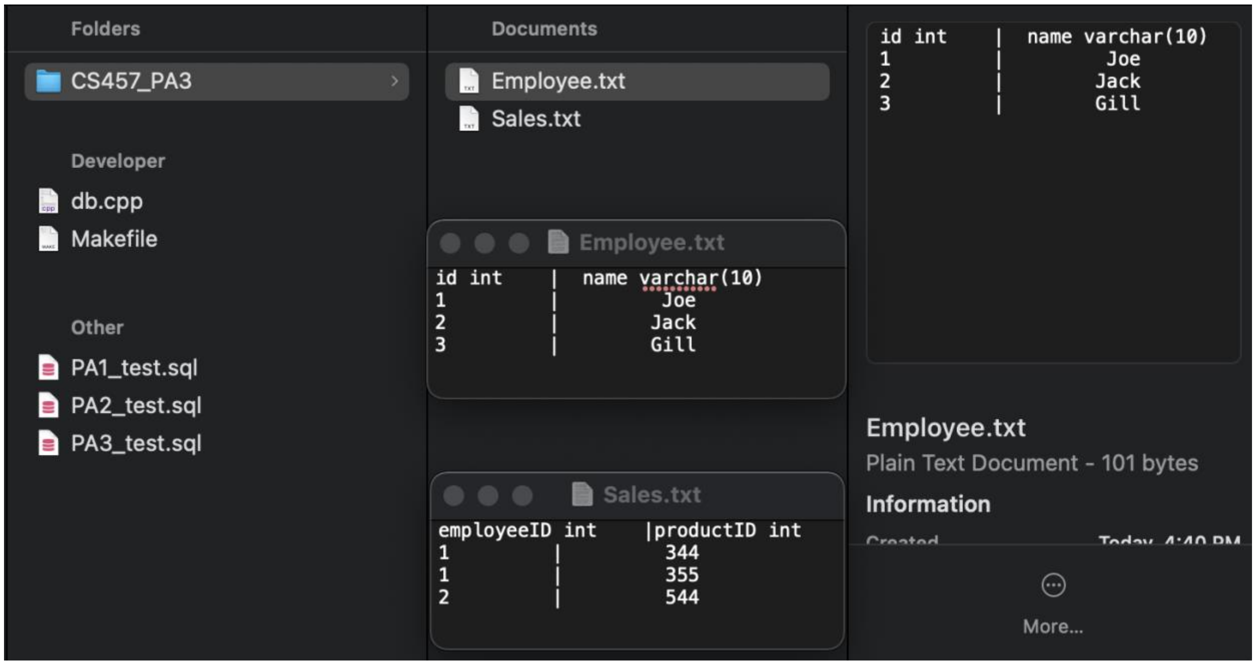


```
[didgit_10@Abrahams-MacBook-Pro-2 ameza2_pa3 % make
g++ -Wall -std=c++11  -c db.cpp -o db.o
g++ -Wall -std=c++11  db.o -lm  -o db
g++ db.o -o db -Wall -std=c++11
[didgit_10@Abrahams-MacBook-Pro-2 ameza2_pa3 % ./db < PA3_test.sql

/////////////////////////////
/////  Database Program /////
/////////////////////////////

Database CS457_PA3 created.
Using database CS457_PA3.
Table Employee created.
Table Sales created.
1 new record inserted.
1 new record inserted.
1 new record inserted.
1 new record inserted.
1 new record inserted.
1 new record inserted.
id int    |  name varchar(10)    |employeeID int    |productID int
1    |   Joe    |     1    |   344
1    |   Joe    |     1    |   355
2    |  Jack    |     2    |   544
id int    |  name varchar(10)    |employeeID int    |productID int
1    |   Joe    |     1    |   344
1    |   Joe    |     1    |   355
2    |  Jack    |     2    |   544
id int    |  name varchar(10)    |employeeID int    |productID int
1    |   Joe    |     1    |   344
1    |   Joe    |     1    |   355
2    |  Jack    |     2    |   544
3    |  Gill    |  NULL    |  NULL
All done.
```

File System View:



Employee.txt

```
id int      |   name varchar(10)
1           |          Joe
2           |         Jack
3           |         Gill
```

Sales.txt

```
employeeID int     |productID int
1             |         344
1             |         355
2             |         544
```

Employee.txt
Plain Text Document - 101 bytes
Information

Appendix A

CREATE:

- Database Creation
  - [FORMAT: keyword, structure, name]
- Table Creation
  - [FORMAT: keyword, structure, name, attr. name, attr. type, attr. name, …]

USE:

- Database Use
  - [FORMAT: keyword, name]

INSERT:

- Table Insertion
  - [FORMAT: keyword indication name values(attri.values)]

SELECT:

- Table Selection
  - [FORMAT: keyword query indication name (where attribute condition operand)]
- Joined Table Selection
  - [FORMAT: keyword query indication table_1 acronym_1 join_statement table_2 acryonym_2 on (attribute condition operand)]

Appendix B

Program Libraries:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <iomanip>
#include <cstring>
#include <algorithm>
#include <string>
#include <climits>
#include <regex>
```

Program Headers:

```
#include <unistd.h>
#include <sys/stat.h>
```