Deliverables

Please submit the files for grading via Moodle.

Pair Programming

You are required to work with a student partner on this assignment. Send me a message on Slack with your name and your partner's name for approval[1]. You must observe the pair programming guidelines outlined in the course syllabus — failure to do so will be considered a violation of the Davidson Honor Code. *Collaboration across teams is prohibited and at no point should you be in possession of any work that was completed by a person other than you or your partner.*

1 Introduction

In this assignment, we will collect voting data related to the 119th Senate, 1st session, and populate a database. This assignment will give you experience collecting and parsing XML data and creating/manipulating a database using SQL.

1.1 (10 pts) Required Reading

- ☐ Syllabus
- ☐ Textbook Chapters 1 (Introduction), 2 (Introduction to the Relational Model), 3 (Introduction to SQL).

Each member of the group should provide a statement indicating which of the readings above were done.

1.2 Material

Examine the page

`https://www.senate.gov/legislative/LIS/roll_call_lists/vote_menu_119_1.htm`

In this page, you see 659 links representing the issues voted by members of the 119th Senate, 1st session. In the left, under the "Vote (Tally)" column, you find links to individual voting sessions. Each voting session has an associated XML description. All voting session XML

[1]This is a hard constraint: if you don't send me the name of the group, you are not formally enrolled in the assignment.

files have been downloaded and are provided in the `XML.zip` file[2]. Create a new directory HW1, and extract the `XML.zip` file inside of it.

1.3 XML Structure and Semantics

Examine the file `vote_119_1_00597.xml`. The file represents a voting session (called simply `Vote`) in which multiple members (`Senators`) *cast a vote* on a particular issue.

In the top-level hierarchy, there is information pertaining the vote. Under the unique `<members>` tag, you have multiple `<member>` tags, each describing the vote cast by a particular senator on this vote.

A **Senator** is defined as having a first name, a last name, party, and state. Each senator also has a unique identifier, noted in the tag `<lis_member_id>`. This identifier is consistent across all files.

A **Vote** is defined as having a congress number, congress session, year, date, and a number x. A vote with number x in the appropriate $\{\text{congress number}, \text{congres session}\}$ uniquely identifies a voting issue.

Each **VoteCast** represents the vote cast by a **Senator** in a specific **Vote**, along with his/her voting option: a *Yea*, a *Nay*, and, for our purposes, an *Absent* (which captures any other scenario, including the "Not Voting" indication found in the XML files). So, a **VoteCast** has an attribute referencing a senator, and attributes referencing a vote (congress number, congress session, vote number). It also has the value 'Y', 'N', or 'A' (for absent/not voting) representing the Senator's actual voting option.

It is part of your task to understand the structure of these files.

2 (20 pts) Database Schema

Deliverable 1. Create a file called `Schema.sql` that performs the following:

1. Creates a schema called "SenatorVotes", and makes it the default schema.
2. Creates all the relations representing Senators, Votes, and Vote Cast, paying attention to the definition of Sec. 1.3.
   (a) Name your relations and attributes consistently.

[2] Curious about how those files have been downloaded? See the `download.py` script along the XML files.

(b) Carefully identify the domain for each attribute (integer? decimal? strings?), as well as the number of bytes allocated for representation.

(c) Indicate **all** the primary keys (using `PRIMARY KEY` in SQL) and foreign keys (using `FOREIGN KEY ... REFERENCES` in SQL) in each relation.

3 (25 pts) Populating the Database

Deliverable 2. Write a parser using Python that navigates over all the voting files in the XML directory and generates a SQL script that inserts Senators, Votes, and vote cast tuples into the database you created above.

The Senator information (ID, name, party, …) is duplicated across different votes. Your script must perform **exactly** 102 SQL insert statements to insert Senators into the appropriate table[3], **exactly** 659 SQL statements to insert Votes into the appropriate table, and **exactly** 65891 SQL statements to insert each vote cast into the appropriate table.

You **must not parse** the names, party, and state of a Senator that has **already been identified** in a previous file. You can decide if you have seen a senator by maintaining a data structure containing the IDs of the previously seen Senator. **The choice of the data structure is absolutely crucial and it is worth many points here: do not choose a data structure that perform queries in $\Theta(n)$ time.**. If you are in doubt about this statement, please see the professor in office hours (you are going to obtain your answer, but I will have the opportunity to make sure that you understand how crucial this choice of data structure is and what are the design choices).

4 (25 pts) Playing CSV Tennis

Please look at the dataset in:

`https://github.com/JeffSackmann/tennis_atp/blob/master/atp_matches_2024.csv`

We often need to model database schemas based on functional requirements specified by application users. However, application users typically describe their functional requirements very informally, and it is up to the database designer to eliminate any ambiguities in the system description as the model is created.

[3]You may have more or less than 100 because of substitutions.

Deliverable 3. Create an Entity-Relationship model (in a PDF file called `ERTennis.pdf`) of a database to manage the information about the tennis matches using SQL, based on the informal requirements below:

1. List all players, tournaments, matches. Identify reasonable attributes for players, tournaments, matches. In addition to the obvious arguments, matches should store the score, number of sets.
2. The statistics about winners and losers, provided in the CSV files, refer to players in the match. We would like to obtain aggregate statistics about a specific player, for example: "how many double faults player X had, in average, on matches won between 1971 and 1975". You should *not represent* this particular information, but your model should be able to *support* this kind of queries.
3. Aggregate statistics about rank, hand (leftie vs. non-leftie) should be supported. For example: "which percentage of matches in tournaments with draw size bigger than 64 were won by lefties?".

Note that the CSV file might induce you that information about the winner is an attribute of the match. It is not: a winner is a player, and that player should be identifiable, and aggregate statistics for that player for multiple matches should be attainable. Do not store a player's age multiple times, at multiple matches. A player should have a derived attribute *age()*, which should be calculated dynamically from his/her date of birth. So, do not plan to store information about a player's age. Do not forget to identify attributes and primary keys in the entities. Finally, there is no need to store information about seed entries.

The description above is certainly vague and even open-ended. I am precisely evaluating your ability to parse such kind of specification, using Entity-Relationship in order to produce a working model that fully supports the queries requested by users. Vagueness and open-endness are part of the problem. **Consulting the professor in office hours to check the model is allowed and encouraged.**

5 (20 pts) Schema and Integrity Constraints

Deliverable 4. You should create a database schema (in a SQL script `TennisSchema.sql`) that is a physical realization of your model above.

Make sure to identify all primary and foreign keys. Also, the IDs in the CSV files are not something we can reasonably trust (for example, see the description for `match_num` in the `matches_data_dictionary.txt` file. Make a web search for `AUTO INCREMENT` and be amazed.

6 Style

I will allocate up to **30 points** to style. The SQL scripts should be well-indented and organized. The entity-relationship models are drawn neatly. Every new function you create should have comments. Variables and methods should be named consistently, and follow a standard coding conventions. The code should be well-indented and organized (use Cmd-P → Format in VSCode) to auto-indent your code). There are no absolute file references (so no "/Users/elmo/College/Fall37/..." anywhere in your code).

Good luck,
- Hammurabi