ECE 356 Final Project Report

Car Dealership Database

Pratham Thukral
Simon Yan
Leon Zhu

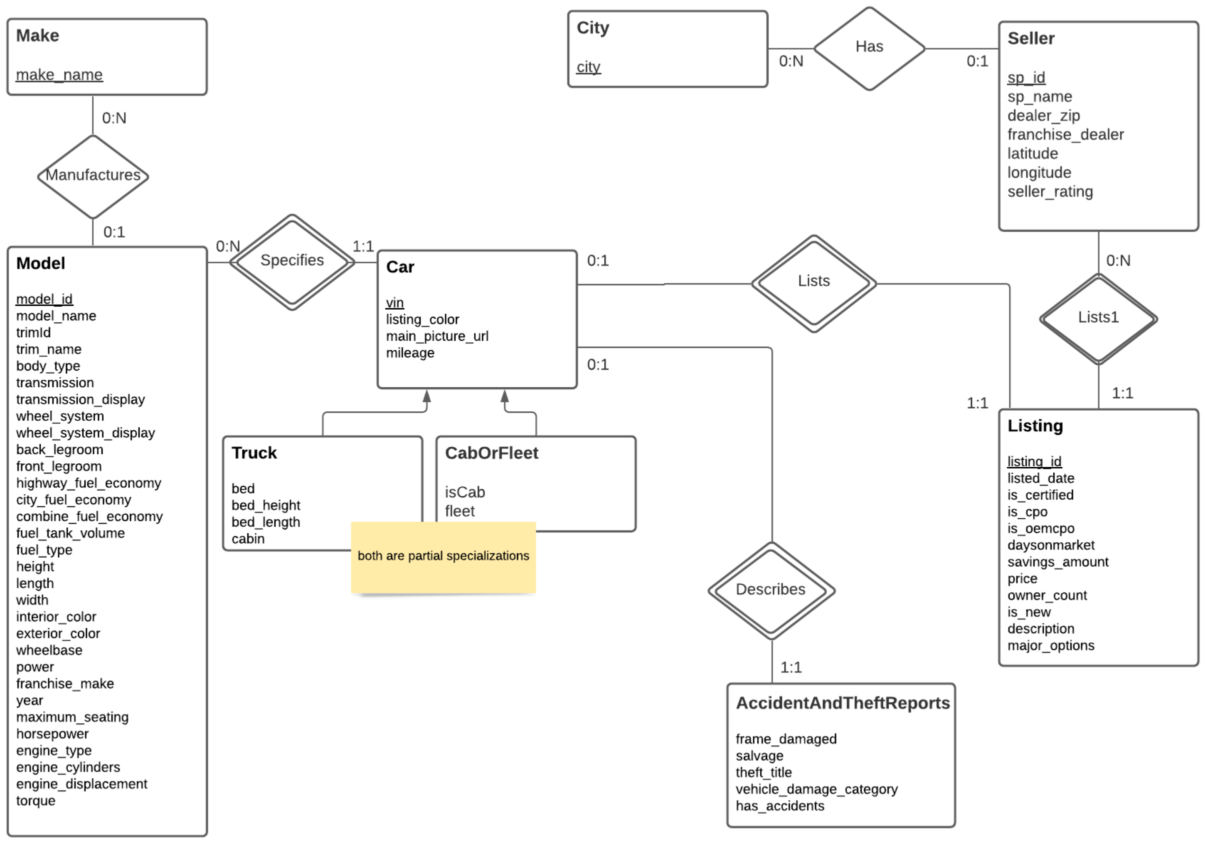
Github Repo: https://github.com/SimonYanGit/ECE356-project

Introduction

The overall idea behind our project was to turn a csv of 500,000 unique vins and listings into a database that could be used by various dealerships to manage their inventory. There is functionality to search for vehicles using various filters, create new listings for vehicles, modify existing listings, and delete listings once a vehicle is sold.

The “UI” of the project is text-based and was created with python. The MySQL python libraries were leveraged to connect the “front” and “backends” of the project. They are both described in further detail throughout this report.

ER Diagram



Entity set Design choices:

- Our choice for a starting place was the Car entity set; which detailed some specifics about the car itself; ie. the vin, colour, url to photos, and mileage. These were not placed in the Model Entity set as we assume these are specific to the car and not the model, which would have details about a set of cars.
- The Car entity was also a generalization of 2 partial specialization sets: Truck and CabOrFleet. As the name suggests, the Truck set was for any vehicle that was also a truck, thus it may have information about the bed and cabin. The CabOrFleet would have booleans regarding whether that car was found to be in a fleet and/or cab service.
- These are partial since they do not form the entire car set, but merely a subset of cars.
- The Model entity set was created to define the model of a car; ie. the description from the manufacturer. This included the name, trim, body_type, transmission, and many more descriptors. The PK was generated by concatenating all of these columns then hashing them. This was chosen because 2 models may have had the same name, same trim, but different transmissions, etc. This was causing conflicts with the PK generation, so one was created.
- A city “has” 0 or more sellers, each seller has 0 or more listings and each listing has exactly one seller.
- The “users” in our application will be the sellers - their unique ID is the sp_id.

Relation set Design choices:

- Many of the relation sets are weak as they rely on the existence of another entity set. For example, the Car-Model relation (the Specifies relation) is weak because there cannot exist a car without a model; 1:1.
- Another weak entity set is the Listing-Car set which requires that every Listing has exactly 1 car associated with it, however there can still be cars that are not for sale; ie. do not have a listing.
- The Make-Model relation is strong because there exist some models with no make, and conversely, there exist some makes with no models.
- Likewise, there can be online or remote sellers that are not associated with a city, and there can also be cities with no sellers. Since there is no dependence, this is a strong entity set.

Pre-Processing

The csv given was 3 million rows, and many of those rows had “messy” data. The csv was shortened to 500,000 rows and some of the cells were processed in order for the SQL LOAD function to work as intended.

- Large integers were converted to their scientific notation (eg. 001000 would be parsed as 1e+3). This was causing an issue because a vin of 001000≠1000 but the scientific notation was causing a PK not unique collision.
    - This was fixed by explicitly converting all of these cells to strings in the csv using double quotes

- Some of the description cells were full sentences so they used commas, backslashes, quotation marks, etc. All of these punctuations were causing sql to parse the comma-separated file incorrectly.
  - This was fixed by replacing all the commas with semi-colons, then switching back in the actual sql table creation scripts where necessary. The backslashes were turned into forward slashes, and the double quotes were turned to single ones.
- We decided to treat all the incoming data as varchar (and TEXT for some larger cells), the empty values from csv were being converted to empty strings: ‘’
  - This was fixed by replacing these back into SQL NULL values when inserting records into our structured tables.
  - Additionally we parsed ‘TRUE’ to boolean trues, as well as other float and integer cells

Database Structuring

1. The initial step was to clean and shorten the csv then upload it to a local MySQL instance using the LOAD INFILE command.
2. Once complete, we had a table (called fullcsv) with all of the csv values as varchar/text objects.
3. We then followed our ER diagram by creating all of the remaining tables, relations, primary keys, and foreign keys.
4. Once these were generated, they were populated using the fullcsv table. They were populated using relevant datatypes, and any missing values were converted to NULLs.
5. Finally, we added indexes to various columns that we were commonly joining and filtering on in our application. More specifically:
   a. body_type
   b. year
   c. maximum_seating
   d. engine_type
   e. listing_color
   f. mileage
   g. listed_date
   h. price

Application UI

Our application was created using Python with the help of the MySQL connector library. Our tool is command line based and is intended for the dealerships to use.
The tool begins by asking the user where they would like to Search, Modify, Create, or Delete.

Search: The search tool brings up a list of various filters the user can use to select a set of vehicles/listings. For example the city, price, make, body type, colour, seating capacity, etc. The user can also select to sort the data (asc or desc) by the listed date, price, or year. All of the user inputs are logged and parsed into a select query that pings the database. The many joins needed to consolidate all this data are completed within a search_view, so the python script just “select * from search_view where…”

Modify: This function asks for a specific listing id and allows the user to change some of the parameters of the listing such as the description, certified (true or false), savings amount, price, major options, etc. These are then parsed into an UPDATE query that updates the listing table. Since there are proper foreign key and cascade options set, the relationships will update based on the listing id that is deleted.

Create: This is by far the most complicated function as there is a lot of necessary data input from the user. The user is forced to enter the vin, colour, new listing id, city, etc. and then they can enter additional column data like description, is new (true or false), owner count, etc. All of these inputs are parsed into 11 different INSERT INTO statements to insert all the necessary data across the database.

Delete: The delete query asks for the listing id in question and proceeds to remove that from the table. Since there are proper foreign key and cascade options set, the relationships will delete based on the listing id that is deleted.

Test cases

- Create a record: we create a new listing using our tool and check if all the tables are updated correctly
- If a row is created but it conflicts with unique, null, or primary key, or foreign key constraints a sql error is thrown and that row is not inserted
- Search for the row that was inserted using the search functionality of our app. This meant using the listing id directly to filter for the correct listing or using various other filters in our code.
- Delete a listing if a car is sold. To verify if this worked we used select queries to see if the rows were actually deleted from the parent table, but also the children table since we were using cascading on our foreign keys.
- Modify an existing record using the tool and selecting to see if that row was edited correctly.

Conclusion

In conclusion, this project taught us a lot about the structuring of a database and the skill needed to build an efficient one. Some of the key takeaways are the importance of indexes for runtime and foreign keys for data management. We do have some potential improvements if given more time:

Data Validation: The code for the UI did not account for certain data validation such as type checking and length

UI Design: If we were to rewrite the code, one thing that could be changed is taking a more object oriented approach around the columns and tables of the sql database. This would create less overhead when specifying which columns that the different CRUD functions were interacting with.

Drop downs: Some of the options available to the user could be converted into drop downs as seen on used car websites. For example, body_type could be turned into a field where the user selects a dropdown list of options rather than typing it in. This would reduce the number of data entry errors.

Frontend: Not required for this project, but a nicer frontend would be a worthwhile investment.