

FunFriends33

CS 327E

Milestone 7

Ranger Motors Database Report

Introduction: Our group, FunFriends33, saw the need for a program that would allow its user to discover summary information about the inventory of a local used car dealership, Ranger Motors Austin. To fill this need, we created a database model and used Python to scrape the dealership's website to acquire information on the inventory, clean and organize the inventory data, store the data in a MySQL database, and create an interface allowing a user to query the database. Here, we describe the background and purpose of our project and our solution, then detail the process of creating each of the above components of the program. We conclude by evaluating the final version of our project and reflecting on our development process.

Background/Motivation: Our project idea arose from the following observation: while Ranger Motors' website allows inventory filtering that is helpful to a customer hoping to buy a car, it lacks some features that would be desired by a person hoping to learn about the properties of the inventory as a whole. Our hypothetical user, then, is someone who is interested in describing the inventory as a whole in terms of aggregations like summary statistics. This might be an employee of Ranger Motors who wants to employ marketing analytics to see what car features are most profitable, or perhaps a competitor hoping to discover the differences between Ranger Motors' inventory and their own.* Whereas the Ranger Motors website allows users to narrow

* Some suspension of disbelief is required to accept that, to meet our stated goals, we had to scrape the website of Ranger Motors Austin. If the solution was indeed meant for the use of Ranger Motors' employees,

down the results until they find a *single* “perfect car,” we want to allow our user to run statistics on customized *groups* of cars. This could include finding the average price of all white Fords, or the number of cars from 2007 that have less than 100,000 miles on them. Such statistics cannot be found using the website’s inventory search. This, then, was our motivation for creating a data pipeline to store the inventory data and an interface to allow users to run custom queries on the database.

Solution/Process: Having defined our problem, we began to outline our solution. Naturally, we began by fleshing out the functionality described in the previous section. In addition to filtering results, we wanted our user to be able to sort results by attributes of the cars, choose the attributes to display in the results and, most importantly, group cars on certain attributes and display basic statistics (e.g., count, max, avg) for each of the groups. Further, we sought to extend the customizability of the attribute filtering, allowing user-entered values to be used to filter both the numeric and text-based attribute values.

We planned for each of the components of our project: we would start by using BeautifulSoup to scrape Ranger Motors’ website for inventory data. Next, we would use Python to clean and organize the data and store it in a MySQL database with PyMySQL. Finally, we would create a text-based user interface in Python to allow custom queries to be run on the database. In an effort to minimize the knowledge required by the user, we made sure that no knowledge of SQL syntax was required to make use of the program, although knowledge of relational database concepts are to some extent necessary in order to understand how to interact with the program.

for example, we would surely have had direct access to the original database upon which the website is based. However, given our lack of such access, we ignore the oddity of this circumstance and simply treat this web scraping as a necessary step in the gathering of our data.

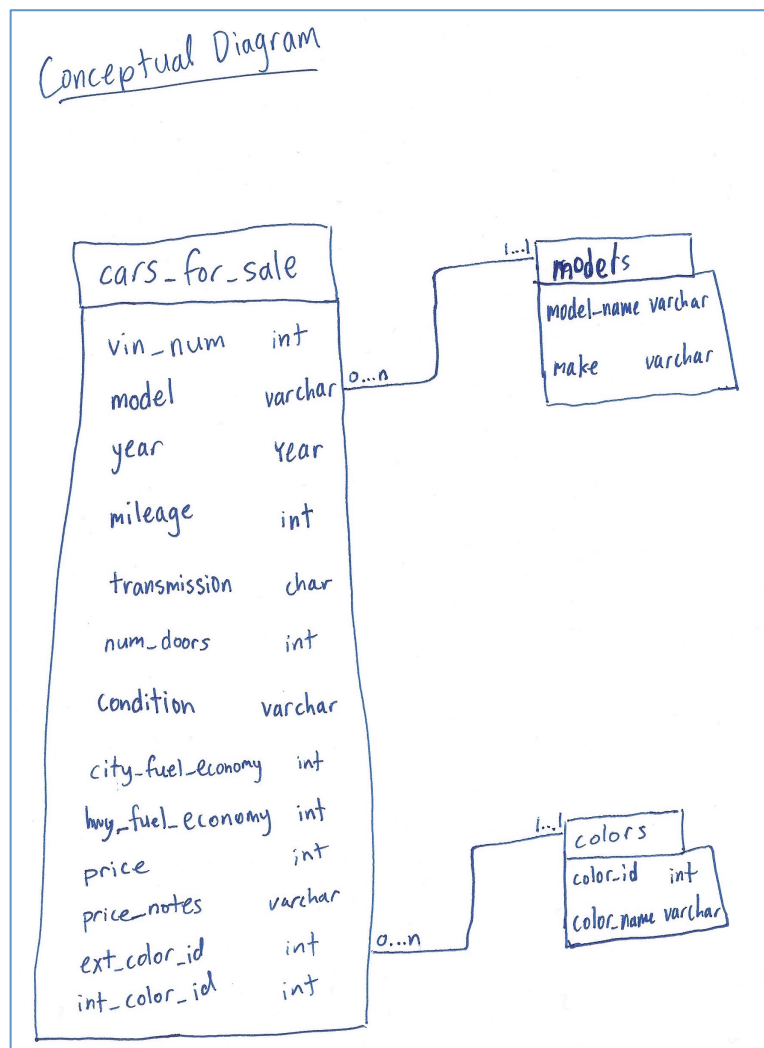


Figure 1: Original Conceptual Diagram

Database Design: Figure 1 shows our initial conceptual diagram for the Ranger Motors Database.

Originally our ERD consisted of three tables. The main table in this database is the CarsForSale table, which contains the bulk of the information on the inventory. Each record in this table corresponds to a vehicle in the Ranger Motors inventory, with attributes that describe the car. The primary key here is the VIN (vehicle identification number), as every vehicle made has a unique VIN.

Our original diagram had two additional tables, the first being the *Makes/Models* table (shown here as *models*), to which the CarsForSale attribute *model* is a foreign key. We added this table to ensure that our CarsForSale table was in third normal form. Since we took *make* to be functionally dependent on *model*, we had to create another table to achieve this, hence the existence of the *Make/Models* table. Having our tables in third normal form was meant to reduce redundancy in our database.

CarsForSale
VIN
Year
Make
Model
Body_Style
Mileage
Transmission
Engine
Exterior
Interior
Doors
Stock
Fuel_Mileage_City
Fuel_Mileage_Hwy
Conditon
Price

Figure 2: Revised Conceptual Diagram

The last table in our original diagram is the *Colors* table, to which the *CarsForSale* attributes *ext_color_id* and *int_color_id* are foreign keys.

This table consists of color identification numbers, which are referenced by the external and internal color attributes for the vehicles, each of which matches the name of a color, in string form. We added this table for the purpose of validation:

given that many different colors exist, using color IDs as the *CarsForSale* attributes, rather than the color names themselves would ensure that all

values input matched to legitimate colors. This

would prevent erroneous input, so that we could not accidentally end up with “multiple versions” of a given color, e.g., “red,” “Red,” and “redd”, resulting from multiple inputs for a color string.

Figure 2 depicts our revised conceptual diagram, which consists of a single table. While the main table from the original diagram remains in the revised edition, the *Makes/Models* and *Colors* tables were removed. We decided to remove the *Makes/Models* table because having the table really did not add any value to our database. The avoidance of redundancy may have been useful had we been dealing with a much larger database, but redundancy in our database is not an issue; the Ranger Motors inventory consists of far fewer than 100 vehicles, which means that the *CarsForSale* table has less than 100 records. We concluded that the addition of the *Makes/Models* table added complexity to no clear benefit.

We also removed the *Colors* table in the revision of our model, as we found validation to be unnecessary. Validation of input for the color attributes would be useful if our data was being manually entered. However, given that we are storing our data automatically using a web-scraping program, the use of a validation table would have no purpose. When checking values against the validation table, our only option for a value not in the table would be to either add it to the table or leave the value as Null. Either way, using such a table would not benefit our database. So we decided to leave the *Colors* table out of our model.

These revision decisions leave us with a model consisting of a single table. This raises the question of whether using a relational database makes any sense for this project. Aside from the obvious factor (that this project is for a Databases course), we think that using a relational database, though not strictly necessary, makes sense here. While the information, as is, does not require a relational model, we can only benefit from having the flexibility to add relational components in the future. For example, the use of a relational database, as opposed to spreadsheet software, leaves us the possibility of adding inventory information for other Ranger Motors Dealership locations. Moreover, we retain all of the necessary aggregation and statistical functionality in using a relational database, so using a spreadsheet or other software would not give us any additional benefit in that sense. Thus, we believe that our use of a relational database is legitimate.

Web Scraping: We scraped the Ranger Motors Austin website to get all of our information on the vehicle inventory. The website has an inventory page that is structured such that it contains the listings for each of the vehicles with some summary information and a picture for each vehicle. The inventory page is shown in Figure 3. Each of these can be clicked (i.e., are

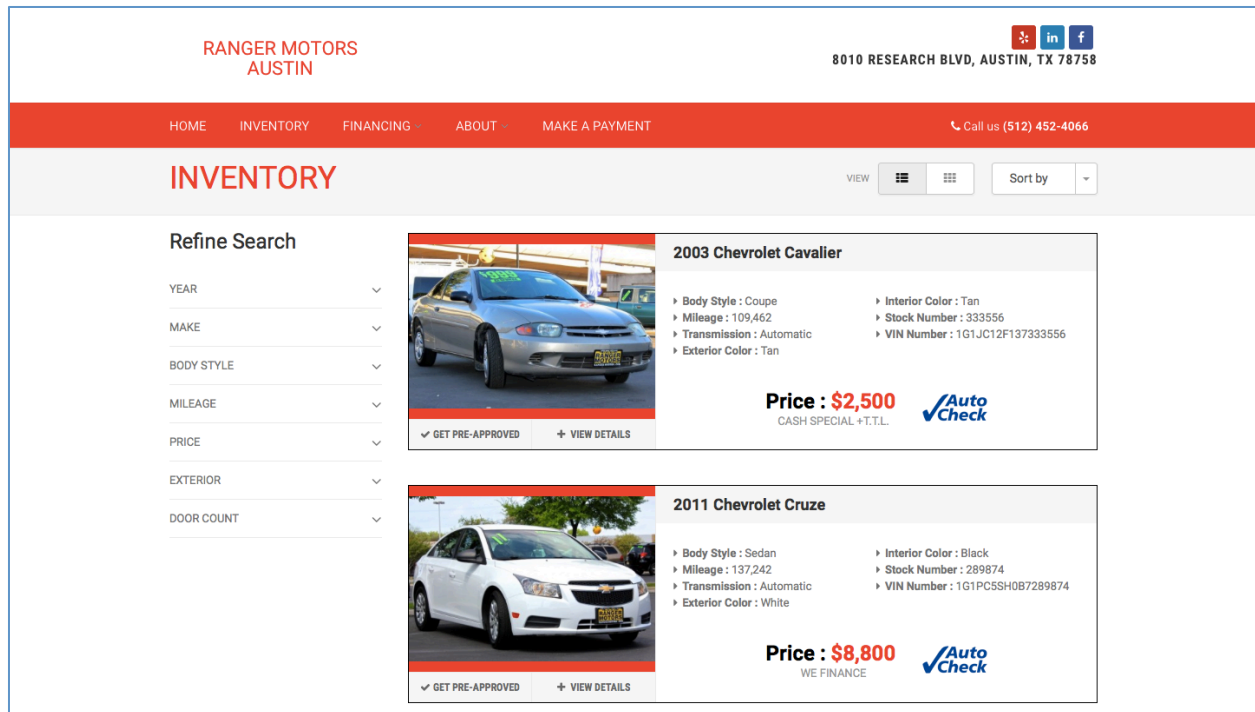


Figure 3: Ranger Motors Austin Inventory Webpage
(<http://www.rangermotorsaustin.com/inventory/>)

hyperlinked) to the individual webpages for the respective vehicles. The individual vehicle pages have more pictures and more complete information on the vehicle. An example of one of these pages is shown in Figure 4.

We used Python’s URL Library with the URL of the Inventory page to get the html and then used BeautifulSoup to parse this html and find the URLs of all of the individual vehicles’ webpages. This was simple once we discovered that a class, “inv-view-details,” existed for such links: we used BeautifulSoup to find all hyperlinks with this class tag. Once on a particular vehicle’s webpage, we had to retrieve information from a few different places. As is clear from Figure 4, most of the attribute information is contained in a table. We got the Attribute names, the left side of this table, by finding matching items with the “list-group” class with our BeautifulSoup object. Then we got the particular values for these attributes by recognizing that the values had the class “list-group-item.” We created Python dictionaries for each vehicle listing




in order to store each of these key-value pairs. Using dictionaries resolved the issue that not all listing pages have tables that match exactly on their Attributes information; dictionaries allowed us to store the Attribute names and values that appeared on each page, rather than having to hard-code a pre-defined set of Attribute names for the database input.

In addition, we had to get the price of the car from the bubble at the top right. This required us to search for a separate class, “details-price,” which we then parsed in Python to get the integer value for the price.

PyMySQL/Database Creation: To prepare the data for insertion into a MySQL database, we first had to finish organizing it. As we mentioned in the previous section, we had to parse some values such as the vehicles’ prices, and we used Python dictionaries so that attribute names could be pulled directly from the webpages. This allowed us to easily discover which vehicles were

RANGER MOTORS
AUSTIN

8010 RESEARCH BLVD, AUSTIN, TX 78758




HOMEINVENTORYFINANCINGABOUTMAKE A PAYMENT

Call us (512) 452-4066

2011 Chevrolet Cruze

PRICE : \$8,800WE FINANCE

Schedule a Test-driveContact UsGet Pre-approvedOffer Trade-InShare listingPrint



Year	2011
Make	Chevrolet
Model	Cruze
Body style	Sedan
Mileage	137,242
Transmission	Automatic
Engine	1.8L 136.0hp
Drivetrain	FWD
Exterior	White
Interior	Black
Doors	4 Doors
Stock	289874
VIN	1G1PC5SH087289874
Fuel Mileage	26.0 City / 36.0 Hwy
Condition	Used

Figure 4: Sample Webpage for Individual Vehicle

missing which attributes, i.e., to find out which values should have Null entries in the database. Once we identified these entries, we set their dictionary values, matching the corresponding Attribute key, to “NULL,” which allowed us to identify null values when inserting records into the database, and to leave out those values accordingly. Several other minor issues required fixing. For example, to automate entry of scraped Attribute names into the database, we had to replace spaces with underscores.

Having organized our data, we then created our database and table, and formed SQL Insert statements for each vehicle listing/record in the database. This consisted of iterating over every key-value pair in each of the vehicle dictionaries and formatting that data to create a SQL Insert command. Executing these statements in PyMySQL, with some additional information such as setting VIN as the primary key, resulted in the population of our database.

Before running the query interface, our program acquires some information about the user’s MySQL connection to set up the database. First, we get the user’s MySQL login information to set up the database connection in PyMySQL. Then we ask whether the user has already created the database and table used by the program. If the user does not have these, we create them for the user. This interaction allows us to prevent the error that would occur if we tried to create a database or table that already existed on the user’s MySQL connection. After this setup is complete, we run the query interface.

Query Interface: We designed our user query interface to meet the needs of our expected user, as described above. This means a focus on aggregation statistics with customizable filter conditions. We aimed to develop a user-friendly command-line interface. To this end, we included text-based descriptions of the process to help the user understand the program. The

program welcomes the user, then builds the query one statement (as in “select statement,” “where statement”) at a time, based on user input. The user is not required to know any SQL, as the program requests the necessary information from the user and worries about SQL syntax “behind the scenes.” However, knowledge of the concepts of relational databases/SQL would be useful, so that the user has a better idea of how to compose aggregation queries. For example, a user naïve about *Group By* and *Having* clauses might erroneously attempt to *Select* an attribute A and *Group By* another attribute B, such that the aggregation is unable to display a single value of A for each B value, resulting in an error. Though we catch this error and report to the user that the query was faulty, the fact remains that some SQL knowledge would be helpful in the creation of a valid query.

After welcoming the user, the program displays the main table’s list of attributes, and prompts the user to choose the attributes to display in the results, to build the select clause.

Next, it asks the user whether he would like to place any conditions on the attributes in order to filter the results (where clause). If so, the conditions are acquired from the user by a prompt that, depending on whether the attribute is numeric or string-like, gets different input from the user. For numeric attributes, the user may choose either to place an equality condition or a boundary condition. If he chooses a boundary condition, he can further choose whether to place an upper bound, a lower bound, or both, and to make these bounds either inclusive or exclusive. Finally, he is prompted to enter the values for the boundaries/equality. For string attributes, the user is asked to enter a string, and then can choose whether he wants the attribute value to be strictly equal to the string, or to merely include the string somewhere in the value.

After the where clause is created, the program shows the user the possible aggregate functions for the query, and allows her to choose whether and which of these to use. If the user

includes aggregation functions, she is asked for the attribute to which the aggregation should be applied and the attribute by which the aggregation should be grouped (group by clause). The user can also specify conditions for grouping (having clause), which are handled in the same way as are the where clause conditions, described above. The aggregations are automatically added to the select clause of the query.

Finally, the user can choose to order the results by any of the attributes and is able to specify whether the ordering should be ascending or descending (order by clause). After this, the query is complete. The program combines each clause of the query, and executes the final version through PyMySQL. The results are then displayed to the user.

Updated Features (since Demo): Since the demo of our project, we have made some updates to make our database more maintainable, as well as to fix some cosmetic issues in the query interface. First, we redesigned our data pipeline so that our database could be stored initially and then updated. Whereas, originally, the entire database would have to be re-scraped and downloaded each time the program was run, the updated pipeline only checks for updates to and new listings in the inventory updates the database to reflect these changes/additions, and leaves the rest of the database untouched. Further, we have added an attribute to the *CarsForSale* table, *DateAdded*, which keeps track of when a listing was added. This gives us more historical information on the inventory. Together, these updates to our project make the database more realistic and maintainable.

We also updated the query interface in places, the most significant revision being in the display of the query results. The original results output was messy and not very descriptive. The revised version shows attribute headers and is thus more readable by the user.

Reflections: Overall, we are pleased with the outcome of our project. Foremost, it solves the issue that we set out to solve with this project; it allows its user to attain an overview of Ranger Motors Austin's Inventory by allowing him or her to discover statistics about the inventory in a customizable, easy-to-use manner. While the project has some existential problems, (e.g., "was a relational database really necessary/the best solution to the problem?", "why did we have to scrape the webpage for this information?"), we think these are minor, and that the program contributes real value, insofar as it meets the demands of our stated goal.

We found this project quite instructive with regard to the practicalities of database design. Dealing with our project's "existential issues," as stated above, actually served to teach us how to think about when a relational database is appropriate. In this sense, the practical aspect of this project helped us put the theoretical material of the "Database Design" part of the class into perspective. Initially, we rushed to think of everything in terms of relational databases and to put everything in 3rd normal form. After dealing with the practical aspects of this project though, we got a better sense of how these concepts come into play. Relational databases are useful in some situations but may be overkill in others. Standards of database design, such as Normal Forms, should be seen in light of the costs and benefits they add to the database design. As there are not necessarily clear-cut answers to such considerations, we think that the process of developing this project has helped us learn how to appropriately weight our options under such considerations.