ECE356 Project Report

Yassine Elhedhli, Matthew Leung, Mike Zhang

2023/12/06

Group 66

Table of Contents

Introduction

For our project, we decided to use the Traffic Accidents Dataset[1]. This dataset features 3 main CSV files: Accidents0515.csv, Vehicles0515.csv, Casualties0515.csv, as well as numerous context CSVs, providing look-up tables for specific attributes of the main CSVs. Some context CSVs were missing from the dataset, for example Weather_Conditions.csv. However we were able to recover the missing look-up tables from the UK government website[2]. The rest of this report details the challenges and design choices we faced with each component of the project.

ER Design

Figure 1 shows our entity-relation diagram. We decided to approach our ER diagram by basing our design off how things are related in the real world, rather than how they were represented in our given dataset. It is for this reason that we split appropriate attributes of Accident into the separate entities Road, Location, LocalAuthorityDistrict, LocalAuthorityHighway, Junction, PedestrianCrossing, and PoliceForce. We also omitted the lookup tables for attributes that didn't make sense as their own entity, for example it wouldn't really make sense to include roadType as its own entity since it really only exists as an adjective of a road. We also decided to exclude certain attributes if they didn't provide any additional or relevant information. One such attribute was ageBandOfCasualty. This attribute didn't provide any additional information since the Casualty entity already had the ageOfCasualty attribute. For the Vehicle entity, we decided to create a separate entity called Driver to hold attributes about the driver of the vehicle. Again, this was done to model the real world relations between Vehicles and Drivers. We also noticed that certain attributes of Casualty were only valid if the casualty was a pedestrian, so we created a partial specialisation of Casualty called PedestrianCasualty. Lastly, in the context of this dataset, Vehicle and Casualty only exist if they are linked to an accident, otherwise they wouldn't exist in the dataset. Because of this, we treated them as weak entity sets with vehicleReferenceNumber and casualtyReferenceNumber as their discriminators.

---

[1] https://www.kaggle.com/datasets/silicon99/dft-accident-data

[2] https://data.dft.gov.uk/road-accidents-safety-data/dft-road-casualty-statistics-road-safety-open-dataset-data-guide-2023.xlsx

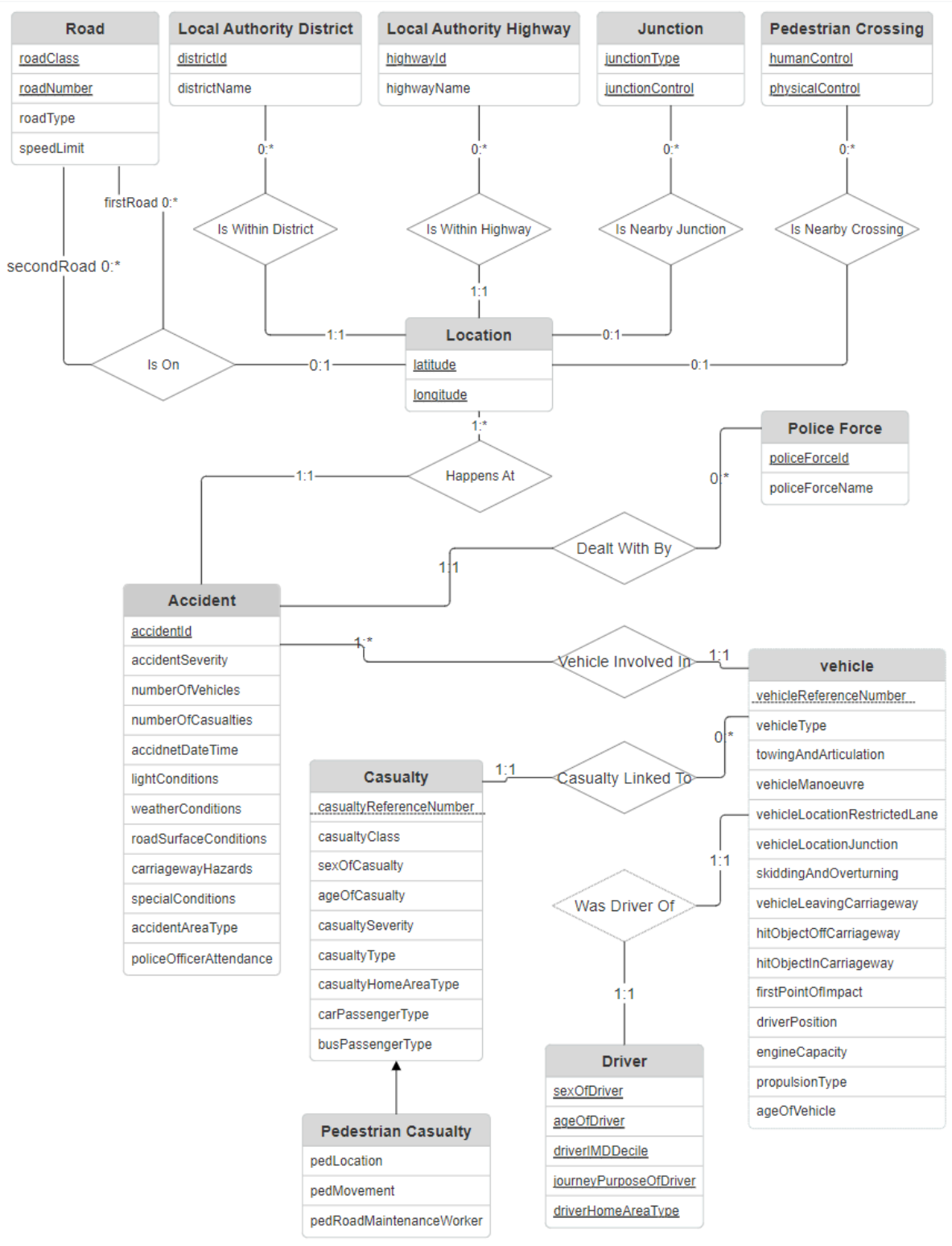

Figure 1. Entity-relation diagram.

Relational Schema

Introduction

When approaching our relational schema we chose to take a more practical approach in designing it over the puristic approach we had when creating the entity relation diagram. We chose to emphasise three qualities when designing the relational schema:

1. Performance
2. Ease of use and simplicity of queries
3. Simplicity of mapping dataset to the database

Translating ERM to RDM

The first step in developing our RDM is dissolving the 1:1 relations and absorbing them into their respective entities. We can do this with the following relations:

1. HappensAt relation linking the Accident and Location entities
2. IsWithinDistrict relation linking the Location and Local Authority District entities
3. IsWithinHighway relation linking the Location and LocalAuthorityHighway entities
4. DealtWithBy relation linking the Accident and PoliceForce entities
5. VehicleInvolvedIn relation linking the Vehicle and Accident entities
6. CasualtyLinkedTo linking the Casualty and Vehicle entities
7. WasDriverOf linking the Driver and Vehicle entities

We can see the greatly simplified ER diagram in Figure 2 below. We are now left with the question of what to do with the 0:1 relations as it is also possible to simply absorb this relation into our entities, although this does mean that some of our entities will then have optional fields which can be considered to be undesirable. Due to the nature of our dataset where nearly all data is already linked to a specific instance of one accident and are not true standalone entities with their own identifiers, we decided to go with this approach. This simplifies our queries and improves performance as we avoid unnecessary inner joins that would have to exist on nearly every query. We can further remove the following relations:

1. IsOn relation linking the Location and Road entities
2. IsNearbyJunction linking the Location and Junction entities
3. IsNearbyCrossing linking the Location and Pedestrian Crossing entities.

This latest round of simplification leaves us with no relations in our ER diagram, as can be seen in Figure 3. Now, we must decide whether to combine the schemas of any of the entities involved in this series of simplifications. Once again, due to the nature of our dataset it makes sense to combine the following schemas:

1. Location absorbs the Road, Junction and Pedestrian Crossing schemas
2. Accident absorbs the Location schema
3. Vehicle absorbs the Driver Schema.

You can see the result of these absorptions in Figure 4.

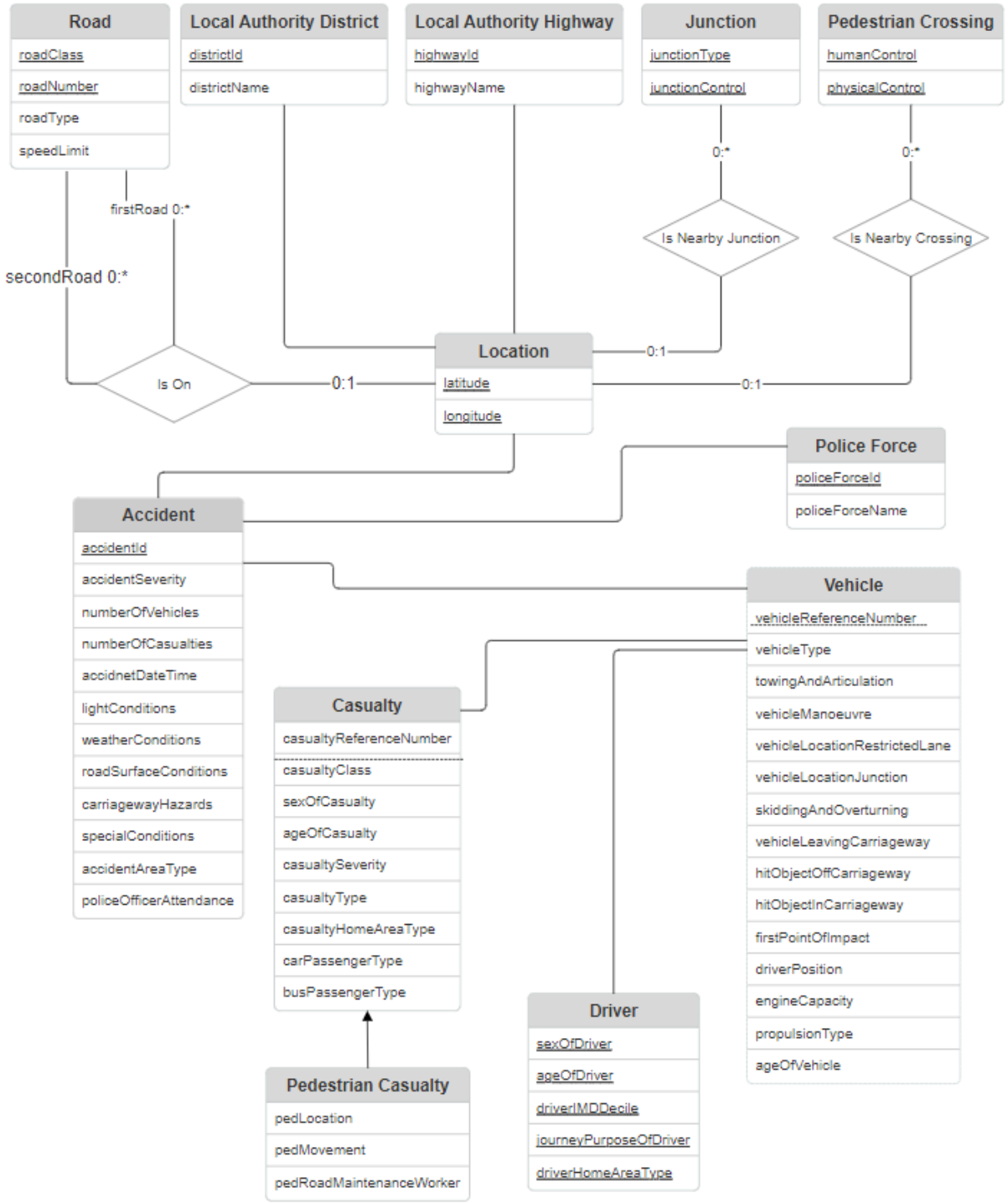

Figure 2. Entity-relation diagram after 1:1 relation simplification.

Finally, the last question we have to answer is what to do about the Pedestrian Casualty specialisation of the Casualty Entity. In our case we decided to introduce optional fields into the Pedestrian schema since this tightly followed the already existing format of the data in our dataset. This is the final operation we must perform and you can see the final result of our relational schema in Figure 5.

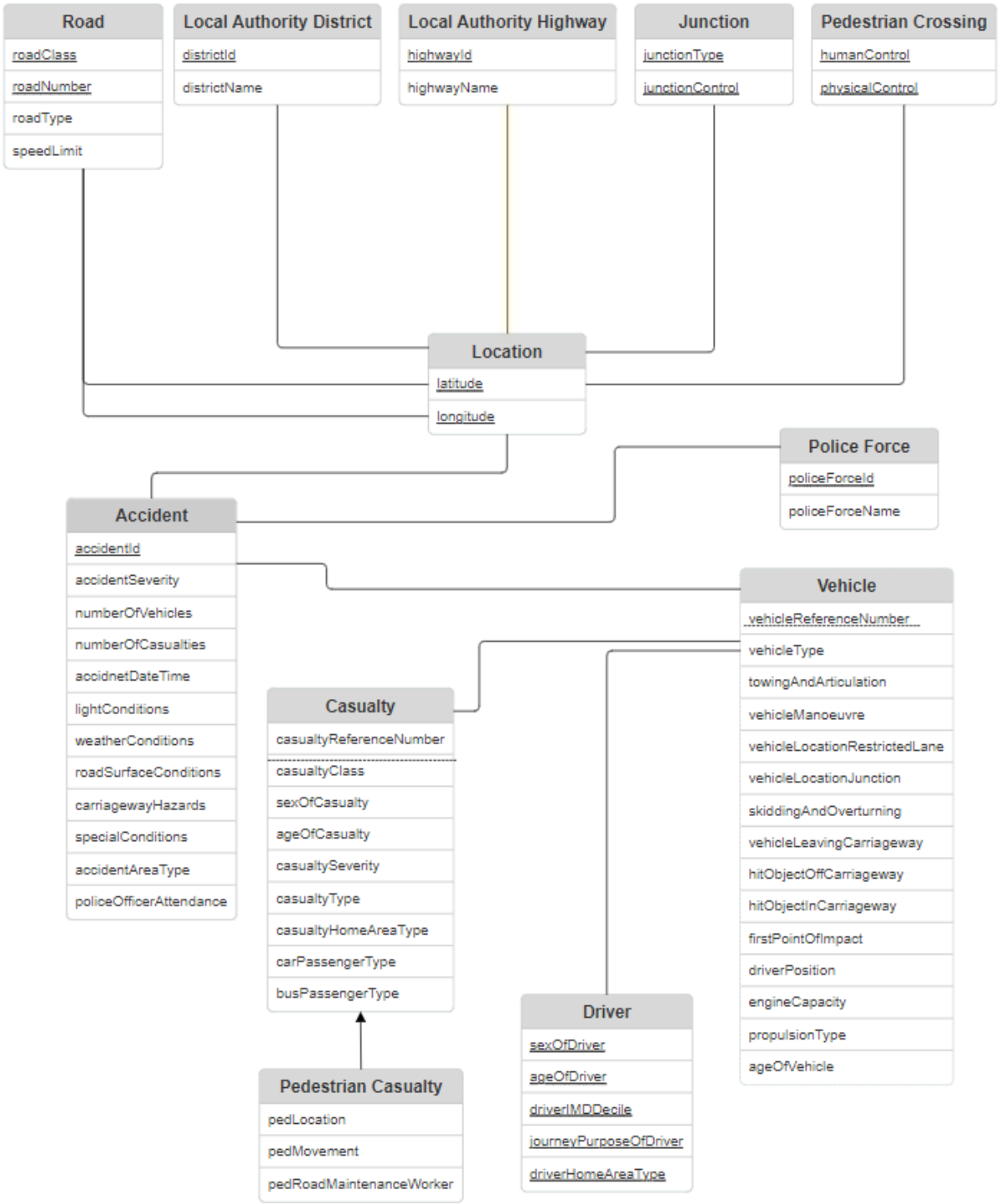

Figure 3. Entity-relation diagram after 1:0 relation simplification.

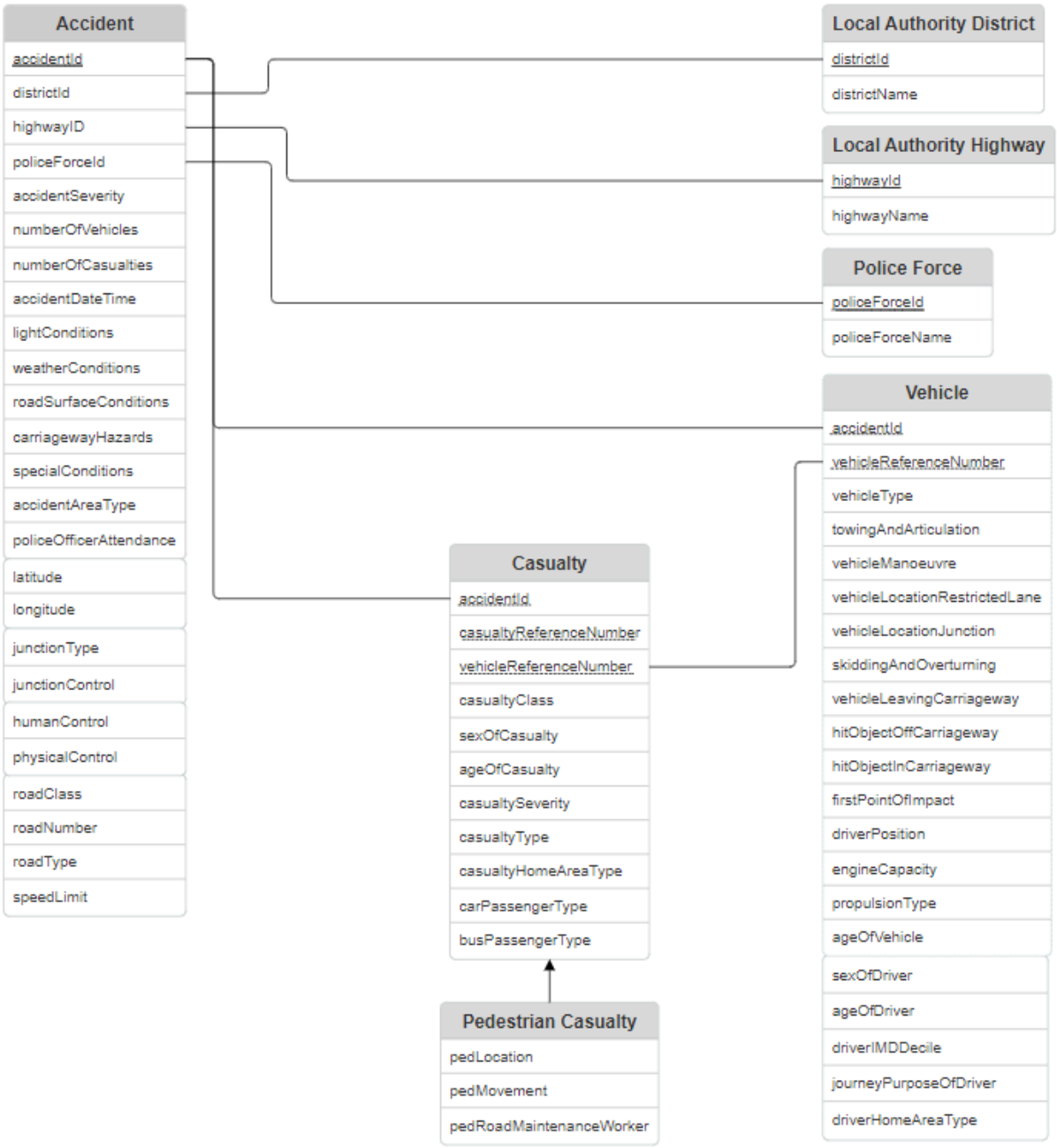

Figure 4. Entity-relation diagram after combining entities

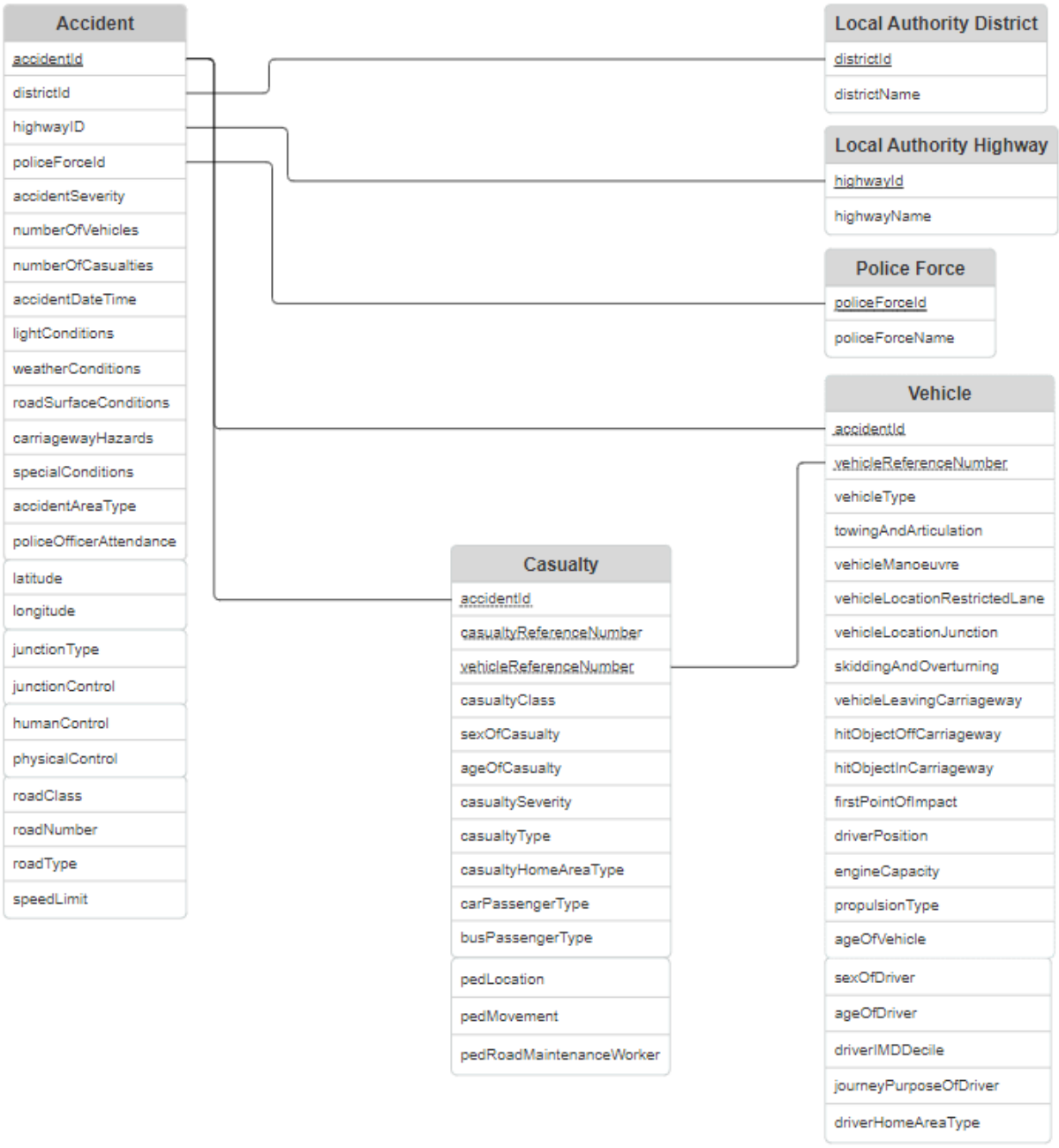

Figure 5. Final DB Schema

Lookup Tables, Indexes and Data Validation

The vast majority of the data in our dataset was encoded using an integer value mapping. To allow us to translate these values back to human readable data we had the choice of either using enums or using lookup tables. We decided to use lookup tables since this was the most straightforward way of translating the context CSVs from the dataset and it provided superior extensibility in case we wanted to add more values to the mapping later on. Each lookup table follows the structure in Figure 6 by convention. This setup also allows us to mark each of the fields in the Accident, Casualty and Vehicle tables as foreign keys to the lookup tables which provides many conveniences such as automatic generation of indexes

to cover nearly the entire database and automatic checks when inserting data to ensure that it is valid.

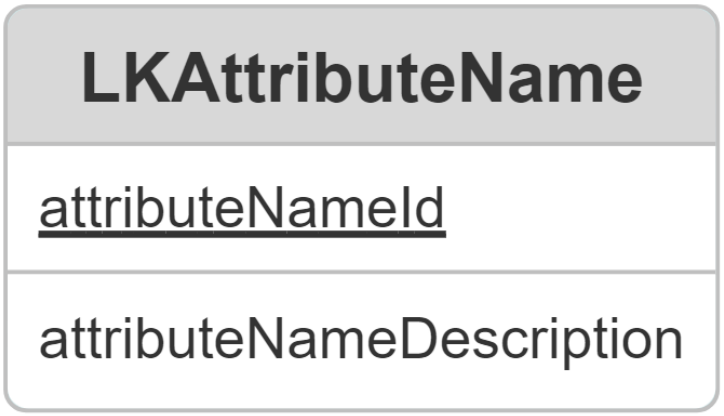

Figure 6. Sample Lookup Table Schema

Client Application

```
# The following items should be present:
# (a) Ideal client requirements
# (b) Actual client proposed
# (c) Actual client implemented
# (d) Test cases necessary to validate the client
# (e) Justification for the client design
# Client should include both the ability to query data and to update/input
data,
# as appropriate to the project area.
```

a) Ideal Client Requirements

The ideal client should allow the user to have completely customizable queries for the search, modify, and create functions. It should also have different levels of access, as modifying and creating accident entries should require a higher clearance than just searching for data.

Users should have customizable queries, which are obtained from the schema, where they can select any number of groups to filter the data. The comparisons on each filter should be adjustable, with minimum, maximum, equals, or a range from min to max. A description of each section should be provided for whichever category they select so the user knows they are filtering for each specific attribute. When returning a value, the user should be able to select which values should be returned, ie. only accidentID, accidentSeverity, and weatherConditions

instead of getting the whole row of data for a single accident back, preventing information overload. Another feature is to save searches, so the user can access previous searches quickly without having to re-input all of the filters. There would also be pre-implemented searches added, and users could choose between a pre-made filter or a custom search.

Users should be able to search for an accident, and using various values come to one accident. All values of an entry inside of Accident, Vehicle, and Casualty should be able to be changed. When changing an ID, the ID is verified to be unique before any changes. If needed, the lookup tables should also be able to be modified, if there was a need for new types of items to be added eg. a new police force in LocalAuthority.

The ideal client should have a way to input entries from either a csv or txt file, as creating an entire entry manually is very tedious with over 50 unique attributes to be added for each accident. New entries will have a new generated accidentID that are both unique and follow the general pattern in the original data.

b) Actual client proposed

Our client proposal is quite similar to the ideal client in several ways. We did not include multiple tiers of user authority, as this would not be useful in our current use case. The primary 3 functions were kept, however some of the functions were modified to be more practical. We had to decide between having preset questions or customizable queries, and we had chosen customizable queries as it provided a much broader range of accessibility. For modification, we made it so that the only value required for targeting was accidentID, which reduces the amount of time taken to get to the accident that the user wants to modify. The client shouldn't have the ability to add new items to the lookup table, while this could be relevant to certain tables such as LocalAuthority, it wouldn't be relevant for most tables such as LKSex. The proposed client would still have the same create function as our ideal client.

c) Actual Client implemented

There were several revisions we had to make as we started creating our actual client. The client app client.py is written in python and has 3 main functions; search, modify, and create. It currently runs on riku.shoshin.uwaterloo.ca, and it uses the db356_m476zhan database which is pre-loaded with the original data.

For the create() function, the single parameter to get a range of values was removed, instead the user can manually enter 2 different filters like x<5 and x>15 [insert screenshot] instead. The description was only added when an attribute required a lookup table, so values such as numberOfCasualties would not have any additional description as they are self explanatory. [insert screenshot] There were further complications in saving searches, so this feature is not included in our current implemented client. We also added a maximum of 50 entries being returned at a time, to prevent the server from returning the entirety of its contents if the user doesn't create strict filters. [insert screenshot]

For modify(), it was quite similar to the proposed model, where it selects an accidentID, and modifies attributes that belong to it. It will print the original value, the lookup table if it's available, and a confirmation that the value has been changes

successfully. A feature we added was to modify multiple attributes of a single accident within one run, to reduce the amount of user inputs. [insert screenshot]

For create(), it automatically generates a new accidentID. The user inputs the date of the accident, and then all 60 attributes after. We assume they will be loading it from a file, or copying it all in, so no lookup tables are shown when doing so. They will enter all contents of Accident, Vehicle, and Casualty. The accidentID and accidentDateTime values will be overwritten with the appropriate values and formatting. The client will send a confirmation message after each table indicating that the table has been created successfully.
[insert screenshot]

d) Test cases

Can't connect until servers are back up
[insert screenshot for each test]
Search:
Invalid category
Invalid attribute
Invalid comparison
Too many items

Modify:
Valid accidentID
Invalid category
Invalid attribute

Create:
Invalid accidentID
Invalid dateTime
Creation Failed

Data-Mining Investigation

For our data-mining investigation, we asked the question "what factors are best able to predict the severity of a car occupant casualty?" This is similar to the suggested question for the accidents dataset, however we wanted to be able to utilise vehicle attributes such as vehicle type, so we restricted our question to vehicle occupants. To answer this question, we employed the classification method, in which

we trained a decision tree to predict the highest casualty severity of a vehicle occupant in a vehicle involved in an accident. We used the library Sci-Kit Learn to train and test or model using K-Fold Cross Validation using a K value of 100. Originally, our model took the input (vehicleType, driverPosition, ageOfVehicle, sexOfDriver, ageOfDriver, propulsionType, driverIMDDecile, driverHomeAreaType) and output the predicted severity of the most severe casualty (light, serious, fatal). While this gave an overall accuracy of 86%, this is not indicative of a well performing model since our data is highly imbalanced (about 90% of casualties are light). The poor performance of our model is evident from the precision and recall of Fatal predictions, which were 2.4% and 1.3% respectively. In order to improve the performance of our model, we added more attributes to the input (for example the speed limit attribute from the Accident), added a “no casualty” class, and combined severe and lethal into a single class. We found that using the following attributes as input gave us the best performing model:

- numberOfVehicles
- lightConditions
- weatherConditions
- roadSurfaceConditions
- carriagewayHazards
- specialConditions
- accidentAreaType
- roadType
- speedLimit
- vehicleType
- vehicleManoeuvre
- vehicleLocationRestrictedLane
- vehicleLocationJunction
- skiddingAndOverturning
- vehicleLeavingCarriageway
- hitObjectOffCarriageway
- hitObjectInCarriageway
- firstPointOfImpact
- ageOfVehicle
- sexOfDriver
- ageOfDriver
- propulsionType
- driverIMDDecile

This gave a lower overall accuracy (around 60%), however for severe/lethal predictions we had a precision and recall of 21% and 23% respectively, with an average precision of 49% across classes. A performance report showing the average metrics across all 100 models is shown in figure 2. Individually, each attribute doesn’t seem to improve the performance of the model very much, but together we can generate a model that is somewhat effective at predicting the casualty severity. Sci-Kit Learn, the library we used, does not seem to handle

categorical data very well and prefers numerical data, so this is probably one of the reasons our model does not perform very well. If we wanted a model that could perform even better, we would likely have to write a custom implementation of a decision tree classifier to better handle categorical data. Even though the performance of our model was mediocre, it performs better than randomly guessing. This suggests that the attributes listed above are somewhat indicative of casualty severity.

	precision	**recall**	**f1-score**
Severe/Lethal	0.20889845915479657	0.2346385589401944	0.22095633729153277
Slight	0.6319893273278027	0.6267730762353396	0.6293572315929847
No Casualty	0.6236235693329925	0.6186023351807735	0.6210865514813811
accuracy			0.599007921981161
macro avg	0.4881704519385303	0.4933379901187692	0.4904667067886329
weighted avg	0.6022547370916324	0.599007921981161	0.6005577894238159

Figure 2. Averaged metrics of final models.

Testing

In order to test the effectiveness of our model we ran K-Fold cross validation with K = 100. After generating the 100 models, we averaged the precision and recall metrics across each class for each model to get an average precision and recall for each class. We also computed the average overall accuracy over all models. We used these metrics to determine the effectiveness of the models that we trained.