# Entity Relationship Design

* Please refer to the ERD-Draft.pdf, ERD-Final.pdf documents inside the Database/ERD path of our git repository to understand this section.
* We will refer to the entity relationship design/diagram/model in this part of the report as ERD for simplicity

ERD-Draft.pdf is an amalgamation of the alternative design choices we considered before creating our final entity relationship model, ERD-Final.

This section will aim to explain our choice of entity sets, multivalued, compound, optional attributes, aggregations, specializations while understanding why certain alternatives were finalized over others. Snippets of the draft and the final design will be compared to provide a holistic view of the design decisions made to build a strong ER-Model that can be converted to a Relational Schema. We will also discuss the cardinality constraints among different entities which will prove to be fundamental in this process.

1. Graphical user interface, application

   Description automatically generatedCar

* Car is an **entity set** that contains a set of specific cars that exist and are distinguishable from other such **entities** but share the same properties.

Graphical user interface, application

Description automatically generated

* The attributes that represent this entity can be divided into two categories: **required** attributes possessed by every single entity in our entity set and “**optional**” attributes that are possessed by some entities in our entity set. When we say Car **exists**, we mean it exists in Plato’s world. Yes, cars in real world certainly need an engine to run but that doesn’t mean every dealer using our database has the specs for every engine specification hence we leave it as an “optional” attribute that when added describes the entity.
* Here we can notice that all the required attributes are **single-valued simple** whereas all the optional attributes appear to be **single-valued composite**.
* **Entity Sets vs. Attributes**: We opted for bodyType, Interior etc. to be attributes rather than entities because the items under consideration are best described as an adjective rather than a noun
* **Keys**: Cars has only one candidate key, (VIN) since none of the other attributes collectively or otherwise uniquely determine a specific car. Hence, we select our only candidate key to be our primary key.

1. Listing
   * Table

     Description automatically generated with low confidenceListing is an **entity set** that contains a set of specific listings in our system that are distinguishable from each other but share common descriptive properties listed as **attributes**.
   * All the attributes are **required**, and we can see that they are **single-valued simple** attributes. We have opted to make them attributes instead of entity sets since they don’t require more expressiveness allowing the representation of additional information about the say, price. Moreover, we only need one price listed under any listing at any time and don’t see it changing in the future.
   * **Keys**: As described in our project scope, we have not placed restrictions on dealers to under details into the description section that are unique in the whole database. If it were, multiple dealers could not have the freedom to describe their car in the same way although it is highly unlikely that they would. Since we haven’t placed such constraints on any other attributes the only logical choice for a candidate key is (listingId) that becomes our primary key.
2. User (higher-level entity) and Dealer, Customer (lower-level entity)
   * + - **Diagram

         Description automatically generatedPartial Generalization**: We have decided that in our system, a User can be both a Dealer and a Customer. Our database will likely be used by more applications in the future and would require more roles like Data Science Analysts, Advertising Analysts etc., that have their own descriptive attributes not shared by others. Therefore, we felt like the specialization was a good strategy compared to just roles and attributes shared by every single user and with NULLs in case they don’t exist. To add, although we say that a User can be both a Customer and Dealer, it is very likely that most Users who join as Customers might never sell their Car.
       - **Disjoint Generalization**: For a specific instance, a User cannot be both a Dealer and a Customer. For example, for a specific listing relating to a specific car, a dealer cannot both create the listing and buy the car themselves. This is represented in the ER Model through combined arrows from lower-level entities to higher-lever entity
       - **User Attributes**: password and email are simple single-valued attributes, name is a composite single valued attribute and phoneNumber is a simple multivalued attribute.

* **Keys**: Our system allows multiple Users to have the same name but have a single emailAddress. Since we are not using the whole email address along with the domain it is reasonable to expect that they would be **unique** by design. For example, if we send an email to only your uwaterloo email address then we can safely assume that you are the only intended recipient of that email. This means that in our system, a User can store multiple phoneNumbers with a single emailAddress. That logically makes emailAddress our **primary key** due to lack of other more favorable choices.
* **Dealer Attributes**: Address is a composite attribute with a mixture of single-valued attributes like zip, city, and composite attributes within that like co-ordinates. We have decided to make Address optional since all the dealers might not be comfortable with sharing such details due to primary concerns. DealerDealers is a required composite attribute that has a franchiseDealer flag style attribute that tells us whether are using our platform on behalf of a franchise and dealerRating is assigned to each use based on customer feedback among other factors to be describes in the Client Interface section.

1. Diagram

   Description automatically generatedDepreciationFactors Entity and EffectsDepreciation relationship set

* **Weak Entity and Identifying Entity**: The existence of DepreciationFactors that EffectsDepreciation of the Car entity has no meaning. The former is predicated on the existence of the latter.
* **Primary Key**: We know that the primary key of a weak entity set is the combination of the primary key of the entity upon which it is dependent. Attributes underlined by the dotted line are the discriminators of our weak entity set and combine with VIN.
* **Cardinality**: Due to compelling arguments towards using better cardinality over traditional cardinality, we’ll be using x:y notation – **Car** has a 0:1 cardinality, it can be in the relation one time and **DepreciationFactors** has 0:1 cardinality, you can have one set of factors effecting a car’s valuation or none, can’t have multiple values for the same factors. The primary key of DepreciationFactors is not just the set of discriminators, it includes the VIN, which means it can be in the relationship only once.

1. CarSale Relationship Set
   * Car, Listing, User entities are in a **ternary relation** with CarSale as the relationship set. ‘date’ is a **descriptive attribute** linked to this relation; it refers to the date a Listing was made by User for the sale of a Car. As desired ‘date’ is a simple single-valued attribute

Diagram

Description automatically generatedCardinality: Using the better cardinality-

* + Listing:Dealer is (1:1); Dealer:Listing is (1:\*) because for a dealer to be in the system they must create a Listing but may have more than one and a Listing can have exactly one Dealer based on our application scope
  + Car:Listing is (1:1); Listing:Car is (1:1) i.e., compulsory participation is enforced on both entities and they are exclusively and uniquely mapped to each other
  + Dealer:Car is (1:\*); Car:Dealer(1:1) because a Dealer must attempt a car sale to be in the relation but can sell as many Cars as they want, while a Car can only be sold by a single Dealer
  + Instead of creating multiple binary relations, we have opted here to use the more effective and information preserving ternary relation because it reflects the actual relationship that these entities share and enforces that they are all connected.

1. Aggregation : meetFor relationship set, Appointment entity set and aggregate entity
   * **Diagram

     Description automatically generatedMotivation**: Relationship sets meetFor and CarSale represent overlapping information. Every meetFor relationship ties together a Car, Listing, Dealer just like CarSale but we cannot remove CarSale since it can exist without any meetings. In addition, we have some extra attributes like description related to the ‘meet’ called Appointment here
   * Our aggregate entity is within the pink rectangle with our CarSale relation and the whole thing is connected to our meetFor relation connecting Appointment entity (with ‘date’ as descriptive attribute) to the aggregate entity

# Relational Model Design

* Refer to the createDb.sql file in the Database folder of our git repository for the SQL code that creates tables and views based on the final relational schema; primary, foreign, and unique key, checks and indexes to maintain the integrity and improve performance and load data from the csv files
* Some relational schema conversions are obvious, so we are going to provide our reasoning behind the motivations for only those that require explanations
* Integrity constraints like unique(), check(), set default, not null are used for several attributes that can be evident by the domain of each attribute and can be seen in our createDb.sql file

1. Car

Graphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generated

1. Let’s start with the easier step, we have the required attributes that the Car entity must have, so the relational schema with have all those attributes as part of the table, Cars.
2. Next, we have to deal with the optional attributes. Now our primary concern with the datasets on Kaggle was all the large number of NULLs and empty strings. By understanding the database and our applications usage we decided to create our entity and relational sets in such a way that the nulls are minimized, and they only have one connotation, missing information and not N/A. Every attribute to the left pertains to a Car. A huge factor in deciding which attributes would be optional was this as well.
3. We wanted to **reduce NULLs** as much as possible because:
   * 1. Normalization concerns: These optional attributes cannot be part of the primary key and are not functionally dependent on it
     2. Simplify the client interface – We did not have to implement three-valued logic
     3. RDBMS Treatment of NULL: From the lectures and personal experience of MySQL, having mitigatable NULLs in the system does not make for a well-maintained error-free application. We perform aggregations, arithmetic operations, and use
     4. subqueries and having nulls just make things unnecessarily complicated considering we have other effective alternate solutions
4. Let’s proceed creating **separate relations for our optional attributes** – we’ll have the composite attribute name as our table name for clarity and since we don’t have any naming conflicts. Our primary key for these tables will be (VIN) which also acts as a foreign key referencing Cars. This gives us Cars(VIN, bodyType, height, year, modelName, franchiseMake, isFleet, isCab), Interior(VIN, backLegroom, frontLegroom, interiorColor, maximumSeating), Wheels(VIN, wheelSystem, wheelSystemDisplay, wheelbase), Engine(VIN, engineCylinders, engineDisplay, engineType, horsepower, transmission, transmissionDisplay), FuelSpecs(VIN, cityFuelEconomy, highwayFuelEconomy, fuelType, fuelTankVolume), TrimPackage(VIN, trimID, trimName).
5. **Note** that we changed ‘fleet’ to ‘isFleet’ for clarity and all the attributes that start with is<attr name> must be either ‘True’ or ‘False’. A check is added to ensure that.
6. Listing

Table

Description automatically generated with low confidence

* Listing (listingId, listingDate, dayOnMarket, mainPictureUrl, majorOptions, price, description). The primary key of the relation is same as the entity set.
* **Diagram

  Description automatically generatedActive** is a flag that indicates when the last time was there was any activity on the Listing. If it exceeds 180 days, then we delete the listing. This feature needs to be finessed to include intermediary email notifications informing the user that the listing has been inactive for 80 days and if they don’t respond to the email (either by going to their listing or taking some action on the communication email, the listing would be deleted). When we say delete, we are speaking from the perspective of the dealer. It will be hidden from them but the system administrators, data science analysts will be able to see all the listings that were ever created. The cleanup of these listings is left to the discretion of those users.

1. User – Partial Disjoint Generalization

* User(emailAddress, firstName, lastName, password). Attributes within the composite attribute, name are now part of the User relation.
* ‘**phoneNumber’** is a multi-valued attribute and is optional so we create another relation called PhoneNumber(emailAddress, phoneNumber) since each user can have none or many numbers of phone numbers stored in our database. emailAddress is a foreign key referencing emailAddress in User.
* Using a **Top-Down approach** and considering that Address is optional as well, our Dealer entity will be called DealerDetails for clarity and DealerDetails(emailAddress, franchiseDealer, sellerRating) with emailAddress as primary key and foreign key referencing emailAddress in User.
* Our **Address** will have Address(emailAddress, zip, city, latitude, longitude) with emailAddress as primary key and foreign key referencing emailAddress in User. But this is not a normalized relation. Now we need to normalize this:
* R = {emailAddress, zip, city, latitude, longitude}
* F = {emailAddress -> zip, city, latitude, longitude; latitude, longitude -> zip (because of accuracy of our co-ordinates it does not give city) }
* Keeping in mind that we are not restricting multiple users from using the same address and most users cannot be reasonably expected to provide us their co-ordinates, here is the relations we finalized on.
* **Address**(emailAddress, city, zip), **Coordinates**(emailAddress, latitude, longitude) with emailAddress as primary key and foreign key referencing emailAddress in User

## DepreciationFactors – Weak Entity

* DepreciationFactors(VIN, frameDamaged, hasAccidents, salvage, savingsAmount) is the final relation. As expected, VIN is the primary key and foreign key referencing VIN in Car because of the nature of weak entities and the cardinality constraints enabling us to identify each depreciation factor with just the associated VIN of the corresponding Car.
* The weak entity previously had isNew attribute, but the relational schema does not due to a dependency. Whenever isNew is false, ownercount is not application. Adhering to our previous commitment of good database design, we have a separate relation called CarOwners(VIN, ownercount) with VIN as the primary key and foreign key referencing VIN in Car.

1. CarSale Relationship Set –

* Diagram

  Description automatically generatedWe have a couple of alternatives to establish this relationship:

CarSale(emailAddress, VIN, listingId, date)

CarSale(VIN, listingId, emailAddress, date)

Cars(VIN, listingId, …), Listing(listingId, dealerEmail, listingDate, …)

Cars(VIN, listingId), CarSale(listingId, dealerEmail, date)

Keeping in mind the cardinality constraints discussed earlier, we felt that option (c) is the most efficient alternative because both the Listing and Dealer have a 1:1 relationship, CarSale serves no special purpose if it’s descriptive attribute gets absorbed and into the Listing entity. The dealerEmail in Listing is a foreign key referencing emailAddress in User and listingId in foreign key referencing listingId in Listing.

1. Aggregation –

* Similar to CarSale relation, we don’t necessarily need a separate meetFor relation if get it’s absorbed into Appointment with the right primary keys to satisfy the cardinality constraints. Therefore, our Appointment(appointmentNumber, dealerEmail, customerEmail, listingId, appointmentDateTime, information). listingId is foreign key referencing listingId in Listing table, dealerEmail and customerEmail are foreign keys referencing emailAddress in User table. listingId is also added to primary key since a dealer and customer pair and meet with regards to multiple car deals and the appointment number is not sufficient enough to make that unique since the appointment number is a suite of numbers that are incremented in the subset of a particular set of dealer, customer, listing
* Similar to Listing entity set, there is an active attribute in appointments. The inactive Appointments and inactive Listings have an expiry date of 60 days and 180 days respectively after which they’ll be hidden from the Dealers and Customers. The clock for a listing or appointment’s expiry begins on the last day there was any activity on the listing.

# Physical Schema Indexes

We have already discussed the primary and foreign key attributes in the relational schema that act as indexes. In this section, we will look at some other indexes added to improve the execution time of our queries to improve the applications performance as the number of records increase.

Our application is defined to use several filters allowing the users to find cars that suite their needs instead of browsing through the 3 million listings in our database. As we have seen in the lectures, one way to increase the efficiency of the record lookup is by using indexes. These filters by definition don’t have to scan the whole tables since they need to return only a subset of the records that match the key. You can see these indexes on our Cars table over maximumSeating, color; DepreciationFactors table over savingsAmount. Some strategically added indexes were able to decrease the execution time on queries that need to perform a lot of joins. In some many tables, we were able to implement covering indexes, for example, counting the number of franchise dealers with sellerRating as 5. To decrease the size of joint attributes in the dataset and increase the efficiency of lookups, we also split the fuel economy into cityFuelEconomy and highwayFuelEconomy.

We saw an improvement in the range of 40-65% after adding these indexes to our database with just primary and foreign keys.

# Changes to the database to support the next phase of the application

* Adding functionality for supporting multiple dealers to a single Listing and a single Appointment
* Right now, we are storing the password for the application in the same table as other User specific attributes without adopting any encryption techniques. If a hacker tries to extract from our database, they will own all the passwords in plain text despite the user having a strong password. This is not a secure methodology for storing passwords since the database administrators and website managers having access to them is a violation of privacy.
  + We will adopt the main principles of secure storage like hash, salt, pepper and iteration and use a reliable hash function for storage.
  + We will use a non-reversible cryptographic function so that the attacker cannot just guess the logic of the encryption and reverse it. While designing this function we will try to protect our passwords against brute force, dictionary, rainbow table attacks.
* Checking the correctness of VIN : Prior to 1981, the length of the VIN ranged from 11 to 17 characters but after that they are strictly 17 characters. As depicted in the picture [1], each segment of the VIN can be checked for correctness with the right database support. We can also split this number into different segments to reduce the length of the attribute which would make the lookup more effective and allow us to use char and decimal types.
  + We also plan to implement LUHN Algorithm to check correctness

Timeline

Description automatically generated

Figure : VIN decoding

* Try to get the user coordinates by using a Geolocation API [2]. It takes care of user consent and allows us to adopt geotagging, tailoring advertisements on the application based on user location etc.

# Bibliography

|  |  |
| --- | --- |
| [1] | AutoCheck Experian, "AutoCheck Experian," 2019. [Online]. Available: https://www.autocheck.com/vehiclehistory/vin-basics. |
| [2] | P. Kinlan, "Web Fundamentals," Google Developers, 12 February 2019. [Online]. Available: https://developers.google.com/web/fundamentals/native-hardware/user-location. [Accessed 15 April 2021]. |