MET CS 669 Database Design and Implementation for Business
Term Project Iteration 6

Name: Edward Myers

Table of Contents

Project Direction Overview

Modifications with justification between iterations are identified as red text.

I would like to create a database that tracks vehicle status, history and maintenance information for a manager of a small vehicle fleet, called "FleetManager". I work for a local township in Pennsylvania, and for many years they have used an application that stores vehicle maintenance file-by-file in an operating system, which works ok for keeping history records on a vehicle but doesn't lend itself to generating meaningful business analytics. When describing the user access below, I anticipate the mechanic user to primarily use a desktop application tied to the database with intuitive option buttons made for ease-of-use. Administrative users from a finance perspective may query the database more directly to gain custom insight on trends and expenses.

For a mechanic user, the database would store maintenance transactions about a vehicle, whether it needed grease/oil, any replacement parts, or got sent out to a vendor for repair. The mechanic would start a maintenance entry in a desktop application linked to the database, add the appropriate details about the work and subsequent parts and save it as an entry. The mechanic could also look at the history of all work done on a particular vehicle sorted by date and pull general information about its parts for ordering and inventory. If the mechanic needed to know what type of oil filter the vehicle required, they would select general information on the desktop application which accesses the known vehicle part data from the database.

For an administrative user, the database would yield long term data regarding a vehicle and its pattern of maintenance. The finance clerk would query the database for information tying a vehicle to its year-over-year expenditure to create a report on cost to maintain vs. replace. A fleet manager would query the database for each vehicle's mileage to identify patterns of uneven wear across the vehicle fleet. A supervisor will query the database for a count and list of what vehicles each mechanic has worked on to even out workload distribution.

I envision that the database will hold:

- General information about each individual vehicle
- Date, parts, cost and other details about each maintenance work order
- Parts inventory which associates which parts go to which vehicles, and how many are in stock.

My interest in the program comes from my desire to help out my home township. I wrote the program that they currently use in Java nearly 20 years ago without any database background. I'm thrilled that they still use the program, but in an era of integrated asset management applications they could be getting so much more out of their data. To that end, I would like to at least design my idea of their next step.

Use Cases and Fields

Original Use Cases:	New Use Cases (Iteration 3)	Acknowledged Use Cases Outside of Scope:
New Vehicle Entry	*Appointment Entry*	*Workload Lookup*
Edit Vehicle Entry		*Expenditure Over Time*
Add Maintenance Entry		*Mileage Reporting*
Add Part Entry		
Add Vendor		

The following are use cases that could be expanded upon that share a similarity to an already-defined use case:

Edit Maintenance Entry	*Delete Maintenance Entry*
Edit Part Entry	*Delete Part Entry*
Edit Vendor	*Delete Vendor*
Add Mechanic	*Delete Vehicle Entry (removed from original use cases)*

From feedback, the 'Delete Vehicle Entry' has been removed as the use case implies that the vehicle and its attributes have already been added into the database. As we are working with fresh data for create/edit transactions, it has been omitted. In its place is a new use case: 'Appointment Entry'.

New Vehicle Entry & Setup Use Case

1) The mechanic clicks the 'Add Vehicle' button on the application interface.
2) The application provides a fill form to accept the data fields.
3) The mechanic specifies the type of vehicle as *CDL vehicle, standard vehicle, accessory,* or *none.*
4) The mechanic enters all other the appropriate information and clicks 'Save'.

Field	What it Stores	Why it's Needed
vehicle_ID	The 4-digit departmental ID code of the vehicle	This is the naming convention already used, and uniquely identifies the vehicle.
make	The brand of the vehicle	Only certain parts work on certain vehicles.
model	The model of the vehicle	Only certain parts work on certain vehicles.
license	The license plate value	You can check which vehicles need renewed registration and pull status of insurance.
VIN	The auto manufacturer vehicle identification number	The coding of the VIN gives exact vehicle specifications, and the VIN is required on nearly all types of registration paperwork.
year	The creation year of the vehicle	The year of the vehicle is the primary statistic used to determine parts availability and to evaluate replacement. The local township participates in a fleet lease program in which a mixed trade in of vehicles 3 years and older reduce the overall fleet age to save on maintenance costs.

inspection_date	The date the vehicle is due for its next inspection.	The inspection date is important to keep the vehicle street-legal, and serves as a parameter by which the application's appointment notifier communicates a necessary vehicle maintenance activity to the mechanic.
vehicle_type	The type of vehicle, being a CDL vehicle, standard vehicle, accessory, or none.	This is a new attribute created for the generalization/specialization rule set. Depending on the type of vehicle, different information will be available and different check flags will apply to appointments.

Notes:

- Vehicle_ID is already established per vehicle as part of a departmental code. (Ex.: all administrative vehicles are '1xxx', police '2xxx', wastewater '3xxx', etc.)

Edit Vehicle Entry

1) The mechanic clicks the 'Edit Vehicle' button on the application interface.
2) The application displays a list of current vehicles and prompts the mechanic for the ID of the vehicle to edit.
3) The mechanic enters the ID of the vehicle and presses 'OK'.
4) The application displays a fill form screen with the current vehicle data populated.
5) The mechanic uses the fill form to edit the populated data as desired.
6) The mechanic presses 'Save', applying the changes to the vehicle instance.

Field	**What it Stores**	**Why it's Needed**
vehicle_ID	The 4-digit departmental ID code of the vehicle	This is the naming convention already used, and uniquely identifies the vehicle. Requested to determine what vehicle's information to extract from the database.
make	The brand of the vehicle	Only certain parts work on certain vehicles. Field that is extracted from the database and may be edited by user.
model	The model of the vehicle	Only certain parts work on certain vehicles. Field that is extracted from the database and may be edited by user.
license	The license plate value	You can check which vehicles need renewed registration and pull status of insurance. Field that is extracted from the database and may be edited by user.
VIN	The auto manufacturer vehicle identification number	The coding of the VIN gives exact vehicle specifications, and the VIN is required on nearly all types of registration paperwork. Field that is extracted from the database and may be edited by user.
year	The creation year of the vehicle	The year of the vehicle is the primary statistic used to determine parts availability and to evaluate replacement. The local township participates in a fleet lease program in which a mixed trade in of vehicles 3 years and older reduce the overall fleet age to save on maintenance costs. Field that is extracted from the database and may be edited by user.

Add Maintenance Entry

1) The mechanic clicks the 'Add Maintenance' button on the application interface.
2) The application provides a fill form to accept information regarding the maintenance.
3) The mechanic selects the vehicle being maintained from a list.
4) The mechanic enters the data about the maintenance, including information about the mechanic, type of maintenance, date, etc. as outlined below.
5) The mechanic enters information about used parts & materials in a sub-window.
6) To add parts used to the maintenance window, the mechanic clicks on the 'include parts' button.
7) The mechanic selects the part by name from a drop-down list or enters the part ID.
8) The application queries the parts inventory table to display the full part ID, name, and description of part.
9) The mechanic enters the number/quantity of the product used.
10) The application references the part record from the inventory again to subtract used quantity from stock quantity. If there is not enough quantity identified in stock, an alert flashes on the screen giving the mechanic the ability to return and change or override and continue.
11) The mechanic presses 'Ok' in the part window, navigating back to the maintenance window.
12) The mechanic presses 'Save' to save the maintenance transaction.

Field	**What it Stores**	**Why it's Needed**
mechanic_name	The mechanic's name	This helps to track which mechanic performed what work for workload analysis.
type	Type of maintenance performed	Classifies the type of maintenance performed. Commonly this field distinguishes if the maintenance was major, minor, vendor or an inspection.
vehicle_ID	The 4-digit departmental ID code of the vehicle	This number identifies which vehicle received the maintenance.
date	The date the maintenance was performed.	This is useful for tracking when maintenance was last completed and the overall frequency of maintenance.
cost	The total cost of the repair considering all fluids, parts, etc.	This is useful for tracking overall maintenance costs for the vehicle. I've disassociated individual cost of parts as that would require the mechanic to constantly update the parts inventory with current pricing.
mileage	Current mileage of the vehicle at time of maintenance	This is useful for comparison with other maintenance records to determine overall miles traveled within a given time frame.
vendor_name	The name of a vendor which performs the repair if maintenance is outsourced	This is useful to compare what vendors get the most business and also to compare how much maintenance vendors do to mechanics in-house.
part_ID	The application's part or product identifier	This number is used to reference the application's unique identifier for the part. This is assigned by the program upon a part entry into the parts inventory.

part_quantity_used	The number or quantity of the part or product used during the maintenance	This number is referenced against the database's known quantity of the item in question. If more of the item is used than the database knows is within possession, a flag is triggered to the user.

Add Part Entry

*Note: The scope of *Add Part* now removes the ability to create a new part from the *Add Maintenance* window. All creating and editing of parts is performed from the *Parts Inventory* window of the application.

1) If a mechanic user would like to create part information, they click the 'parts inventory' button on the application interface.
2) The mechanics clicks 'add part'.
3) The application provides a fill form to accept information regarding the part.
4) The mechanic enters the data about the part, including ID, name, serial #, etc. by text input as outlined below.
5) The mechanic presses 'save'.
6) The application checks to make sure no duplicate part ID's exist, after which the program saves the new part into the appropriate database table.

Field	**What it Stores**	**Why it's Needed**
part_name	The parts's name	Quick reference or jargon related to the identification of what the part is used for.
part_serial	The specific manufacturer's part or product identifier	This number is used to reference the manufacturer's specification on how and where the part or product can be used.
description	Additional notes on the part or product	If the identification or use of the part or product is not clear from the name or researching the serial #, then additional notes to describe the part and its use are helpful.
quantity_in_stock	The quantity of the part on hand.	This is useful for determining when to order more supply. This value is also compared to quantities deducted during an *Add Maintenance* event.

Add Vendor Use Case

1) The mechanic clicks the 'Add Vendor' button on the application interface.
2) The application provides a fill form to accept information regarding the vendor.
3) The mechanic enters the data about the vendor.
4) The mechanic presses 'Save' to save the entry.

Field	**What it Stores**	**Why it's Needed**
vendor_name	The vendor's name	This helps to track which vendor performed what work for workload analysis.

vendor_address (street_number, name, town, zipcode)	Address variables	This helps identify where the vendor is located. Proximity can be a factor for vendor repair selection.
phone_number (area_code)	Phone number variables	This stores telephone contact information for the vendor.
agent_name	Vendor representative, if applicable	This is useful in case the vendor has an area sales representative that receives business from your area.
email	Email contact address	Similar to phone number, it is another way to get in contact with the vendor.

Appointment Entry Use Case

The application has a feature that notifies the mechanic of a scheduled maintenance by either an automatic parameter (ex./ the application triggers its own required maintenance alert if the current date is within 3 months of a vehicle's inspection deadline), or by manual parameter where the mechanic has entered a 'Scheduler Entry' vehicle appointment. For manual entry, the mechanic does the following:

1) The mechanic clicks a button on the application interface called "New Appointment".
2) A fill form screen appears for the mechanic to enter details about the appointment.
3) The form asks the mechanic to select the vehicle by its vehicle ID code from a drop menu.
4) The form asks for the mechanic to be assigned the maintenance activity from a drop menu.
5) The form requires a maintenance due date to be filled in, as well as a checkbox to select if the mechanic would like a one-week reminder sent to them.
6) The form allows for the mechanic to put in a note regarding useful details about the maintenance, such as type, ordered parts, etc.

This use case as well as the 'Maintenance Entry' use case identifies that a 'mechanic' data entity will need to be created. Not only will a mechanic's first name, last name, certified license, etc. be needed, but phone number, email, or preferred method of contact so that the application can send the mechanic a message required by the fleet scheduler.

Field	**What it Stores**	**Why it's Needed**
appointment_date	The desired date of the upcoming maintenance	This is the primary variable of the fleet scheduler, communicating what vehicles need maintenance attention to specific mechanics.
mechanic name (fname & lname menu selection)	The first and last name of the mechanic	Even though this information is pulled from the known registry of mechanics, the selected mechanic name is attached to the selected appointment record. That mechanic is now assigned the task.
vehicle_id	The 4-digit departmental ID code of the vehicle	This number identifies which vehicle received the maintenance. Even though it is pulled from the registry of vehicle, upon creation of this appointment record the vehicle gets assigned the pending maintenance.

Part History Use Case

The application will have a section where the mechanic will be able to generate custom reports that are useful to the maintenance of the vehicle fleet. One of these reports relates to the usage frequency of all parts, which can give valuable insight into wear patterns and restock frequency. The mechanic should be able to generate this report through the following steps:

1) The mechanic clicks the 'Reports' button on the application.
2) The mechanic uses a scroll menu to view the available reports. To pull a part usage history, the mechanic would select 'Parts Usage Report'.
3) The report prompt asks the mechanic to enter a start and end date for the reporting bounds. Alternatively, the mechanic could select 'All' which would print the entire usage history.
4) The application generates the report, converting it into a digital file and uses the workstation operating system's default file manager options to save it in the desired location.

For each part usage record entry, the program will need to monitor the Part entity already established in the database with some sort of trigger that fires when a change happens to the part inventory. The part_name is necessary attribute to explain what part is experiencing the change. Presumably the quantity is changing, therefore what the quantity used to be and what the quantity now is needs to be recorded. Lastly, the other data makes no sense without the date in which the change occurred.

Field	**What it Stores**	**Why it's Needed**
part_name	The parts's name	Quick reference or jargon related to the identification of what the part is used for.
part_old_quantity	The previous quantity of the part	This is half of the reason that the change existed. Gives what the quantity used to be.
part_new_quantity	The new quantity of the part	This is the other half of the reason that the change existed. Gives what the quantity has become.
change_date	The date of the change	Gives the time dimension to the change in quantity. The other variables do not make sense without a timeframe to describe them.

Structural Database Rules

Associative Database Rules

Use Case #1: New Vehicle Entry

- The mechanic clicks the 'Add Vehicle' button on the application interface.
- The application provides a fill form to accept the data fields.
- The mechanic enters all the appropriate information and clicks 'Save'.

From the above use case, I identify a 'mechanic' and data about a 'vehicle'. I know that there is an interaction between the two entities at some point, but this is an interaction between a mechanic and a third-party program which accesses the database. The mechanic is an obvious entity, as is vehicle. Regarding vehicle, the following two attributes can warrant their own entities:

Field	What it Stores
make	The brand of the vehicle
model	The model of the vehicle

In a local municipality's fleet, often the manager or mayor makes the decision to acquire vehicles of the same type or similar enough that they take a lot of common parts. This cuts down on the cost of keeping repair materials in stock, and most of the vehicle's maintenance timings and needs will line up. So lets look at the relationship between vehicle, make and model. A unique relationship exists between vehicle/make and vehicle/model. While an argument can be made that make (1->M) model (1->M) vehicle relationships occur as I used in my previous iteration, a simpler way is to show that a vehicle must have a unique make and a unique model, while a make may produce many vehicles and many vehicles may exist of a certain model. The two modified rules are:

	Entity #1	Participation	Relationship	Plurality	Entity #2
1a	A vehicle make	May	Apply to	Many	Vehicles
1b	A vehicle	Must	Have	One	Vehicle make

	Entity #1	Participation	Relationship	Plurality	Entity #2
2a	A vehicle model	May	Correspond to	Many	Vehicles
2b	A vehicle	Must	Correspond to	One	Vehicle model

Use Case #2 does not tell us any more about the relationships. Moving on…

Use Case #3: Add Maintenance Entry

- The mechanic clicks the 'Add Maintenance' button on the application interface.

  …
- The mechanic selects the vehicle being maintained from a list.

  …
- The mechanic enters information about used parts & materials in a sub-window.

A few relationships are now revealed. Now we know that Mechanics perform a maintenance activity on vehicles, which they record as a maintenance entry in the program. One mechanic enters data about a particular maintenance activity that they performed, but multiple maintenance activities can be performed by a mechanic. If a maintenance activity exists, it had to be performed by either a mechanic or a vendor (further ahead). A mechanic does not necessarily have to perform a maintenance activity.

*Note: my program purposely limits the performance of one maintenance or repair to one mechanic.

	Entity #1	Participation	Relationship	Plurality	Entity #2
3a	A mechanic	May	Perform	Many	Maintenance Activities
3b	A maintenance activity	May	Be performed by	One	mechanic

We know from the steps in the program that the mechanic is forced to select one vehicle to log the maintenance activity toward. Thus, every maintenance activity maps to only one vehicle. It follows that a maintenance activity cannot exist without a vehicle identified to work on. Also, it makes sense that a vehicle can have multiple maintenance activities performed on it over time, or none.

	Entity #1	Participation	Relationship	Plurality	Entity #2
4a	A maintenance activity	Must	Apply to	One	Vehicle
4b	A vehicle	may	Have	Many	Maintenance activities

As part of the maintenance entry into the application, the mechanic enters what parts were used during the maintenance activity. Depending on the type of maintenance, it is possible for no parts, one type, or multiple types of parts to be used. Conversely, a part may be included in none, one or many maintenance activities.

	Entity #1	Participation	Relationship	Plurality	Entity #2
5a	A maintenance activity	May	Use	Many	Parts
5b	A part	May	Be used in	Many	Maintenance activities

Use Case #4: Add Part Entry

- If a mechanic user would like to create part information, they click the 'parts inventory' button on the application interface.

  …
- The mechanic enters the data about the part, including ID, name, serial #, etc. by text input as outlined below.

  …
- The application checks to make sure no duplicate part ID's exist, after which the program saves the new part into the appropriate database table.

This information, like Use Case 1, gives us the foundation of more entity tables. The third-party application uses a construct called 'Parts Inventory', which calls on a table of parts with the database to add, edit and delete from. However, we can infer that these parts are inventoried for a reason: that they fit into vehicles. A vehicle can have a lot of parts, and some parts can fit into multiple vehicles. For the purposes of the program, it would not make sense to force populating all the parts of a vehicle into the database as there would be thousands, most of which would never receive maintenance. Likewise, we would not include all possible vehicles that the part applied to since most of them are not in the vehicle fleet.

	Entity #1	Participation	Relationship	Plurality	Entity #2
6a	A vehicle	May	Contain	Many	Parts
6b	A part	May	Be contained in	Many	Vehicles

Use Case #5: Add Vendor

- The application provides a fill form to accept information regarding the vendor.
- The mechanic enters the data about the vendor.

Although the use case does not specifically say what the vendor does or why we are recording data about vendors, we can infer that a vendor interacts with a vehicle or part in some way. In my original version of this program, the mechanics at my local Township referred to vendors as both parts sellers and repair shops at which vehicles were sent for maintenance activities. This gives vendors a similar role as mechanics.

	Entity #1	Participation	Relationship	Plurality	Entity #2
7a	A vendor	May	Perform	Many	Maintenance Activities
7b	A maintenance activity	May	Be performed by	One	vendor

*Note: like for mechanics, I purposely limit the scope of a single maintenance activity to a sole vendor. Also, the 'may be performed by' references the fact that a maintenance activity could be performed by either a mechanic or a vendor. There is a chance that since both relationships use 'may', the entity performing the maintenance could be left NULL. It would then be a responsibility of the third-party application code to enforce an entry.

Additionally, vendors supply parts. If a part exists, it must have a vendor/source. A vendor could also supply many types of parts, such as for a specific brand of vehicle. If a vendor is listed, they may or may not sell parts since some vendors may only perform repairs.

	Entity #1	Participation	Relationship	Plurality	Entity #2
8a	A vendor	May	Supply	Many	Parts
8b	Part	Must	Be supplied by	One	vendor

Use Case #6: Appointment Entry

- The form asks the mechanic to select the vehicle by its vehicle ID code from a drop menu.

- The form asks for the mechanic to be assigned the maintenance activity from a drop menu.
- The form requires a maintenance due date to be filled in, as well as a checkbox to select if the mechanic would like a one-week reminder sent to them.
- The form allows for the mechanic to put in a note regarding useful details about the maintenance, such as type, ordered parts, etc.

The data entry from the steps above creates an appointment record, which the application processes and in some manner calls status on to trigger an interaction with the mechanic. The database therefore needs a table to log these appointment logs in. From the steps above, appointments establish a relationship with mechanics and vehicles. In the terms of an appointment, an appointment is assigned to one mechanic. However, a mechanic can be assigned many appointments as there may be many vehicles. A mechanic does not have to be scheduled any appointments (may or may not), but if an appointment exists, it must be given an appointee.

	Entity #1	Participation	Relationship	Plurality	Entity #2
9a	A mechanic	May	Be assigned to	Many	Appointments
9b	An appointment	Must	Be assigned to	One	mechanic

Similarly, nothing says that a vehicle needs to be assigned an appointment. A vehicle entity can exist without an appointment assigned. Also, over time a vehicle may be assigned dozens of appointments. If an appointment exists however, it must relate to one vehicle:

	Entity #1	Participation	Relationship	Plurality	Entity #2
10a	A vehicle	May	Be assigned to	Many	Appointments
10b	An appointment	Must	Be assigned to	One	vehicle

Derived from New Specialization Rule

- A maintenance activity is a vendor repair or an inspection or none of these.

Concerning the vendor repair created through the process of subtyping maintenance activity, we can make the argument that a vendor repair will be performed by one vendor. A vendor can make multiple repairs over time.

	Entity #1	Participation	Relationship	Plurality	Entity #2
11a	A vendor repair	Must	Be performed by	One	Vendor
11b	A vendor	May	Perform	Many	Vendor repairs

Derived from Normalization

Normalization of the Inspection subtype under MaintenanceActivity revealed a multitude of tests with a pass, fail, 'see comments' repetition to them. A many to many relationship between inspections and tests was revealed, and a second many to many relationship was found between tests and results:

	Entity #1	Participation	Relationship	Plurality	Entity #2
12a	An inspection	Must	Have	Many	Tests
12b	A test	May	Occur in	Many	Inspections

	Entity #1	Participation	Relationship	Plurality	Entity #2
13a	A test	Must	Have	Multiple	Results
13b	A result	Must	Occur in	Multiple	Tests

Both will require bridge entities to resolve in the physical ERD.

Normalization of the Vehicle entity brought to light two additional sets of repetitions. The attribute 'year' would be repeated constantly, therefore it was extracted into its own entity set with a one-to-many relationship with vehicle. Additionally, the 'type' attribute repeats one of the three options: CDL vehicle, standard vehicle, or accessory. I gave 'type' its own entity with a one-to-many relationship with vehicle as well.

Note: The Year to Vehicle entity relationship can be denormalized. Year may remain as an attribute of the Vehicle entity.

	~~Entity #1~~	~~Participation~~	~~Relationship~~	~~Plurality~~	~~Entity #2~~
~~14a~~	~~A year~~	~~May~~	~~Apply to~~	~~Multiple~~	~~Vehicles~~
~~14b~~	~~A vehicle~~	~~Must~~	~~Associate with~~	~~One~~	~~Year~~

	Entity #1	Participation	Relationship	Plurality	Entity #2
14a	A type	May	Apply to	Multiple	Vehicles
14b	A vehicle	Must	Have	One	Type

Derived for History Table

- A PartHistory table will monitor the Parts table to track the history of quantity changes to different parts over time.

As each part's quantities will fluctuate with use, sometimes wildly, we know that a single part can have multiple quantity changes. There is an extremely rare case where a part never gets used despite the shop having quantity on the shelf, so we will use optional participation 'may'. Looking in reverse, each record in the PartHistory table reflects the quantity-at-time status of only one part. If the record exists, a quantity change 'must' have occurred and likewise the part 'must' have existed.

	Entity #1	Participation	Relationship	Plurality	Entity #2
H-a	A part	May	Have	Multiple	Quantity changes
H-b	A quantity change	Must	Apply to	One	Part

The resulting PartHistory table will have a synthetic key ID, a foreign key back to the part it references, the old part quantity, the new part quantity and the date of the change as attributes. The mechanism for update will be the creation of a trigger that monitors any update activity on the Parts table where changes in quantity have occurred.

Specialization / Generalization Rules

Supertype: Maintainer ***Subtypes: Mechanic, Vendor***

Justification

- Identified above in the vendor/maintenance activity and mechanic/maintenance activity relationships, vendor and mechanic may function the same. A mechanic can work on a vehicle in-house, or a vendor can work on a vehicle sent out to their remote garage. We can generalize an entity supertype that encapsulates these two entities as subtypes and call it 'Maintainer'.

Completeness

- In terms of the database, we only want maintenance activities performed by either a mechanic or a vendor. We do not support any third-party option, as anyone not a mechanic can be configured in the application program as a 'vendor' of service if need be. Therefore, the relationship should be totally complete.

Disjointedness

- There is no case where an outside vendor would also be an in-house mechanic in this program. Regarding the local municipality application, it is not a good practice for a mechanic on hourly payroll to take money as a vendor after hours or moonlighting to perform a vehicle maintenance for the municipality. As such, the subtypes will be disjoint.

	Supertype		Subtype #1	Subtype #2	Disjointedness?	Completeness?
15	A maintainer	Is	A mechanic	Or a vendor	--	--

Supertype: Appointment Subtypes: Automatic Appointment, Manual Appointment

Justification

- In the appointments use case, it states that an appointment can either be created manually by a mechanic who fills in the details, or auto generated by the application based on some monitoring factor. Common flags for automatic appointments are mileage (3,000 to 5,000 miles from last oil change flags another maintenance), and inspection date (starting 3 months prior to a vehicle's inspection deadline).

Completeness

- These two subtypes will be the only types of appointment made by this program. Therefore, we will say it is totally complete.

Disjointedness

- An appointment cannot be manual and automatic at the same time. Either the application creates it or a mechanic does.

	Supertype		Subtype #1	Subtype #2	Disjointedness?	Completeness?
16	An appointment	Is	An automatic appointment	Or a manual appointment	--	--

~~***Supertype: Vehicle Subtypes: CDL Vehicle, Standard Vehicle, Accessory***~~

I have decided to remove the subtypes previously associated with Vehicle. When considering the need to extract out CDLVehicle, StandardVehicle and Accessory, I found that none will require unique attributes not already found in the supertype. Therefore, upon implementation I will only use vehicle but add in a 'vehicle_type' attribute which will discern the types of vehicles.

Supertype: Maintenance_activity Subtypes: Vendor_repair, Inspection

Justification

- In reviewing my attributes for week 4, I realized that there are a few different styles of maintenance activities that are significantly different from the rest. For instance, an Inspection should have fields that check tire treads, brakes, engines, lights, etc. that are all part of a state inspection requirement. Normal maintenance might not go into that level of detail. Also, a vendor repair might have repair warranty information or repair invoice linking number not otherwise found in maintenance activities.

Completeness

- The scenario I'm envisioning is that most of the activities will be standard maintenance activities, handled by the supertype. Occasionally a maintenance will be an inspection or vendor repair, which will be handled by the subtypes. Thus, the relationship is partially complete, allowing for optional subtype membership.

Disjointedness

- For the purposes of the database, a maintenance activity should be standard (the supertype itself), a vendor repair or an inspection. Since maintenance activities are performed my maintainers, and maintainers may include vendors, this takes care of the possibility an inspection could be done by an outside vendor, which is rare. If we get into the weeds concerning shared repairs, those are two different maintenance activities as viewed by the program. As such, the subtypes are disjoint.

	Supertype		Subtype #1	Subtype #2	Disjointedness?	Completeness?
17	A maintenance activity	Is	A vendor repair	Or an inspection	--	Or none of these.

Here are my final structural database rules:

Associative Database Rules

- **A vehicle make may apply to many vehicles; each vehicle must have one vehicle make.**
- **A vehicle model may correspond to many vehicles; each vehicle must correspond to one vehicle model.**
- **A maintenance activity must apply to one vehicle; each vehicle may have many maintenance activities.**
- **A maintenance activity may use many parts; each part may be used in many maintenance activities.**
- **A vehicle may contain many parts; each part may be contained in many vehicles.**
- **A vendor may supply many parts, each part must be supplied by one vendor.**
- **A mechanic may be assigned to many appointments; each appointment must be assigned to one mechanic.**
- **A vehicle may be assigned to many appointments, each appointment must be assigned to one vehicle.**
- **A vendor repair must be performed by one vendor; each vendor may perform many vendor repairs.**
- **An inspection must have many tests; each test may occur in many inspections.**
- **A test must have many results; each result must occur in many tests.**
- ~~**A year may apply to multiple vehicles; each vehicle must associate with one year. (Removed)**~~
- **A type may apply to multiple vehicles; each vehicle must have one type.**
- **A part may have multiple quantity changes; each quantity change must apply to one part.**

By nature of the generalization rule to follow, the mechanic/maintenance activity and vendor/maintenance activity can now be replaced by maintainer/maintenance activity. Note, maintenance activity previously had an optional relationship toward both mechanic and vendor. This was to allow for the fact that it may be a mechanic or a vendor doing the maintaining. With the creation of the *Maintainer* supertype, it now encapsulates all possible performers of the maintenance, so a maintenance activity's relationship toward maintainer is mandatory.

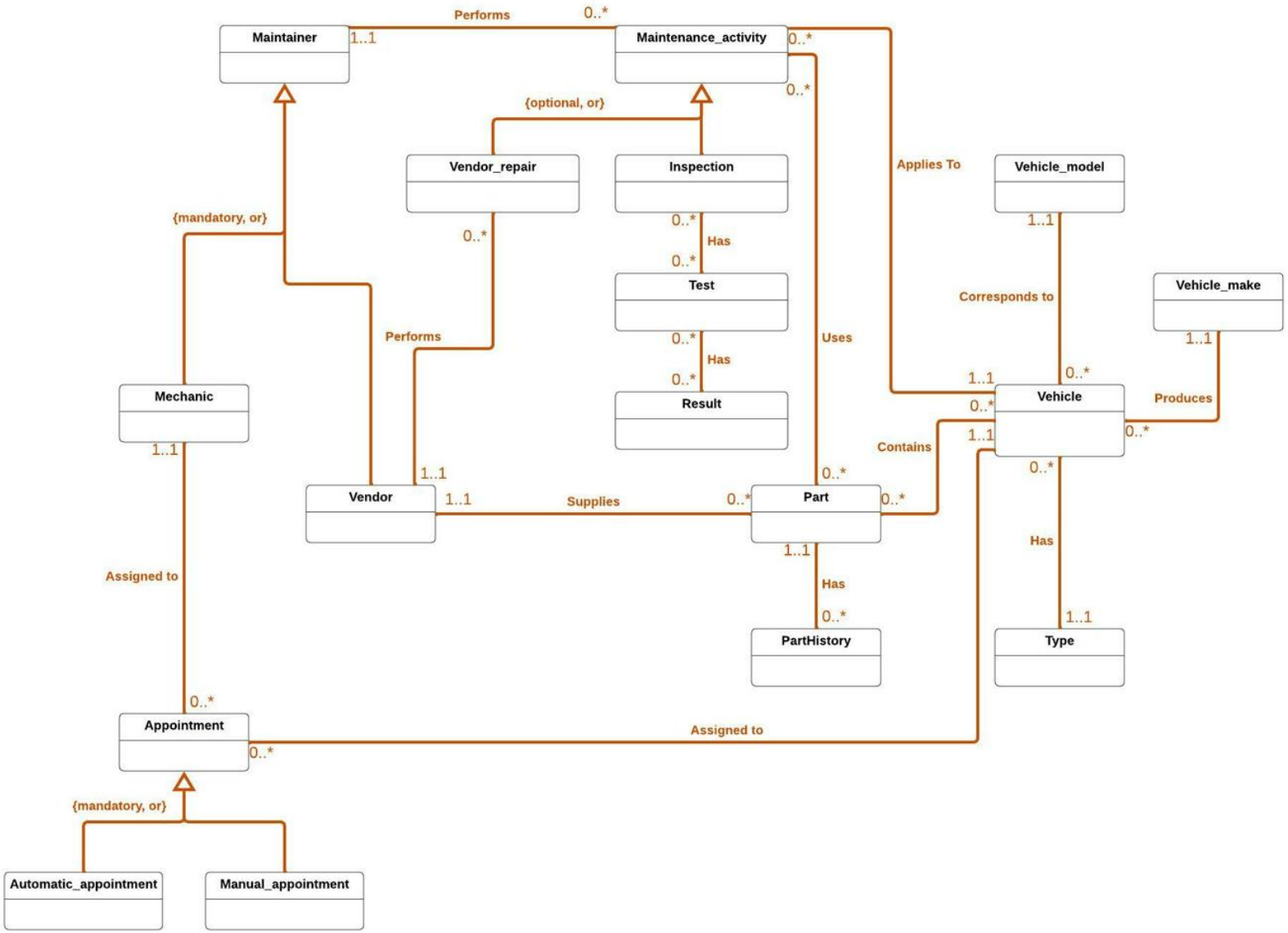
- ~~**A mechanic may perform many maintenance activities; each maintenance activity may be performed by one mechanic. (Removed)**~~
- ~~**A vendor may perform many maintenance activities; each maintenance activity may be performed by one vendor. (Removed)**~~
- **A maintainer may perform many maintenance activities, each maintenance activity must be performed by one maintainer.**

Specialization / Generalization Database Rules

- **A maintainer is a mechanic or a vendor.**
- **An appointment is an automatic appointment or a manual appointment.**
- ~~**A vehicle is a CDL vehicle, a standard vehicle, an accessory, or none of these. (Removed)**~~
- **A maintenance activity is a vendor repair or an inspection or none of these.**

Conceptual Entity-Relationship Diagram

CS 669 O2 – Term Iteration #4 Conceptual ERD, Ed Myers



Associative Database Rules

- A vehicle make may apply to many vehicles; each vehicle must have one vehicle make.
- A vehicle model may correspond to many vehicles; each vehicle must correspond to one vehicle model.
- A maintenance activity must apply to one vehicle; each vehicle may have many maintenance activities.
- A maintenance activity may use many parts; each part may be used in many maintenance activities.
- A vehicle may contain many parts; each part may be contained in many vehicles.
- A vendor may supply many parts, each part must be supplied by one vendor.
- A mechanic may be assigned to many appointments; each appointment must be assigned to one mechanic.
- A vehicle may be assigned to many appointments, each appointment must be assigned to one vehicle.
- A maintainer may be assigned to many appointments; each appointment must be assigned to one maintainer.
- An inspection must have many tests; each test may occur in many inspections.
- A test must have many results; each result must occur in many tests.
- A type may apply to multiple vehicles; each vehicle must have one type.
- A part may have multiple quantity changes; each quantity change must apply to one part.

Specialization / Generalization Database Rules

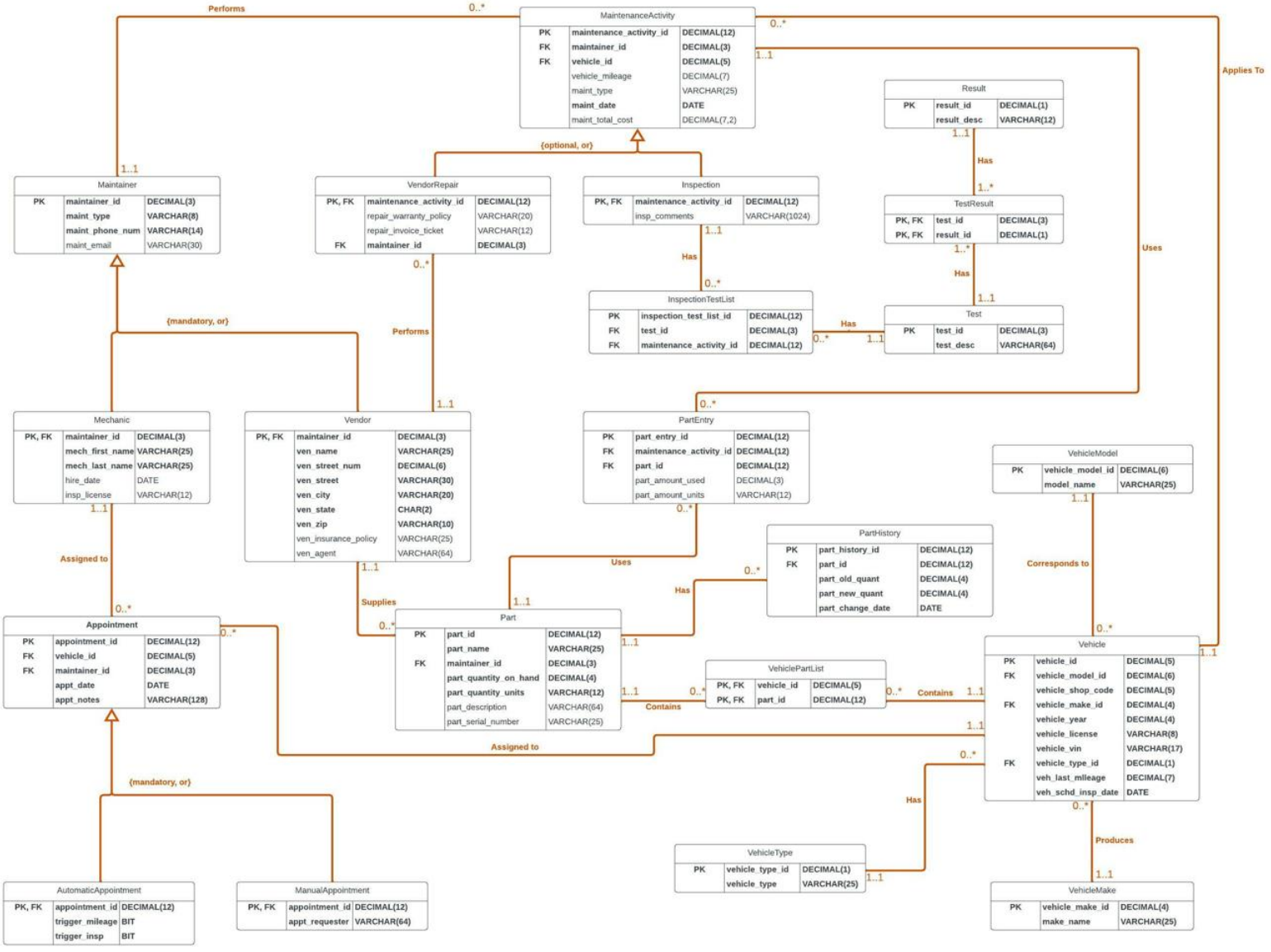
- A maintainer is a mechanic or a vendor.
- An appointment is an automatic appointment or a manual appointment.
- A maintenance activity is a vendor repair or an inspection or none of these.

From previous instances of the conceptual ERD, the following changes have been made:

- The addition of the *Appointment* entity, along with the expansion of its subsets *Automatic_appointment* and *Manual_appointment.* The relationship is {mandatory, or} as an appointment must be one of the two subtypes and cannot be more than one subtype at the same time.
- The *Vehicle* entity is no longer expanded into subsets *CDL_vehicle, Standard_vehicle* and *Accessory.*
- The addition of *Vehicle_make* and *Vehicle_model* above *Vehicle* explain relationships between those entities. These will be necessary tables within the database to reduce redundancy of data. The relationships have been adjusted so that both relate back to the main *Vehicle* entity rather than each other.
- The similarity between mechanics and vendors performing a maintenance activity was solved by creating the generalization *Maintainer. Maintainer* now possesses the subtypes *Mechanic* and *Vendor.* The relationship is {mandatory, or} as a maintainer must be classified as one of its subtypes within the program, and a mechanic cannot be a vendor at the same time.
- With the addition of inspection, entities Test and Result were discovered in normalization. As their relationships were Inspection (M->N) Test (M->N) Result, two more bridge entities InspectionTestList and TestResult will be created in the physical ERD to reconcile.
- The Year entity has been collapsed back within Vehicle for denormalization.
- The PartHistory entity has been added with a (1->M) relationship to the part entity per the new rule.

Full DBMS Physical ERD

CS 669 O2 – Term Iteration #4 Physical ERD, Ed Myers



Justification of Attributes

Base Entity	Attribute	Data Type/Precision	Reasoning
Maintainer	maintainer_id	DECIMAL(3) PK	Adequate synthetic PK for maintainer
	maint_type	VARCHAR(8)	Allows for entry 'mechanic' or 'vendor'
	maint_phone_num	VARCHAR(14)	'x-xxx-xxx-xxxx' format, including dashes
	maint_email	VARCHAR(30)	Captures most long emails
Maintainer is a supertype that holds common attributes for the subtypes 'Mechanic' and 'Vendor'. The PK maintainer_id is mandatory, as is type of maintainer and phone number. Maintainer_id is 3 long as I don't anticipate more than 999 mechanics/vendors in this database. These should be required to identify who is doing the maintaining and how to get a hold of them. Maintainer email is nullable only because there are a select few who do not use email. **Maintainer_id is a synthetic primary key. (maint_first_name, maint_last_name) should not be considered a composite candidate key as two individuals or vendors. No determinants exist that are not candidate keys. Maintainer is in BCNF.**			
Mechanic	maintainer_id	DECIMAL(3) PK,FK	Subtype of Maintainer
	hire_date	DATE	Establishes seniority and experience of mechanic
	mech_first_name	VARCHAR(25)	Captures a variety of first names
	mech_last_name	VARCHAR(25)	Captures a variety of last names
	insp_license	VARCHAR(12)	given 12 characters for letter/number formatting and multi-class licenses
Mechanic is a subtype of Maintainer, which holds the common attributes. The mandatory fields here are maintainer_id, mech_first_name and mech_last_name. The other variables are nullable, as hire_date is not especially necessary except for inter-personnel uses and some mechanics do not have or are working toward acquiring an inspection license. **Maintainer_id is the only primary key, and all other attributes rely on it to map back to a specific mechanic. Mechanic is in BCNF.**			
Vendor	maintainer_id	DECIMAL(3) PK,FK	Subtype of Maintainer
	ven_name	VARCHAR(25)	Name of vendor
	ven_street_num	DECIMAL(6)	Street number of vendor address
	ven_street	VARCHAR(30)	Street name of vendor address
	ven_city	VARCHAR(20)	City name of vendor address
	ven_state	CHAR(2)	State 2-char abbreviation of address
	ven_zip	VARCHAR(10)	Allows for 'xxxxx-xxxx' zipcode format
	ven_insurance_policy	VARCHAR(25)	Township requests proof of insurance from vendors who we work with. Allows for multiple character types
	ven_agent	VARCHAR(64)	Personal or regional representative of the vendor

Vendor is a subtype of Maintainer, which holds the common attributes. Maintainer_id, name, street number, street name, city, state, and zip are all mandatory. Insurance policy is nullable as a vendor may be large and reputable enough not to require an on-hand policy id. We also may not have a dedicated vendor assigned as agent. **A note here about NF: there will be inevitable redundancy most likely with city, state and zip. I recognize to attain BCNF these should be extracted out to remove repetition. I made a design choice to keep them the way it is given the limited number of vendors used with a nod to efficiency.**			
Appointment	appointment_id	DECIMAL(12) PK	Synthetic PK for appointment
	vehicle_id	DECIMAL(5) FK	FK to vehicle entity
	maintainer_id	DECIMAL(3) FK	FK to maintainer entity
	appt_date	DATE	Date of appointment
	appt_notes	VARCHAR(128)	Ample space for a decent note for why the appointment was scheduled.
Appointment is a supertype to AutomaticAppointment and ManualAppointment. The PK and FK values are mandatory, as well as the appointment date and notes. An appointment would be useless without a date to apply to and a reason to have it. **Appointment_id is the sole determiner of the appointment to which all other attributes tie to in describing an appointment instance. Appointment is in BCNF.**			
Automatic-Appointment	appointment_id	DECIMAL(12) PK,FK	Subtype of Appointment
	trigger_mileage	BIT	Trigger flag that application sets/checks to generate an automatic appointment. An application event triggers code that looks at the mileage from a vehicle's current activity and compares it to the vehicles last mileage attribute. Given the mileage difference and the vehicle type, the application set the flag appropriately.
	trigger_insp	BIT	Trigger flag that application sets/checks to generate an automatic appointment. The application checks the vehicle's scheduled inspection date against the current date to determine if it is ready to set this flag off.
AutomaticAppointment is a subtype of Appointment, which holds the common attributes. The two triggers are single BIT flags meant to act as Booleans for the application and are mandatory. If an automatic appoint fires off, it should have one or both of these flags set to 1 to populate the note attribute message properly upon appointment generation. **Appointment_id is the sole PK, with the other attributes depending on it, thus AutomaticAppointment is in BCNF.**			
Manual-Appointment	appointment_id	DECIMAL(12) PK,FK	Subtype of Appointment

	appt_requester	VARCHAR(64)	Holds person or entity responsible for requesting the maintenance, if not the maintainer themselves.
ManualAppointment is a subtype of Appointment, which holds the common attributes. The appt_requester attribute is mandatory as there must be a person responsible for initializing the request if not automatic. **The supertype appointment_id is the only determinant in the entity table, thus ManualAppointment is in BCNF.**			
Part	part_id	DECIMAL(12) PK	Synthetic PK for part
	part_name	VARCHAR(25)	Short part name identifier
	maintainer_id	DEICMAL(3) FK	FK to maintainer entity
	part_quantity_on_ hand	DECIMAL(4)	Quantity of part in stock. Not expecting beyond 9999 of any part in stock.
	part_quantity_units	VARCHAR(12)	Units describing part quantity (gals, pieces, tubes, quarts, etc.)
	part_description	VARCHAR(64)	Short description of part and what its for
	part_serial_number	VARCHAR(25)	Alphanumeric ID code usually vendor-dependent for ordering supply.
Part contains all attributes used to identify a vehicle part and where its used. Mandatory attributes are part_id, part_name and the foreign_key maintainer_id, which is used to identify which vendor the part came from. Quantity on hand is also required so that the application can show how much of a part is in stock during a maintenance activity entry. The application deducts the number used from this quantity to keep a running total. Units to qualify the quantity are also required. Part description and serial number are not required, as something like a bolt or nut may be self-explanatory and not have a proper vendor number. **Part_serial_number may act as a candidate key to Part, but is unreliable as you are taking one vendor's numbering convention against the other vendors' conventions and hoping there isn't a uniqueness violation. Thus, part_id is the sole reliable determinant and Appointment is in BCNF.**			
Maintenance-Activity	maintenance_- activity_id	DECIMAL(12) PK	Synthetic PK for the maintenance activity
	maintainer_id	DECIMAL(3) FK	FK linking back to the maintainer
	vehicle_id	DECIMAL(5) FK	FK linking back to the vehicle
	vehicle_mileage	DECIMAL(7)	Mileage input for the maintenance entry
	maint_type	VARCHAR(25)	Maintenance type, either "Standard Maintenance" (*supertype), "*Vendor Repair" or "Inspection"
	maint_date	DATE	Date of maintenance activity
	maint_total_cost	DECIMAL (7,2)	Total cost of parts for that specific activity
Maintenance activity has the required fields, maintenance_activity_id, maintainer_id, vehicle_id, and maint_date. Vehicle mileage is nullable since a maintenance may be on a piece of equipment where			

mileage isn't tracked or a mileage appointment flag is undesired. If maint_type is left null, then the application assumes the activity is a standard maintenance (supertype). Cost is nullable if no parts or materials were used for the maintenance or of such low quantity that it was not tracked. **Maintenance_activity_id is the only determinant of the MaintenanceActivity entity, with all other attributes relying on it to describe the activity instance. Thus, MaintenanceActivity is in BCNF.**			
VendorRepair	maintenance_-activity_id	DECIMAL(12) PK,FK	Subtype of maintenance activity
	repair_warranty_-policy	VARCHAR(20)	Work guarantee policy identifier, if it exists.
	repair_invoice_ticket	VARCHAR(12)	Invoicing number for document tracking. Actual cost should be entered into the maint_total_cost in the supertype.
	maintainer_id	DECIMAL(3) FK	FK linking back to the maintainer
VendorRepair has maintenance_activity_id and maintainer_id as mandatory fields. The first is a synthetic PK, while the second satisfies the fact that a vendor repair requires linking with a known vendor. Warranty information may or may not exist depending on the type of vendor repair, similar to an invoicing ticket. **The maintenance_activity_id PK is the only determinant that describes the entire VendorRepair instance, thus VendorRepair is in BCNF.**			
Inspection	maintenance_-activity_id	DECIMAL(12) PK,FK	Subtype of maintenance activity
	insp_comments	VARCHAR(1024)	Ample room for inspection comments
The Inspection entity originally took a form that possessed many tests and attributes. The process of normalization created two more entity tables, Test and Result. These tables to be explained below, handled the realization that Inspection (M->N) Test, and Test (M->N) Results. Thus two bridge entities, InspectionTestList, and TestResult were created to handle those cases. **Maintenance_activity_id is the only determinant for Inspection, thus Inspection is in BCNF.**			
Test	test_id	DECIMAL(3) PK	Synthetic PK for the test activity
	test_desc	VARCHAR(64)	Modest explanation of the test
The Test entity builds out the unique test table which InspectionTestList pulls from to generate unique inspections. All fields are mandatory. **As test_id is the sole determinant of the entity, Test is in BCNF.**			
Inspection_-Test_List	inspection_test_-list_id	DECIMAL(12) PK	Synthetic PK for the InspectionTestList bridge entity
	test_id	DECIMAL(3) FK	FK linking back to test
	maintenance_-activity_id	DECIMAL(12) FK	FK linking back to the inspection (subtype of maintenance activity)
The sizing of test_id relates to there being less than a thousand tests in practicality that can apply to an inspection. **Inspection_Test_List is a bridge entity using FKs to link back to establish its 1-M relationships. I've given it a synthetic PK for tracking, and since it is the only determinant it is in BCNF.**			

Result	result_id	DECIMAL(1) PK	Synthetic PK for the result
	result_desc	VARCHAR(12)	'pass', 'fail', 'see comments'
As there are only three options for result (pass, fail, see comments), those options can be represented using one decimal value. **Result has one PK as a determinant, thus it is in BCNF.**			
TestResult	test_id	DECIMAL(3) PK,FK	PK bridge, FK linking back to the test
	result_id	DECIMAL(1) PK,FK	PK bridge, FK linking back to the result
TestResult is a bridge entity that I am not planning on tracking, so I will leave it to be populated only by its foreign keys. **Thus TestResult is in BCNF.**			
PartEntry	part_entry_id	DECIMAL(12) PK	Synthetic PK for part entry
	maintenance_-activity_id	DECIMAL(12) FK	FK linking to maintenance activity
	part_id	DECIMAL(12) FK	FK linking back to part
	part_amount_used	DECIMAL(3)	Quantity of part used
	part_amount_units	VARCHAR(12)	Units that part quantity is measured in
I made a synthetic PK for PartEntry even though it is a bridge entity between Part and MaintenanceActivity as it additionally tracks amount used. Part_amount_used is of DECIMAL(3) size because I can't see using more than 999 of a part or material in a single maintenance activity. **Part_id is the sole determinant, therefore PartEntry is in BCNF.**			
VehiclePartList	vehicle_id	DECIMAL(5) PK,FK	PK bridge, FK linking back to vehicle
	part_id	DECIMAL(12) PK,FK	PK bridge, FK linking back to part
VehiclePartList is a bridge entity possessing two foreign keys. I do not intend to track it with a primary key as it holds no other additional data. Therefore it is in BCNF.			
Vehicle	vehicle_id	DECIMAL(5) PK	Synthetic PK for vehicle_id
	vehicle_model_id	DECIMAL(6) FK	FK linking to vehicle model entity
	vehicle_shop_code	DECIMAL(5)	In-house vehicle code number
	vehicle_make_id	DECIMAL(4) FK	FK linking to vehicle make entity
	vehicle_year	DECIMAL(4)	Holds year value for vehicle year
	vehicle_license	VARCHAR(8)	Vehicle license plate value
	vehicle_vin	VARCHAR(17)	Vehicle Identification Number, created by the manufacturer
	vehicle_type_id	DECIMAL(1) FK	FK linking to the type entity
	veh_last_mileage	DECIMAL(7)	Last mileage recorded for the vehicle
	veh_schd_insp_date	DATE	Scheduled upcoming inspection date
In normalizing the Vehicle entity, repeated values were found with the 'year' and 'type' attributes. As these were transitive dependencies, I've split them out into their own entities with many-to-one relationships back to the vehicle. The vehicle_ID is a synthetic key, but the vehicle shop code is an internal numeric id using the naming convention of the municipality (ex./ 4002 is the 2nd vehicle within			

the highway fleet, identified by the leading 4). It is allowed because it acts like a candidate key. Vehicle license and VIN are sized appropriately to their length. Vehicle last mileage and vehicle scheduled inspection date are used by the appointment entity as checks to determine if a mileage or inspection appointment is needed. **All transitive dependencies have been removed from Vehicle to their own entities. Vehicle_id remains the PK primary determinant. An argument can be made that the VIN due to uniqueness is a candidate key. Even though this is allowed, it will not be used for indexing within this application. Therefore Vehicle is in BCNF.**			
VehicleModel	vehicle_model_id	DECIMAL(6) PK	Synthetic PK for vehicle model
	model_name	VARCHAR(25)	Description of model
Vehicle_model_id and model_name are sized within reason for their uses. I went a little large with model, but seeing as the uses may extend beyond traditional vehicles and into other equipment, its better to be safe. **Vehicle_model_id is the only PK determinant, therefore VehicleModel is in BCNF.**			
VehicleMake	vehicle_make_id	DECIMAL(4) PK	Synthetic PK for vehicle make
	make_name	VARCHAR(25)	Description of make
Same idea and description as for VehicleModel. **Vehicle_make_id is the only PK determinant, therefore VehicleMake is in BCNF.**			
~~**VehicleYear**~~	~~vehicle_year_id~~	~~DECIMAL(2) PK~~	~~Synthetic PK for year~~
	~~vehicle_year~~	~~DECIMAL(4)~~	~~Holds year value for vehicle year~~
Vehicle Year entity has been removed and folded into Vehicle by denormalization.			
VehicleType	vehicle_type_id	DECIMAL(1) PK	Synthetic PK for type
	vehicle_type	VARCHAR(25)	'CDL Vehicle', 'Standard Vehicle' or 'Accessory'
There are only three anticipated types as listed, so the type_id having one decimal satisfies the indexing need. In the event more are created in the future, six index placement remain. **Vehicle type_id is the only PK determinant, therefore VehicleType is in BCNF.**			
PartHistory	part_history_id	DECIMAL(12) PK	Synthetic PK for part history
	part_id	DECIMAL(12) FK	FK linking to part entity
	part_old_quant	DECIMAL(4)	Old, replaced quantity value
	part_new_quant	DECIMAL(4)	New quantity value
	part_change_date	DATE	Date of quantity change
This entity serves as a history table to track quantity changes of parts over time. All fields are mandatory. Part_history is sized to reflect the possibility of there being many part quantity changes in the history table, and part_id's sizing reflects the key it references in Part. The old and new quantity fields are sized the same as the active quantity field within the Part entity. **As part_history_id is the only PK and no other attribute can serve as a candidate key to the group, PartHistory is in BCNF.**			

The DBMS physical ERD greatly resembles the conceptual ERD created and updated earlier. Differences of note:

- Two bridge entities had to be created to deal with the M:N relationships between *Maintenance Activity/Part* and *Part/Vehicle.*
    - *PartEntry* represents the bridge for the first set, which makes sense if you think of a maintenance entry initiating a unique part entry into a maintenance activity record for each part used per each separate maintenance. *PartEntry* contains the foreign keys for both *MaintenanceActivity* and *Part.*
    - *VehiclePartList* reconciles the relationship for the second set, which works because each individual vehicle should have a customized part list attached to it, regardless of whether a part on the list could fit in a different vehicle. *VehiclePartList* contains the foreign keys for both *Vehicle* and *Part*.

- From the conceptual ERD, subtypes VendorRepair and Inspection were added to specify the supertype MaintenanceActivity. This spawned an additional one to one relationship between Vendor and VendorRepair, as for our case only one vendor can make a repair that counts as a single maintenance activity.

- In the process of creating the Inspection entity, due to transitive repetition I found the need to create a Test entity which represents the numerous tests an inspector would need to perform to pass a vehicle's inspection maintenance activity. Recursively, I found again that a test could have multiple results, so I created a Result entity to hold the possibilities 'pass', 'fail', and 'see comments'.

- From the use cases built for these entities, I found that Inspection <-> Test and Test <-> Result were both many-to-many relationships. This necessitated two bridge entities to be created, one called 'InspectionTestList' and the other 'TestResult', which solved the problems.

- From the normalization of Vehicle, I found that the attributes 'year' and 'type' were subject to much repetition. Therefore I extracted them out of the Vehicle entity and created one-to-many relationships between each back to vehicle.

- VehicleYear is now denormalized to become an attribute within Vehicle.

- Primary keys for VehiclePartList and TestResult have been created.

- Part_name was added as an attribute to the Part entity.

- PartHistory was added as a history tracking table with a 1-M relationship with Part. No bridge entities are required.

Many of my relationships are 1:M, so where you see that type of relationship I have placed a foreign key reference in the entity of the many side back to the entity ID of the one side.

Style-wise, I used the entity relationship structures within LucidChart rather than the stock UML class shapes. I liked how each structure fit the key type, entity ID and data type in a more organized way.

Stored Procedure Execution and Explanations

Use Case: Add Vehicle

The scenario exists where a user needs to enter a new vehicle into the program application. A refinement of the *Add Vehicle* use case is given below:

1) The mechanic clicks the 'Add Vehicle' button on the application interface.
2) The application provides a fill form to accept the data fields.

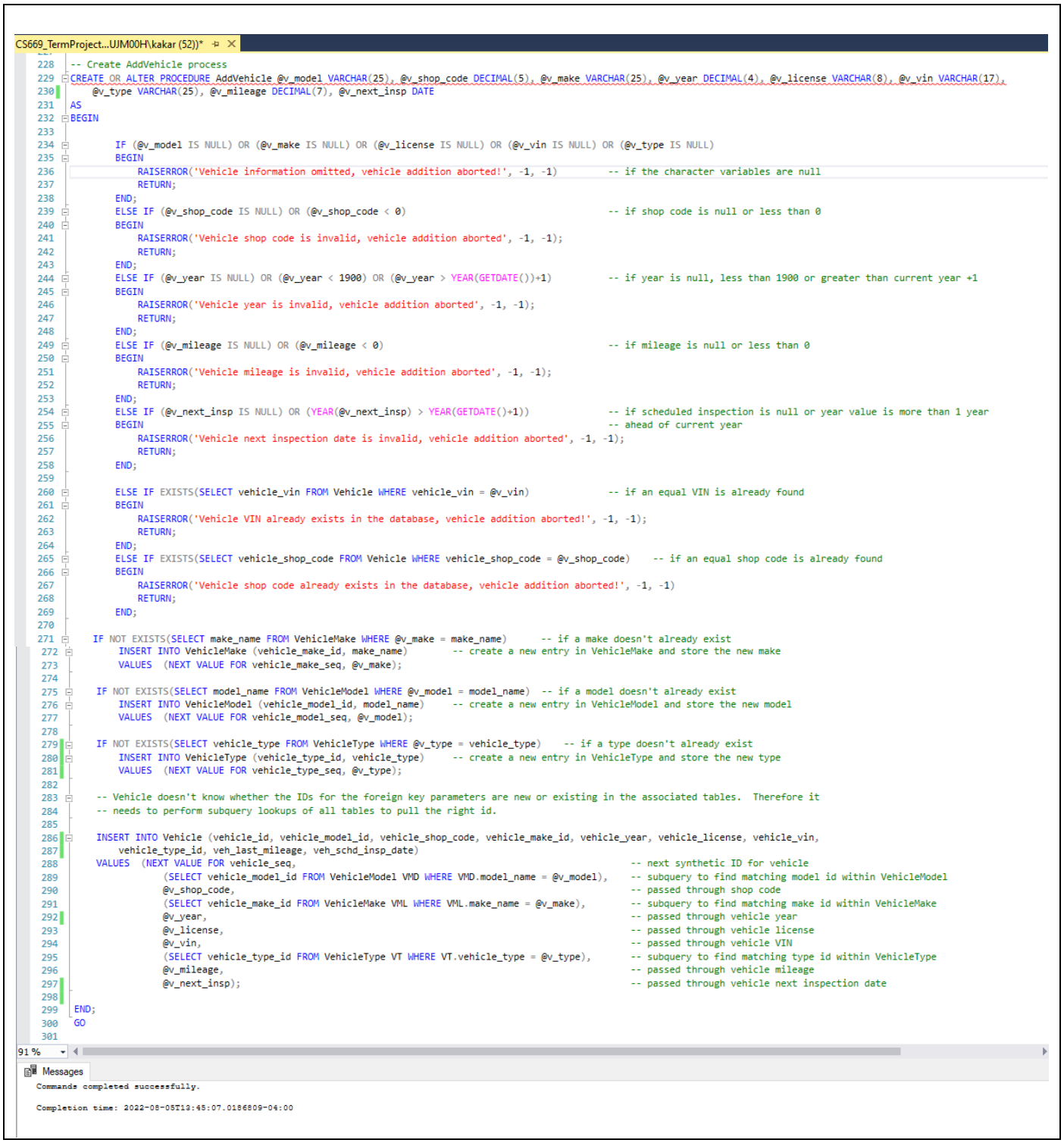
model	The model of the vehicle	Only certain parts work on certain vehicles.
vehicle ID (shop code)	The 4-digit departmental ID code of the vehicle	This is the naming convention already used, and uniquely identifies the vehicle.
make	The brand of the vehicle	Only certain parts work on certain vehicles.
year	The creation year of the vehicle	The year of the vehicle is the primary statistic used to determine parts availability and to evaluate replacement. The local township participates in a fleet lease program in which a mixed trade in of vehicles 3 years and older reduce the overall fleet age to save on maintenance costs.
license	The license plate value	You can check which vehicles need renewed registration and pull status of insurance.
VIN	The auto manufacturer vehicle identification number	The coding of the VIN gives exact vehicle specifications, and the VIN is required on nearly all types of registration paperwork.
vehicle type	The type of vehicle, being a CDL vehicle, standard vehicle, accessory, or none	This is a new attribute created for the generalization/specialization rule set. Depending on the type of vehicle, different information will be available and different check flags will apply to appointments.
last mileage	The last recorded mileage of the vehicle	This is used as a comparison with a new mileage entered from latest activity to throw a flag that triggers an automatic appointment to be created by the program if satisfied.
Inspection date	The date the vehicle is due for its next inspection	The inspection date is important to keep the vehicle street-legal, and serves as a parameter by which the application's appointment notifier communicates a necessary vehicle maintenance activity to the mechanic.

3) The mechanic specifies the type of vehicle as *CDL vehicle, standard vehicle, accessory,* or *none.*
4) The mechanic enters all other the appropriate information and clicks 'Save'.

The stored procedure accepts the vehicle model, shop code, make, year, license, VIN, type (*CDL Vehicle, Standard Vehicle,* or *Accessory*) mileage and next inspection date. The procedure attempts to be robust by blocking null variables, out of range numeric values and unrealistic years/dates. It also prevents duplicate inputs of the VIN and shop code, which are the closest things to alternate candidate keys.

The core Vehicle entity possesses four foreign key dependencies to lookup tables, so those tables must be sequenced appropriately. To maintain uniqueness, each lookup table insertion first checks for a duplicate entry prior to creating a new sequence id. If none is found, it creates a new entry. Lastly, when entering all information as an insertion into the Vehicle entity, all sequence IDs for the foreign keys must be looked up by subquery.

Here is my stored procedure definition using SQL Server:



```
CS669_TermProject...UJM00H\kakar (52))*

-- Create AddVehicle process
CREATE OR ALTER PROCEDURE AddVehicle @v_model VARCHAR(25), @v_shop_code DECIMAL(5), @v_make VARCHAR(25), @v_year DECIMAL(4), @v_license VARCHAR(8), @v_vin VARCHAR(17),
    @v_type VARCHAR(25), @v_mileage DECIMAL(7), @v_next_insp DATE
AS
BEGIN

        IF (@v_model IS NULL) OR (@v_make IS NULL) OR (@v_license IS NULL) OR (@v_vin IS NULL) OR (@v_type IS NULL)
        BEGIN
            RAISERROR('Vehicle information omitted, vehicle addition aborted!', -1, -1)         -- if the character variables are null
            RETURN;
        END;
        ELSE IF (@v_shop_code IS NULL) OR (@v_shop_code < 0)                                    -- if shop code is null or less than 0
        BEGIN
            RAISERROR('Vehicle shop code is invalid, vehicle addition aborted', -1, -1);
            RETURN;
        END;
        ELSE IF (@v_year IS NULL) OR (@v_year < 1900) OR (@v_year > YEAR(GETDATE())+1)          -- if year is null, less than 1900 or greater than current year +1
        BEGIN
            RAISERROR('Vehicle year is invalid, vehicle addition aborted', -1, -1);
            RETURN;
        END;
        ELSE IF (@v_mileage IS NULL) OR (@v_mileage < 0)                                        -- if mileage is null or less than 0
        BEGIN
            RAISERROR('Vehicle mileage is invalid, vehicle addition aborted', -1, -1);
            RETURN;
        END;
        ELSE IF (@v_next_insp IS NULL) OR (YEAR(@v_next_insp) > YEAR(GETDATE()+1))              -- if scheduled inspection is null or year value is more than 1 year
        BEGIN                                                                                   -- ahead of current year
            RAISERROR('Vehicle next inspection date is invalid, vehicle addition aborted', -1, -1);
            RETURN;
        END;

        ELSE IF EXISTS(SELECT vehicle_vin FROM Vehicle WHERE vehicle_vin = @v_vin)              -- if an equal VIN is already found
        BEGIN
            RAISERROR('Vehicle VIN already exists in the database, vehicle addition aborted!', -1, -1);
            RETURN;
        END;
        ELSE IF EXISTS(SELECT vehicle_shop_code FROM Vehicle WHERE vehicle_shop_code = @v_shop_code)    -- if an equal shop code is already found
        BEGIN
            RAISERROR('Vehicle shop code already exists in the database, vehicle addition aborted!', -1, -1)
            RETURN;
        END;

    IF NOT EXISTS(SELECT make_name FROM VehicleMake WHERE @v_make = make_name)      -- if a make doesn't already exist
        INSERT INTO VehicleMake (vehicle_make_id, make_name)        -- create a new entry in VehicleMake and store the new make
        VALUES  (NEXT VALUE FOR vehicle_make_seq, @v_make);

    IF NOT EXISTS(SELECT model_name FROM VehicleModel WHERE @v_model = model_name)  -- if a model doesn't already exist
        INSERT INTO VehicleModel (vehicle_model_id, model_name)     -- create a new entry in VehicleModel and store the new model
        VALUES  (NEXT VALUE FOR vehicle_model_seq, @v_model);

    IF NOT EXISTS(SELECT vehicle_type FROM VehicleType WHERE @v_type = vehicle_type)    -- if a type doesn't already exist
        INSERT INTO VehicleType (vehicle_type_id, vehicle_type)     -- create a new entry in VehicleType and store the new type
        VALUES  (NEXT VALUE FOR vehicle_type_seq, @v_type);

    -- Vehicle doesn't know whether the IDs for the foreign key parameters are new or existing in the associated tables.  Therefore it
    -- needs to perform subquery lookups of all tables to pull the right id.

    INSERT INTO Vehicle (vehicle_id, vehicle_model_id, vehicle_shop_code, vehicle_make_id, vehicle_year, vehicle_license, vehicle_vin,
        vehicle_type_id, veh_last_mileage, veh_schd_insp_date)
    VALUES  (NEXT VALUE FOR vehicle_seq,                                                    -- next synthetic ID for vehicle
             (SELECT vehicle_model_id FROM VehicleModel VMD WHERE VMD.model_name = @v_model),    -- subquery to find matching model id within VehicleModel
             @v_shop_code,                                                                  -- passed through shop code
             (SELECT vehicle_make_id FROM VehicleMake VML WHERE VML.make_name = @v_make),       -- subquery to find matching make id within VehicleMake
             @v_year,                                                                       -- passed through vehicle year
             @v_license,                                                                    -- passed through vehicle license
             @v_vin,                                                                        -- passed through vehicle VIN
             (SELECT vehicle_type_id FROM VehicleType VT WHERE VT.vehicle_type = @v_type),      -- subquery to find matching type id within VehicleType
             @v_mileage,                                                                    -- passed through vehicle mileage
             @v_next_insp);                                                                 -- passed through vehicle next inspection date

END;
GO

91 %

Messages
Commands completed successfully.

Completion time: 2022-08-05T13:45:07.0186809-04:00
```

Here is execution of the transaction along with the initial entry of the appropriate tables:

CS669_TermProject...UJM00H\kakar (52))*

```sql
-- Add Vehicle
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'Mustang', 1266, 'Ford', 1901, 'FGN2098', '4Y1HL65848Z411492', 'Standard Vehicle', 3, '07-MAR-2022';
COMMIT TRANSACTION AddVehicle;

--Multiple inserts to fill out Vehicle table
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'Fusion', 1272, 'Ford', 2022, 'HGG2739', '581HL65848GH11481', 'Standard Vehicle', 673, '12-MAR-2022';
COMMIT TRANSACTION AddVehicle;
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'F350', 2098, 'Ford', 2002, 'FGN2059', '6Z1HL65848Z411492', 'CDL Vehicle', 64556, '10-MAY-2022';
COMMIT TRANSACTION AddVehicle;
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'Silverado', 2021, 'Chevrolet', 1997, 'GHU2848', '781HL65848Z414598', 'Standard Vehicle', 35234, '23-APR-2022';
COMMIT TRANSACTION AddVehicle;
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'Ram 1500', 1006, 'Dodge', 2005, 'JRT2093', '3XYHL65848Z436741', 'Standard Vehicle', 13542, '12-NOV-2022';
COMMIT TRANSACTION AddVehicle;
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'Mustang', 3002, 'Ford', 2018, 'PQN3483', '21AHL65848Z452287', 'Standard Vehicle', 52344, '01-DEC-2022';
COMMIT TRANSACTION AddVehicle;
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'Fusion', 1200, 'Ford', 2005, 'HGG2739', '3RGTY65848GH11481', 'Standard Vehicle', 45645, '21-MAR-2022';
COMMIT TRANSACTION AddVehicle;
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'F550', 2679, 'Ford', 2021, 'FGN2059', '9YHGR25848Z411492', 'CDL Vehicle', 84556, '05-MAY-2022';
COMMIT TRANSACTION AddVehicle;
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'Silverado', 0026, 'Chevrolet', 2016, 'GHU2848', '6YGWQ65848Z414598', 'Standard Vehicle', 23344, '28-APR-2022';
COMMIT TRANSACTION AddVehicle;
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'Ram 2500', 1016, 'Dodge', 2007, 'JRT2093', '7UJGH65848Z436741', 'Standard Vehicle', 542545, '01-NOV-2022';
COMMIT TRANSACTION AddVehicle;
BEGIN TRANSACTION AddVehicle;
EXECUTE AddVehicle  'Mustang', 3015, 'Ford', 2012, 'PQN3483', '5IONTW5848Z452287', 'Standard Vehicle', 8523, '25-DEC-2022';
COMMIT TRANSACTION AddVehicle;
```

91 %

Messages

```
(1 row affected)

Completion time: 2022-08-05T13:49:26.9433794-04:00
```

Messages

```
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
(1 row affected)
```

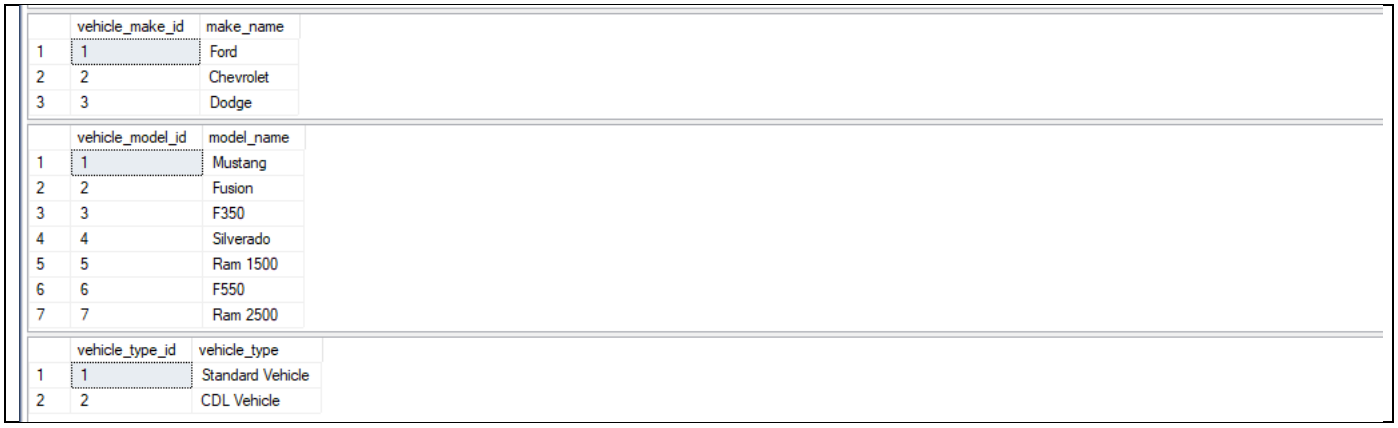
CS669_TermProject...UJM00H\kakar (52))*

```sql
SELECT * FROM Vehicle
SELECT * FROM VehicleMake
SELECT * FROM VehicleModel
SELECT * FROM VehicleType
```

91 %

Results | Messages

	vehicle_id	vehicle_model_id	vehicle_shop_code	vehicle_make_id	vehicle_year	vehicle_license	vehicle_vin	vehicle_type_id	veh_last_mileage	veh_schd_insp_date
1	1	1	1266	1	1901	FGN2098	4Y1HL65848Z411492	1	3	2022-03-07
2	2	2	1272	1	2022	HGG2739	581HL65848GH11481	1	673	2022-03-12
3	3	3	2098	1	2002	FGN2059	6Z1HL65848Z411492	2	64556	2022-05-10
4	4	4	2021	2	1997	GHU2848	781HL65848Z414598	1	35234	2022-04-23
5	5	5	1006	3	2005	JRT2093	3XYHL65848Z436741	1	13542	2022-11-12
6	6	1	3002	1	2018	PQN3483	21AHL65848Z452287	1	52344	2022-12-01
7	7	2	1200	1	2005	HGG2739	3RGTY65848GH11481	1	45645	2022-03-21
8	8	6	2679	1	2021	FGN2059	9YHGR25848Z411492	2	84556	2022-05-05
9	9	4	26	2	2016	GHU2848	6YGWQ65848Z414598	1	23344	2022-04-28
10	10	7	1016	3	2007	JRT2093	7UJGH65848Z436741	1	542545	2022-11-01
11	11	1	3015	1	2012	PQN3483	5IONTW5848Z452287	1	8523	2022-12-25



	vehicle_make_id	make_name
1	1	Ford
2	2	Chevrolet
3	3	Dodge

	vehicle_model_id	model_name
1	1	Mustang
2	2	Fusion
3	3	F350
4	4	Silverado
5	5	Ram 1500
6	6	F550
7	7	Ram 2500

	vehicle_type_id	vehicle_type
1	1	Standard Vehicle
2	2	CDL Vehicle

Use Case: Add Vendor (Vendor is a subtype of Maintainer)

Upon setup of the original application, a user will need to register an in-house mechanic or an outside vendor into the system for them to be selectable in the maintenance activity entry screen. For this example, the user wants to set up a new vendor. A refinement of the *Add Vendor* use case is given below:

1) The mechanic clicks the 'Add Vendor' button on the application interface.
2) The application provides a fill form to accept information regarding the vendor.

Field	**What it Stores**	**Why it's Needed**
vendor_name	The vendor's name	This helps to track which vendor performed what work for workload analysis.
street number	Vendor address variables	This helps identify where the vendor is located. Proximity can be a factor for vendor repair selection.
street name		
town/city		
state		
zip code		
phone_number	Phone number variables	This stores telephone contact information for the vendor.
agent_name	Vendor representative, if applicable	This is useful in case the vendor has an area sales representative that receives business from your area.
insurance policy	Insurance policy number if applicable	A place to store proof of insurance carried by vendor for the work that they do.
email	Email contact address	Similar to phone number, it is another way to get in contact with the vendor.

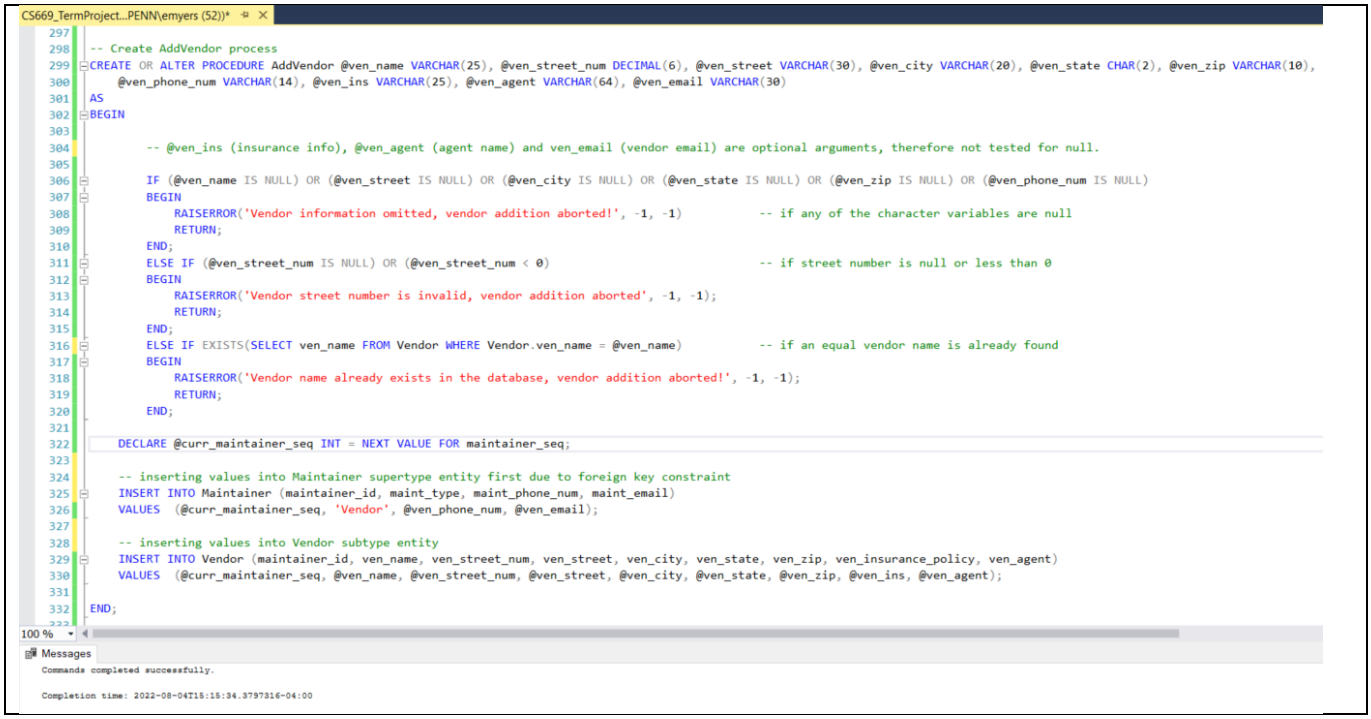
3) The mechanic enters the data about the vendor.
4) The mechanic presses 'Save' to save the entry.

This procedure takes the vendor name, street number, street name, city, state, zip-code, phone number, insurance policy number (if given), agent name (if given), and email (if given). The last three parameters are optional as determined by the Vendor subtype and Maintainer supertype entities.

Preliminary checks for null values in the mandatory fields are applied. A check for a less than zero street number and already-existing vendor name is also applied. The attributes associated with the Maintainer

supertype are inserted first to maintain the foreign key constraint, then exclusive attributes to the Vendor subtype are inserted.

Here is my stored procedure definition using SQL Server:



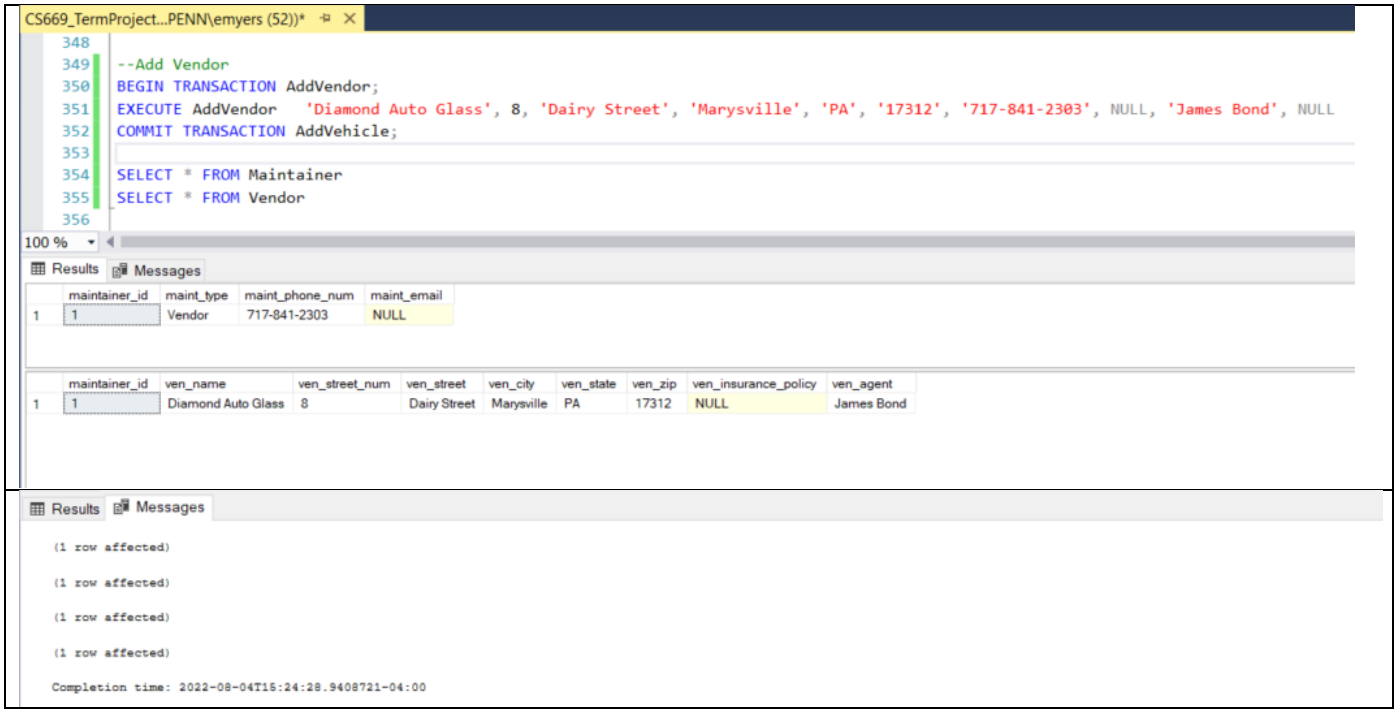
```
CS669_TermProject...PENN\emyers (52))*

297
298  -- Create AddVendor process
299  CREATE OR ALTER PROCEDURE AddVendor @ven_name VARCHAR(25), @ven_street_num DECIMAL(6), @ven_street VARCHAR(30), @ven_city VARCHAR(20), @ven_state CHAR(2), @ven_zip VARCHAR(10),
300      @ven_phone_num VARCHAR(14), @ven_ins VARCHAR(25), @ven_agent VARCHAR(64), @ven_email VARCHAR(30)
301  AS
302  BEGIN
303
304          -- @ven_ins (insurance info), @ven_agent (agent name) and ven_email (vendor email) are optional arguments, therefore not tested for null.
305
306          IF (@ven_name IS NULL) OR (@ven_street IS NULL) OR (@ven_city IS NULL) OR (@ven_state IS NULL) OR (@ven_zip IS NULL) OR (@ven_phone_num IS NULL)
307          BEGIN
308              RAISERROR('Vendor information omitted, vendor addition aborted!', -1, -1)          -- if any of the character variables are null
309              RETURN;
310          END;
311          ELSE IF (@ven_street_num IS NULL) OR (@ven_street_num < 0)                          -- if street number is null or less than 0
312          BEGIN
313              RAISERROR('Vendor street number is invalid, vendor addition aborted', -1, -1);
314              RETURN;
315          END;
316          ELSE IF EXISTS(SELECT ven_name FROM Vendor WHERE Vendor.ven_name = @ven_name)       -- if an equal vendor name is already found
317          BEGIN
318              RAISERROR('Vendor name already exists in the database, vendor addition aborted!', -1, -1);
319              RETURN;
320          END;
321
322      DECLARE @curr_maintainer_seq INT = NEXT VALUE FOR maintainer_seq;
323
324      -- inserting values into Maintainer supertype entity first due to foreign key constraint
325      INSERT INTO Maintainer (maintainer_id, maint_type, maint_phone_num, maint_email)
326      VALUES  (@curr_maintainer_seq, 'Vendor', @ven_phone_num, @ven_email);
327
328      -- inserting values into Vendor subtype entity
329      INSERT INTO Vendor (maintainer_id, ven_name, ven_street_num, ven_street, ven_city, ven_state, ven_zip, ven_insurance_policy, ven_agent)
330      VALUES  (@curr_maintainer_seq, @ven_name, @ven_street_num, @ven_street, @ven_city, @ven_state, @ven_zip, @ven_ins, @ven_agent);
331
332  END;

100 %
Messages
Commands completed successfully.

Completion time: 2022-08-04T15:15:34.3797316-04:00
```

Here is execution of the transaction along with the initial entry of the appropriate tables:



```
CS669_TermProject...PENN\emyers (52))*

348
349  --Add Vendor
350  BEGIN TRANSACTION AddVendor;
351  EXECUTE AddVendor   'Diamond Auto Glass', 8, 'Dairy Street', 'Marysville', 'PA', '17312', '717-841-2303', NULL, 'James Bond', NULL
352  COMMIT TRANSACTION AddVehicle;
353
354  SELECT * FROM Maintainer
355  SELECT * FROM Vendor
356

100 %
Results  Messages

   maintainer_id  maint_type  maint_phone_num  maint_email
1  1              Vendor      717-841-2303     NULL

   maintainer_id  ven_name          ven_street_num  ven_street   ven_city    ven_state  ven_zip  ven_insurance_policy  ven_agent
1  1              Diamond Auto Glass  8              Dairy Street  Marysville  PA         17312    NULL                  James Bond

Results  Messages

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

Completion time: 2022-08-04T15:24:28.9408721-04:00
```

Question Identification and Explanations

Scenario: Vehicle Trade-Ins

Query 1: How many vehicles qualify for trade in through the municipal lease program? Select all vehicles with 5,000 miles or greater and are more than 3 years old. Show how many distinct types of vehicles apply, giving the make, model and type in the result. Order the results descending by mileage and give a count of how many vehicles are within each output row and percentage of the overall fleet.

An administrator of the vehicle fleet wants information regarding how many vehicles are eligible to be traded in for new ones per a lease agreement. A municipality can enter a lease agreement with a vehicle supplier to prevent the average age of its vehicles from getting too old. As a vehicle gets older, it often takes more time, parts, fluids, and money to maintain. To prevent the 'nickel and dime' effect, it is advantageous to have a replacement plan to save on maintenance costs. Moreover, knowing the make and model of the aging vehicles within the fleet will help the administrator identify purchasing trends. Knowing the percentage of vehicles that are tradeable out of the whole allows the administrator to accurately plan how many replacement leases they should schedule in the next few budget years. Type of vehicle is important since 'CDL Vehicles' are usually more expensive to replace given their size, capacity and special registration.

Scenario: Ordering Parts from a Vendor

Query 2: Generate a list of parts that have a quantity 10 'units' or under. List them in a results table alongside the vendor which sells the part, their phone number, email if known and agent if known. Due to intra-state shipping surcharges, administration doesn't want anyone purchasing parts from out-of-state (PA in my case). Disqualify any parts from the list which have vendors outside of PA.

The in-house mechanic wants to stock the parts room for the month and would like to replenish materials that are running low in the shop. This is a standard supply ordering activity that can happen per week or per month in a mechanic's shop. Its very useful to know not only what you are running low on, but the right contacts to order the supplies from. I have personally seen a purchasing freeze similar to this scenario, and everyone can relate to delivery surcharges related to fuel cost increases.

Scenario: Monthly Inspection List

Query 3: Generate a list of how many vehicle inspections are due in which months. There should be 12 results, one for each month with the number of inspections given in each month, even if there are none. The list should be chronologically correct in order of month. CDL Vehicle inspections run on a 6-month interval, where Standard Vehicle inspections run on a 12-month interval.

The in-house mechanic runs a report at the beginning of the year which forecasts how many vehicle inspections are due and on what month they land. This is a workload query to judge proper time management in forecasting what inspections land where. As inspections can be shifted within a 3-month window prior to the deadline, the mechanic can use this report to balance out the workload on the books if needed.

Query Executions and Explanations

Here is a screenshot of Query #1:

```
/*  Query #1
How many vehicles qualify for trade in through the municipal lease program?  Select all vehicles with 5,000 miles or greater and are
more than 3 years old.  Show how many distinct types of vehicles apply, giving the make, model and type in the result.  Order the
results descending by mileage and give a count of how many vehicles are within each output row and percentage of the overall fleet.
*/

SELECT vmk.make_name AS 'Vehicle Name', vmd.model_name AS 'Vehicle Model', vt.vehicle_type AS 'Vehicle Type',
       format(AVG(v.veh_last_mileage), 'N0') AS 'Average Mileage', COUNT(v.vehicle_year) AS 'Number',
       format(CAST(COUNT(v.vehicle_id) AS FLOAT)  / (SELECT COUNT(vehicle_id) FROM Vehicle), 'P') AS '% of Fleet'
FROM   Vehicle v
    JOIN    VehicleType vt  ON  vt.vehicle_type_id = v.vehicle_type_id
    JOIN    VehicleMake vmk ON  vmk.vehicle_make_id = v.vehicle_make_id
    JOIN    VehicleModel vmd ON  vmd.vehicle_model_id = v.vehicle_model_id
WHERE (v.veh_last_mileage >= 5000) AND (v.vehicle_year < YEAR(GETDATE())-3)
GROUP BY vmk.make_name, vmd.model_name, vt.vehicle_type
ORDER BY AVG(v.veh_last_mileage) DESC
```

	Vehicle Name	Vehicle Model	Vehicle Type	Average Mileage	Number	% of Fleet
1	Dodge	Ram 2500	Standard Vehicle	542,545	1	9.09%
2	Ford	F350	CDL Vehicle	64,556	1	9.09%
3	Ford	Fusion	Standard Vehicle	45,645	1	9.09%
4	Ford	Mustang	Standard Vehicle	30,434	2	18.18%
5	Chevrolet	Silverado	Standard Vehicle	29,289	2	18.18%
6	Dodge	Ram 1500	Standard Vehicle	13,542	1	9.09%

```
(6 rows affected)

Completion time: 2022-08-05T14:27:05.2905011-04:00
```

To achieve the results, I joined the **Vehicle** table to the **VehicleMake**, **VehicleModel** and **VehicleType** tables by foreign key ties. I then chose aggregate functions to perform an average mileage calculation, count, and fleet percentage on the returned rows. I excluded any vehicle entries with less than 5,000 miles and less than 3 years old. I grouped the results by common vehicle make, model and type, ordered by average mileage descending.

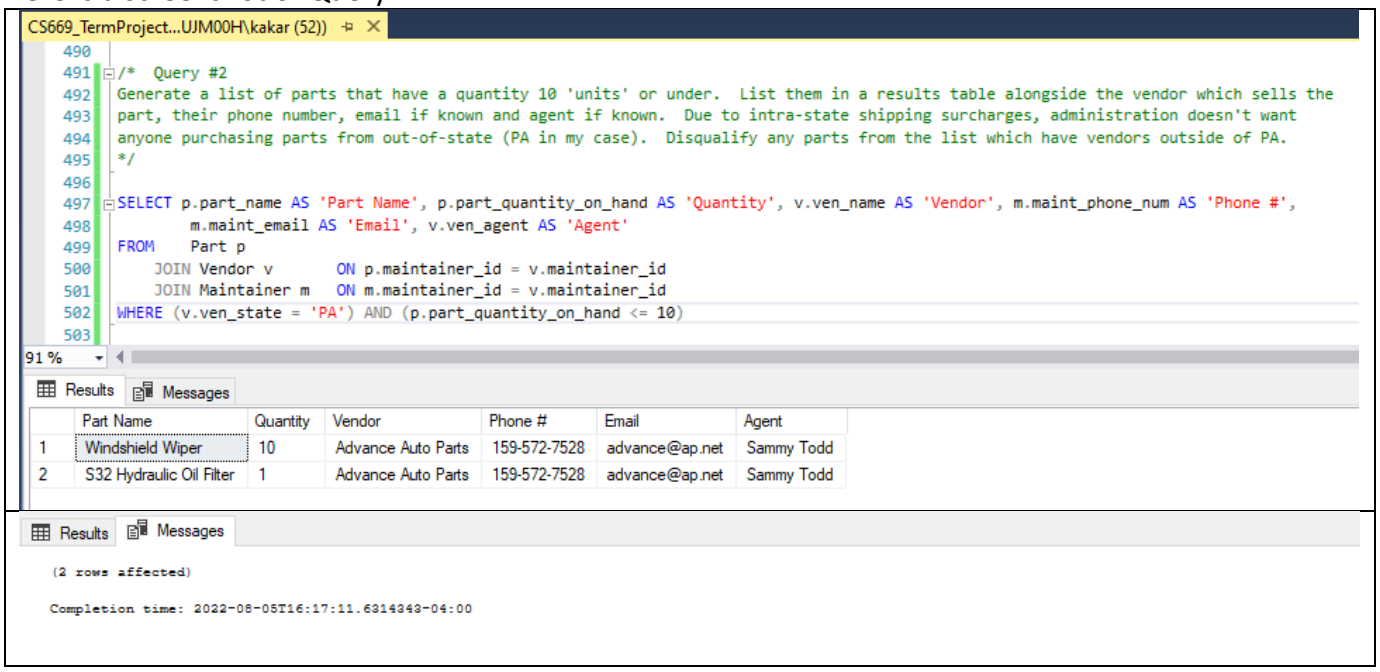
I created a proofing query to demonstrate that Query #1 worked as intended. Below is a simple query and table generated to output the year, make model, type and mileage of all vehicles inserted into the Vehicle table:

```
-- Proofing query to Query #1, an output of all vehicles for manual calculation
SELECT v.vehicle_year AS 'Vehicle Year', vmk.make_name AS 'Vehicle Name', vmd.model_name AS 'Vehicle Model',
       vt.vehicle_type AS 'Vehicle Type', format(v.veh_last_mileage, 'N0') AS ' Mileage'
FROM   Vehicle v
    JOIN    VehicleType vt  ON  vt.vehicle_type_id = v.vehicle_type_id
    JOIN    VehicleMake vmk ON  vmk.vehicle_make_id = v.vehicle_make_id
    JOIN    VehicleModel vmd ON  vmd.vehicle_model_id = v.vehicle_model_id
```

	Vehicle Year	Vehicle Name	Vehicle Model	Vehicle Type	Mileage
1	1901	Ford	Mustang	Standard Vehicle	3
2	2022	Ford	Fusion	Standard Vehicle	673
3	2002	Ford	F350	CDL Vehicle	64,556
4	1997	Chevrolet	Silverado	Standard Vehicle	35,234
5	2005	Dodge	Ram 1500	Standard Vehicle	13,542
6	2018	Ford	Mustang	Standard Vehicle	52,344
7	2005	Ford	Fusion	Standard Vehicle	45,645
8	2021	Ford	F550	CDL Vehicle	84,556
9	2016	Chevrolet	Silverado	Standard Vehicle	23,344
10	2007	Dodge	Ram 2500	Standard Vehicle	542,545
11	2012	Ford	Mustang	Standard Vehicle	8,523

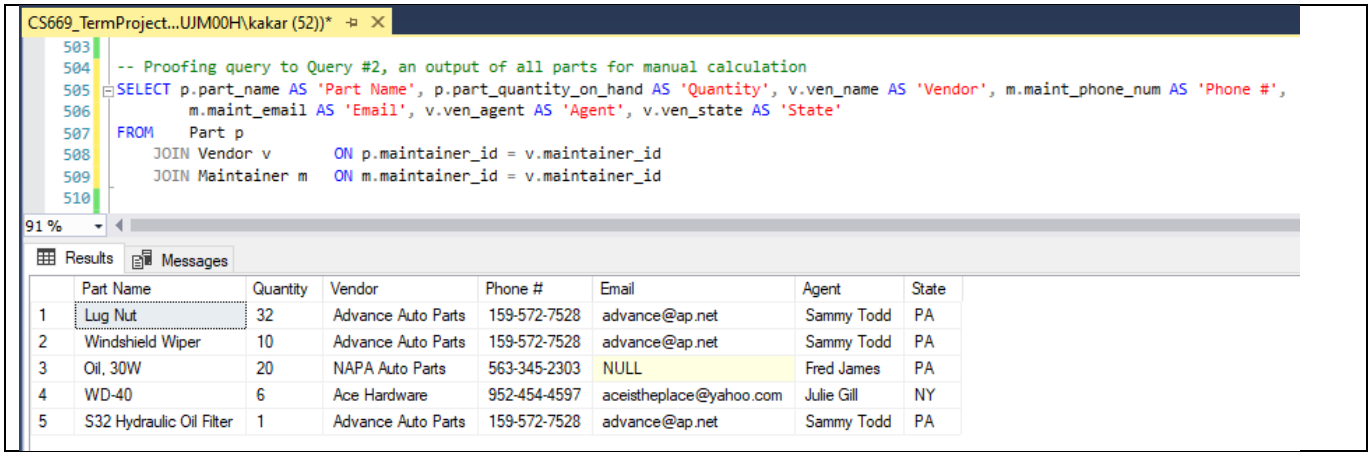
As a sample, there are two vehicles of the type ‘Chevrolet Silverado, Standard Vehicle’. Both have mileage greater than 5,000 (35,234 and 23,344 respectively) and are both over 3 years old (1997 and 2016 respectively). This would create a single row when grouped by make and model. The count would be 2 as there are two vehicles of that style, and the average mileage would be (35,234 + 23,344) / 2, or 29,289. There are a total of 11 vehicles in the fleet listed, so the percentage of qualified Chevy Silverados would be 2 / 11 or 18.18%. All values are correct.

Here is a screenshot of Query #2



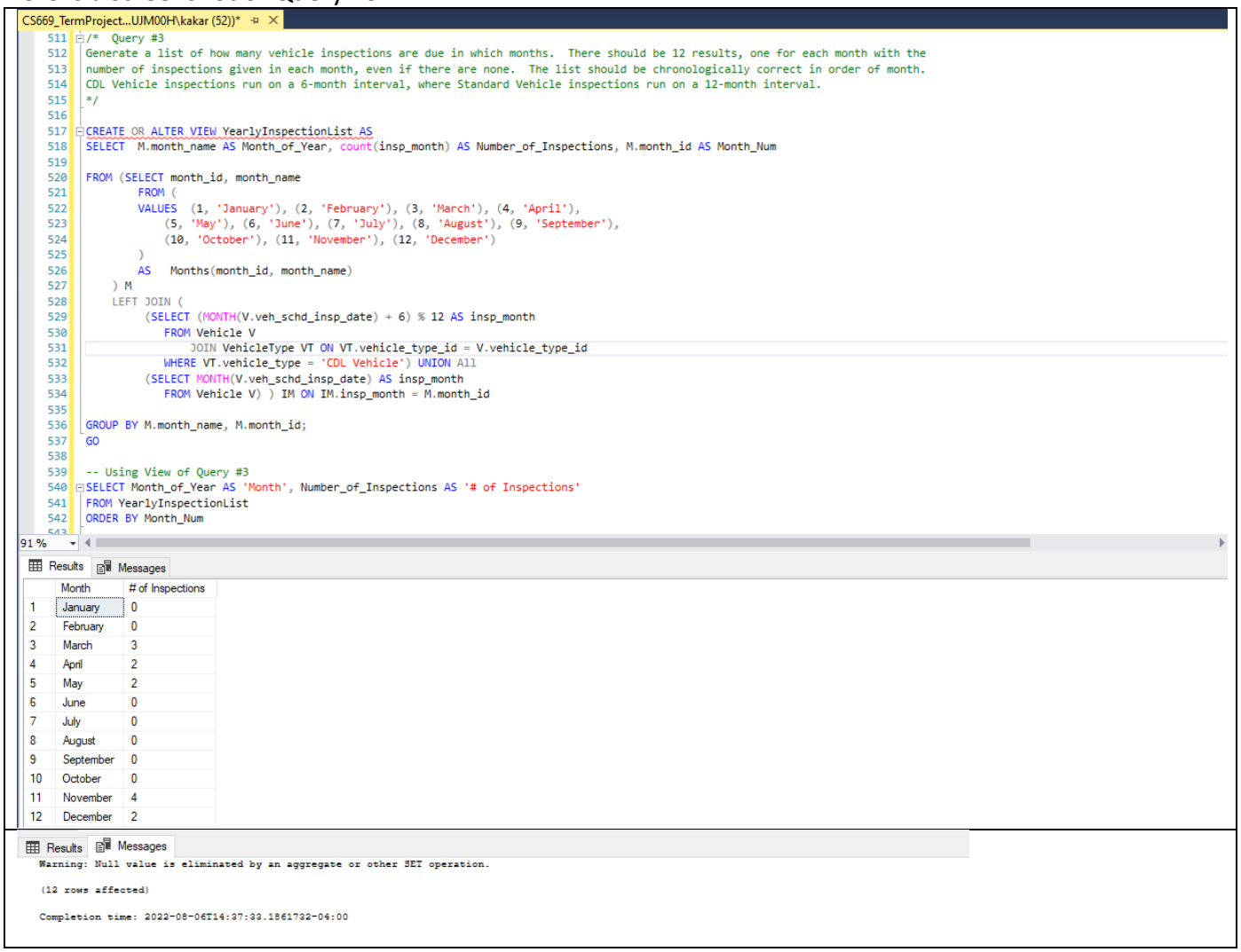
To achieve the results, I joined a populated Parts table to Vendors which connected the parts to the vendors which sold them. Next, I joined the **Vendor table (which is a subtype)** to the **Maintainer table (the supertype)** to connect related attributes between the two. From numerous tables I specified attributes that gave me part name, quantity, supplying vendor, phone number, email if known and agent if known. Entries in which part quantities were greater than 10 or the suppliers were outside of PA were parsed out.

To validate the above result, I’ve created a proofing query which outputs all parts in the database, matched up with their supplying vendor and pertinent information:



From the above output, the 'WD-40' part entry is invalid because its supplying vendor ships from NY. 'Lug Nut' and 'Oil, 30W' are out because their quantities in stock are above 10. The last two remaining parts qualify for restock, which are 'Windshield Wiper' and 'S32 Hydraulic Oil Filter'. These are the values returned from my query; thus it is correct.

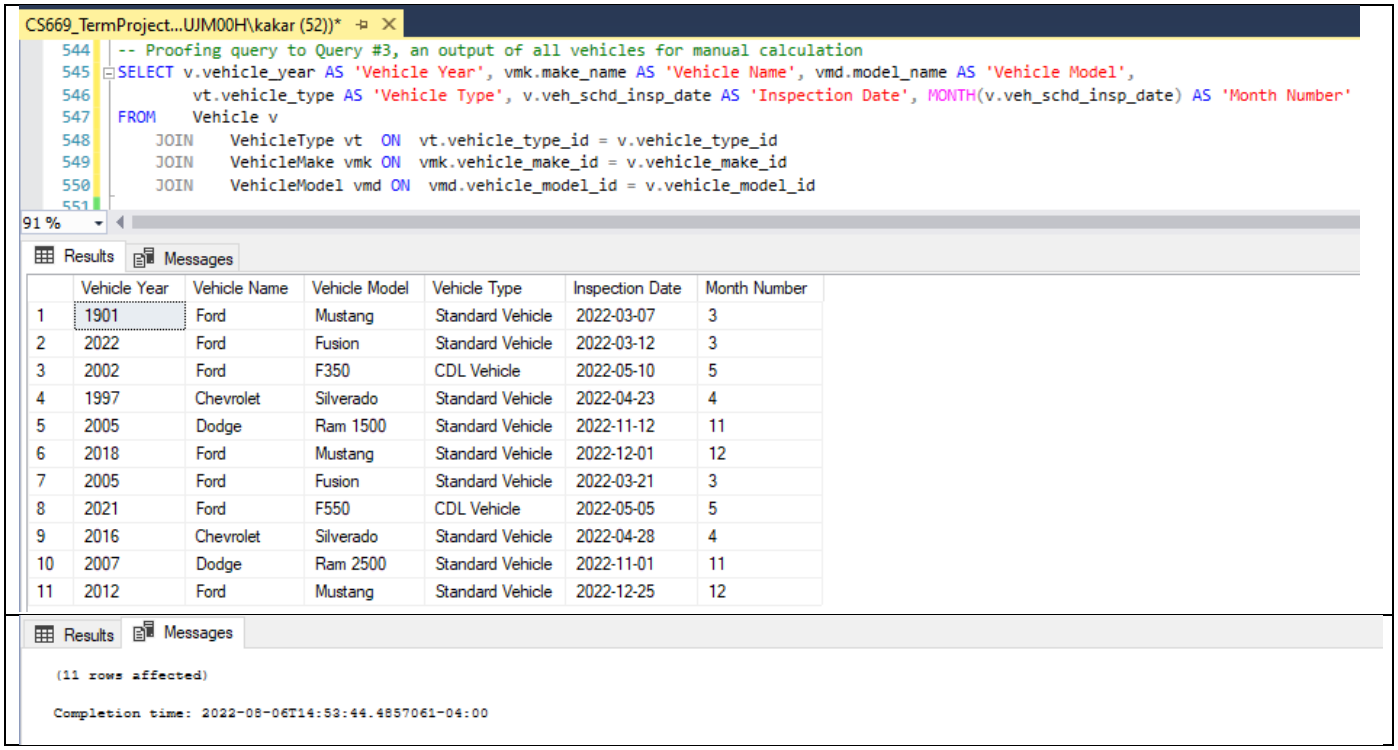
Here is a screenshot of Query #3:



To achieve the results, within my view I decided to use an aggregate **COUNT()** function to count the number of inspections that fell within each month upon grouping. I created an inline SELECT pseudo lookup table to associate month names with month id numbers. I performed a **LEFT JOIN** with a **UNION ALL** of two tables derived from the Vehicle table. The first half of a union was a **second JOIN** between the inspection date attribute of the Vehicle table where values returned were applicable to vehicles of the 'CDL Vehicle' type via a **WHERE** clause. I modified the values here to return the remainder of the (old month + 6) / 12. This takes care of the fact that CDL vehicles have two inspections a year, 6 months apart. The second half of the union was to the unaltered month values of the original vehicle list. The purpose of this union was to include one additional inspection per CDL Vehicle to acknowledge the fact that they get two inspections per year. Multiple **SUBQUERIES** were used throughout.

Outside the view, a simple query calls for the month and number of inspections, **ORDERING** by ascending month value.

To validate the above result, I've created a proofing query which outputs all vehicles in the database, matched up with vehicle type, inspection date and other pertinent information (such as month number for convenience):



Across all vehicle entries, from the counting of the month numbers we can see that there are 3 in March, 2 in April, 2 in May, 2 in November and 2 in December. Going back and counting the 'CDL Vehicle' entries, both have their original inspections in May. Adding 6 months onto these, we know that these two CDL vehicles need inspected again in November. Adding two more inspections to November and assigning 0 to the rest, we have:

Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
0	0	3	2	2	0	0	0	0	0	4	2

This matches the output; therefore the view query is correct.

Index Identification and Creations

From Primary Keys (already indexed)

Maintainer.maintainer_id	Inspection_Test_List.inspection_test_list_id
Vendor.maintainer_id	Result.result_id
Mechanic.maintainer_id	TestResult.test_id
Appointment.appointment_id	TestResult.result_id
AutomaticAppointment.appointment_id	PartEntry.part_entry_id
ManualAppointment.appointment_id	VehiclePartList.vehicle_id
Part.part_id	VehiclePartList.part_id
MaintenanceActivity.maintenance_activity_id	Vehicle.vehicle_id
VendorRepair.maintenance_activity_id	VehicleModel.vehicle_model_id
Inspection.maintenance_activity.id	VehicleMake.vehicle_make_id
Test.test_id	VehicleType.vehicle_type_id

From Foreign Keys (to be indexed)

Entity.Attribute	Unique?	Reason for Indexing
Appointment.vehicle_id	Not unique	Foreign key, not unique because a vehicle can have many appointments over its lifetime.
Appointment.maintainer_id	Not unique	Foreign key, not unique because a maintainer (mechanic) can be assigned many appointments.
Part.maintainer_id	Not unique	Foreign key, not unique because a single maintainer (vendor) can supply many types of parts.
MaintenanceActivity.maintainer_id	Not unique	Foreign key, not unique because a maintainer can perform many maintenance activities.
MaintenanceActivity.vehicle_id	Not unique	Foreign key, not unique because a vehicle can have many maintenance activities performed on it.
Inspection_Test_List.test_id	Not unique	Foreign key, not unique because a test can be a part of many inspections test lists.
Inspection_Test_List.maintenance_activity_id	Not unique	Foreign key, not unique because a maintenance activity (inspection) can have many different inspections test lists.
PartEntry.maintenance_activity_id	Not unique	Foreign key, not unique because a maintenance activity can include many part entries.

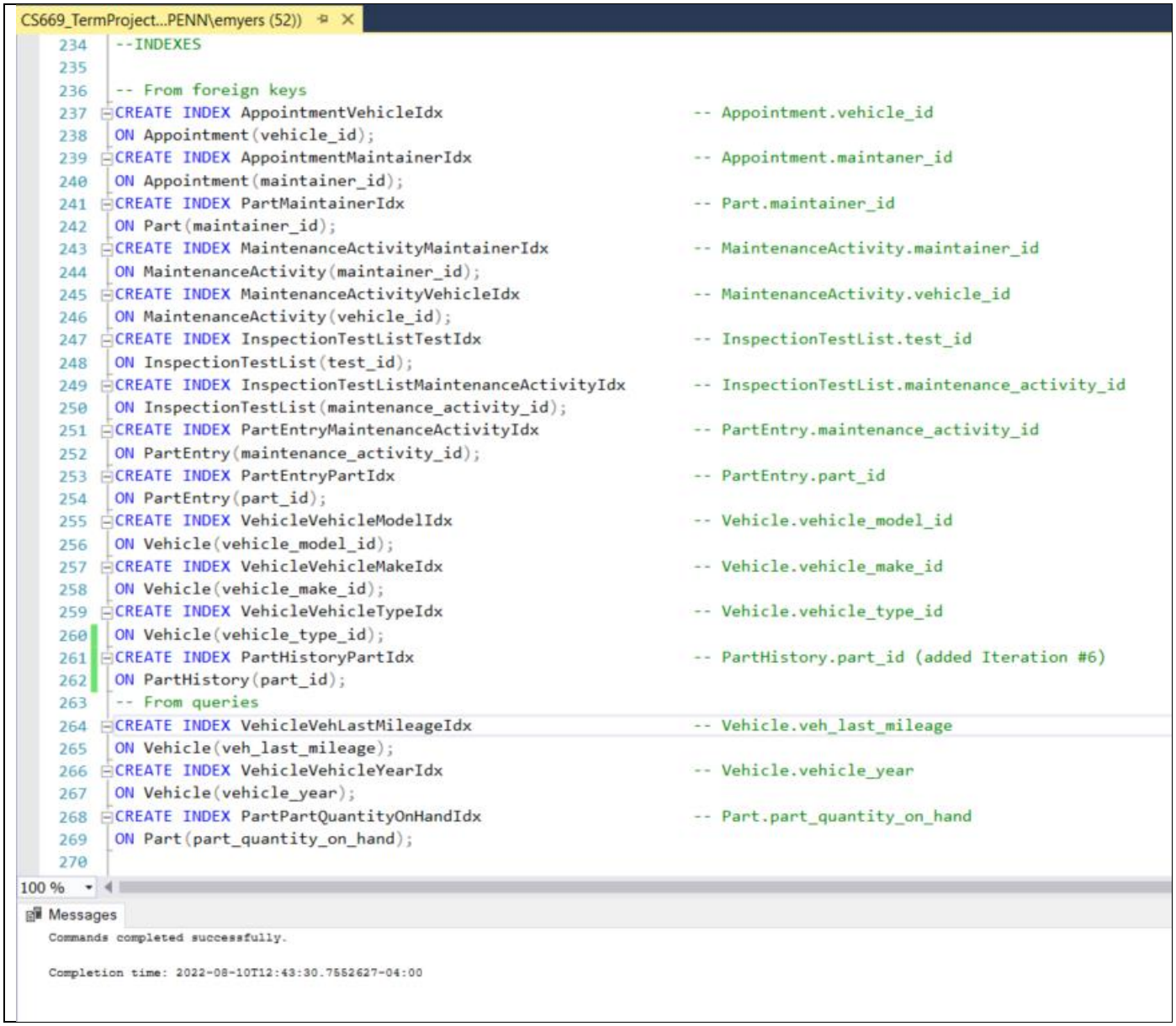
PartEntry.part_id	Not unique	Foreign key, not unique because a part can be within many parts entries.
Vehicle.vehicle_model_id	Not unique	Foreign key, not unique because a vehicle model can be repeated across many vehicles.
Vehicle.vehicle_make_id	Not unique	Foreign key, not unique because a vehicle make can be repeated across many vehicles.
Vehicle.vehicle_type_id	Not unique	Foreign key, not unique because a vehicle type can be repeated across many vehicles.
PartHistory.part_id	Not unique	Foreign key, not unique because a part can have multiple part history records.

From Queries (to be indexed)

In query #1, vehicle mileage was one of the important attributes that the mechanics used to determine if a vehicle should qualify for trade in. It is located in the WHERE clause after the joins. The vehicle table could possibly get large as the vehicle fleet grows, and the mileages will possess many different values. Both of these factors qualify **Vehicle.veh_last_mileage** to be indexed. Although mileages often vary, the possibility exists for two vehicle's mileages to become equal at some point in time. The index must be **not unique.**

In the same query, the second determining factor for a vehicle trade-in was a vehicle's year. It is also contained in the Where clause. For the same reasons above, **Vehicle.vehicle_year** is a good candidate for indexing. It must be **not unique** as multiple vehicles can have the same vehicle year.

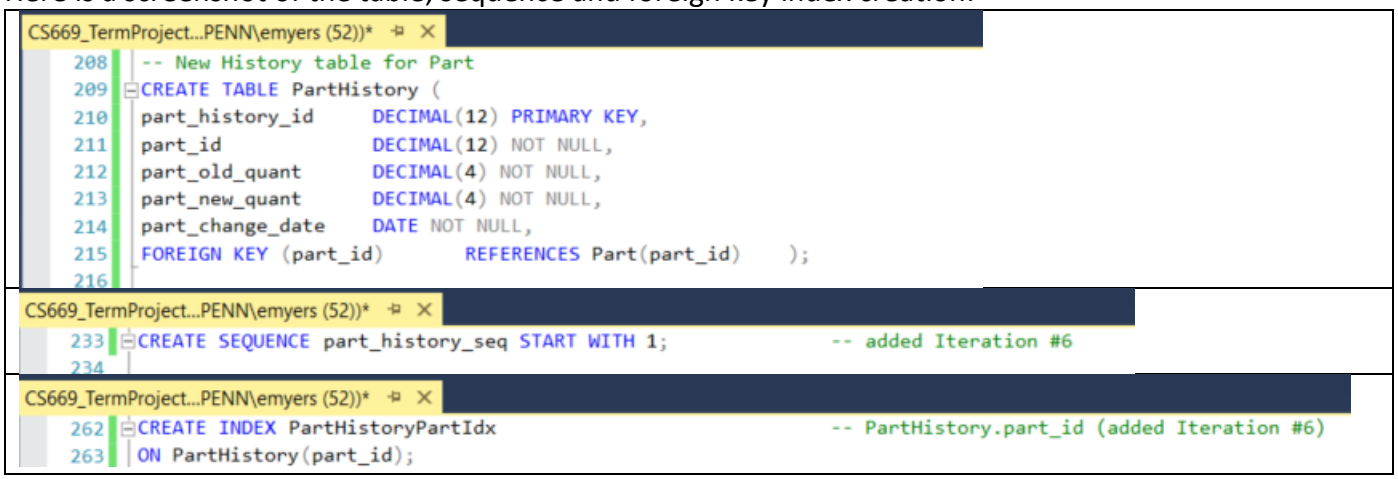
In query #2, part quantity on hand was an important statistic that the mechanics use to determine when to order more stock. It is located in the WHERE clause after the joins. The parts table can get quite large, and the stock quantities can vary but not always be unique. Therefore, **Part.part_quantity_on_hand** is a good candidate for a **not unique** index.


CS669_TermProject...PENN\emyers (52))
--INDEXES
-- From foreign keys
CREATE INDEX AppointmentVehicleIdx -- Appointment.vehicle_id
ON Appointment(vehicle_id);
CREATE INDEX AppointmentMaintainerIdx -- Appointment.maintaner_id
ON Appointment(maintainer_id);
CREATE INDEX PartMaintainerIdx -- Part.maintainer_id
ON Part(maintainer_id);
CREATE INDEX MaintenanceActivityMaintainerIdx -- MaintenanceActivity.maintainer_id
ON MaintenanceActivity(maintainer_id);
CREATE INDEX MaintenanceActivityVehicleIdx -- MaintenanceActivity.vehicle_id
ON MaintenanceActivity(vehicle_id);
CREATE INDEX InspectionTestListTestIdx -- InspectionTestList.test_id
ON InspectionTestList(test_id);
CREATE INDEX InspectionTestListMaintenanceActivityIdx -- InspectionTestList.maintenance_activity_id
ON InspectionTestList(maintenance_activity_id);
CREATE INDEX PartEntryMaintenanceActivityIdx -- PartEntry.maintenance_activity_id
ON PartEntry(maintenance_activity_id);
CREATE INDEX PartEntryPartIdx -- PartEntry.part_id
ON PartEntry(part_id);
CREATE INDEX VehicleVehicleModelIdx -- Vehicle.vehicle_model_id
ON Vehicle(vehicle_model_id);
CREATE INDEX VehicleVehicleMakeIdx -- Vehicle.vehicle_make_id
ON Vehicle(vehicle_make_id);
CREATE INDEX VehicleVehicleTypeIdx -- Vehicle.vehicle_type_id
ON Vehicle(vehicle_type_id);
CREATE INDEX PartHistoryPartIdx -- PartHistory.part_id (added Iteration #6)
ON PartHistory(part_id);
-- From queries
CREATE INDEX VehicleVehLastMileageIdx -- Vehicle.veh_last_mileage
ON Vehicle(veh_last_mileage);
CREATE INDEX VehicleVehicleYearIdx -- Vehicle.vehicle_year
ON Vehicle(vehicle_year);
CREATE INDEX PartPartQuantityOnHandIdx -- Part.part_quantity_on_hand
ON Part(part_quantity_on_hand);
100 %
Messages
Commands completed successfully.
Completion time: 2022-08-10T12:43:30.7552627-04:00

History Table Demonstration

I have chosen to keep a history table for the Parts entity, to track the change in part quantities over time. It is very important for a mechanic or shop user to know the nature of how parts and materials get used when working on different vehicles. Data on usage, specifically on how quickly parts are used allow the mechanic to better forecast ordering frequency before a part runs out of stock.

Here is a screenshot of the table, sequence and foreign key index creation:

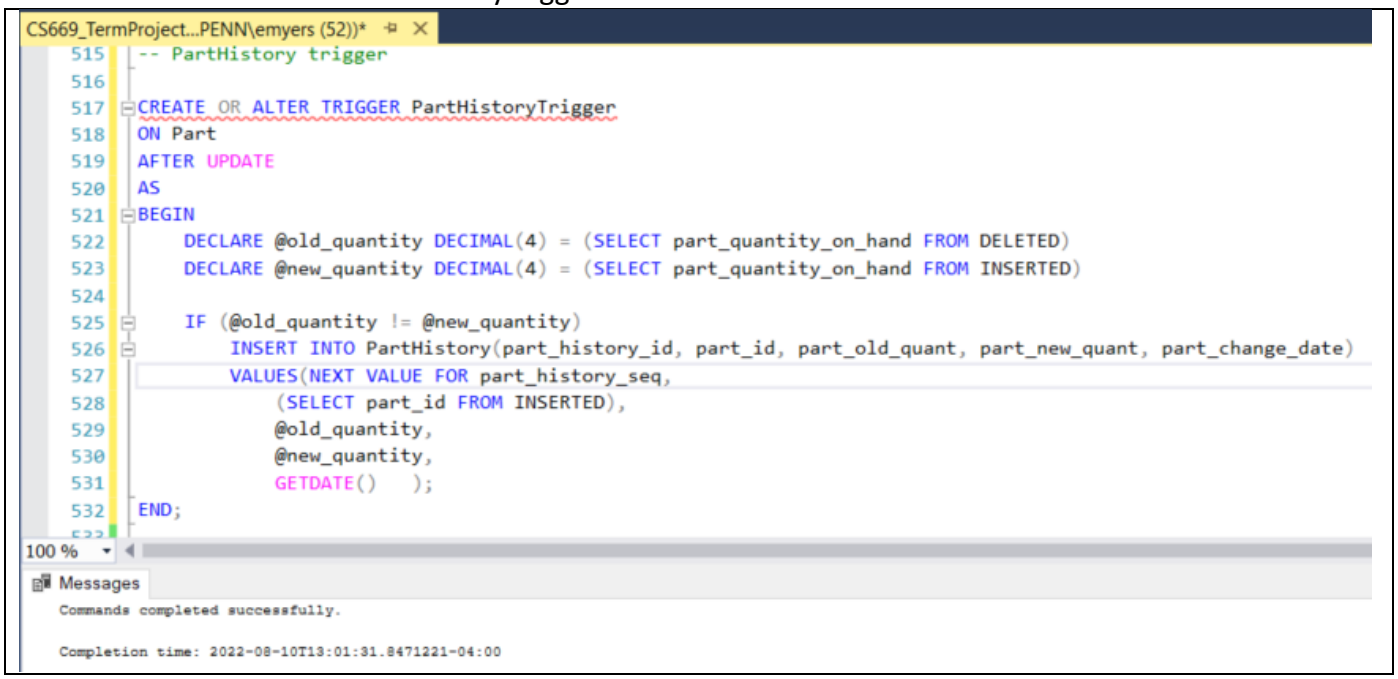


```
CS669_TermProject...PENN\emyers (52))*
208  -- New History table for Part
209  CREATE TABLE PartHistory (
210  part_history_id     DECIMAL(12) PRIMARY KEY,
211  part_id             DECIMAL(12) NOT NULL,
212  part_old_quant      DECIMAL(4) NOT NULL,
213  part_new_quant      DECIMAL(4) NOT NULL,
214  part_change_date    DATE NOT NULL,
215  FOREIGN KEY (part_id)       REFERENCES Part(part_id)    );
216

CS669_TermProject...PENN\emyers (52))*
233  CREATE SEQUENCE part_history_seq START WITH 1;              -- added Iteration #6
234

CS669_TermProject...PENN\emyers (52))*
262  CREATE INDEX PartHistoryPartIdx                             -- PartHistory.part_id (added Iteration #6)
263  ON PartHistory(part_id);
```

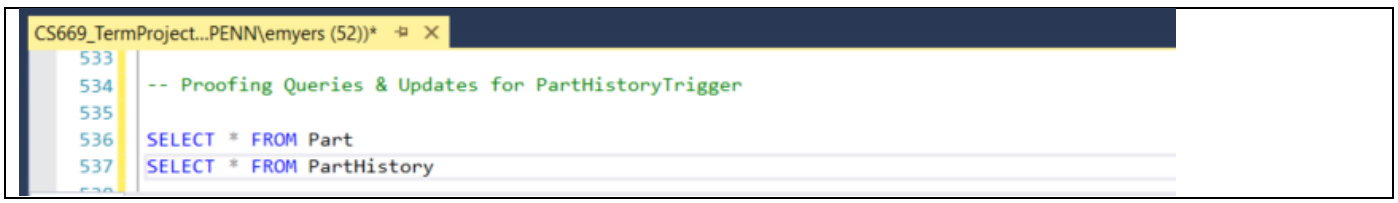
Here is a screenshot of the PartHistoryTrigger:



```
CS669_TermProject...PENN\emyers (52))*
515  -- PartHistory trigger
516
517  CREATE OR ALTER TRIGGER PartHistoryTrigger
518  ON Part
519  AFTER UPDATE
520  AS
521  BEGIN
522      DECLARE @old_quantity DECIMAL(4) = (SELECT part_quantity_on_hand FROM DELETED)
523      DECLARE @new_quantity DECIMAL(4) = (SELECT part_quantity_on_hand FROM INSERTED)
524
525      IF (@old_quantity != @new_quantity)
526          INSERT INTO PartHistory(part_history_id, part_id, part_old_quant, part_new_quant, part_change_date)
527          VALUES(NEXT VALUE FOR part_history_seq,
528              (SELECT part_id FROM INSERTED),
529              @old_quantity,
530              @new_quantity,
531              GETDATE()   );
532  END;
100 %
Messages
Commands completed successfully.

Completion time: 2022-08-10T13:01:31.8471221-04:00
```

To prove that the trigger works, here is a starting query of the Parts and PartHistory tables before any changes to quantities are made:



```
CS669_TermProject...PENN\emyers (52))*
533
534  -- Proofing Queries & Updates for PartHistoryTrigger
535
536  SELECT * FROM Part
537  SELECT * FROM PartHistory
```

Results | Messages

	part_id	part_name	maintainer_id	part_quantity_on_hand	part_quantity_units	part_description	part_serial_number
1	1	Lug Nut	5	32	nut	NULL	NULL
2	2	Windshield Wiper	5	10	wiper blade	NULL	WS321-5
3	3	Oil, 30W	4	20	quart	for fall/winter oil changes	CS-0W30
4	4	WD-40	6	6	can	NULL	NULL
5	5	S32 Hydraulic Oil Filter	5	1	filter	for Ford CDL Vehicles only	S3202

part_history_id	part_id	part_old_quant	part_new_quant	part_change_date

Results | Messages

```
(5 rows affected)

(0 rows affected)

Completion time: 2022-08-10T13:04:51.6044141-04:00
```

Now I will execute several updates on multiple parts to simulate quantity updates in the application:

CS669_TermProject...PENN\emyers (52))*

```
538
539 UPDATE Part
540 SET     part_quantity_on_hand = 30
541 WHERE   part_name = 'Lug Nut';
542 UPDATE Part
543 SET     part_quantity_on_hand = 25
544 WHERE   part_name = 'Lug Nut';
545 UPDATE Part
546 SET     part_quantity_on_hand = 3
547 WHERE   part_name = 'Oil, 30W';
548 UPDATE Part
549 SET     part_quantity_on_hand = 28
550 WHERE   part_name = 'Windshield Wiper';
551 UPDATE Part
552 SET     part_quantity_on_hand = 10
553 WHERE   part_name = 'S32 Hydraulic Oil Filter';
554
555 SELECT * FROM Part
556 SELECT * FROM PartHistory
```

100 %

Results | Messages

	part_id	part_name	maintainer_id	part_quantity_on_hand	part_quantity_units	part_description	part_serial_number
1	1	Lug Nut	5	25	nut	NULL	NULL
2	2	Windshield Wiper	5	28	wiper blade	NULL	WS321-5
3	3	Oil, 30W	4	3	quart	for fall/winter oil changes	CS-0W30
4	4	WD-40	6	6	can	NULL	NULL
5	5	S32 Hydraulic Oil Filter	5	10	filter	for Ford CDL Vehicles only	S3202

	part_history_id	part_id	part_old_quant	part_new_quant	part_change_date
1	1	1	32	30	2022-08-10
2	2	1	30	25	2022-08-10
3	3	3	20	3	2022-08-10
4	4	2	10	28	2022-08-10
5	5	5	1	10	2022-08-10

Results | Messages

```
(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(5 rows affected)

(5 rows affected)
```

```
Completion time: 2022-08-10T13:23:24.9200377-04:00
```

I executed two quantity changes on ‘Lug Nuts’, from the original 32 to 30 then to 25. Then I changed ‘Oil, 30W’ from 4 to 3, ‘Windshield Wiper’ from 10 to 28 and ‘S32 Hydraulic Oil Filter’ from 1 to 10. This simulates up and down changes for part usage and restock. The PartHistory table logged each transaction correctly when queried for its values.

Data Visualizations

Include and explain data visualizations and stories that tell effective stories about the information in a way that people quickly and accurately understand it.

Data Story #1: Evaluating the Vehicle Fleet Makeup using Age vs. Make

Scenario:

The administration in charge of the vehicle fleet for a municipality recognizes that their fleet is in rough shape. Reports from the mechanics come in weekly about another truck breaking down and requiring a lot of parts for service. The accounts payable secretary keeps complaining of the number of checks she has to write to parts vendors. The fleet manager recently pulled the second quarter budget for the highway vehicle repairs and found that expenditures were already at 70% of the anticipated yearly line cap, in *April!*

Fortunately, the manager has been working with a well-known car retailer in the area and recently entered into a fleet leasing program. The idea is to slowly replace vehicles from a mix of medium to advanced ages. That way, the administration can recoup some sellback money from younger vehicles that have a trade-in value while also phasing out the extremely old ones. Eventually, as the fleet gets 'younger', the trade-ins become more lucrative, and the lease program pays for itself. To this end, the manager wants to sort the vehicles into age 'buckets' to find out how many are considered newer (less than 3 years of age), average (between 3 and 8 years of age, inclusive) and older (over 8 years of age). He pulls up the maintenance program that the mechanic's use and wants to create a query that does the following:

Create a query that groups the vehicles into age 'buckets'. The categories will be 'newer' (less than 3 years old), 'average' (between 3 and 8 years old, inclusive), and 'older' (over 8 years old). The query will return the count of how many vehicles fit each bucket. The results should be fed into a graphical display for analysis. Note: management knows that the mechanics keep a vintage toy '1901 Ford Sweepstakes Racer' in the shop that they put in the program as a Mustang. Please omit this result…

The query uses a nested subquery which creates the buckets with the appropriate condition statements against the vehicle age, obtained by subtracting the current year from the vehicle year. The second column is the count aggregate column which counts the vehicles qualifying for each group. A second case statement parallels the first to create a third, 'row counter' column that is meant for ordering the results only. The results are grouped by the case statement conditions. In the exterior select, only the category and vehicle columns are chosen from the inner select table, with the results sorted by the inner select table's rowCounter value.

```
CS669_TermProject...UJM00H\kakar (52))
-- Data visualization query #1

/* Create a query that groups the vehicles into age 'buckets'.  The categories will be 'newer' (less than 3 years old), 'average'
(between 3 and 8 years old, inclusive), and 'older' (over 8 years old).  The query will return the count of how many vehicles fit
each bucket.  The results should be fed into a graphical display for analysis.   Note:  management knows that the mechanics keep
a vintage toy '1901 Ford Sweepstakes Racer' in the shop that they put in the program as a Mustang.  Please omit this result…
*/

SELECT V.Category, V.Vehicles FROM
    (SELECT
            CASE
                WHEN YEAR(GETDATE()) - v.vehicle_year < 3   THEN 'newer (0-2 years)'
                WHEN (YEAR(GETDATE()) - v.vehicle_year) >= 3 AND (YEAR(GETDATE()) - v.vehicle_year) <= 8 THEN 'average (3-8 years)'
                ELSE 'older (9+ years)'
            END AS Category, COUNT (v.vehicle_year) AS Vehicles,
            CASE
                WHEN YEAR(GETDATE()) - v.vehicle_year < 3   THEN 1
                WHEN (YEAR(GETDATE()) - v.vehicle_year) >= 3 AND (YEAR(GETDATE()) - v.vehicle_year) <= 8 THEN 2
                ELSE 3
            END AS rowCounter

    FROM    Vehicle v
    WHERE v.vehicle_year > 1990
    GROUP BY
            CASE
                WHEN YEAR(GETDATE()) - v.vehicle_year < 3   THEN 'newer (0-2 years)'
                WHEN (YEAR(GETDATE()) - v.vehicle_year) >= 3 AND (YEAR(GETDATE()) - v.vehicle_year) <= 8 THEN 'average (3-8 years)'
                ELSE 'older (9+ years)'
            END,
            CASE
                WHEN YEAR(GETDATE()) - v.vehicle_year < 3   THEN 1
                WHEN (YEAR(GETDATE()) - v.vehicle_year) >= 3 AND (YEAR(GETDATE()) - v.vehicle_year) <= 8 THEN 2
                ELSE 3
            END

) V
ORDER BY V.rowCounter ASC
```

90 %

Results | Messages

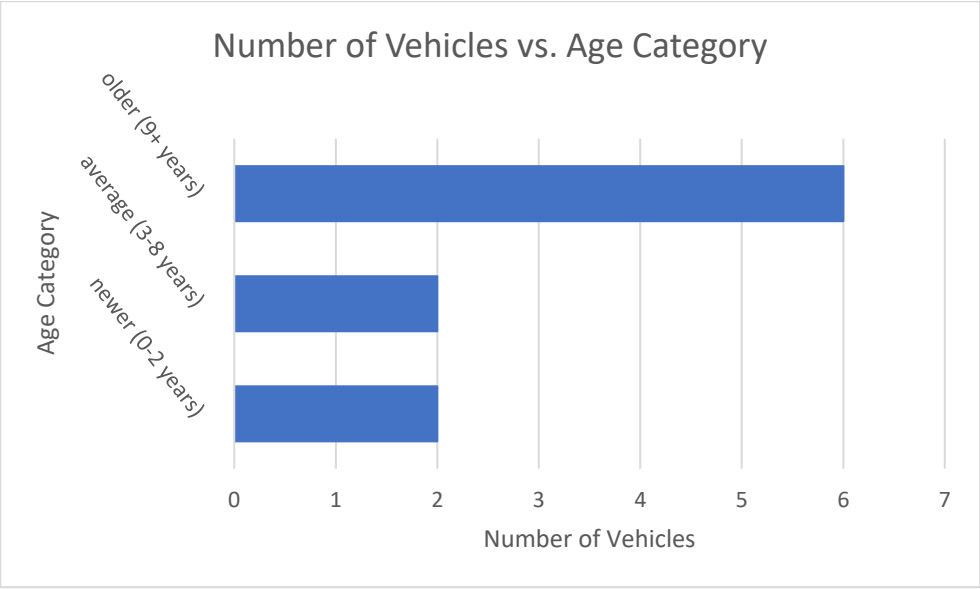
	Category	Vehicles
1	newer (0-2 years)	2
2	average (3-8 years)	2
3	older (9+ years)	6

Results | Messages

```
(3 rows affected)

Completion time: 2022-08-11T21:17:49.6493181-04:00
```

The output is simple and can be encapsulated in a bar graph:



Summary and Reflection

The goals of my original 'FleetManager' program have not changed. The mechanic user should be able to add, edit and delete vehicles within a fleet inventory. The user may also assign maintenance transactions to those vehicles, including data on parts, costs, and vendors. Several administrative use cases that involve data reporting are now omitted from the project since the scope should focus more on insertion and manipulation interactions with the database.

My favorite part of the activity this week was writing the queries. Thinking through the scenarios on how this database will be used is very interesting and gave me a good feeling that all the tables, relationships and logic were purposeful. My perception of query and stored procedure writing is that you can do a lot of logical back-end work for the programmer of the application in setting up shortcuts and pathways to the most-used information ahead of time. The application program still has to do plenty with the user interface and input/output formatting, but coming from making a version of this program that was fully application-end dependent I very much appreciate the structure and usefulness of a database.

I don't have any worries at this point which I hope is a good thing. I'm still enjoying the full process even though it is a lot!

Thanks for your time and I look forward to your critique.