Assignment 2 – Logical Modelling – Wheels4U

Student Name: **Joel Pillar-Rogers** Student ID: **2017545** Student FAN: **pill0032**

Introduction

To build my Wheels4U Logical Model, I closely follow the logical modelling steps outlined in Steps 2.1 to 2.4 of the COMP8711 Database Modelling and Knowledge Engineering course content. I follow the nine sub-steps from 2.1 in order. I apply the referential integrity constraints from Step 2.4 throughout so that I can present the logical model in database description language (DBDL). I then briefly discuss Step 2.2, normalisation. I build and populate the database in SQLite using database definition language (DDL), following the constraints of Step 2.4. Finally, as in Step 2.3, I present queries and results for the requested user transactions.

Part A: Deriving Relations for the Logical Model

The relations on this section are derived from the World Champion Entity Relational Diagram (ERD) of Wheels4U developed in Assignment 1 (see Appendix A). I generally follow the process from Step 2.1 of the lecture notes, although I apply referential integrity throughout so I can present my relations in DBDL.

Step 2.1.1: Strong entities

A strong entity type is one that is not dependent on some other entity for its existence. These are the initial strong entities from the ERD:

- Depot
- Client
- Vehicle
- VehicleType
- Invoice
- Insurance
- HiredVehicle
- DailyTariff
- Booking
- ServiceHistory
- ScheduledService

I decided VehicleType was independent of Vehicle because it could exist on its own in this database and has its own primary key and relationships.

Step 2.1.2: Weak entities

A weak entity type is dependent on another entity for its existence; it can't exist without it. It is identified through its role and its sharing of a primary key with a strong entity.

These are the weak entities:

- PersonalClient
- CompanyClient

- policyNumber
- rentalPrice

PersonalClient and CompanyClient share the same primary key as Client and I think are a composition specialisation – Client is composed of PersonalClients and CompanyClients and they have a coincidental lifetime.

I wasn't sure if relationships with attributes were entities, but I've included policyNumber and rentalPrice here for completeness. It seems clear that they wouldn't exist without the entities on either side.

Step 2.1.3: One-to-many (1:*) relationships

Now that I've identified the entities, I can start to build relations based on the multiplicities of their relationships. The one-to-many relationship is the most common in relational databases, including this one. The design of the relation in a one-to-many relationship is defined by its cardinality. The entity on the 'one' side of the relationship is the parent while the 'many' side is the child. The primary key of the parent is inserted into each child. This is because one parent can have many children, whereas each child has at most one parent and a relation/table can only have one element for each intersection of column and row. Below I work through each one-to-many relationship, starting with the "most parenty".

Client, Depot and Insurance only have one-to-many relationships and they are on the 'one' side of them all. VehicleType and DailyTariff have one-to-many relationships on the 'one' side and a many-to-many relationship with each other which will create a separate relation. This means these relations are always the parent and won't contain any foreign keys. They can be directly represented in DBDL with further changes unlikely.

Here is the DBDL for Client. It shows the attributes and primary key. It has no foreign keys as it is always the parent and it has no derived attributes. It has a composite attribute, *address*, but only its constituent parts are attributes in the final relation. It has a multivalued attribute, *phone* which I've left out of this table and will address later. It also has subclasses which will be addressed later.

Client (clientID, street, postcode)
Primary Key clientID

Here is the DBDL for Depot. Like Client, it has a composite attribute *address* with only its constituents listed. It also has a multivalued *phone* that will be dealt with later.

Depot (depotID, street, postcode, fax)
Primary Key depotID

Here is the DBDL for VehicleType. It has simple attributes, but it has a composite primary key made up of *make* and *model*. The many-to-many relationship with DailyTariff will be discussed later.

VehicleType (make, model, doors, body, trim)
Primary Key make, model

Here is the DBDL for DailyTariff. It has simple attributes and no foreign keys. The many-to-many relationship with VehicleType will be discussed later.

DailyTariff (tariffID, conditions)
Primary Key tariffID

Here is the DBDL for Insurance. It has simple attributes and no foreign keys.

Insurance (insuranceID, policyType, cost)
Primary Key insuranceID

HiredVehicle is the central relation of this database, with the most relationships. Client, Depot, Insurance, Vehicle and DailyTariff all have one-to-many relationships with HiredVehicle so we can post their primary keys into that relation. Because Depot has two relationships with HiredVehicle, it will post two primary keys into it: one for the pickup Depot and one for the return Depot.

Below is the 'in-progress' DBDL for HiredVehicle (it has other relationships that could affect it later). It has no composite attributes. It contains *policyNumber*, which is an attribute on the relationship from Insurance. Attributes on relationships always follow the primary key into the child. The ERD-specified primary key for HiredVehicle was "date", but that seems insufficient given that many vehicles may be hired on the same day. I could choose to make *regNum*, from Vehicle, a combined primary key with "date", which should make all HiredVehicle tuples unique (I assume one vehicle won't be hired twice on the same date). Instead, however, I'm going to give HiredVehicle a surrogate primary key, *hireID*, because HiredVehicle will be the parent in other relationships and a surrogate will be easier to use and identity unique bookings.

Each foreign key needs to specify what action to take if the parent modifies or deletes its primary key. In general, an updated foreign key will CASCADE, which means changes flow through from the attribute in the parent to the attribute in the child. I think this is appropriate for the relationships here since the related primary keys are all surrogate keys and this won't affect hire conditions at all.

Deletion need further thought, however. Two of the most common options are NO ACTION, which stops deletions of the primary key in the parent until no child references it, and SET NULL, which sets the foreign key in the child to null. This second option is usually more appropriate when the relationship from the child to the parent is optional.

For HiredVehicle, I've decided to make all the foreign keys ON UPDATE CASCADE and ON DELETE NO ACTION. HiredVehicle does have an optional relationship with Insurance (the rest are mandatory) but it doesn't make sense to delete an insurance policy reference if one has been taken out. All foreign keys are NOT NULL apart from *insuranceID*, as the Client can choose not to take out insurance.

HiredVehicle (hireID, date, cardType, cardNo, kilometrage, days, clientID, pickupID, returnID, regNum, tariffID, insuranceID, policyNumber)
Primary Key hireID
Foreign Key clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key pickupID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION

Foreign Key returnID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key regNum **references** Vehicle(regNum) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key tariffID **references** DailyTariff(tariffID) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key insuranceID **references** Insurance(insuranceID) ON UPDATE CASCADE ON DELETE NO ACTION

Booking is a way for Clients to reserve Vehicles in the future (they actually reserve a VehicleType). The DBDL for Booking is below. Client, Depot and VehicleType are the parents in one-to-many relationships with Booking, so their primary keys are posted. As VehicleType has a composite primary key, *make* and *model*, the Booking foreign keys for these need to reference *make* and *model* from the same tuple (i.e. it can't get *make* from one tuple and *model* from another). Therefore, these two attributes are grouped in the DBDL instead of listed on separate lines.

I have chosen ON UPDATE CASCADE and ON DELETE NO ACTION for the foreign keys, since all the relationships have mandatory participation. For this reason they are also NOT NULL. For the VehicleType foreign key, I think using ON UPDATE CASCADE should have an email notification attached, since that will directly impact the car the client will be hiring.

Booking has no other relationships, so this should be its final form.

Booking (startDate, hireDays, colour, clientID, depotID, make, model)
Primary Key startDate
Foreign Key clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key depotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key make, model **references** VehicleType(make, model) ON UPDATE CASCADE ON DELETE NO ACTION

There are a few remaining one-to-many relationships. Depot and VehicleType are both parents to Vehicle. The Vehicle DBDL is below. Vehicle has simple attributes and a single primary key. It has a composite foreign key from VehicleType which describes what kind of Vehicle it is. It has a foreign key from Depot which lists where it is currently available for hire.

For the *make/model* foreign key, I chose NO ACTION on update or delete because these attributes are essential to the car. You can't suddenly change the make of an existing car, which is why it has mandatory participation in the relationship (and is NOT NULL). This relationship could possibly have been modelled as a superclass/subclass relationship of composition form.

For the *depotID* foreign key, I chose CASCADE on update and SET NULL on delete. I chose the latter because Vehicle has an optional relationship with Depot; cars will frequently be unavailable at any depot due to hires and services (and record a null foreign key for the duration). It seems of no consequence to SET NULL if the current depot is deleted.

Vehicle (regNum, fleetNum, colour, make, model, depotID)
Primary Key regNum
Foreign Key make, model **references** VehicleType(make, model) ON UPDATE NO ACTION ON DELETE NO ACTION
Foreign Key depotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE SET NULL

Vehicle and Depot are both parents in one-to-many relationships with ServiceHistory. The DBDL for ServiceHistory is below. It has simple attributes and foreign keys from Vehicle and Depot. Similar to HiredVehicle, *date* is probably insufficient to uniquely identify a service, since many services would likely occur each day. Therefore, I have added regNum to make a composite primary key here as well.

I have chosen the standard options for updates and deletes of the foreign keys, as updates won't change any relevant details and it has mandatory participation in both relationships. This also means both foreign keys are NOT NULL.

ServiceHistory has no other relationships, so this should be its final form.

ServiceHistory (date, cost, description, regNum, depotID)
Primary Key date, regNum
Foreign Key regNum **references** Vehicle(regNum) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key depotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION

Finally, Depot has a one-to-many relationship with ScheduledService. This relation has a couple of simple attributes and a single primary key. It has a foreign key from the Depot at which the future service will occur, and this has mandatory participation (so NOT NULL). It also has a one-to-one relationship with Vehicle that will be addressed in the next section, so this DBDL will likely be modified then. The foreign key is CASCADE on update and NO ACTION on delete.

ScheduledService (date, kilometrage, depotID)
Primary Key date
Foreign Key depotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION

Step 2.1.4: One-to-one (1:1) relationships

One-to-one relationships can lead to several types of relations. Since cardinality is the same on both sides, it is necessary to look at participation.

- If both entities have mandatory participation, it makes sense to combine them into one relation since they will always be one-to-one.
- If one is mandatory and the other is optional, the mandatory participant will the child since it will always have a value for the foreign key, whereas the parent would have nulls. As usual, the primary key from the parent is placed in the child.
- If both are optional, the data should indicate what makes sense. The one with more tuples will likely make a better parent as a foreign key in this entity would have many nulls. In some cases, if the two entities are of similar sizes, it could make sense to model them as a many-to-many relationship in a separate table.

There is a one-to-one relationship from Vehicle to ScheduledService, which is the last relation detailed in the previous section. Because it is mandatory on both sides, these two relations will be combined. The new relation, replacing both Vehicle and ScheduledService needs a new name. Since Vehicle has a more prominent role in the database, it makes sense to name the new entity Vehicle.

The updated DBDL for Vehicle is below. It now includes the attributes from ScheduledService, renamed to maintain their meaning and that relation no longer exists. The primary key from ScheduledService is no longer necessary. The foreign key from ScheduledService to Depot has been maintained, but it has been renamed to distinguish it from the Depot where the Vehicle is currently available (which has also been renamed).

The update and delete options for the foreign keys are maintained from their previous entities. There are no other relationships for this entity, so this should be the final form of this relation.

Vehicle (regNum, fleetNum, colour, make, model, currentDepotID, nextServiceDate, nextServiceKilometrage, nextServiceDepotID)
Primary Key regNum
Foreign Key make, model **references** VehicleType(make, model) ON UPDATE NO ACTION ON DELETE NO ACTION
Foreign Key currentDepotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE SET NULL
Foreign Key nextServiceDepotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION

There is a one-to-one relationship between HiredVehicle and Invoice. Invoice has mandatory participation with HiredVehicle, but HiredVehicle has only optional participation in return. This is because invoices are generated only after a hired vehicle is returned, so this foreign key needs to be null for a short period.

As Invoice has mandatory participation, it makes sense to put the foreign key there. The DBDL for Invoice is below. It has three simple attributes, including the primary key, *invoiceID*. It also has a derived attribute, *finalCost*, which is calculated as the number of days hired (from HiredVehicle) multiplied by the *rentalPrice*, which is an attribute on the relationship between VehicleType and DailyTariff.

The foreign key in Invoice is the primary key of HiredVehicle, *hireID*. Since Invoice has mandatory participation, I have chosen the standard options CASCADE on update and NO ACTION on delete. This also means the foreign keys should be NOT NULL.

Invoice (invoiceID, qualityCheck, datePaid, hireID)
Primary Key invoiceID
Foreign Key hireID **references** HiredVehicle(hireID) ON UPDATE CASCADE ON DELETE NO ACTION
Derived finalCost (HiredVehicle.days * [tba].rentalPrice)

There is another one-to-one relationship between PersonalClient and CompanyClient. It is optional for a PersonalClient to be a CompanyClient, but it is mandatory in the opposite direction. This means CompanyClient should store the primary key of PersonalClient. Since these are both subclasses of Client, this is similar to a recursive relationship, but I will leave deriving their relations till the following sections.

Step 2.1.5: One-to-one (1:1) recursive relationships

There are no recursive relationships in this database.

Step 2.1.6: Superclass/subclass relationships

There is one superclass/subclass relationship identified in the ERD. This is between Client (superclass) and PersonalClient and CompanyClient (subclass). This is specified as a {Mandatory, Or} relationship, which means Clients must specialise into one of these, but can't specialise into both. Typically, {Mandatory, Or} relationships are designed as two separate tables, each taking the attributes of the superclass.

In this case, however, because of the strong mandatory relationship between Client and PersonalClient, the optional relationship between PersonalClient and CompanyClient and the existential dependence of the two subclasses on the superclass, I think it makes more sense to design these more like a {Mandatory, And} relationship. This usually has a single table for all three entities, with flags to indicate which type of subclass it would have been. Given that many Clients won't be CompanyClients, a single table would probably have a lot of nulls, which is inefficient use of space. I instead only combine Client and PersonalClient and leave CompanyClient as a weak entity with optional specialisation.

The updated definition of the Client relation is below. It now includes the attributes of PersonalClient, which include the composite attribute *name*, represented only by its constituents. Since PersonalClient had a mandatory relationship with Client, there is no need to have a flag for it. Client continues to have no foreign keys, although the previous relationship between PersonalClient and HiredVehicle will need to be addressed in the next section.

CompanyClient lists *clientID* as its primary key, which is also a foreign key from Client and for both reasons is NOT NULL. Since it is a weak entity, I have chosen to CASCADE on both update and delete. Its only other attribute is simple.

Client (clientID, fName, lName, title, driversNum, street, postcode)
Primary Key clientID

CompanyClient (clientID, cName)
Primary Key clientID
Foreign Key clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE CASCADE

Step 2.1.7: Many-to-many (*:*)relationships

For many-to-many relationships, it is usually unwieldy to combine them in one of the entities. Generally, a new relation is created which takes a primary key from either entity, and sometimes adds a relationship attribute.

There are two many-to-many relationships in this database: between HiredVehicle and Client (now) and between DailyTariff and VehicleType.

HiredVehicle must specify at least one Client to be a driver but it can also specify multiple drivers. This is messy to include in the HiredVehicle relation so I will create a new relation to combine pairs of HiredVehicles and drivers(Clients). This is similar to how multivalued attributes are handled in the next section. I have called this new relation Drivers.

Drivers only has two attributes, which are the foreign keys provided by Client and HiredVehicle. They are both NOT NULL. I have chosen default options for update and cascade.

Drivers (clientID, hireID)
Primary Key clientID, hireID
Foreign Key clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key hireID **references** HiredVehicle(hireID) ON UPDATE CASCADE ON DELETE NO ACTION

For the many-to-many relationship between DailyTariff and VehicleType I have created a new relation called VehicleTariff, which is the daily rental tariff (in dollars) for each vehicle type for each set of conditions. The *make, model* and *tariffID* attributes are all foreign keys and together form a composite primary key in this relation, while *rentalPrice* is the attribute on the relationship. I have chosen default options for updates and deletions.

VehicleTariff (make, model, tariffID, rentalPrice)
Primary Key make, model, tariffID
Foreign Key make, model **references** VehicleType(make, model) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key tariffID **references** DailyTariff(tariffID) ON UPDATE CASCADE ON DELETE NO ACTION

Step 2.1.8: Complex relationship types

All relationships in the ERD are binary; there are no complex relationships.

Step 2.1.9: Multi-valued attributes

There are two multi-value attributes, which are the phone numbers from Client and Depot. These form weak entity relations, with the primary key of the parent and the phone number paired to uniquely identify each tuple. In this way, Clients and Depot can have as many or as few phone numbers as they want (not limited by the database anyway). The DBDL for these two relations are below.

Each only has the two attributes as described, with the foreign key inserted from their respective parents. The foreign key and phone number create a composite primary key. Since these are dependent entities, I have chosen CASCADE on both update and delete. All attributes are NOT NULL as they are primary keys.

ClientPhoneNo (clientID, phoneNo)
Primary Key clientID, phoneNo
Foreign Key clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE CASCADE

DepotPhoneNo (depotID, phoneNo)
Primary Key depotID, phoneNo
Foreign Key depotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE CASCADE

Complete Logical Model

The complete logical model is presented below.

Client (clientID, fName, lName, title, driversNum, street, postcode) **Primary Key** clientID	**ClientPhoneNo** (clientID, phoneNo) **Primary Key** clientID, phoneNo **Foreign Key** clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE CASCADE
Drivers (clientID, hireID) **Primary Key** clientID, hireID **Foreign Key** clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** hireID **references** HiredVehicle(hireID) ON UPDATE CASCADE ON DELETE NO ACTION	**CompanyClient** (clientID, cName) **Primary Key** clientID **Foreign Key** clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE CASCADE
Depot (depotID, street, postcode, fax) **Primary Key** depotID	**DepotPhoneNo** (depotID, phoneNo) **Primary Key** depotID, phoneNo **Foreign Key** depotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE CASCADE
Vehicle (regNum, fleetNum, colour, make, model, currentDepotID, nextServiceDate, nextServiceKilometrage, nextServiceDepotID) **Primary Key** regNum **Foreign Key** make, model **references** VehicleType(make, model) ON UPDATE NO ACTION ON DELETE NO ACTION **Foreign Key** currentDepotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE SET NULL **Foreign Key** nextServiceDepotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION	**HiredVehicle** (hireID, date, cardType, cardNo, kilometrage, days, clientID, pickupID, returnID, regNum, tariffID, insuranceID, policyNumber) **Primary Key** hireID **Foreign Key** clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** pickupID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** returnID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** regNum **references** Vehicle(regNum) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** tariffID **references** DailyTariff(tariffID) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** insuranceID **references** Insurance(insuranceID) ON UPDATE CASCADE ON DELETE NO ACTION
VehicleType (make, model, doors, body, trim) **Primary Key** make, model	**Insurance** (insuranceID, policyType, cost) **Primary Key** insuranceID
DailyTariff (tariffID, conditions) **Primary Key** tariffID	
VehicleTariff (make, model, tariffID, rentalPrice) **Primary Key** make, model, tariffID **Foreign Key** make, model **references** VehicleType(make, model) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** tariffID **references** DailyTariff(tariffID) ON UPDATE CASCADE ON DELETE NO ACTION	**Invoice** (invoiceID, qualityCheck, datePaid, hireID) **Primary Key** invoiceID **Foreign Key** hireID **references** HiredVehicle(hireID) ON UPDATE CASCADE ON DELETE NO ACTION **Derived** finalCost (HiredVehicle.days * VehicleTariff.rentalPrice)
Booking (startDate, hireDays, colour, clientID, depotID, make, model) **Primary Key** startDate **Foreign Key** clientID **references** Client(clientID) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** depotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** make, model **references** VehicleType(make, model) ON UPDATE CASCADE ON DELETE NO ACTION	**ServiceHistory** (date, cost, description, regNum, depotID) **Primary Key** date, regNum **Foreign Key** regNum **references** Vehicle(regNum) ON UPDATE CASCADE ON DELETE NO ACTION **Foreign Key** depotID **references** Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION

Step 2.2 Validate relations using normalization

A logical model developed with an ERD and following the steps of logical modelling should be quite well normalised.

- It should definitely meet first normal form (1NF), where each column and row intersection has only one value.
- It should probably meet second normal form (2NF), where each non-primary key attribute is fully functionally dependant on the primary key attribute – this is why we choose unique keys!
- It will likely be quite close to third normal form (3NF), where no non-primary key attribute is transitively dependent on the primary key, due to appropriate design of entities when building the entity-relationship model. A quick scan of my logical model suggests this seems to be the case.

Part B: Build an SQL model with the Database

In this section I use SQLite to build the database from the logical model and populate it with customers, vehicles and hires. First, I define the integrity constraints on the database. Next, I use Data Definition Language (DLL) to construct the database in SQLite (see attached wheels_bld.sql). I then use Data Manipulation Language (DML) to INSERT data into the table (see attached wheels_dat.sql). Finally, I use Structure Query Language (SQL) to validate the user transactions and integrity constraints (see attached wheels_queries.sql).

Step 2.4 Define integrity constraints

In this section I check and set integrity constraints on the database. Much of this has been determined in the ERD and its documentation. There are five areas to address for each relation:

Required data. This is the identification of data that cannot be NULL which I have largely already done.

Attribute domain constraints. This sets the domains, or set of possible values, for each attribute. Sometimes these are generic, like all integers; sometimes they are specific, like gender can only be represented by 'M', 'F' or 'U'. These domains have been largely specified in the ERD documentation already. The SQL ISO standards define the types that can be used in DDL. SQL groups these into more generic types behind the scenes but I will use the ISO types as good practice.

Entity integrity. This says that all primary keys need to be unique, not null and only one per relation. This is achieved by design.

Referential integrity. This says all foreign keys must refer to an existing tuple of another relation or be null. This guards against dataset errors when inserting or updating content. This has been done in Part A.

General Constraints. These are logical constraints on the data. For instance, requiring an attribute to have all UNIQUE values, or fulfilling some other ASSERTION.

Step 2.4 CREATE TABLES and INSERT data

Below is an example of the code for building a relation, Vehicle, which also demonstrates its integrity constraints. The rest can be found in the attachment “wheels_bld.sql”.

```
CREATE TABLE Vehicle (
regNum                      CHAR(7)      NOT NULL,
fleetNum                    INT(3),
colour                      CHAR(12),
make                        CHAR(8)      NOT NULL,
model                       CHAR(8)      NOT NULL,
currentDepotID              INT(2)       NOT NULL,
nextServiceDate             DATE,
nextServiceKilometrage      INT(6),
nextServiceDepotID          INT(2)       NOT NULL,
PRIMARY KEY (regNum)
FOREIGN KEY (make, model) REFERENCES VehicleType(make, model) ON UPDATE NO ACTION ON DELETE NO ACTION,
FOREIGN KEY (currentDepotID) REFERENCES Depot(depotID) ON UPDATE CASCADE ON DELETE SET NULL,
FOREIGN KEY (nextServiceDepotID) REFERENCES Depot(depotID) ON UPDATE CASCADE ON DELETE NO ACTION
);
```

The Invoice relation has a derived attribute, *finalCost*, which is not stored in the database but calculated as needed. This is an ideal case to create a VIEW, which is like a permanent query that lives in the database. The code for it is below and is run when the database is built. It is quite a complicated query as the required data are several tables apart.

```
CREATE VIEW InvoiceWithCost (invoiceID, make, model, dateHired, daysHired, dailyTariff, finalCost, datePaid)
AS
SELECT i.invoiceID, vt.make,  vt.model, hv.date AS 'dateHired', hv.days AS 'daysHired', t.rentalPrice AS 'dailyTariff',
hv.days * t.rentalPrice AS finalCost, i.datePaid
FROM Invoice i, HiredVehicle hv, Vehicle v, VehicleType vt, VehicleTariff t, DailyTariff dt
WHERE i.hireID = hv.hireID
        AND hv.tariffID = dt.tariffID
        AND hv.regNum = v.regNum
        AND v.make = vt.make AND v.model = vt.model
        AND vt.model = t.model AND vt.make = t.make AND dt.tariffID = t.tariffID;
```

Finally, I added data to the database using INSERT instructions, which are in the attached “wheels_dat.sql”. Below is an example of some of the INSERT code:

```
INSERT INTO Client VALUES ('AAAAAAAA', 'JOEL', 'PILLAR', 'SIR', 'LZ3148', '23 MadeUp Road', '5050');
INSERT INTO Depot VALUES (01, '11 Town St', '5000', '555-1234');
INSERT INTO VehicleType VALUES ('Holden', 'Commodore', 2, 'Hatch', 'Luxury');
INSERT INTO DailyTariff VALUES ('T1', 'People under 25');
INSERT INTO Vehicle VALUES ('AAA 111', 111, 'BlueSteel', 'Ford', 'Falcon', 01, '2020-04-04', 120000, 01);
```

Step 2.3 Validate relations against user transactions

In database development, like in all project delivery, it's necessary to check the final product against the needs of the user. The design document specified many questions that the database should be able to answer. I used SQL to produce results for these questions. The full set of queries are attached (see wheels_queries.sql) and a couple are below.

```
--1. Add a new hire:

INSERT INTO HiredVehicle VALUES ('19--CCBBAA', '2016-02-29', 'MA', '4992585470986666', 90000, 13, 'BBBBBBBB', 02,
01, 'CCC 333', 'T2', 12346, 'BBBBB-33333');

--2. List all [vehicle types] that do not currently have a future hire booked.

SELECT make, model
FROM VehicleType v
WHERE NOT EXISTS (
        SELECT *
        FROM Booking b
        WHERE v.make = b.make
          AND v.model = b.model);
```

I also ran a few tests of my integrity rules. These INSERTs should generally fail. For example:

```
--TEST: double FOREIGN KEY of Vehicle (both make and model exist but in different tuples)

INSERT INTO Vehicle
VALUES ('ZZZ 999', 120, 'Yellow', 'Ford', 'Commodore', 01, '2020-04-04', 120000, 01);

--TEST: CONSTRAINT rules of ENUM in VehicleType ('Bug' was not an enumerated value of make)

INSERT INTO VehicleType
VALUES ('Volkswagen', 'Beetle', 2, 'Bug', 'Standard');
```

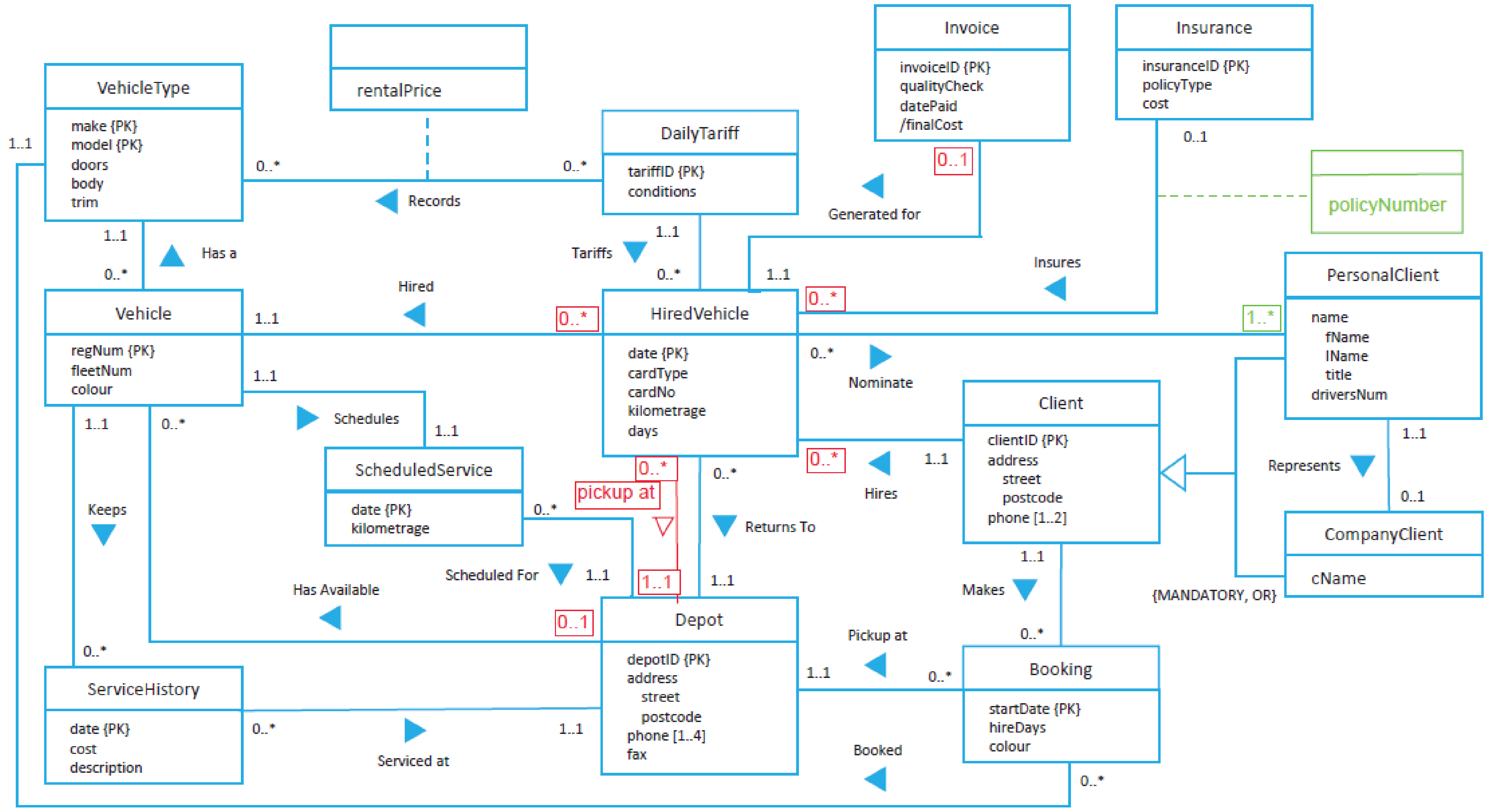
Finally, I tested the ON UPDATE and ON DELETE functions. For example:

```
--TEST: UPDATE DepotID to test CASCADE to Vehicle (should succeed because ON UPDATE CASCADE specified)

(print vehicles)
UPDATE Depot
SET depotID = 06
WHERE depotID = 01;
(print vehicles again)

--TEST: DELETE a Client to test CASCADE to Drivers (should fail because ON DELETE NO ACTION specified)

(print drivers)
DELETE FROM Client
WHERE clientID = "BBBBBBBB";
(print drivers again)
```

Conclusion

In this assignment, I followed the steps of logical modelling outlined in the lecture notes to produce a logical model for Wheels4U. Specifically, I identified entities, created relations based on multiplicities and set update and delete options for foreign keys. I then built and populated an SQLite database using this logical model. I tested that the database could run the required operations from the user and I also tested whether my constraints and foreign key update/delete options were working correctly.

Appendix A – Entity Relationship Diagram – Wheels


VehicleType
make {PK}
model {PK}
doors
body
trim
rentalPrice
DailyTariff
tariffID {PK}
conditions
Invoice
invoiceID {PK}
qualityCheck
datePaid
/finalCost
Insurance
insuranceID {PK}
policyType
cost
policyNumber
PersonalClient
name
fName
lName
title
driversNum
Vehicle
regNum {PK}
fleetNum
colour
HiredVehicle
date {PK}
cardType
cardNo
kilometrage
days
Client
clientID {PK}
address
street
postcode
phone [1..2]
CompanyClient
cName
ScheduledService
date {PK}
kilometrage
Depot
depotID {PK}
address
street
postcode
phone [1..4]
fax
ServiceHistory
date {PK}
cost
description
Booking
startDate {PK}
hireDays
colour
Records
Has a
Tariffs
Generated for
Insures
Hired
Nominate
Schedules
Hires
Represents
Keeps
pickup at
Returns To
Scheduled For
Has Available
Makes
Pickup at
Serviced at
Booked
{MANDATORY, OR}
1..1
0..*
0..1
1..*