MySQL Bike Share Model Relational Database

The database was based on a model of Bike Share (figure 1). The model assumes that bike rotation, repairs and payments are all connected on a singular system. The isolated tables, *payment methods* and *customer details* were created first without the requirement for foreign keys. A one-to-one relationship was created between *customer details* and *customers*. The only way I found in MySql was using the primary key as the foreign key to create this relationship. The three tables represented all the customer’s information which was required for subscribing to the service. The next table created was *payments* which used the two new primary keys as foreign keys. This was because customers will have many payments and many different methods will be used to pay.

The next two tables created were *stations* and *staff* as they were independent from surrounding entities not requiring a foreign key in creation from a non-existing table. Once *station’s* primary key was available it was used as the foreign key in *station status* and *bikes* tables as stations will have many states and bike rotation respectively. *Bikes’* primary key was used as a foreign key in *bike status*. A trigger was also created in *bikes* to upgrade *bike status* information automatically once a bike was activated. This was the most difficult element to incorporate. The *stations* and *bikes* information was grouped as static information as they will be relatively fixed attributes.

The company information was expanded upon next using the *staff* and *stations* foreign keys in *vans* table as staff members will use many *vans* and *stations* will be rotated by several vans. The primary key for *vans* was based on the license plate present. *Repairs* used the foreign keys of *staff* and *bike status* as staff members will engage in several repairs due to several different individual bike states overtime. Any updates to staff will be recorded as a trigger to the *old staff* table. The relationship was one to many as one staff member could have several amendments such as changes of address. A procedure (*GetOld\_Staff*) was created to verify that an update to *staff* would create an entry in *old staff*. This completed all the customer information table.

The final section was dynamic information completed by the introduction of the *bike rentals* table. The table has three one to many connects. The *customer* will have many *bike rentals* requiring many *payments* and using many different *bikes* overtime. The cyclical structure between *customers*, *payments* and *bike rentals* requires data recorded to be unilaterally cross-referenceable (customer’s events=payments=bike rentals) due to shared foreign keys and the unique event recorded by repeatable *customers*. This was verified using a view (*rental*) which looked at the results from the *bike rentals* table. The *customer* (10) made two successive rentals using two different *payments* for two different *bikes* using two different *payment* transactions.

The database could be improved upon using more triggers to updates status tables and calculating geometry between journey from stations. Once a bike reaches a certain mileage, it would active a bike status which requires a mechanic to repair elements from the bike. The database can currently be used to assess which staff member has worked the most or which customer uses the service the longest. The database was complicated to construct but the organisation of the entity-relationship schema aided in providing context to each tables connectivity and functionality.

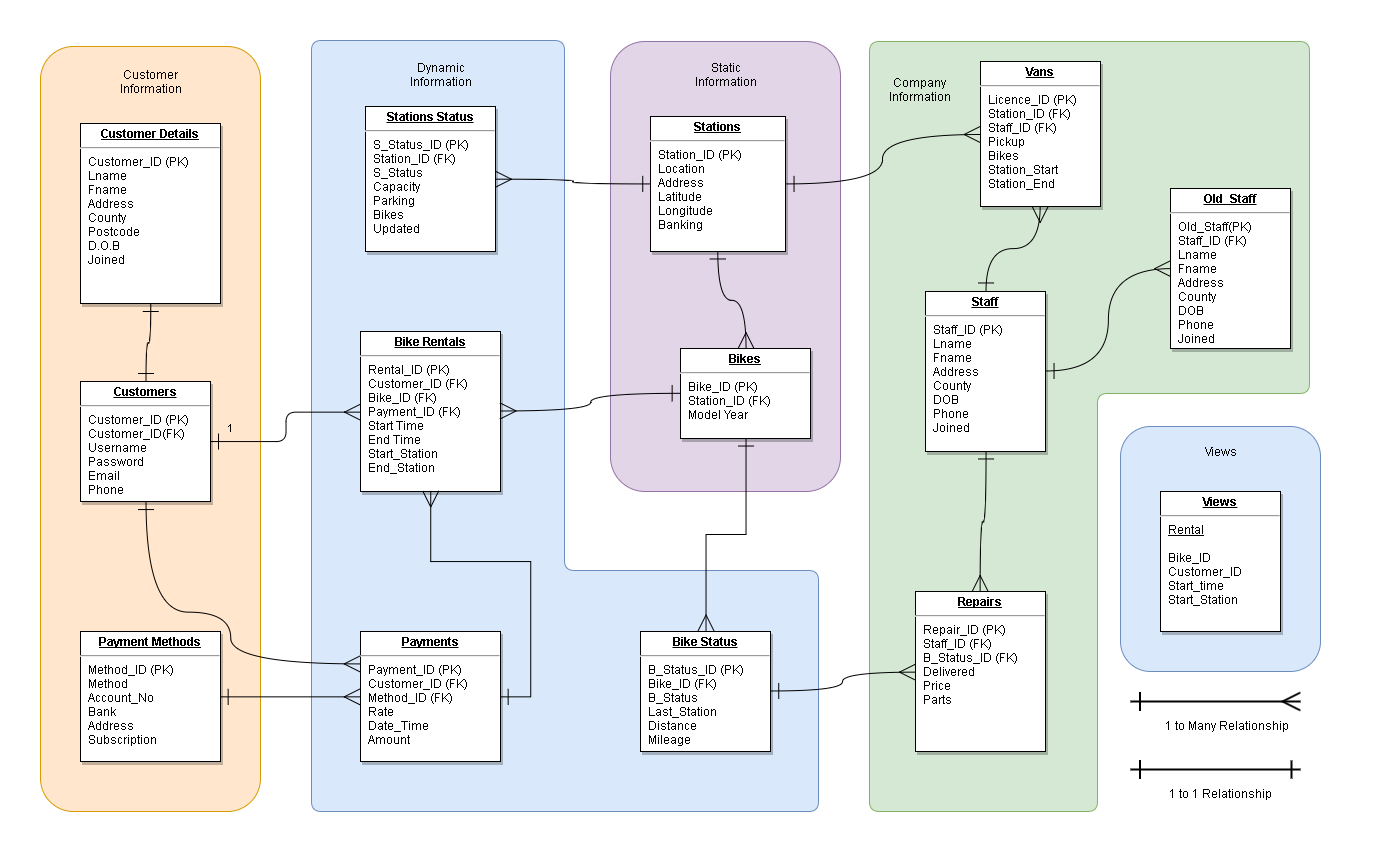


Figure 1: Entity – Relationship schema diagram showing structure between tables using crow’s feet