CS27020
Modelling Persistent Data
Assignment: Ski Lifts and Pistes

Author:
James Euesden (jee22)

Address:
Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

Date: February 21, 2014

1 Introduction

This task set is to build a relational database, using PostgreSQL. To begin this task, I have been provided with sample data of Pistes and Lifts, which must be viewed in Unnormalized Form (UNF), and then taken through the normalization process to reach Third Normal Form (3NF). This will be done by determining the Functional Dependencies, constructing Primary Keys and understanding other Candidate Keys, then using these to help with the normalization steps. With the resulting model, I will then create the Database, and provide suitable commands for conducting a series of queries on the database, with screenshot evidence.

2 Analysis

2.1 Unnormalized Structure

An Unnormalized Structure is data that has not been Normalized, or ready to be put safely and logically into a relational database. This is data that we might find in the 'real world' during situations where we are provided large quantities of data that, while it might make logical sense to a client, may not be appropriate to be implemented into a relational database.

Based upon the sample data provided[**?**], the unnormalized structure of the Database is as follows. A thing to initially note is how there appears to be some correlation between the data in 'rise' in Lift and 'fall' in Piste. However, this only happens for a few of the entries, and with multiple values in attributes, is not something to be taken into consideration.

2.1.1 Piste

piste_name
grade
length_km
fall_m
lif_name
piste_open

2.1.2 Lift

lift_name
lift_type
summit_m
rise_m
length_m
operating

2.2 Functional Dependencies

A functional dependency is where data relies depends upon another piece of data in order to be determined. This can be expressed as FD: X -> Y. To see the functional dependencies, I made a number of assumptions about the data, based upon the sample data provided.

2.2.1 Piste

piste_name ->grade, length_km, fall_m, piste_open

We assume that this information about a Piste is functionally dependant upon the piste_name, and this is how it should be accessed. Each attribute provides a 'fact' about the Piste, based on the piste_name.

piste_name ->lift_name

In order to find the name of Lifts (lift_name) servicing a Piste, we must know the piste_name. However, it should be noted that Lift is in itself it's own relation. On top of this, a Lift can serve many Pistes, and so it should be noted that this could also be expressed as:

piste_name <-> lift_name

2.2.2 Lift

lift_name ->lift_type, summit_m, rise_m, length_m, operating

Similar to Piste, we assume that the data about a Lift is functionally dependant upon lift_name. Each of these items is functionally dependant on the lift_name, and gives data, a 'fact', about a Lift.

2.3 Primary & Candidate Keys

A Primary Key (PK) is a key that is unique to each record in a relation, and that will never be repeated in the data set. This key can be a single attribute, or a composite key, comprised of multiple attributes. This key is used as the unique identifier of a relation. When picking a Primary Key, it must conform to being unique to that relation, while also being as unchaging as possible (immutable - could change, but shouldn't) and there may also be Candidate Keys.

When deciding the Primary Keys for these relations, I found it challenging to decide the correct course of action. My initial thought was to use the piste_name and lift_name. However, while these are likely to be unique, it is also possible that in the future, a user may want to change or update the names, causing potential update issues. With this in mind, my choice was to make an auto-incremented integer value for both Piste and Lift in order to represent a unique identification number. These are Piste(piste_uid) and Lift(lift_uid).

This choice to use a UID means that a user has the freedom of adding a number of different Pistes, while still giving them the ability to alter the names of a Piste at a

later date. With this considered, I decided to continue using piste_name and lift_name as Candidate Keys. This means that while they are not the true Primary Key, they are a Candidate for it, and it should be noted along with the functional dependencies when running the normalization process.

It could be considered that now piste_name and all it's functionally dependant attributes are now all functionally dependent upon piste_uid, and the same for Lift. Since the name values are staying as Candidate keys, and the UID is merely for unique representaton purposes though, the original functional dependencies still hold.

As an extra precaution when creating the database, I will constrain the lift_name and piste_name to be unique, enforcing the rule that no two Pistes or Lifts should be named the same, but their names are free for change.

3 Normalizing the Data

As previously mentioned, Normalizing data is the process of working through 'Normalization steps' to reach certain forms. These forms are good for use in relational database models, and each form removes some form of issue when implementing the data into a database. For this task, the data will be taken to 3NF. When normalizing this data, I will be dealing with each relation (Piste, Lift) separately, and will bring them together when it seems appropriate.

3.1 First Normal Form

The act of taking data from UNF into 1NF is by disallowing attributes to have multiple values. This means that an attribute could not contain two values, such as 2 phone numbers for one person. In the sample data provided, it can be seen that within 'Piste', there are multiple values in 'lift_name'. This violates 1NF rule.

In order to solve this, we can move lift_name out of the Piste and into a new relation, 'Connection'. This new relation contains the Primary Key of Piste and the attribute lift_name. Piste no longer contains lift_name, while otherwise staying the same. This brings Piste into 1NF.

Lift does not have any multiple values within it's attributes, nor could it be assigned any in the future. This means that Lift is already in 1NF and does not need anything doing. There are no more sets of multiple values in Lift or Piste, and Connection is also acceptable, at this stage.

The end result of the 1NF operations are below, with the current three relations shown. Those attributes underlined represent Primary Key components. Those attributes with an asterisk () are foreign keys. These relations still have some anomalies however, that will be dealt with in 2NF.

3.1.1 Piste

piste_uid
piste_name
grade
length_km
fall_m
open

3.1.2 Lift

lift_uid
lift_name
lift_type
summit_m
rise_m

length_m
operating

3.1.3 Connection

piste_uid*
lift_name*

(piste_uid* references piste_uid from relation Piste). (lift_name* references lift_name from relation Lift).

3.2 Second Normal Form

Achieving Second Normal Form relies upon two things, the first being that 1NF is already achieved, the second being that every non-Primary Key attribute of the relation is dependent on the whole of a candidate key. As we can see from the new relation 'Connection', and our functional dependencies, lift_name depends on piste_name, which is not in connection. In order to give the new relation content, fulfill 2NF and satisfy the functional dependencies, we make connection into a Many-to-Many table.

To do this, we bring the Primary key of Lift into connection, and bring piste_name into connection too. This table now contains 4 attributes, all foreign keys, two UIDs as a composite Primary Key, and the two names as Candidate keys. Together, these form the link between Piste and Lift, while also giving context to the UIDs for queries and for the functional dependencies.

This is valid with 2NF, as we can assume that the full PK of Lift and the full PK of Piste are dependent upon one another in this relation. The attributes left within Lift and Piste are wholly dependant on the Primary Key, and so are also valid. The current relationships are:

3.2.1 Piste

piste_uid
piste_name
grade
length_km
fall_m
open

3.2.2 Lift

lift_uid
lift_name
lift_type
summit_m
rise_m

length_m
operating

3.2.3 Connection

piste_uid*
lift_uid*
piste_name*
lift_name*

(piste_uid* and piste_name* reference piste_uid and piste_name from relation Piste). (lift_uid* and lift_name* reference lift_uid* and lift_name from relation Lift).

3.3 Third Normal Form

For a database to be valid for Third Normal Form, it must first conform to 2NF, and also have no transitive dependencies. This requires that all non-key attributes rely upon only the PK, and nothing but the key, providing a fact about the PK and nothing else.

If we look at our current relations, we can see that this is already the case, where each attribute of data provides a fact about the PK. When it comes to Connection, the lift_name and piste_name both do provide a fact each about the PK, indication what the name of the Lift or Piste is corresponding to the respective UID.

Each attributes relies soley upon the Primary Key of its relation, and provides a fact about that Primary Key, providing no information about any other aspect of the database or of itself. From all this, we can see that Lift has been in 3NF throughout the whole process, and the database structure is the same as it was in 2NF.

4 PostgreSQL

With the data now in 3NF, it is suitable to be put into a database. For this task, it must be placed into a PostgreSQL table. I have created this on my personal filestore at Aberystwyth University. Below are the typescripts of the commands I used to create these, and screenshots to demonstrate their use and the results of my queries.

4.1 Creating the tables

The typescript for creating the database:

```
-- Create a sequence for incrementing the piste_uid
CREATE SEQUENCE piste_uid_seq;

-- Create a type for grading the Piste
CREATE TYPE grade_rank as
        ENUM('EASY', 'MEDIUM', 'HARD', 'DIFFICULT');

-- Create the Piste table.
-- PK UID. Unique Names.
CREATE TABLE piste (
        piste_uid int DEFAULT nextval('piste_uid_seq')
                NOT NULL PRIMARY KEY,
        piste_name varchar(50) UNIQUE NOT NULL,
        grade grade_rank NOT NULL DEFAULT 'EASY',
        length_km real NOT NULL,
        fall_m integer NOT NULL,
        open_piste boolean
);
ALTER SEQUENCE piste_uid_seq OWNED BY piste.piste_uid;

-- Create a sequence for incrementing the lift_uid
CREATE SEQUENCE lift_uid_seq;

-- Create a type for the type and operating Lifts
CREATE TYPE type_lift as
        ENUM('GONDOLA', 'CHAIR', 'TOW');

-- Create the Lift table.
-- PK UID. Unique Names.
CREATE TABLE lift (
        lift_uid int DEFAULT nextval('lift_uid_seq')
                NOT NULL PRIMARY KEY,
        lift_name varchar(60) UNIQUE NOT NULL,
```

```
        lift_type type_lift NOT NULL,
        summit_m integer NOT NULL,
        rise_m integer NOT NULL,
        length_m integer NOT NULL,
        operating boolean
);
ALTER SEQUENCE lift_uid_seq OWNED BY lift.lift_uid;

-- Create the table Connections (M-M).
-- PK is UIDs, Names for content.
CREATE TABLE connections (
        piste_uid integer NOT NULL,
        piste_name varchar(50) NOT NULL,
        lift_uid integer NOT NULL,
        lift_name varchar(60) NOT NULL,
        CONSTRAINT fkey_piste_uid
                FOREIGN KEY (piste_uid)
                REFERENCES piste (piste_uid),
        CONSTRAINT fkey_piste_name
                FOREIGN KEY (piste_name)
                REFERENCES piste (piste_name),
        CONSTRAINT fkey_lift
                FOREIGN KEY (lift_uid)
                REFERENCES lift (lift_uid),
        CONSTRAINT fkey_lift_name
                FOREIGN KEY (lift_name)
                REFERENCES lift (lift_name),
        PRIMARY KEY (piste_uid, lift_uid)
);
```

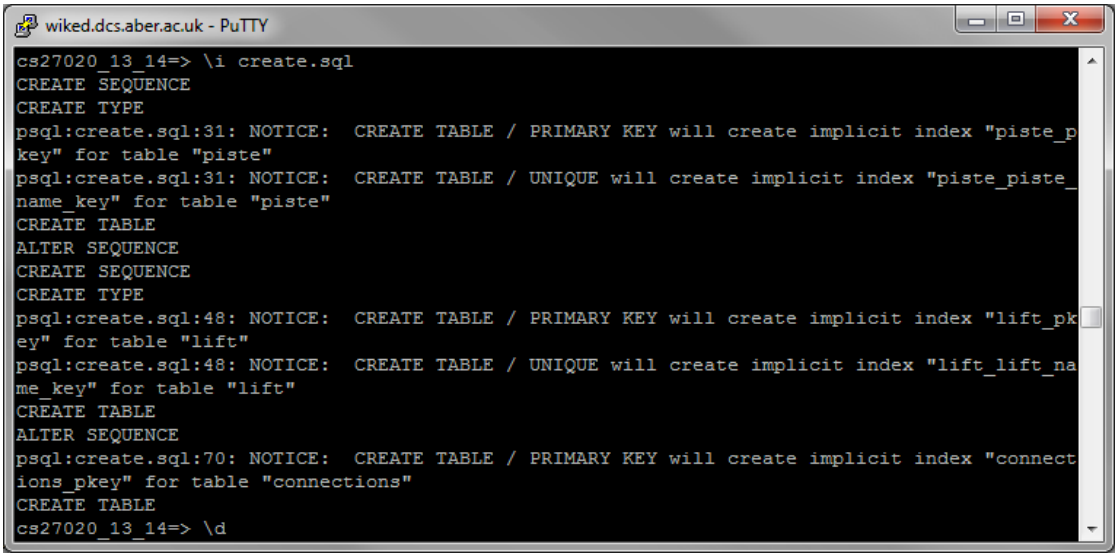

Figure 1: The output when creating tables.

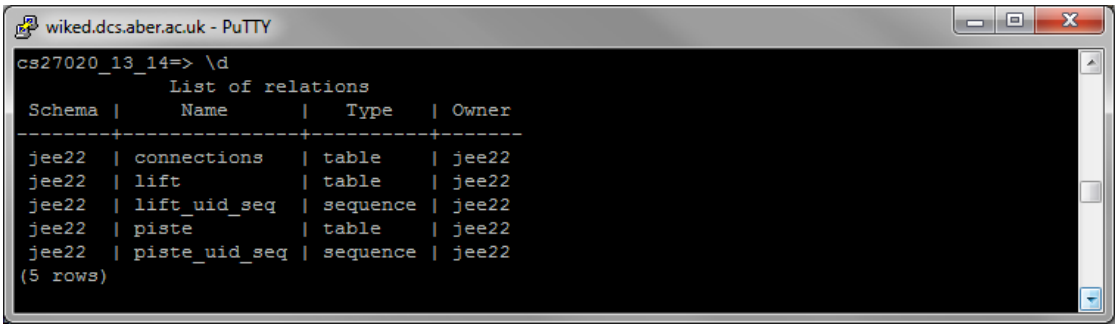

Figure 2: The list of relations after running the create typescript.

4.2 Data Types Justification

Most data types have the contraint that they are 'NOT NULL'. When dealing with certain aspects of real data, such as a Piste or Lift, that definitely does have a specfic state (open/operating), specific numbers for lengths, rises and summits, and must have a name, I felt that NOT NULL should be included. There are no items in the sample data that have empty values for attributes.

4.2.1 Piste

- piste_uid - int NOT NULL The piste_uid is kept as an integer to be a Unique Identification number. This way of using an integer allows it to be incremented for the number of values.

- piste_name - varchar(50) UNIQUE NOT NULL varchar allows the input of text, so the user can specify the name of the Lift. Setting the limit at 60 allows for a decent amount of characters without allowing unreasonable amounts. By using the constraint 'UNIQUE', it is enforcing the user to never have two of the same named Pistes.
- grade - grade_rank NOT NULL I made a custom TYPE of ENUM for grade_rank, allowing the user to select from the different types of ranks, and using an ENUM allows for future expansion on these grades if necessary.
- length_km - real NOT NULL Using a real allows some floating point precision. Since the km is used to represent the length, where there can be numbers after the decimal, a real seemed appropriate to reprsent this data.
- fall_m - integer NOT NULL A way to represent the fall as a whole number.
- open_piste - boolean Since a piste can either be open or not open, a true/false boolean seemed appropriate to determine either/or.

4.2.2 Lift

- lift_uid - int NOT NULL
  The lift_uid is kept as an integer to be a Unique Identification number. This way of using an integer allows it to be incremented for the number of values.
- lift_name - varchar(60) UNIQUE NOT NULL varchar allows the input of text, so the user can specify the name of the Lift. Setting the limit at 60 allows for a decent amount of characters without allowing unreasonable amounts. By using the constraint 'UNIQUE', it is enforcing the user to never have two of the same named Lifts.
- lift_type - type_lift NOT NULL I made a custom TYPE of ENUM for lift_type, allowing the user to select from the types of lift in the sample data, and using an ENUM allows for future expansion on types if necessary.
- summit_m - integer NOT NULL A way to represent the summit as a whole number.
- rise_m - integer NOT NULL A way to represent the rise as a whole number.
- length_m - integer NOT NULL A way to represent the length as a whole number.
- operating - boolean Since a lift can either be operating or not operating, a true/false boolean seemed appropriate to determine either/or.

4.3 Quering the Database

4.3.1 Test Data

In order to test my data, I wrote the sample data into a typescript to insert it into my database. I will not include the typescript here. However, here are the contents of each relation after the insertion:

Where you might see <and >in the Test Query typescripts, it should be assumed that here is where a value would go for the query. An example would be '<lift_name >' could be imagined to be 'Rastlift', or any other lift_name, when executing the query.

4.3.2 Erronous Data Entry

In order to check that my database was working correctly, I conducted a series of tests where I attempted to INSERT incorrect data. Doing this confirmed that my database was robust and built as intended. Below are a series of screenshots and captions detailing my testing.

4.3.3 Test Query 1

"Return the pistes served by a given lift."

```
SELECT *
FROM piste WHERE piste_uid IN
        (SELECT piste_uid
                FROM connections WHERE lift_uid IN
        (SELECT lift_uid
                FROM lift WHERE lift_name='<lift_name>'));
```

4.3.4 Test Query 2

"Return the lift(s) that provide access to a given piste."

```
SELECT *
FROM lift WHERE lift_uid IN
        (SELECT lift_uid
                FROM connections WHERE piste_uid IN
        (SELECT piste_uid
                FROM piste WHERE piste_name='<piste_name>'));
```

4.3.5 Test Query 3

"Return the lifts that are currently operating."

```
SELECT lift_name, operating
        FROM lift WHERE operating='t';
```

4.3.6 Test Query 4

"Return the pistes that are currently open, together with the lifts that are currently operating and that provide access to those pistes."

```
SELECT * FROM connections WHERE lift_uid IN
        (SELECT lift_uid FROM lift WHERE operating='t')
AND piste_uid IN
        (SELECT piste_uid FROM piste WHERE open_piste='t');
```

References

[1] Edel Sherratt, *CS27020 Assignment: Ski Lifts and Pistes*. Computer Science Department, Aberystwyth University, 2014.