

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 23, 2014

Copyright © Aberystwyth University 2014

1 Introduction

The task set for this assignment was to build a relational database, using PostgreSQL, based upon Pistes and Lifts. To begin this task, I was 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 had to be done by determining the Functional Dependencies, selecting Primary Keys and Candidate Keys, then using these to help with the normalization steps.

With the resulting model, I then had to create an appropriate database in PostgreSQL. Provided with the sample data was a number of queries requests that had to be written. These queries had to show the database working, and evidence of their correct operation would support the design and functional dependencies determined in my analysis. The evidence of this testing was to be provided in a series of screenshots and typescripts of the SQL commands.

2 Analysis

2.1 Unnormalized Structure

An Unnormalized Structure/Form (UNF) is data that has not yet 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 and cause issues during use.

The sample data provided[1] involved the attributes of Pistes, and Lifts that serviced those Pistes. Each of these items had attributes about them, and are displayed in my UNF below. A thing I initially noted is how there appeared 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 often occurs where there are multiple values in records, leaving no true connection between these two.

The UNF Data:

2.1.1 Piste

Piste (piste_name)
Grade (grade)
Length(Km) (length_km)
Fall (m) (fall_m)
Lifts (lift_name)
Open? (piste_open)

2.1.2 Lift

Lift (lift_name)
Type (lift_type)
Summit (m) (summit_m)
Rise (m) (rise_m)
Length (m) (length_m)
Operating (operating)

2.2 Functional Dependencies

A functional dependency is where data depends upon another piece of data in order to be determined. This can be expressed as FD: $X \rightarrow Y$. If we know what 'X' is, we can find out the value of 'Y' from it. 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 \rightarrow grade, length_km, fall_m, piste_open

We assume that the data describing a Piste (grade, length, fall and open) are 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 \rightarrow 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 (Many-to-Many), and so it should be noted that this could also be expressed as:

piste_name \rightarrow lift_name

lift_name \rightarrow piste_name

Where we can find the Pistes connected to a Lift, or the Lifts servicing a Piste. This does not come into effect until our Normalization process however. It should be noted now though, for use later.

2.2.2 Lift

lift_name \rightarrow 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. If we know the name of a Lift, we can get the other data associated with it.

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 unchanging as possible (immutable - could change, but shouldn't).

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 unique Pistes and Lifts, while still giving them the ability to alter the names 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 now be considered that `piste_name` and all its functionally dependant attributes are now all functionally dependent upon `piste_uid`, and the same applies for Lift:

`piste_uid -> piste_name`

`piste_name -> grade, length_km, fall_m, piste_open`

`lift_uid -> lift_name`

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

`piste_uid -> lift_uid`

`lift_uid -> piste_uid`

These new UIDs will affect our normalization process slightly, but as it can be seen, the attributes have not changed much from the originally found functional dependencies.

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 issues when implementing the data into a database. For this task, the data will be taken to 3NF.

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 same record. 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 its 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*

(Foreign key piste_uid* references piste_uid from relation Piste)

(Foreign key 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', lift_name is not wholly dependent upon 'piste_uid'. It is dependent upon lift_uid in the functional dependencies.

With this in mind, the structure of Connections should instead be representing a Many-to-Many relationship, with a full Primary Key comprised of the two foreign keys 'piste_uid' and 'lift_uid', and removing lift_name from 'Connections' entirely.

This results in no non-key attributes within Connections, and the Primary Key UIDs supporting the link between Lift and Piste. The three relations now only have non-key attributes that do solely rely upon the Candidate Keys, and all are valid for 2NF.

The new form of the relations now looks like this:

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*

(Foreign key piste_uid* references piste_uid from relation Piste)

(Foreign key lift_uid* references lift_uid* 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. A transitive dependencies is where an attribute relies on only partial of the Primary Key. It can be described that all non-key attributes rely upon only the PK, and nothing but the PK, 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.

Each attributes relies solely upon the Primary Key of its relation, and provides a fact about that whole 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 does not need to change further. The structure is now in 3NF.

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:

```
/*
    CS27020 Pistes and Lifts Assignment
    PostgreSQL commands for creating the database.
    Author: James Euesden (jee22@aber.ac.uk)
    Date: 20/2/14
*/

/*
    Create the database:
    psql -h db.dcs.aber.ac.uk -U jee22 -d cs27020_13_14
    < create.sql
    Also a script for removing all relations:
    drop_tables.sql
    And a script for adding sample data to the database:
    insert_sample.sql
    All required test queries contained in:
    test_queries.sql
*/

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

-- Create a type for grading for 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,
length_km real NOT NULL
        CONSTRAINT length_must_be_positive
        CHECK (length_km > 0),
fall_m integer NOT NULL
        CONSTRAINT fall_must_be_positive
        CHECK (fall_m > 0),
open_piste boolean NOT NULL
);
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 of 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
                CONSTRAINT summit_must_be_positive
                CHECK (summit_m > 0),
        rise_m integer NOT NULL
                CONSTRAINT rise_must_be_positive
                CHECK (rise_m > 0),
        length_m integer NOT NULL
                CONSTRAINT length_must_be_positive
                CHECK (length_m > 0),
        operating boolean NOT NULL
);
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,
        lift_uid integer NOT NULL,

```

```

CONSTRAINT fkey_piste_uid
    FOREIGN KEY (piste_uid)
    REFERENCES piste (piste_uid),
CONSTRAINT fkey_lift
    FOREIGN KEY (lift_uid)
    REFERENCES lift (lift_uid),
PRIMARY KEY (piste_uid , lift_uid)
);

```

```

cs27020_13_14=> \i create.sql
CREATE SEQUENCE
CREATE TYPE
psql:create.sql:34: NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "piste_pkey" f
or table "piste"
psql:create.sql:34: NOTICE: CREATE TABLE / UNIQUE will create implicit index "piste_piste_name_k
ey" for table "piste"
CREATE TABLE
ALTER SEQUENCE
CREATE SEQUENCE
CREATE TYPE
psql:create.sql:54: NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "lift_pkey" fo
r table "lift"
psql:create.sql:54: NOTICE: CREATE TABLE / UNIQUE will create implicit index "lift_lift_name_key
" for table "lift"
CREATE TABLE
ALTER SEQUENCE
psql:create.sql:69: NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "connections_p
key" for table "connections"
CREATE TABLE
cs27020_13_14=>

```

Figure 1: The output when creating tables.

```

cs27020_13_14=> \d
          List of relations
Schema |      Name      | Type   | Owner
-----+-----+-----+-----
jee22  | connections    | table  | jee22
jee22  | lift           | table  | jee22
jee22  | lift_uid_seq   | sequence | jee22
jee22  | piste         | table  | jee22
jee22  | piste_uid_seq  | sequence | jee22
(5 rows)
cs27020_13_14=>

```

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

4.2 Data Types Justification

For each of the attributes, I carefully selected particular data types that I felt would best represent them. On each data type, I have constrained the type or amount of data that they can take in order to reduce the potential for issues in the databases use. I

feel confident in my choices for data types in that they represent the data from the real world as best as they can within my structure.

Most data types have the constraint that they are 'NOT NULL'. When dealing with certain aspects of real data, such as a Piste or Lift, that definitely do have a specific data, I felt that NOT NULL should be included. There are no items in the sample data that have empty values for attributes. I felt that having the majority of attributes constrain to not null ensures that data is correctly input into the table, avoiding potential issues with updates and missing data for future users. This helps to ensure the integrity of the data throughout its lifetime.

For those attributes which are numbers, I have also chosen to constrain them to being greater than 0. Doing this stops a user from being allowed to enter negative values. Each of these has also been named using the CONSTRAINT command, in order to better inform the user of where their error lies, such as 'rise_must_be_positive'. These help a user who may have made a slight typing error see where they have gone wrong in an informative error message.

4.2.1 Piste Data Types

- `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 records entered.
- `piste_name` - varchar(50) UNIQUE NOT NULL PRIMARY KEY
varchar allows the input of text, so the user can specify the name of the Piste. Setting the limit at 50 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. I based the choices of ranks on those provided in the sample data: Easy, Medium, Hard, Difficult.
- `length_km` - real NOT NULL CHECK (`length_km > 0`)
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 represent this data. Example: 4.5km.
- `fall_m` - integer NOT NULL CHECK (`fall_m > 0`)
A way to represent the fall as a whole number in meters.
- `open_piste` - boolean NOT NULL
Since a piste can either be open or not open, a true/false boolean seemed appropriate to determine either/or.

4.2.2 Lift Data Types

- 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 records entered.
- 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. I based the choices of ranks on those provided in the sample data: Gondola, Tow and Chair.
- summit_m - integer NOT NULL CHECK (summit_m > 0)
A way to represent the summit as a whole number in meters.
- rise_m - integer NOT NULL CHECK (rise_m > 0)
A way to represent the rise as a whole number in meters.
- length_m - integer NOT NULL CHECK (length_m > 0)
A way to represent the length as a whole number in meters.
- operating - boolean NOT NULL
Since a lift can either be operating or not operating, a true/false boolean seemed appropriate to determine either/or.

The data types from Connection are the same as those that they Reference, and so the justification of there choices is already included. The attributes comprise the Primary Key as Foreign Keys, and are piste_uid and lift_uid.

4.3 Checking the Database

4.3.1 Sample Data

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

```
cs27020_13_14=> SELECT * FROM piste;
```

piste_uid	piste_name	grade	length_km	fall_m	open_piste
1	Zwischenholzabfahrt	MEDIUM	3	440	t
2	Moeseralmabfahrt	MEDIUM	2.5	400	f
3	Schoenjochabfahrt	MEDIUM	4	510	t
4	Sattelkopf-Suedabfahrt	MEDIUM	4	350	t
5	Sattelkopf-Nordabfahrt	DIFFICULT	1.5	220	t
6	Moeserabfahrt	EASY	0.5	80	t
7	Wonneabfahrt	MEDIUM	1.5	280	f
8	Rastabfahrt	MEDIUM	1	150	f
9	Waldabfahrt	HARD	3	420	t
10	Ladisabfahrt	EASY	3.5	290	t
11	Verbindungsabfahrt	EASY	2	70	t
12	Plazoerabfahrt	MEDIUM	3	360	f
13	Schoengampabfahrt	MEDIUM	3.5	420	t
14	Schoenjochpiste	EASY	1	70	t
15	Almabfahrt	MEDIUM	4	370	f

(15 rows)

```
cs27020_13_14=>
```

Figure 3: The contents of the Piste table.

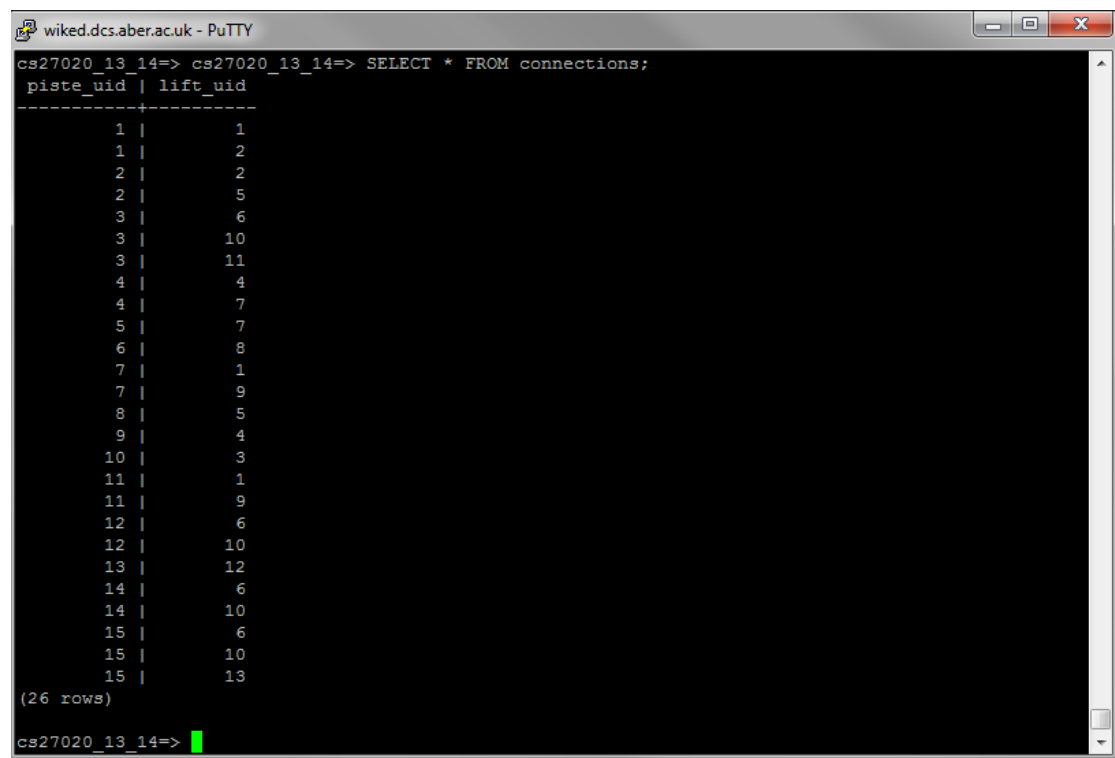
```
cs27020_13_14=> cs27020_13_14=> SELECT * FROM lift;
```

lift_uid	lift_name	lift_type	summit_m	rise_m	length_m	operating
1	Schoenjochbahn I	GONDOLA	1920	440	1600	t
2	ESL-Fiss-Moeseralm	CHAIR	1850	400	1700	f
3	ESL-Ladis-Fiss	CHAIR	1510	290	2700	f
4	Waldlift	TOW	1850	420	1200	t
5	Rastlift	TOW	1900	150	400	t
6	Schoenjochbahn II	GONDOLA	2436	516	1350	t
7	Sattelkopflift	TOW	2100	220	1000	f
8	Moeserlift	TOW	1930	80	400	t
9	Wonnelifft	TOW	2080	280	1000	t
10	Plazoerlift	TOW	2450	360	1350	t
11	Schoenjochlift	TOW	2509	70	420	f
12	Schoengamplift	TOW	2509	420	1340	t
13	Almlift	TOW	2250	370	1180	f

(13 rows)

```
cs27020_13_14=>
```

Figure 4: The contents of the Lift table.



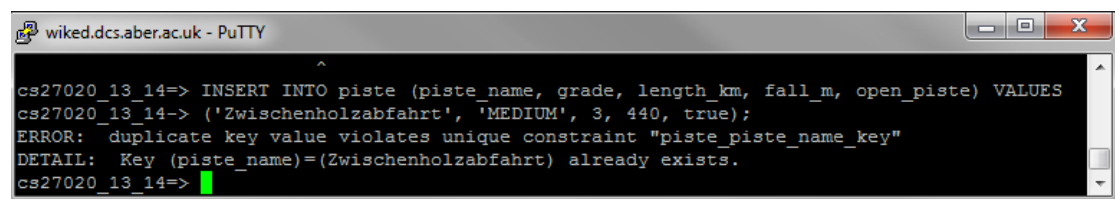
```
cs27020_13_14=> cs27020_13_14=> SELECT * FROM connections;
piste_uid | lift_uid
-----+-----
1 | 1
1 | 2
2 | 2
2 | 5
3 | 6
3 | 10
3 | 11
4 | 4
4 | 7
5 | 7
6 | 8
7 | 1
7 | 9
8 | 5
9 | 4
10 | 3
11 | 1
11 | 9
12 | 6
12 | 10
13 | 12
14 | 6
14 | 10
15 | 6
15 | 10
15 | 13
(26 rows)
cs27020_13_14=>
```

Figure 5: The contents of the Connections table.

4.3.2 Erronous Data Entry

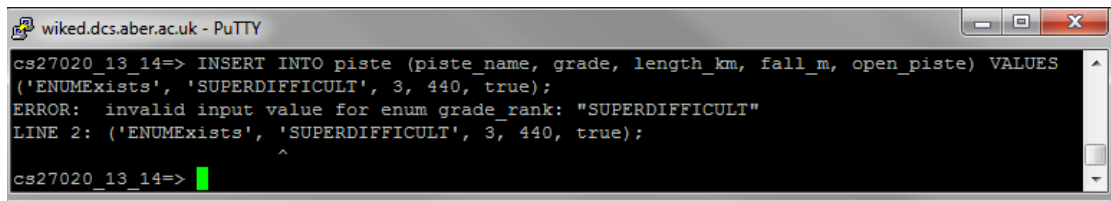
In order to check that my database was working correctly, I conducted a series of tests where I attempted to INSERT incorrect data. Doing this confirmed that my database was robust and built as intended. Below are a series of screenshots and captions detailing some my testing.

I have shown this testing in brief, where I can demonstrate my correctly operating ENUMs, CHECKS and UNIQUE constraints. The ones I have shown are examples, and should be taken as a representation for every other attribute that share the same constraints.



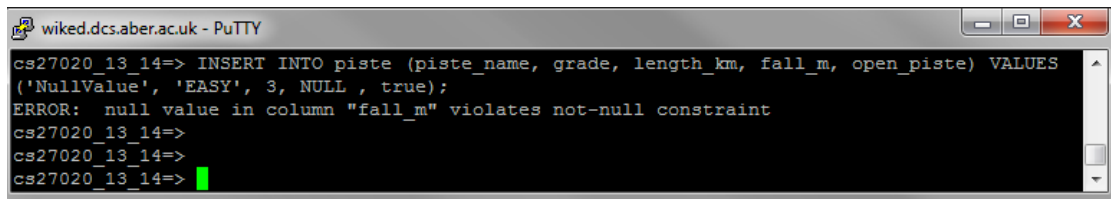
```
cs27020_13_14=> INSERT INTO piste (piste_name, grade, length_km, fall_m, open_piste) VALUES
cs27020_13_14-> ('Zwischenholzabfahrt', 'MEDIUM', 3, 440, true);
ERROR: duplicate key value violates unique constraint "piste_piste_name_key"
DETAIL: Key (piste_name)=(Zwischenholzabfahrt) already exists.
cs27020_13_14=>
```

Figure 6: An entry cannot have the same piste_name as an already existing Piste - UNIQUE constraint. The item can be renamed, but should not share the same name with another of it's type.



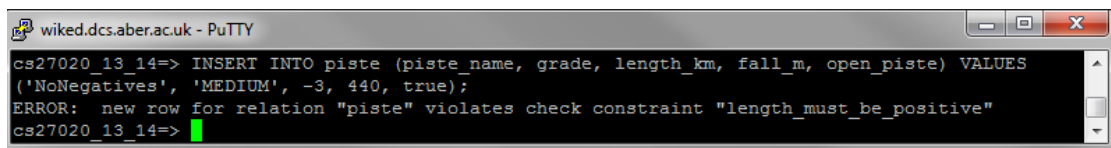
```
wiked.dcs.aber.ac.uk - PuTTY
cs27020_13_14=> INSERT INTO piste (piste_name, grade, length_km, fall_m, open_piste) VALUES
('ENUMExists', 'SUPERDIFFICULT', 3, 440, true);
ERROR:  invalid input value for enum grade_rank: "SUPERDIFFICULT"
LINE 2: ('ENUMExists', 'SUPERDIFFICULT', 3, 440, true);
                        ^
cs27020_13_14=>
```

Figure 7: An entered Grade should conform to an existing ENUM value. 'SUPERDIFFICULT' is not an existing ENUM value in this database.



```
wiked.dcs.aber.ac.uk - PuTTY
cs27020_13_14=> INSERT INTO piste (piste_name, grade, length_km, fall_m, open_piste) VALUES
('NullValue', 'EASY', 3, NULL, true);
ERROR:  null value in column "fall_m" violates not-null constraint
cs27020_13_14=>
cs27020_13_14=>
cs27020_13_14=>
```

Figure 8: If a value is attempted to be entered as NULL, it will be rejected, as values are specific and so should not be recorded into the database as such. This violates the constraint and would potentially cause issues in the databases lifecycle, such as causing errors in queries that might attempt to assess NULL data.



```
wiked.dcs.aber.ac.uk - PuTTY
cs27020_13_14=> INSERT INTO piste (piste_name, grade, length_km, fall_m, open_piste) VALUES
('NoNegatives', 'MEDIUM', -3, 440, true);
ERROR:  new row for relation "piste" violates check constraint "length_must_be_positive"
cs27020_13_14=>
```

Figure 9: If a numeric value is attempted to be entered as negative, the operation is cancelled, the user is informed of their mistake and through the use of CONSTRAINT, is told where their mistake is and why. When looking at the sample data, there are no negative numbers. Know what information the data provides shows that there should never be negative values.

4.4 Test Queries

There were 4 queries provided that needed to be successfully carried out in order to test the structure of my database:

- Return the piste(s) served by a given lift
- Return the lift(s) that provide access to a given piste
- Return the lifts that are currently operating
- Return the pistes that are currently open, together with the lifts that are currently operating and that provide access to those pistes

It should be noted that 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.

For queries where it requires multiple queries from multiple tables, I have opted to use 'INNER JOIN', to connect my relations together in order to access the data. I felt that by explicitly stating that I wished to use INNER JOIN, as opposed to having an implied JOIN, was much more robust.

Using the INNER JOIN should improve performance of the database over an implied JOIN, which would be an important factor were the database to be very large. I also feel that using INNER JOIN makes it easier to read as a human than if the queries were comprised of multiple SELECT statements or large lists of data selected from a WHERE clause.

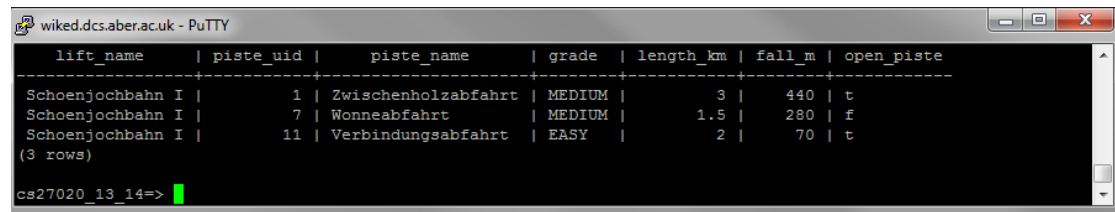
4.4.1 Test Query 1

"Return the piste(s) served by a given lift."

— *SELECT the Pistes serviced by a particular Lift.*

```
SELECT lifts.lift_name , pistes.*
FROM piste pistes
INNER JOIN connections conns
ON pistes.piste_uid=conns.piste_uid
INNER JOIN lift lifts
ON lifts.lift_uid=conns.lift_uid
AND lifts.lift_name='<lift_name>';
```

This query selects all the data concerning the piste(s) that are served by a given lift. The query gets all of the Piste uids from Connections relation which are linked to the Lifts who have uids matching the same uid for the Lift in the Lift relation that share the given lift_name.



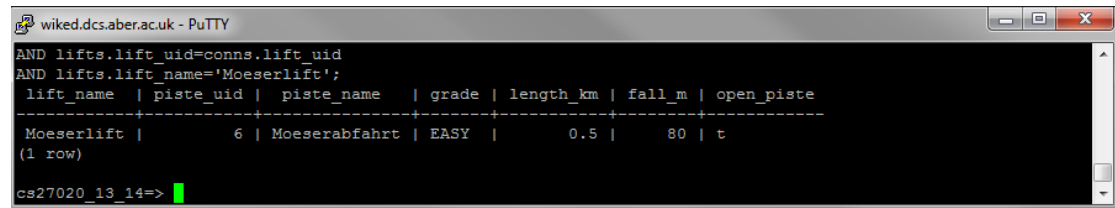
```
wiked.dcs.aber.ac.uk - PuTTY
```

lift_name	piste_uid	piste_name	grade	length_km	fall_m	open_piste
Schoenjochbahn I	1	Zwischenholzabfahrt	MEDIUM	3	440	t
Schoenjochbahn I	7	Wonneabfahrt	MEDIUM	1.5	280	f
Schoenjochbahn I	11	Verbindungsabfahrt	EASY	2	70	t

(3 rows)

```
cs27020_13_14=>
```

Figure 10: The result of looking for the Pistes serviced by the Lift: 'Schoenjochbahn I'.



```
wiked.dcs.aber.ac.uk - PuTTY
```

```
AND lifts.lift_uid=conns.lift_uid
AND lifts.lift_name='Moeserlift';
```

lift_name	piste_uid	piste_name	grade	length_km	fall_m	open_piste
Moeserlift	6	Moeserabfahrt	EASY	0.5	80	t

(1 row)

```
cs27020_13_14=>
```

Figure 11: The result of looking for the Pistes serviced by the Lift: 'Moeserlift'.

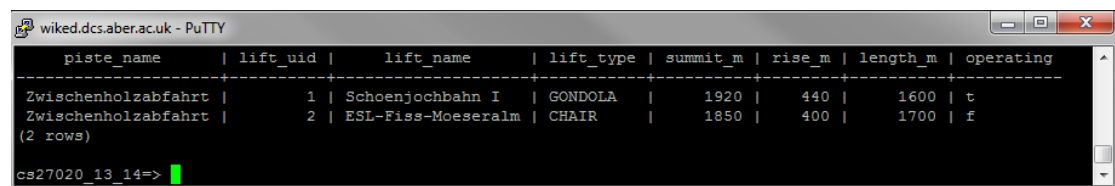
4.4.2 Test Query 2

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

— *SELECT the Lifts that provide access to a given Piste.*

```
SELECT pistes.piste_name, lifts.*
FROM lift lifts
      INNER JOIN connections conns
      ON lifts.lift_uid=conns.lift_uid
      INNER JOIN piste pistes
      ON pistes.piste_uid=conns.piste_uid
      AND pistes.piste_name='<piste_name>';
```

This query is the same as the first query, but looking for Lifts providing access to a particular Piste. The logic for the query is identical for the first and so should not need to be explained again here.



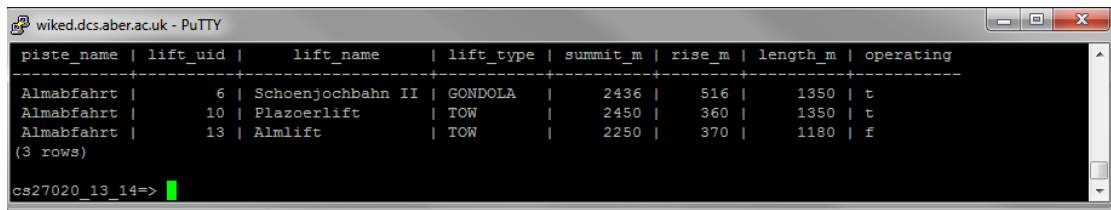
```
wiked.dcs.aber.ac.uk - PuTTY
```

piste_name	lift_uid	lift_name	lift_type	summit_m	rise_m	length_m	operating
Zwischenholzabfahrt	1	Schoenjochbahn I	GONDOLA	1920	440	1600	t
Zwischenholzabfahrt	2	ESL-Fiss-Moeseralp	CHAIR	1850	400	1700	f

(2 rows)

```
cs27020_13_14=>
```

Figure 12: The result of looking for the Lifts servicing the Piste: 'Zwischenholzabfahrt'.



```
wiked.dcs.aber.ac.uk - PuTTY
```

piste_name	lift_uid	lift_name	lift_type	summit_m	rise_m	length_m	operating
Almabfahrt	6	Schoenjochbahn II	GONDOLA	2436	516	1350	t
Almabfahrt	10	Plazzerlift	TOW	2450	360	1350	t
Almabfahrt	13	Almlift	TOW	2250	370	1180	f

(3 rows)

```
cs27020_13_14=>
```

Figure 13: The result of looking for the Lifts servicing the Piste: 'Almabfahrt'.

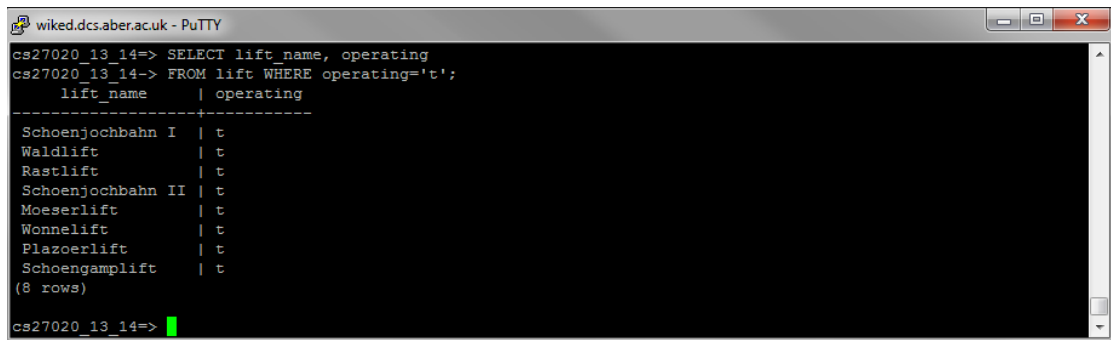
4.4.3 Test Query 3

"Return the lifts that are currently operating."

— *SELECT the Lifts that are currently operating.*

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

Look for the Lifts that are currently operating. In this case, look for those that are marked 'TRUE', or 't', in the operating attribute. I chose to only display the lift_name and it's operational status for this query. However, by exchanging the two requested attributes for an '*', 'SELECT * FROM...', it would return all the data for each of the operating lifts. I also understood the requested query as asking for all Lifts operating, regardless of whether the Pistes they service are open or not.



```
wiked.dcs.aber.ac.uk - PuTTY
```

```
cs27020_13_14=> SELECT lift_name, operating
cs27020_13_14-> FROM lift WHERE operating='t';
```

lift_name	operating
Schoenjochbahn I	t
Waldlift	t
Rastlift	t
Schoenjochbahn II	t
Moeserlift	t
Wonnelt	t
Plazzerlift	t
Schoengamplift	t

(8 rows)

```
cs27020_13_14=>
```

Figure 14: The Lifts that are currently operating.

4.4.4 Test Query 4

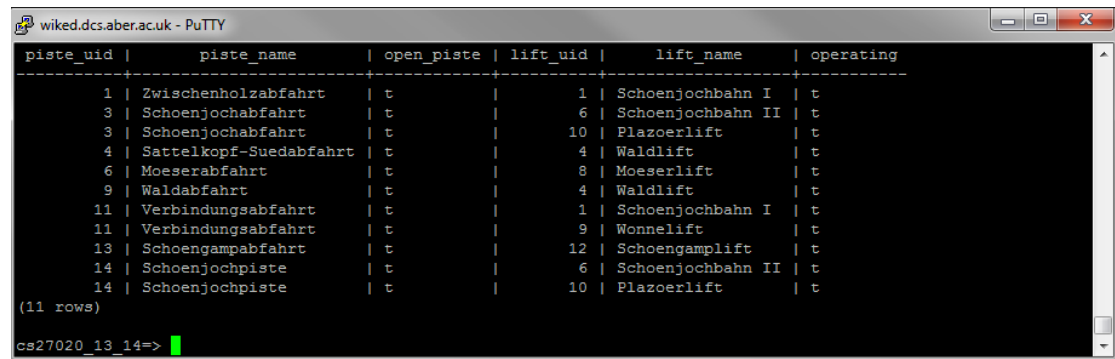
”Return the pistes that are currently open, together with the lifts that are currently operating and that provide access to those pistes.”

— *SELECT the Pistes that are currently open, together with the lifts that are currently operating and that provide access to those pistes.*

```
SELECT pistes.piste_uid , pistes.piste_name , pistes.open_piste ,
lifts.lift_uid , lifts.lift_name , lifts.operating
FROM
    connections conns
    INNER JOIN piste pistes
ON pistes.open_piste='t'
    INNER JOIN lift lifts
ON lifts.operating='t'
    AND pistes.piste_uid=conns.piste_uid
    AND lifts.lift_uid=conns.lift_uid ;
```

The final query requests to return all of those Pistes that are currently open. With those Pistes, those listed should also display the Lifts that service them, but only those Lifts that are currently operating. To do this, the query looks for those Pistes that are open, then looks for the Lifts that are also operating, and returns those Lifts that are servicing the open Pistes.

I chose to only return the UIDs, Name and open/operating status of the Pistes and Lifts for the sake of space. As with the previous query, it could easily be modified to provide all data associated with the relevant Lifts and Pistes.



piste_uid	piste_name	open_piste	lift_uid	lift_name	operating
1	Zwischenholzabfahrt	t	1	Schoenjochbahn I	t
3	Schoenjochabfahrt	t	6	Schoenjochbahn II	t
3	Schoenjochabfahrt	t	10	Plaoerlift	t
4	Sattelkopf-Suedabfahrt	t	4	Waldlift	t
6	Mooserabfahrt	t	8	Moeserlift	t
9	Waldabfahrt	t	4	Waldlift	t
11	Verbindungsabfahrt	t	1	Schoenjochbahn I	t
11	Verbindungsabfahrt	t	9	Wonnelifft	t
13	Schoengampabfahrt	t	12	Schoengamplift	t
14	Schoenjochpiste	t	6	Schoenjochbahn II	t
14	Schoenjochpiste	t	10	Plaoerlift	t

(11 rows)

cs27020_13_14=>

Figure 15: The Pistes that are open, with the corresponding connecting Lifts that are currently operating.

5 Conclusion

With my completed Database, evidence of testing with the sample data, checking for errors and working queries, I feel I have fulfilled the task set. My database correctly takes in the desired data, refusing any data that might violate the conditions or cause future errors, and stores the data in a way that is in 3NF. I have followed the Normalization process to reach 3NF, based on my choices of the Primary/Candidate keys and the functional dependencies. The end result is a fully functional database that could be used for storing data on connected Pistes and Lifts, and queried for the data inside with relative ease.

References

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