CS27020

Assignment: Ski Lifts and Pistes

Introduction

The aim of this assignment is to design and implement a back-end relational database to support this. The overall goal of my input into this assignment is for me to produce a normalised database. This means that there is a separation of concerns, which often eliminates redundancy in the relations.

The procedure to allow me to end with a normalised set of data for my database, is to start with defining the source data as an un-normalised. A functional dependency shows a relationship that exists when one attribute uniquely determines another attribute. In a database, it serves as a constraint between two sets of attributes. These functional dependencies will also be determined at this stage of the process. The next step of the process was to use a technique called bottom up analysis, to first result in First Normal Form – where the un-normalised structure is brought to this by transferring repeating groups of items to separate relation schemes. The next intention of the process is to result in Second Normal Form, by eliminating partial dependencies on keys. Finally, the relation schemes are brought to Third Normal Form by eliminating transitive dependencies on keys.

The normalised set of data is then to be used to implement a PostgreSQL database with normalised relations and appropriate primary and foreign key constraints. This database implementation will also accompany a number of SQL queries to show functionality of the data, and be shown as a table of data on my University filestore webpages.

Un-Normalised Structures

The first step of the process to develop a set of normalised data is to represent the source as an un-normalised structure. This was done by identifying all the fields and defining an initial relation for the normalisation process.

PISTE (Piste, Grade, Length, Fall, Lifts, Open?)

LIFTS (Lift, Type, Summit, Rise, Length, Operating)

PISTE

Piste Name **(PK)**
Grade
Length (km)
Fall (m)
Open?

LIFTS

Lift Name **(PK)**
Type
Summit (m)
Rise (m)
Length (m)
Operating

The choice of choosing Piste as the Primary Key for my table Piste, is due to the fact of observing the sample data and deducing that every Piste is unique and therefore makes perfect sense to be used as a Primary Key.

The choice of choosing the Primary Key in the Lifts table is much the same, as the Lift allows me to uniquely identify each record – and therefore is the ideal option for usage as a Primary Key in this un-normalised structure.

<u>Functional Dependencies</u>

The functional dependencies help to provide the basis for normalisation. These functional dependencies reflect the concepts which must not be confused within single relations, and allows the process of normalisation to proceed.

Piste Name → Grade, Length, Fall, Open?

Lift Name → Type, Summit, Rise, Length, Operating

The attributes above are all dependant on their relevant table (Piste Name or Lift Name) – as without that dependency, the attributes alone will mean nothing. This gives a clear relation between the values and defines the functional dependencies of the data set.

<u>First Normal Form</u>

The objective of this stage, is to bring the relation to First Normal Form – by transferring the repeating groups of items to separate relation schemes. Each of the new schemes includes the key of the original un-normalised structure as part of its own key. In the case of any repeating groups, they must be split off into another relation and there must be a key to link them.

PISTE (<u>PisteName</u>, Grade, Length, Fall, Lifts*, Open?)

The Foreign Key (FK) of **Lifts** references the LIFTS table.

LIFTS (<u>LiftName</u>, Type, Summit, Rise, Length, Operating)

In order to move the relations into First Normal Form, I have transferred the repeating groups to separate relation schemes. Both of the new schemes now include the primary keys of the un-normalised structure. There is a repeating group of Lifts in the PISTE table, and this has been split off into another relation – creating a foreign key link between the two.

<u>Second Normal Form</u>

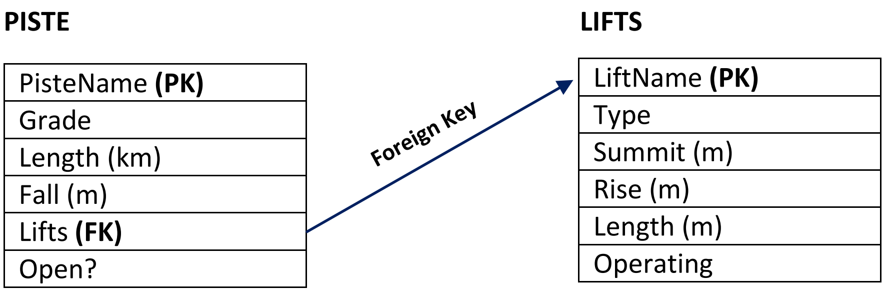
The aim of the this normalisation stage, is to not have a dependency based on part of the primary key – so it split off the attributes determined by part of the key and that part of the key. The plan is to have all of the relations have single attribute primary keys so we have removed all partial dependencies.

PISTE (<u>PisteName</u>, Grade, Length, Fall, Lifts*, Open?)

The Foreign Key (FK) of **Lifts** references the LIFTS table.

LIFTS (<u>LiftName</u>, Type, Summit, Rise, Length, Operating)

There is not much change between First and Second Normal Form in this particular assignment, however it has one subtle change – where in the Second Normal Form, it now has the ability to have more than one lift value in an individual record. This is not the case in First Normal Form, where multiple values of the same field would be a breach of the rules.



Third Normal Form

In order to complete the normalisation process, we need to remove all of the transitive dependencies – this splits off the non-key determinant and the attribute determined, leaving the determinant as a foreign key. This step completes normalising the set of sample data.

After much deliberation and observations, it has been concluded that there are no transitive dependencies to complete the process of normalisation and therefore Second Normal Form and Third Normal Form are the same.

Implementation of Database

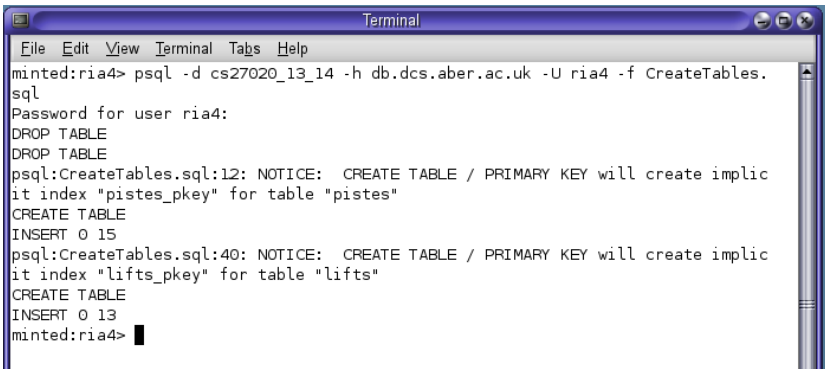
Once formed a fully normalised set of data, the implementation of the database can progress – this begins with the setup of the PostgreSQL database by using wiked or minted operating systems to first log in with:

psql –U ria4 –d cs27020_13_14 –h db.dcs.aber.ac.uk

The –f option is then used to run an SQL script to create and initialize the database creation, this script is my own file of CreateTables.sql – this contains the relevant INSERT INTO SQL commands, to first create the tables and then populate the tables with the sample data given in the assignment brief. The command used in order to call this SQL script is as follows:

psql –d cs27020_13_14 –h db.dcs.aber.ac.uk –U ria4 –f CreateTables.sql

When input into the minted terminal, the following result will be shown:



```
minted:ria4> psql -d cs27020_13_14 -h db.dcs.aber.ac.uk -U ria4 -f CreateTables.
sql
Password for user ria4:
DROP TABLE
DROP TABLE
psql:CreateTables.sql:12: NOTICE:  CREATE TABLE / PRIMARY KEY will create implic
it index "pistes_pkey" for table "pistes"
CREATE TABLE
INSERT 0 15
psql:CreateTables.sql:40: NOTICE:  CREATE TABLE / PRIMARY KEY will create implic
it index "lifts_pkey" for table "lifts"
CREATE TABLE
INSERT 0 13
minted:ria4> 
```

The CREATE TABLE statements that I have used within my CreateTables.sql are shown below along with the appropriate primary and foreign key constraints to implement the database relations.

```
        CREATE TABLE pistes
```

```
(
	piste varchar(30) primary key,
	grade varchar(9),
	length varchar(3),
	fall varchar(4),
	lifts text,
	open varchar(3)
);
```

The choice behind using 'varchar' as an option as the datatype for the piste field, is because this it is the name of the pistes that are stored within the database. From my observations, there are no piste fields that are above 30 characters and therefore there is no need to declare it as more than 30. This was also similar for the datatype option for 'grade' – where the longest possible character value was 'difficult' and therefore of 9 characters in length (hence varchar(9)). These declaration thoughts are largely the same for length, which is set in KM. This could have been set as a decimal datatype, or smallint – however for ease of implementation, I have used the varchar to ensure correct function. The fall datatype has been put in as 'varchar(4)' to allow for 4 characters in length, to ensure that the relevant data can be input correctly without error. The lifts field has been implemented as a 'text' datatype, due to the unpredictability of the information being used in that field. A possible choice for this, could have been varchar – however, I felt more confident using 'text' for this due to no real requirement for memory storage. The choice of varchar for the 'open' datatype was a cause of concern – I thought at first for the use of boolean, however the input obviously have had to be True or False and this was not what was required. Therefore, I have used varchar(3) to allow for 3 characters for 'yes' – the longest possible value of the field.

```
CREATE TABLE lifts
(
	lift varchar(30) primary key,
	type varchar(7),
	summit int,
	rise int,
	length int,
	operating varchar(3)
);
```

The choices are largely similar to those of the datatypes in the pistes – I have once again used a varchar(30) for the lift field, which is essentially the Lift Name. Therefore, from my observations I have found that there is once again no lift name that exceeds the 30 character limit and for these reasons, the varchar(30) is sufficient as a datatype. The varchar(7) choice for the type field, once again falls under the category of character limit – and once again there is no risk of it being exceeded due to my careful input of the database content. The following three fields of summit, rise and length have all been categorised as 'int' datatypes. These all contain integer values and therefore is no need for a float, double or other numerical datatype. The operating field was similar to the choice for the open field in the last table, as all is needed is a character value of 3 – and therefore not much deliberation is needed.

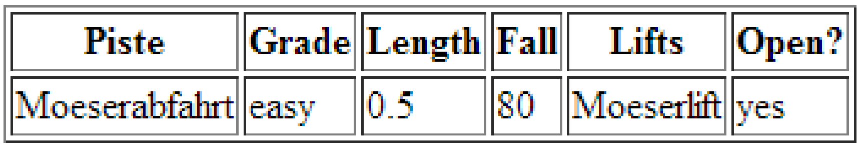
Following the correct implementation of the database, via the use of the CreateTables.sql file – the relevant SQL statement files could be created to query the statements mentioned in the assignment brief.

- The pistes served by a given lift;

The SQL statement that I created in order to query this particular statement is as follows:

```
$res = pg_query($conn, "SELECT * FROM pistes WHERE lifts='Moeserlift'");
```

This query uses a simple SELECT statement and makes use of the * option to then choose all of the fields of the database from the table called ‘pistes’. The test data input of ‘Moeserlift’ has been used for testing purposes. The expected output of this statement, is to produce all of the details (Piste Name, Grade, Length, Fall, Lifts and Open?) in a table once queried. The graphical output of this SQL statement is shown on my University filestore, via use of PHP to create a table to output the queried data.



Piste	Grade	Length	Fall	Lifts	Open?
Moeserabfahrt	easy	0.5	80	Moeserlift	yes

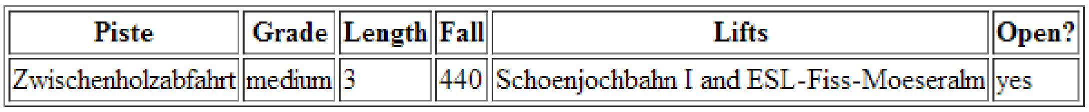
Figure 1 - (http://users.aber.ac.uk/ria4/cs27020/Statement1.php).

- The lift(s) that provide access to a given piste;

The SQL statement that I created in order to query this particular statement is as follows:

```
$res = pg_query($conn, "SELECT * FROM pistes WHERE piste='Zwischenholzabfahrt'");
```

This query once again uses the simple SELECT * to display all of the table fields from ‘pistes’ – despite the brief only asking for the ‘lift(s)’ of a given piste. The output looks much better given a SELECT * statement as opposed to a SELECT lifts, where only one field is populated. The relevant information is still provided with my statement correctly. The test data input of ‘Zwischenholzabfahrt’ is used in this example, and the expected outcome of this test is to produce the following values of ‘Lifts’ as ‘Schoenjochbahn I and ESL-Fiss-Moeseralm’. The graphical output of this SQL statement is shown on my University filestore, via use of PHP to create a table to output the queried data.



Piste	Grade	Length	Fall	Lifts	Open?
Zwischenholzabfahrt	medium	3	440	Schoenjochbahn I and ESL-Fiss-Moeseralm	yes

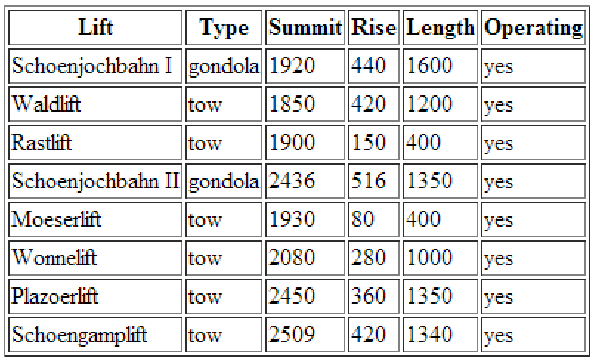
Figure 2 - (http://users.aber.ac.uk/ria4/cs27020/Statement2.php).

- The lifts that are currently operating;

The SQL statement that I created in order to query this particular statement is as follows:

```
$res = pg_query($conn, "SELECT * FROM lifts WHERE operating='yes'");
```

This query uses the SELECT * to provide the list of lifts from the database table ‘Lifts’ – where the operating value is equal to ‘yes’. There is no need for test data in this example, and the expected outcome should be for all of the lifts that are in operation (operating=’yes’) – to be output along with their corresponding fields to the webpage. The graphical output of this SQL statement is shown on my University filestore, via use of PHP to create a table to output the queried data.



Lift	Type	Summit	Rise	Length	Operating
Schoenjochbahn I	gondola	1920	440	1600	yes
Waldlift	tow	1850	420	1200	yes
Rastlift	tow	1900	150	400	yes
Schoenjochbahn II	gondola	2436	516	1350	yes
Moeserlift	tow	1930	80	400	yes
Wonnelift	tow	2080	280	1000	yes
Plazoerlift	tow	2450	360	1350	yes
Schoengamplift	tow	2509	420	1340	yes

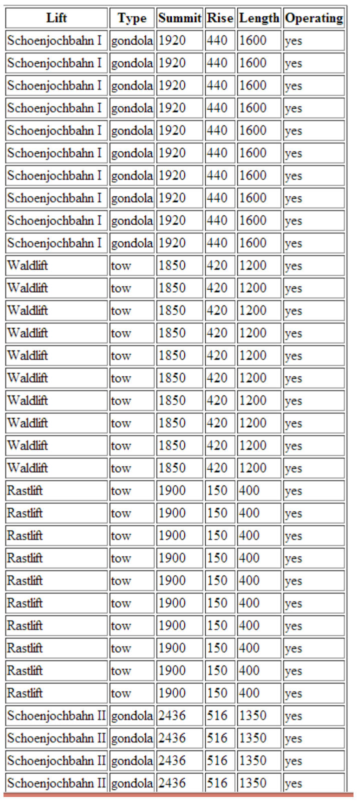
Figure 3 - (http://users.aber.ac.uk/ria4/cs27020/Statement3.php).

- The pistes that are currently open, together with the lifts that are currently operating and that provide access to those pistes;

The SQL statement that I created in order to query this particular statement is as follows:

```
$res = pg_query($conn, "SELECT * FROM pistes,lifts WHERE open='yes' AND operating='yes'");
```

This query makes use of the SELECT * statement to find values when the relevant open and operating fields of the tables pistes and lifts respectively to produce an output of pistes where they are currently open. There is no inputted test data that has been used in this example, and the graphical output of this SQL statement is shown on my University filestore, via use of PHP to create a table to output the queried data. The expected outcome, is to shown a list of pistes/lifts including their corresponding table values for the values that are operating and open. As you can see from Figure 4 below, the SQL statement produces multiples of the fields – however it does produce the correct fields. However, I didn't have the time to fix the duplication issue at this point.



Lift	Type	Summit	Rise	Length	Operating
Schoenjochbahn I	gondola	1920	440	1600	yes
Schoenjochbahn I	gondola	1920	440	1600	yes
Schoenjochbahn I	gondola	1920	440	1600	yes
Schoenjochbahn I	gondola	1920	440	1600	yes
Schoenjochbahn I	gondola	1920	440	1600	yes
Schoenjochbahn I	gondola	1920	440	1600	yes
Schoenjochbahn I	gondola	1920	440	1600	yes
Schoenjochbahn I	gondola	1920	440	1600	yes
Schoenjochbahn I	gondola	1920	440	1600	yes
Schoenjochbahn I	gondola	1920	440	1600	yes
Waldlift	tow	1850	420	1200	yes
Waldlift	tow	1850	420	1200	yes
Waldlift	tow	1850	420	1200	yes
Waldlift	tow	1850	420	1200	yes
Waldlift	tow	1850	420	1200	yes
Waldlift	tow	1850	420	1200	yes
Waldlift	tow	1850	420	1200	yes
Waldlift	tow	1850	420	1200	yes
Waldlift	tow	1850	420	1200	yes
Waldlift	tow	1850	420	1200	yes
Rastlift	tow	1900	150	400	yes
Rastlift	tow	1900	150	400	yes
Rastlift	tow	1900	150	400	yes
Rastlift	tow	1900	150	400	yes
Rastlift	tow	1900	150	400	yes
Rastlift	tow	1900	150	400	yes
Rastlift	tow	1900	150	400	yes
Rastlift	tow	1900	150	400	yes
Rastlift	tow	1900	150	400	yes
Rastlift	tow	1900	150	400	yes
Schoenjochbahn II	gondola	2436	516	1350	yes
Schoenjochbahn II	gondola	2436	516	1350	yes
Schoenjochbahn II	gondola	2436	516	1350	yes
Schoenjochbahn II	gondola	2436	516	1350	yes

Figure 4 - http://users.aber.ac.uk/ria4/cs27020/Statement4.php).