Database Coursework - COMP1204

Jack Corbett - jc11g17 - 29244102

May 2018

A report discussing the use of a database to store reviews for Trip Advisor. Including modelling the relationships, normalising, creating the database and querying.

1 ERD and Normalisation

1.1 EX1

First we must identify the relation we are trying to model, we will call this R1. It contains all of the information to link hotels to reviews. For demonstration purposes I have included a column of example data. The primary key had to include content as well as the hotelID and author of the review to cater for the anonymous reviewers in the data set. In practice it isn't sensible to include content as it will be difficult to query given it's text format and it will be hard to enforce it's restriction; however, it is the best we can do as we cannot use date instead, as there is an occurrence in our data set where two anonymous authors write a review for the same hotel on the same day, which is a likely occurrence.

Key	Attribute	Data Type	Example Data
PK	hotelID	int	72572
	hotelOverallRating	tinyint	*0 to 5*
	hotelAveragePrice	decimal	$173
	hotelURL	varchar	http://www.tripadvisor.com...
PK	author	varchar	Marilyn1949
PK	content	text	Great location for...
	dateWritten	date	Dec 1, 2008
	noReader	int	*-1 ...*
	noHelpful	int	*-1 ...*
	overallRating	tinyint	*-1 to 5*
	value	tinyint	*-1 to 5*
	rooms	tinyint	*-1 to 5*
	location	tinyint	*-1 to 5*
	cleanliness	tinyint	*-1 to 5*
	checkIn_frontDesk	tinyint	*-1 to 5*
	service	tinyint	*-1 to 5*
	businessService	tinyint	*-1 to 5*

1.2 EX2

Now we know the data we want to store, it is important to consider the functional dependencies:

- hotelID→hotelURL, hotelOverallRating, hotelAveragePrice
- hotelID, author, content→hotelURL, hotelOverallRating, hotelAveragePrice, date, noReader, noHelpful, overall, value, rooms, location, cleanliness, checkIn_frontDesk, service, businessService

We only have one candidate key for this relation as hotelURL is null for some hotels, so it cannot be used as part of a candidate key and, as we discussed above, we cannot use date with hotelID and author to uniquely identify tuples:

- hotelID, author, content

1.3 EX3

We can now normalise our tables to avoid data redundancy.

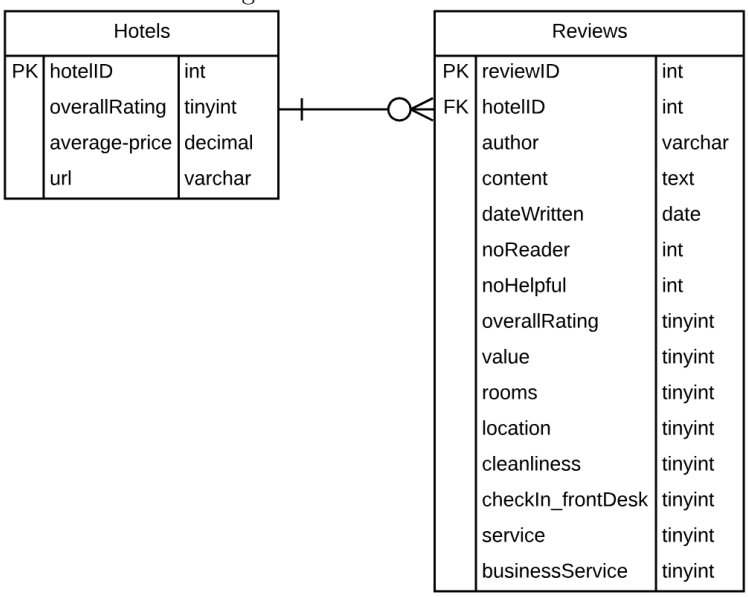
- 1NF: The relation is already in first normal as it contains no repeating columns.

- 2NF: Not all attributes of the relation are dependent on the whole key as the hotel data is not reliant on the author and content of the review. To remedy this I split the relation into: Hotels and Reviews.
- 3NF: In the new schema there were no transitive dependencies so no changes were required.
- BCNF: If hotelURL was a candidate key it would need to be removed from the hotels relation as it would overlap with hotelID. However as it is null for some hotels no changes are required.

Finally, I added a surrogate key to the reviews table: reviewID. This avoided using a composite key of author, content and hotelID which will make it much simpler when writing queries. It also avoids imposing restrictions on the content of reviews to keep the primary key unique.

1.4 EX4

Figure 1: Normalised ERD



2 Relational Algebra

2.1 EX5

We can use selection to find all the reviews by the same user by querying the reviews table. In this case we don't need to use the projection operator as we want all columns of each review:

$$\sigma_{author=\$author}(Reviews)$$

2.2 EX6

To find all the users who have written more than two reviews and get their name and the number of hotels they have reviewed we can make use of the aggregate function count(). This time we do project to get just the author and count:

$$\pi_{author,reviewCount}(\sigma_{reviewCount>2}({}_{author}\gamma_{count(reviewID)->reviewCount}(Reviews)))$$

2.3 EX7

To find all the hotels with more than 10 reviews we can again use count() on the Reviews relation, this time projecting only hotelID:

$$\pi_{hotelID}(\sigma_{reviewCount>10}(_{hotelID}\gamma_{count(reviewID)->reviewCount}(Reviews)))$$

2.4 EX8

To find all the hotels with an overall rating greater than 3 and an average cleanliness greater than or equal to 4.5 it required joining the Reviews and Hotels tables. The avg() aggregate function was then used to calculate the average cleanliness for each hotel:

$$\pi_{hotelID}(\sigma_{(Hotels.overallRating>3 \wedge averageCleanliness>=4.5)}(_{hotelID}\gamma_{avg(Reviews.cleanliness)->averageCleanliness}(Reviews \bowtie Hotels)))$$

3 SQL

3.1 EX9

First I created the HotelReviews table to hold all of the unnormalised data from the data files. This follows the structure of R1 (EX1).

```
CREATE TABLE HotelReviews
( hotelID             integer,
  hotelOverallRating  tinyint,
  hotelAveragePrice   integer,
  hotelURL            varchar(1000),
  author              varchar(20),
  content             text,
  dateWritten         date,
  noReader            integer,
  noHelpful           integer,
  overallRating       tinyint,
  value               tinyint,
  rooms               tinyint,
  location            tinyint,
  cleanliness         tinyint,
  checkIn_frontDesk   tinyint,
  service             tinyint,
  businessService     tinyint,
  PRIMARY KEY (hotelID, author, content));
```

3.2 EX10

The bash script to add the data to an sql file is found in *Appendix A*.

3.3 EX11

Now I have the data in the main table I created my normalised tables from EX4.

```
CREATE TABLE Reviews
( reviewID          integer,
  hotelID           integer,
  author            varchar(20),
  content           text,
  dateWritten       date,
  noReader          integer,
  noHelpful         integer,
  overallRating     tinyint,
  value             tinyint,
  rooms             tinyint,
  location          tinyint,
  cleanliness       tinyint,
  checkIn_frontDesk tinyint,
  service           tinyint,
  businessService   tinyint,
  PRIMARY KEY (reviewID AUTOINCREMENT),
  CONSTRAINT Reviews_Hotels_hotelID_fk
  FOREIGN KEY (hotelID) REFERENCES Hotels (hotelID));

CREATE TABLE Hotels
( hotelID       integer,
  overallRating tinyint,
  averagePrice  decimal,
  url           varchar(1000),
  PRIMARY KEY (hotelID));
```

3.4 EX12

Next I moved the data from HotelReviews into the newly created tables using:

```
INSERT INTO Hotels
SELECT DISTINCT hotelID, hotelOverallRating, hotelAveragePrice, hotelURL
FROM HotelReviews;

INSERT INTO Reviews (hotelID, author, content, dateWritten, noReader, noHelpful,
    overallRating, value, rooms, location, cleanliness, checkIn_frontDesk, service,
    businessService)
SELECT hotelID, author, content, dateWritten, noReader, noHelpful, overallRating, value,
    rooms, location, cleanliness, checkIn_frontDesk, service, businessService
FROM HotelReviews;
```

3.5 EX13

I added indexes on author and hotelID in the Reviews table and overallRating in the Hotels table as these will be most commonly used to fetch data. For example if you want to fetch: reviews by a certain author, the reviews for a certain hotel or the hotels with a certain overall rating. These queries are likely to be used to order hotel results when users search.

```
CREATE INDEX author_index ON Reviews(author);
CREATE INDEX hotelID_index ON Reviews(hotelID);
CREATE INDEX overallRating_index ON Hotels(overallRating);
```

3.6 EX14

Now the data has been added to the normalised tables we can write the SQL versions of the queries we modelled using relational algebra in EX5-EX8.

EX5: The first was a simple select statement using * to specify that we want every column. For the sake of this example I used the anonymous TripAdvisor Member author as it returned the most results.

```
SELECT * FROM Reviews WHERE author = 'A TripAdvisor Member';
```

EX6: For the second query I used the count aggregate function, as in the relational algebra query, and added an order by clause to display the authors with the greatest number of reviews at the top of the results to make the output more useful.

```
SELECT author, COUNT(reviewID) AS reviewCount FROM Reviews GROUP BY author HAVING
    reviewCount > 2 ORDER BY reviewCount DESC;
```

EX7: The third query used a very similar struacture this time not including the count in the results.

```
SELECT hotelID FROM Reviews GROUP BY hotelID HAVING COUNT(hotelID) > 10;
```

EX8: For the final query I performed an inner join on the Reviews and Hotels tables to allow me to include comparisons on data from both tables in my having clause. I also used shorthand aliasing to avoid having to write out the table names in full each time.

```
SELECT h.hotelID FROM Hotels h
INNER JOIN Reviews r on h.hotelID = r.hotelID GROUP BY h.hotelID
HAVING h.overallRating > 3 AND AVG(r.cleanliness) >= 4.5;
```

3.7 EX15

It wasn't only the 'A TripAdvisor Member' reviewers that caused problems. There were also 12,517 reviews by author 'lass=' which appears to be a parsing error when the data has been scraped, where part of the HTML class tag has been accepted by mistake. Other names also caused issues including one that contained a speech mark, as this terminated the string within the insert statement early, preventing the file from being read without error by sqlite.

Another problem was that the full review text was often not included, instead the first few words followed by a showReview() call. This is again I expect a parsing error where it has been scraped from the source code of the website, as it is likely a PHP method call. However, this does mean it would be impossible to fetch the entire review content by web scraping alone.

In order to calculate average scores correctly I also had to replace -1 with NULL to ensure it was ignored by the avg() function as the negative values understandably skewed the results. Finally, the average price value caused an issue in data file 93230 as 'Unkonwn' (incorrect spelling) was stored instead of a $ followed by the value. This lead to me having to include a special case in my script to remove it.

4 Conclusion

The database schema I created makes it much easier to access the hotel review data and decreases data redundancy, making the system more maintainable. Although there were some problems with the data files as discussed above, these were only edge cases and the database produced, I believe, modelled the data effectively.

A

EX10 - Bash Script

```
#!/bin/bash

#Set the internal field seperator to the new lines and return special characters. This prevents data entries with spaces being
    split over multiple lines when using grep.
IFS=$'\n\r';

{
  #First drop the current table from the database and then recreate the table
  echo "DROP TABLE IF EXISTS 'HotelReviews';"
  echo "CREATE TABLE 'HotelReviews' ('hotelID' integer, 'hotelOverallRating' tinyint, 'hotelAveragePrice' integer, 'hotelURL'
    varchar(1000), 'author' varchar(20), 'content' text, 'dateWritten' date, 'noReader' integer, 'noHelpful' integer, '
    overallRating' tinyint, 'value' tinyint, 'rooms' tinyint, 'location' tinyint, 'cleanliness' tinyint, 'checkIn_frontDesk'
    tinyint, 'service' tinyint, 'businessService' tinyint, PRIMARY KEY('hotelID', 'author', 'content'));"
} > hotelreviews.sql;

#Loop through each hotel data file
for file in $1/*.dat;
do
  #First store the id, overallRating, averagePrice and URL as these will be the same for that file. Trimming off new line
    characters.
  hotelID=$(echo "$file" | sed "s/.*hotel_//; s/.dat//");
  hotelOverallRating=$(grep "<Overall Rating>" $file | sed "s/<Overall Rating>//" | tr -d '\n\r');
  #For the average price we also have to trim Unkonwn as this is found in data file 93230
  hotelAveragePrice=$(grep "<Avg. Price>" $file | sed "s/<Avg. Price>//" | tr -d '$\n\rUnkonwn,');
  #Use regex to add speech marks if the URL tag is found, this ensures nulls are submitted correctly
  hotelURL=$(grep "<URL>" $file | sed "s/<URL>\(.*\)$/'\1'/" | tr -d '\n\r');

  #There are multiple entries for the following variables per data file so grep is used to add each to an array.
  #This makes it easy to iterate through when creating our insert array
  author=($(grep "<Author>" $file | sed "s/<Author>//" | tr -d \'\"));
  content=($(grep "<Content>" $file | sed "s/<Content>//" | tr -d \'\"));
  #The date command reformats the date to abide by SQLite formatting
  date=($(grep "<Date>" $file | sed "s/<Date>//" | date +"%Y-%m-%d" -f -));
  noReader=($(grep "<No. Reader>" $file | sed "s/<No. Reader>//; s/-1/NULL/"));
  noHelpful=($(grep "<No. Helpful>" $file | sed "s/<No. Helpful>//; s/-1/NULL/"));
  overall=($(grep "<Overall>" $file | sed "s/<Overall>//; s/-1/NULL/"));
  value=($(grep "<Value>" $file | sed "s/<Value>//; s/-1/NULL/"));
  rooms=($(grep "<Rooms>" $file | sed "s/<Rooms>//; s/-1/NULL/"));
  location=($(grep "<Location>" $file | sed "s/<Location>//; s/-1/NULL/"));
  cleanliness=($(grep "<Cleanliness>" $file | sed "s/<Cleanliness>//; s/-1/NULL/"));
  checkInFrontDesk=($(grep "<Check in / front desk>" $file | sed "s/<Check in \/ front desk>//; s/-1/NULL/"));
  service=($(grep "<Service>" $file | sed "s/<Service>//; s/-1/NULL/"));
  businessService=($(grep "<Business service>" $file | sed "s/<Business service>//; s/-1/NULL/"));
```

```
  #Now all the data has been read from the hotel.dat file we can count up to the size of the author array (representing the
    number of reviews)
  #Echoing each value and using the default operator to write NULL if the variable has not been assigned
  for ((i=0;i<${#author[@]};i++));
  do
    echo "INSERT INTO 'HotelReviews' VALUES ($hotelID,${hotelOverallRating:-NULL},${hotelAveragePrice:-NULL},${hotelURL:-NULL
    },'${author[$i]}','${content[$i]}','${date[$i]}',${noReader[$i]:-NULL},${noHelpful[$i]:-NULL},${overall[$i]:-NULL},${value[
    $i]:-NULL},${rooms[$i]:-NULL},${location[$i]:-NULL},${cleanliness[$i]:-NULL},${checkInFrontDesk[$i]:-NULL},${service[$i]:-
    NULL},${businessService[$i]:-NULL});" >> hotelreviews.sql;
  done
done
```