BIDD: Assignment 4

Malthe Kirkbro
maek@itu.dk

Mikkel Gaub
mikg@itu.dk

Omar Khan
omsh@itu.dk

Kenneth Ry Ulrik
kulr@itu.dk

December 1, 2014

Data cleaning

During data cleaning, we found that there was indeed references pointing from the user table toward zipcode tuples that did not exist. We thus deleted these users, to avoid any errors that may occur in the future.
After this, we moved on to look at the rating table, making sure that the ratings of the users we just deleted were also cleaned. Finally, we went ahead and checked the moviegenre table for invalid references, however there appeared to be none.
Following this, we went ahead and applied proper structure to the database, by adding constraints that were otherwise not present - this was not mentioned in the assignment, but made our work a tad easier when executing various queries.
On the subject of the age data, we initially took the minimum age, and added it to a random value which was multiplied with the interval's length, to give some value in between. It appeared, however, that we had very few cases of the minimum age within each age group when we computed these ages. This was a problem, so we altered our update queries to 'FLOOR' the result, so that a RAND() computed value of 0.999, multiplied by 10, would, instead of becoming 10, become 9. This also meant that we could write the age intervals as their minimum up to their maximum allowed value minus one, since we can count on the flooring functionality to give us the results we want, as long as we define our numbers correctly.
On a side-note, we decided to make the first (youngest users) interval go from the age of 10 to 18, instead of 0-18, to make our estimates of age more realistic.

See the relevant SQL queries for data cleaning below:

Deletion-cleaning

```
-- user
DELETE FROM user
WHERE NOT EXISTS (
        SELECT id FROM occupation
        WHERE id = user.occupation
);

DELETE FROM user
WHERE NOT EXISTS (
        SELECT zip FROM zipcode
        WHERE zip = user.zip
);

-- rating
DELETE FROM rating
WHERE NOT EXISTS (
        SELECT id FROM user
        WHERE id = rating.userId
```

```
);

DELETE FROM rating
WHERE NOT EXISTS (
        SELECT id FROM movie
        WHERE id = rating.movieId
);

-- moviegenre
DELETE FROM moviegenre
WHERE NOT EXISTS (
        SELECT id FROM genre
        WHERE id = moviegenre.genreId
);

DELETE FROM moviegenre
WHERE NOT EXISTS (
        SELECT id FROM movie
        WHERE id = moviegenre.movieId
);

-- zipcodedata
DELETE FROM zipcodedata
WHERE NOT EXISTS (
        SELECT zip FROM zipcode
        WHERE zip = zipcodedata.zip
);
```

Age-value improvements

```
-- age group (10-17)
UPDATE user
SET age=10+FLOOR(RAND()*8)
WHERE age=1;

-- age group (18-24)
UPDATE user
SET age=18+FLOOR(RAND()*7)
 ,WHERE age=18;

-- age group (25-34)
UPDATE user
SET age=25+FLOOR(RAND()*10)
WHERE age=25;

-- age group (35-44)
UPDATE user
SET age=35+FLOOR(RAND()*10)
WHERE age=35;
```

```
-- age group (45-49)
UPDATE user
SET age=45+FLOOR(RAND()*5)
WHERE age=45;

-- age group (50-55)
UPDATE user
SET age=50+FLOOR(RAND()*6)
WHERE age=50;

-- age group (56-85)
UPDATE user
SET age=56+FLOOR(RAND()*30)
WHERE age=56;
```

Enriching

For enriching, we had to first alter the .csv file in a way that we could read it properly. This involved removing the header at first, as we preferred making our own table befitting the data, and then loading the data into this table of the database, meaning the header data was irrelevant. The provided database used CHAR for its zip-values (so that a zip is allowed to start with 0) ahead of our enrichment, and because our .csv file contained zip codes of length 3 and 4, we had a choice to make; either alter the provided database or alter the data. We chose the latter. We thus went ahead and cycled through all 'zip' values in the dataset, adding 0s in front of all zip code entries with a length below 5, so that all values would have a length of 5, thus compatible with the provided database. The alternative would have been to alter the table in the provided database to hold a VARCHAR reference instead of a CHAR one.
Once the .csv file was edited, we had the task of creating a table to contain the data. We encountered the fact that for the exisiting table to be able to reference our new zipcodedata table, the two values (its foreign key column and zipcodedata's primary key column) had to have the exact same typing, including their character set definition. We found the correct character set and defined our new table to follow the norm of the rest of the database.
In the end, all there was left to do, was to load the file into our newly created table, lines separated by semi-colons.

See the relevant SQL queries for the enrichment below:

```
CREATE TABLE zipcodedata (
        zip char(5) CHARACTER SET latin1 COLLATE latin1_swedish_ci,
        mean int(11),
        pop int(11),
        PRIMARY KEY(zip)
);

LOAD DATA LOCAL INFILE --filepath for .csv file.
INTO TABLE zipcodedata
FIELDS TERMINATED BY ';';
```

Description

Below is the fact table that we have come up with. See below for the full explanation of its contents.

movieId	genreId	userId	occupationId	zip
1	42	3	2	06950
1	42	4	2	06950
1	42	1	2	06950

One fact in this fact table represents (possibly only a part of) a single rating. Thus by using this fact table, we can say something about ratings on various movies which have different genres, as well as saying something about the users that made these ratings, specifically regarding age of user, demography and the associated income.
One fact in the fact table represents a single rating because one fact contains both a movieId and a userId, whereas these two combined grant access to rating data, since they form the primary key of the 'rating' table.
As it can be seen in the sample data above, one rating may occur several times. This is because one fact links to a single genreId, which means, in the case that a movie should have more than a single genre connected to it, it will occur several times as several facts - all these facts describe the same rating of the same movie concerning the same user.
The 'zip' value in the fact table links to two different tables; 'zipcode' and 'zipcodedata'. This is a case of vertical partitioning, as we let the zip value refer to two different data sources, all related to our user.

We have the following dimensions and possible measures by using this fact table:
Dimensions:

- modelId
- userId
- genreId
- occupationId
- zip

Measures, deriviated from our dimensions:

- rating (retrievable from movieId, userId (stored in rating table))
- time (date retrievable from movieId, userId (stored in rating table))
- lattitude (retrievable from zip)
- longitude (retrievable from zip)
- salary (retrievable from zip)

- population (retrieve from zip)
- age (retrievable from userId)

Found below are the transcripts for the fact table and the pre-aggregation tables along with the created indices.

Fact table

```
CREATE TABLE ratingfacts (
  movieId INT(11),
  userId INT(11),
  genreId INT(11),
  occupationId INT(11),
  zip CHAR(5) CHARACTER SET latin1 COLLATE latin1_swedish_ci,
  FOREIGN KEY (movieId) REFERENCES movie(id),
  FOREIGN KEY (userId) REFERENCES user(id),
  FOREIGN KEY (genreId) REFERENCES genre(id),
  FOREIGN KEY (occupationId) REFERENCES occupation(id),
  FOREIGN KEY (zip) REFERENCES zipcode(zip)
);

INSERT INTO ratingfacts
SELECT rating.movieId, rating.userId, moviegenre.genreId,
  occupation.id as occupation, zipcode.zip
FROM rating
JOIN user ON user.id = rating.userId
JOIN moviegenre ON moviegenre.movieId = rating.movieId
JOIN occupation ON occupation.id = user.occupation
JOIN zipcode ON zipcode.zip = user.zip;
```

Pre-aggregation table: genreview

```
CREATE TABLE genreview (
  genreName VARCHAR(20),
  movieCount INT(11),
  ratingCount INT(11),
  ratingMean FLOAT,
  userSalaryMean INT(11),
  userAgeMean FLOAT,
  userLattitudeMean FLOAT,
  userLongitudeMean FLOAT,
  zipPopMean INT(11)
);

INSERT INTO genreview (
  SELECT
    genre.name,
    COUNT(DISTINCT facts.movieId),
```

```
    COUNT(DISTINCT facts.movieId, facts.userId),
    AVG(rating.rating),
    AVG(zipcodedata.mean),
    AVG(user.age),
    AVG(zipcode.lattitude),
    AVG(zipcode.longitude),
    AVG(zipcodedata.pop)
  FROM ratingfacts AS facts
  JOIN genre ON genre.id = facts.genreId
  JOIN rating ON rating.userId = facts.userId
    AND rating.movieId = facts.movieId
  JOIN zipcode ON zipcode.zip = facts.zip
  JOIN zipcodedata ON zipcodedata.zip = facts.zip
  JOIN user ON user.id = facts.userId
  GROUP BY genreId
);

CREATE INDEX genreNameIndex ON genreview (genreName);
CREATE INDEX movieCountIndex ON genreview (movieCount);
CREATE INDEX ratingCountIndex ON genreview (ratingCount);
CREATE INDEX ratingMeanIndex ON genreview (ratingMean);
CREATE INDEX userSalaryMeanIndex ON genreview (userSalaryMean);
CREATE INDEX userAgeMeanIndex ON genreview (userAgeMean);
CREATE INDEX userLattitudeMeanIndex ON genreview (userLattitudeMean);
CREATE INDEX userLongitudeMeanIndex ON genreview (userLongitudeMean);
CREATE INDEX zipPopMeanIndex ON genreview (zipPopMean);
```

Pre-aggregation table: zipcodeview

```
CREATE TABLE zipcodeview (
  zip INT(11),
  city VARCHAR(64),
  state CHAR(2),
  population INT(11),
  totalUsers INT(11),
  userAgeMean FLOAT,
  userLattitudeMean FLOAT,
  userLongitudeMean FLOAT,
  totalRatings INT(11),
  ratingMean FLOAT
);

INSERT INTO zipcodeview (
  SELECT
    facts.zip,
    zipcode.city,
    zipcode.state,
    zipcodedata.pop,
    COUNT(DISTINCT user.id),
    AVG(user.age),
    AVG(zipcode.lattitude),
```

```
    AVG(zipcode.longitude),
    COUNT(DISTINCT rating.userId, rating.movieId),
    AVG(rating.rating)
  FROM ratingfacts AS facts
  JOIN zipcode ON zipcode.zip = facts.zip
  JOIN zipcodedata ON zipcodedata.zip = facts.zip
  JOIN user ON user.id = facts.userId
  JOIN rating ON rating.userId = facts.userId
    AND rating.movieId = facts.movieId
  GROUP BY zipcode.zip
);

CREATE INDEX zipIndex ON zipcodeview (zip);
CREATE INDEX cityIndex ON zipcodeview (city);
CREATE INDEX stateIndex ON zipcodeview (state);
CREATE INDEX populationIndex ON zipcodeview (population);
CREATE INDEX totalUsersIndex ON zipcodeview (totalUsers);
CREATE INDEX userAgeMeanIndex ON zipcodeview (userAgeMean);
CREATE INDEX userLattitudeMeanIndex ON zipcodeview (userLattitudeMean);
CREATE INDEX userLongitudeMeanIndex ON zipcodeview (userLongitudeMean);
CREATE INDEX totalRatingsIndex ON zipcodeview (totalRatings);
CREATE INDEX ratingMeanIndex ON zipcodeview (ratingMean);
```

Pre-aggregation table: userview

```
CREATE TABLE userview (
  id INT(11),
  age FLOAT,
  occupation VARCHAR(20),
  lattitudeMean FLOAT,
  longitudeMean FLOAT,
  zip INT(11),
  city VARCHAR(64),
  state CHAR(2),
  population INT(11),
  totalRatings INT(11),
  ratingMean FLOAT
);

INSERT INTO userview (
  SELECT
    user.id,
    user.age,
    occupation.description,
    zipcode.lattitude,
    zipcode.longitude,
    zipcode.zip,
    zipcode.city,
    zipcode.state,
    zipcodedata.pop,
    COUNT(DISTINCT rating.userId, rating.movieId),
```

```
    AVG(rating.rating)
  FROM ratingfacts AS facts
  JOIN user ON user.id = facts.userId
  JOIN occupation ON occupation.id = user.occupation
  JOIN zipcode ON zipcode.zip = user.zip
  JOIN zipcodedata ON zipcodedata.zip = user.zip
  JOIN rating ON rating.userId = facts.userId
    AND rating.movieId = facts.movieId
  JOIN moviegenre ON moviegenre.movieId = rating.movieId
  GROUP BY facts.userId
);

CREATE INDEX idIndex ON userview (id);
CREATE INDEX ageIndex ON userview (age);
CREATE INDEX occupationIndex ON userview (occupation);
CREATE INDEX lattitudeMeanIndex ON userview (lattitudeMean);
CREATE INDEX longitudeMeanIndex ON userview (longitudeMean);
CREATE INDEX zipIndex ON userview (zip);
CREATE INDEX cityIndex ON userview (city);
CREATE INDEX stateIndex ON userview (state);
CREATE INDEX populationIndex ON userview (population);
CREATE INDEX totalRatingsIndex ON userview (totalRatings);
CREATE INDEX ratingMeanIndex ON userview (ratingMean);
```

Pre-aggregation tables

GenreView
One of our views of the data retrievable from the fact table construction is a view based on the genres of the movies that were rated. This gives us the ability to see ratings in relation to the genres of the movies rated. Thus we can say something about what the best rated genre is as well as which zipcode areas have given the most rates of highest average ratings for each genre.

UserView
This other view takes a look at ratings from each user's perspective, meaning we can sort this pre-aggregation table to find out for instance which user has casted the most votes on movies of the genre "action". Additionally we can find out which user is the most active overall, or perhaps find out about some interesting users, in the sense that there may for instance some users that were simply created to give a 1 or 5 star rating to a single movie and afterwards never rated a movie again.

ZipCodeView
This final view is a view which takes the perspective of a single zip code area. This means that we can say something about how many raters there are from each zip code area, as well as for instance their average age and the average ratings the various zip code areas.
Note that we talk about "zip code areas" rather than "zip codes" alone, as there are several sections of for instance a single zip code within a city.

Indexes created
As the assignment simply states to create indexes onto the pre-aggregation tables that we have created, there is nothing much to describe, aside from the fact that we of course have added indexes onto all values in all of our pre-aggregation tables for ease of access of this aggregated data, thus making it aggregated data-retrieval faster in all cases, no matter what the requester may be interested in finding out about the data.

0.1 Four fun facts

- The guy who has rated the highest number of times is 45-56 years old and lives in Kansas City, Leavensworth. He rated 2314 times.

```
SELECT totalRatings, age, city, state, occupation
FROM userview
ORDER BY totalRatings DESC
LIMIT 1;
```

- The profession (Thus excluding retired people, among others) of users that has rated the least number of times as well as having given the lowest average scores for movies, is farmers.

```
SELECT AVG(totalRatings) meanRatings, occupation
FROM userview
GROUP BY occupation
ORDER BY meanRatings DESC;

SELECT AVG(ratingMean) meanRatingScore, occupation
FROM userview
GROUP BY occupation
ORDER BY meanRatingScore DESC;
```

- The occupation of users that have given the highest average scores for movies of the genre "Sci-Fi", is programmers.

```
SELECT genre.name, occupation.description,
       AVG(rating.rating) meanRating
FROM ratingfacts AS facts
JOIN genre ON genre.id = facts.genreId
JOIN user ON user.id = facts.userId
JOIN occupation ON occupation.id = user.occupation
JOIN rating ON rating.userId = facts.userId
       AND rating.movieId = facts.movieId
WHERE genre.name = "Sci-Fi"
GROUP BY occupation.id
ORDER BY meanRating DESC
LIMIT 1;
```

- The zip code area with the lowest average income rate movies with the genre "Action" the most, in order to spice up their otherwise extremely dull lives.

```
SELECT facts.zip, zipcode.city,
        zipcode.state, genre.name genre,
        zipcodedata.mean meanIncome,
        COUNT(
        DISTINCT facts.userId, facts.movieId
        ) ratings
```

```
FROM ratingfacts AS facts
JOIN zipcode ON zipcode.zip = facts.zip
JOIN zipcodedata ON zipcodedata.zip = facts.zip
JOIN moviegenre ON moviegenre.movieId = facts.movieId
JOIN genre ON genre.id = moviegenre.genreId
WHERE facts.zip = (
        SELECT zip
        FROM (
                SELECT zip, mean meanSalary
                FROM zipcodedata
        WHERE EXISTS (
                SELECT *
                FROM rating JOIN user
                        ON user.id = rating.userId
                WHERE user.zip = zipcodedata.zip
        )
        ORDER BY meanSalary DESC
        LIMIT 1
        ) AS s1
)
GROUP BY genre.id
ORDER BY ratings DESC
LIMIT 1;
```