Mini Project in Database Systems

Elad Harizy and Abraham Murciano

October 5, 2020

Contents

1 First Stage **2**

1.1 Proposal . . . 2

1.2 Entity Relationship Diagram . . . 3

1.2.1 Entities . . . 3

1.2.2 Relations . . . 4

1.2.3 Diagram . . . 4

1.3 Logical Schema . . . 6

1.4 Normalization . . . 8

1.4.1 Functional Dependencies . . . 8

1.4.2 Third Normal Form . . . 9

1.5 Physical Schema . . . 9

1.5.1 Users Table . . . 9

1.5.2 Posts Table . . . 10

1.5.3 Questions Table . . . 10

1.5.4 Answers Table . . . 10

1.5.5 Comments Table . . . 11

1.5.6 Topics Table . . . 11

1.5.7 Votes Table . . . 12

1.5.8 Follows Table . . . 12

1.5.9 Relates To Table . . . 12

1.6 Report . . . 13

1.7 Populating Tables With Data 15
1.7.1 Populating Users 15
1.7.2 Populating Posts 15
1.7.3 Populating Questions 16
1.7.4 Populating Answers 16
1.7.5 Populating Comments 16
1.7.6 Populating Topics 16
1.7.7 Populating Votes 19
1.7.8 Populating Follows 19
1.7.9 Populating Relates To 19
1.8 Validating the Data 19
1.8.1 Negative Points for Certain Questions 19
1.8.2 Circular References in Comment Hierarchy 20
1.9 Backup and Restore 20

2 Second Stage **20**

Introduction

In this document we outline the process we underwent in order to design and create a database. Most of the referenced files can be found on https://github.com/abrahammurciano/ask-us. File paths will be relative to the repository root directory.

1 First Stage

1.1 Proposal

Ask Us is a hypothetical organization who want to create a website where people can ask and answer questions to the general public on any topic. Our task is to analyse the data needs of such a website, to the ends of creating a database for them along with a website to interface with it.

Ask Us will need to store information on its users, as well as all the questions and answers. They would also like each question and answer to have its own comment thread, for people to contribute helpful information regarding answers as well as for short follow up questions.

All questions, answers, and comments should be scored by the users in some way, so that people can see what posts were considered good by the general public and which were disliked. This should help people evaluate the answers and comments they receive.

Additionally, they would like questions to be organized by topic, where each question can be associated with multiple topics. Users should also be able to follow topics they are interested in, so they can see questions related to their interests.

1.2 Entity Relationship Diagram

1.2.1 Entities

We will have six entities: user, post, question, answer, comment, and topic.

A user is an individual writing a post on Ask Us. The user entity will have five attributes: ID, username, email, password, and points. A user's points are the sum of all the points of all their posts. ID will be the primary key.

A post is anything a user writes. This can either be a question, answer, or comment. The post entity will only have an ID, which is the primary key. The purpose of this entity is mainly so that comments can have their parent post be of any type. (We can comment on questions, answers, and even other comments.)

Questions, answers, and comments will all have the following attributes, aside from the ones we specify below. They will have a body, which is the main text that makes up that post. They will also have a count of the points that they have been awarded by users. They will also have a timestamp which will be the exact date and time a post was made.

The question entity has one other attribute called title, aside from the attributes listed above. Since it is a weak entity, its primary key is the post's ID.

The answer entity has a boolean attribute called accepted, apart from the attributes it has in common with the other post types. Similarly, since it is a weak entity, its primary key is the post's ID.

A comment is typically a short piece of text that a user writes beneath any kind of post. A comment itself has no other attributes, other than the ones above. Once again, since it is a weak entity, its primary key is its post's ID.

A topic is a way for users to categorize their questions, thus getting better targeted responses. A user can also follow a topic of interest. The topic entity will have three attributes: ID, username, and description. ID will once again be the primary key.

1.2.2 Relations

A user can 'follow' a topic. In this relation, there can be many topics a user can follow, but it is not mandatory for a user to follow any topic. In the same light, topics can be followed by any number of users.

A user is related to the posts that they write. In this relation, the user can write anywhere from many to no posts. However, a post must be written by exactly one user.

A user is also related to a post by voting on it. The vote relation is a many to many relation, in that a user can vote on many to no posts and a post can be voted on by many to no users.

The comment entity has two relations with the post entity. One relation is called belongs to. This relation represents the fact that a specific comment has a parent post (which may also be a comment). A post can optionally have many comments, but a comment must be a child of exactly one post.

The other relation is an inheritance relation. It is an identifying relation, and is labelled 'is a' in figure 2. In this relation, a particular comment is related to exactly one post, meaning that this comment entity 'extends' that post, because they are in fact the same post. In this relation, a post does not necessarily have to be related to a comment, since not all posts are comments, but if it is, it must be related to exactly one comment.

Since questions and answers are also types of post, they also have an inheritance relation between themselves and post. This relation is identical to the inheritance relation between comment and post.

Answers must also be related to the questions which they answer. A question can have zero to any number of answers, yet an answer must be related to exactly one question.

Lastly, questions are related to topics. A question must have at least one topic, and a topic can have zero or more questions related to it.

1.2.3 Diagram

There are a few possible ways we can design an entity relationship diagram to suit our needs. One option is what is shown in figure 1. The main aspect to note is that the Post entity has the attributes 'body', 'timestamp', and 'points', which all the three post types don't need to have, since they 'inherit' those attributes from Post.

However, this may cause inefficiency when retrieving data for any type of post from the database. This is because the attributes of each post would be divided into two tables, requiring us to join the tables to obtain all the data on any particular post.

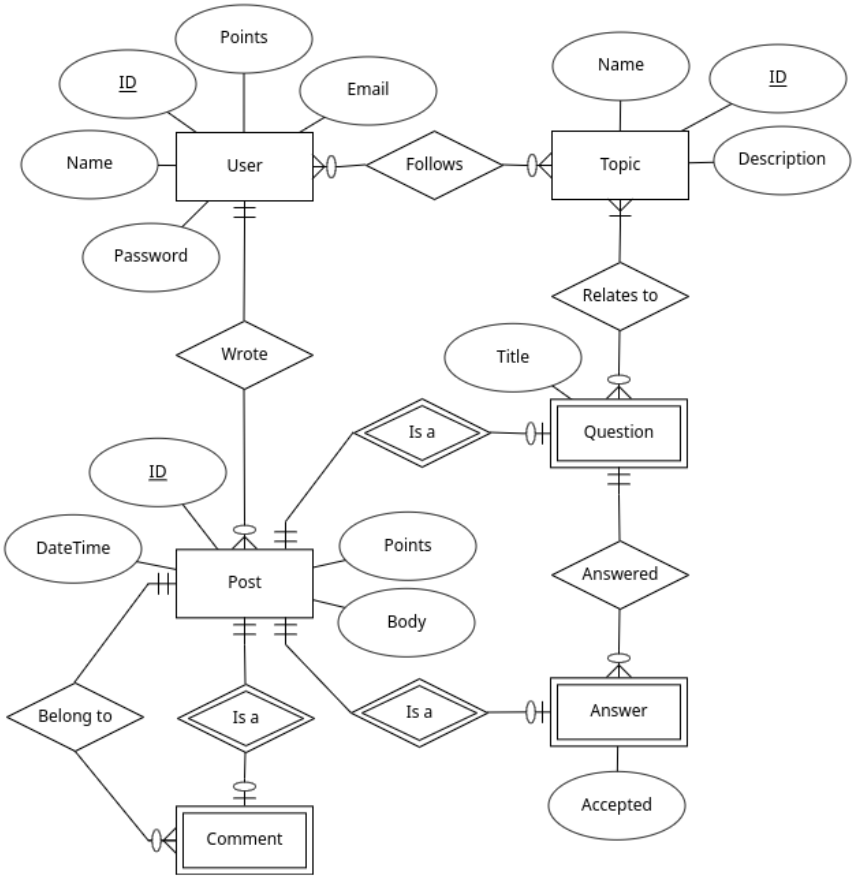

Figure 1: The first design of the ERD

We therefore tweak the design so that each post type contains all of their fields in the same table, even those that all the post types have in common. This solves the inefficiency because there is no longer a need to join the tables, and this solution still maintains the tables normalized to the same extent that they were prior.

Another consideration we must make is that since a comment is a type of post, and a comment can have child comments, we encounter a cycle in the ERD. Specifically, this means that the depth of a comment thread would have no limit. This can cause issues if we try to query all the descendant comments of any post, since we would need a recursive query for this.

We may consider applying certain techniques such as materialized paths or using a closure table, which would make this kind of query trivial. However, if we give our system some consideration, we can be fairly certain that such queries need not be made.

For example, when the website needs to load all the child comments of any particular question, we are only interested in the top-level comments at first. Retrieving those is also a simple query, since each comment knows its direct parent. Once we have those, the application level can recursively query all the top-level comments of each of those comments, meaning it would obtain all the second-level comments of the question. Similarly, our application can obtain all the comments, regardless of how many levels deep it can be found.

This approach is preferable in our scenario, since obtaining all the descendant comments in one bulk, as techniques like materialized path and closure tables would allow is unhelpful to us. This is because we need to display them in a hierarchical structure, and retrieving them in bulk would then require further processing by the application in order to structure them accordingly.

Another possible modification which we may consider to make the retrieval of comments more efficient is to only allow a single level of comments, or in other words, not allowing for comments to have child comments. However, this would cause a detriment in user experience, as they would be limited to conducting much discussion on some questions or answers.

Therefore we can conclude that the optimal approach would be to maintain the design as we have described above.

Figure 2 shows the final design of the entity relationship diagram for the database we are to make for Ask Us. Although the ERD is more complex, it provides the facility to use simpler and more efficient queries to retrieve the data that our system is interested in.

1.3 Logical Schema

We are to convert the ERD in the previous section into a logical schema. First we will convert the entities into tables, and then we will add tables for some of the relations where required.

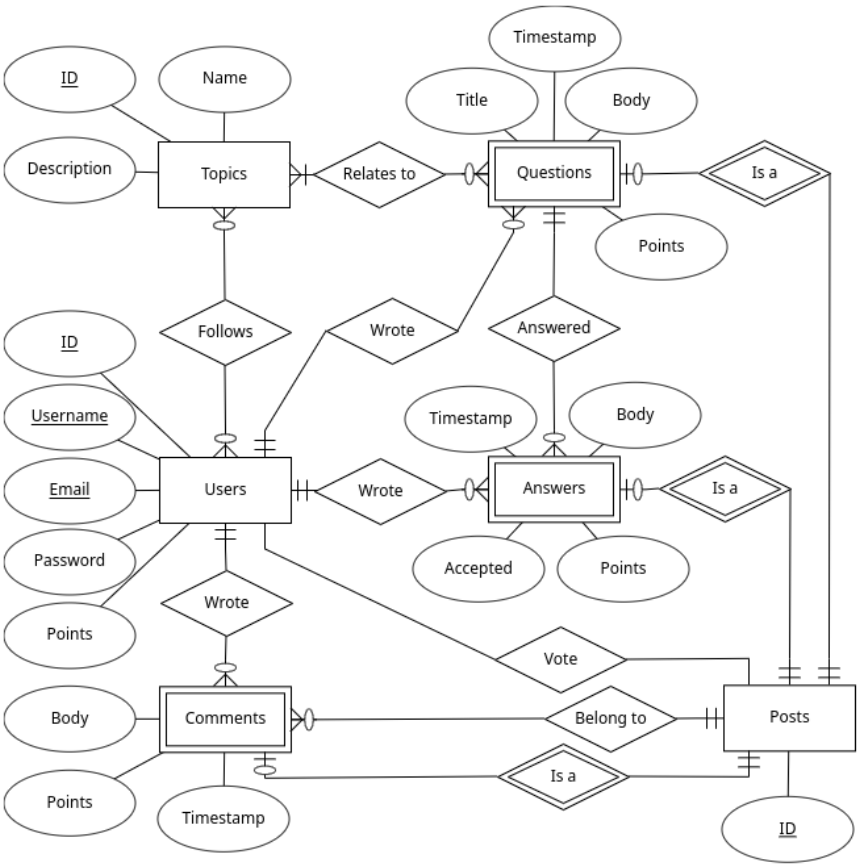

Figure 2: The final ERD for the database for Ask Us

See the entity tables below. An underlined value indicates a primary key. An *italicized* value indicates a foreign key.

User (ID, Name, Email, Password, Points)

Topic (ID, Name, Description)

Post (ID)

Comment (*Post ID*, *Parent Post ID*, Body, Points, TimeStamp)

Question (*Post ID*, Title, Body, Points, TimeStamp)

Answer (*Post ID*, Accepted, *Question ID*, Body, Points, TimeStamp)

Follows (*User ID*, *Topic ID*)

Relates to (*Post ID*, *Topic ID*)

Vote (*User ID*, *Post ID*)

1.4 Normalization

1.4.1 Functional Dependencies

These are the functional dependencies for the User table.

ID → (Name, Email, Password, Points)
Email → (ID, Name, Password, Points)

These are the functional dependencies for the Topic table.

ID → (Name, Description)

These are the functional dependencies for the Comment table.

Post ID → (Parent Post ID, Body, Points, TimeStamp)

These are the functional dependencies for the Question table.

Post ID → (Title, Body, Points, TimeStamp)

These are the functional dependencies for the Answer table.

Post ID → (Accepted, Question ID, Body, Points, TimeStamp)

The tables for Post, Follows, Relates to and Vote have no non trivial functional dependencies.

1.4.2 Third Normal Form

All the tables above are in 3NF because all the functional dependencies of the form $X \to Y$ satisfiy the condition "X is a superkey".

The tables for POST, FOLLOWS, RELATES TO and VOTE to are also in 3NF since they have no non-trivial functional dependencies.

For the same reason, the tables are all in BCNF.

1.5 Physical Schema

Here we can see the SQL statements which can be used to create the tables we defined above. These statements are specific to Oracle Database.

1.5.1 Users Table

Displayed here is the SQL statement that creates the `users` table. An 8 digit number for the ID (and all ID fields of subsequent tables) is large enough since we will not have more than one million records in the entire database. The ID field is to be generated by the database.

Usernames we limit to twenty characters, and email addresses to 320. This is because 320 characters is the length of the longest possible email address. The `password` field will actually store a base 64 encoded SHA256 hash of the password, which takes 44 bytes.

A reasonable cap for the total number of points a user can accumulate is a ten digit number. This conclusion is based off researching the highest number of points any user has ever accumulated on popular websites which implement similar scoring systems such as Stack Overflow and Reddit, then adding a couple of digits to be on the safe side. By default, a user starts with no points.

`/sql/create-table/users.sql`

```
create table users(
        id number(8) generated always as identity primary key,
        username varchar(20) unique not null,
        email varchar(320) unique not null,
        password char(44) not null,   -- base64 sha256 is 44 chars
        points number(10) default 0 not null
);
```

1.5.2 Posts Table

Below is the SQL statement which creates our table that contains the IDs of all the posts.

/sql/create-table/posts.sql

```
create table posts (
        id number(8) generated always as identity primary key
);
```

1.5.3 Questions Table

This is the SQL command that generates the questions table. The field `body` is of the CLOB data type, which is short for 'character large object'. The `varchar` data type has much smaller length restrictions which would not suffice to allow for long questions.

In this table, we have introduced for the first time the `timestamp` field in SQL. Its data type is `date`, which gives us a date and time representation, accurate up to one second, which is enough for us to record the creation dates of questions (as well as answers and comments). By default, it is assigned the current date and time at the time of insertion of each row.

/sql/create-table/questions.sql

```
create table questions(
        post_id number(8) primary key references posts(id),
        title varchar(255) not null,
        body clob not null,
        points number(10) default 0 not null,
        timestamp date default sysdate not null
);
```

1.5.4 Answers Table

Now we come to the SQL statement for the `answers` table. It is mostly identical to the `questions` table, with the exceptions of the field `accepted` which is a boolean, and the field `question_id` which is the question that this answer is responding. Since Oracle does not have a built in boolean data type, we use the number data type and limit it to one digit, and we decide that 0 means false and anything else means true.

/sql/create-table/answers.sql

```
create table answers(
        post_id number(8) primary key references posts(id),
        accepted number(1) default 0 not null,
        question_id number(8) references questions(post_id) not null,
        body clob not null,
        points number(10) default 0 not null,
        timestamp date default sysdate not null
);
```

1.5.5 Comments Table

This table is similar to the two previous tables. The only new column is the `parent_post_id` column. This column is a foreign key to the comment's direct parent.

/sql/create-table/comments.sql

```
create table comments(
        post_id number(8) primary key references posts(id),
        parent_post_id number(8) references posts(id) not null,
        body clob not null,
        points number(10) default 0 not null,
        timestamp date default sysdate not null
);
```

1.5.6 Topics Table

The `topics` table contains an `id` column like most of the previous tables, as well as a label and a description. The label must be unique and is limited to twenty characters.

/sql/create-table/topics.sql

```
create table topics(
        id number(8) generated always as identity primary key,
        label varchar(255) unique not null,
        description varchar(255) not null
);
```

1.5.7 Votes Table

This table consists of two columns which together form the primary key, and each of which are a foreign key in their own right. Each row signifies that a particular user voted on a particular post.

/sql/create-table/votes.sql

```
create table votes(
        user_id references users(id),
        post_id references posts(id),
        primary key (user_id, post_id)
);
```

1.5.8 Follows Table

With this SQL statement we can create the `follows` table. It has a very similar structure to the `votes` table. Here, each row represents that a certain user follows some topic.

/sql/create-table/follows.sql

```
create table follows(
        user_id references users(id),
        topic_id references topics(id),
        primary key (user_id, topic_id)
);
```

1.5.9 Relates To Table

This SQL command creates the `relates_to` table. Once again, it has an almost identical structure to the previous two tables. Each of these rows records that a specific question relates to some topic.

/sql/create-table/relates_to.sql

```
create table relates_to(
        post_id references posts(id),
        topic_id references topics(id),
        primary key (post_id, topic_id)
);
```

1.6 Report

Below we can see the report which was generated by PL/SQL Developer after creating all the tables with the SQL statements mentioned in the previous section.

Tables

table owner = AMURCIAN
table name = %

Owner	Table Name	Tablespace Name	Cluster Name	Iot Name	Status	Pct Free	Pct Used	Ini Trans	Max Trans	Initial Extent	Next Extent	Min Extents	Max Extents	Pct Increase	Freelists	Freelist Groups	Logging	Backed Up	Num Rows	Blocks	Empty Blocks	Avg Space	Chain Cnt	Avg Row Len	Avg Space Freelist Blocks	Num Freelist Blocks	Degree	Instances	Cache	Table Lock	Sample Size	Last Analyzed	Partitioned	Iot Type	Temporary	Secondary	Nested	Buffer Pool	Flash Cache	Cell Flash Cache	Row Movement	Global Stats	User Stats	Duration	Skip Corrupt	Monitoring	Cluster Owner	Dependencies	Compression	Compress For	Dropped	Read Only	Segment Created	Result Cache	Clustering	Activity Tracking	Dml Timestamp	Has Identity	Container Data	Inmemory	Inmemory Priority	Inmemory Distribute	Inmemory Compression	Inmemory Duplicate
AMURCIAN	ANSWERS	USERS			VALID	10		1	255								YES	N	0	0	0	0	0	0	0	0	1	1	N	ENABLED	0	06/09/2020 06:00:17	NO		N	N	NO	DEFAULT	DEFAULT	DEFAULT	DISABLED	YES	NO		DISABLED	YES		DISABLED	DISABLED		NO	NO	NO	DEFAULT	NO			NO	NO	DISABLED				
AMURCIAN	COMMENTS	USERS			VALID	10		1	255								YES	N	0	0	0	0	0	0	0	0	1	1	N	ENABLED	0	06/09/2020 06:00:17	NO		N	N	NO	DEFAULT	DEFAULT	DEFAULT	DISABLED	YES	NO		DISABLED	YES		DISABLED	DISABLED		NO	NO	NO	DEFAULT	NO			NO	NO	DISABLED				
AMURCIAN	FOLLOWS	USERS			VALID	10		1	255								YES	N	0	0	0	0	0	0	0	0	1	1	N	ENABLED	0	06/09/2020 06:00:17	NO		N	N	NO	DEFAULT	DEFAULT	DEFAULT	DISABLED	YES	NO		DISABLED	YES		DISABLED	DISABLED		NO	NO	NO	DEFAULT	NO			NO	NO	DISABLED				
AMURCIAN	POSTS	USERS			VALID	10		1	255								YES	N	0	0	0	0	0	0	0	0	1	1	N	ENABLED	0	06/09/2020 06:00:17	NO		N	N	NO	DEFAULT	DEFAULT	DEFAULT	DISABLED	YES	NO		DISABLED	YES		DISABLED	DISABLED		NO	NO	NO	DEFAULT	NO			YES	NO	DISABLED				
AMURCIAN	QUESTIONS	USERS			VALID	10		1	255								YES	N	0	0	0	0	0	0	0	0	1	1	N	ENABLED	0	06/09/2020 06:00:17	NO		N	N	NO	DEFAULT	DEFAULT	DEFAULT	DISABLED	YES	NO		DISABLED	YES		DISABLED	DISABLED		NO	NO	NO	DEFAULT	NO			NO	NO	DISABLED				
AMURCIAN	RELATES_TO	USERS			VALID	10		1	255								YES	N	0	0	0	0	0	0	0	0	1	1	N	ENABLED	0	06/09/2020 06:00:17	NO		N	N	NO	DEFAULT	DEFAULT	DEFAULT	DISABLED	YES	NO		DISABLED	YES		DISABLED	DISABLED		NO	NO	NO	DEFAULT	NO			NO	NO	DISABLED				
AMURCIAN	TOPICS	USERS			VALID	10		1	255								YES	N	0	0	0	0	0	0	0	0	1	1	N	ENABLED	0	06/09/2020 06:00:17	NO		N	N	NO	DEFAULT	DEFAULT	DEFAULT	DISABLED	YES	NO		DISABLED	YES		DISABLED	DISABLED		NO	NO	NO	DEFAULT	NO			YES	NO	DISABLED				
AMURCIAN	USERS	USERS			VALID	10		1	255								YES	N	0	0	0	0	0	0	0	0	1	1	N	ENABLED	0	06/09/2020 06:00:17	NO		N	N	NO	DEFAULT	DEFAULT	DEFAULT	DISABLED	YES	NO		DISABLED	YES		DISABLED	DISABLED		NO	NO	NO	DEFAULT	NO			YES	NO	DISABLED				
AMURCIAN	VOTES	USERS			VALID	10		1	255								YES	N	0	0	0	0	0	0	0	0	1	1	N	ENABLED	0	06/09/2020 06:00:17	NO		N	N	NO	DEFAULT	DEFAULT	DEFAULT	DISABLED	YES	NO		DISABLED	YES		DISABLED	DISABLED		NO	NO	NO	DEFAULT	NO			NO	NO	DISABLED				

1.7 Populating Tables With Data

Now that we have successfully created all the necessary data, it is time to fill them with data. We will use a variety of different methods to import data to our tables.

1.7.1 Populating Users

We retrieved a large excel file containing over one hundred thousand users' data from the internet, but since we had no remaining space in the database we had to shrink that file to only one thousand records. Using the PL/SQL Developer tool called ODBC importer we imported the data which we needed into the users table. Not all of the columns provided were necessary to us, so figure 3 shows which columns we used. The excel file which we used can be found in `/data/users/user_list.xlsx`.

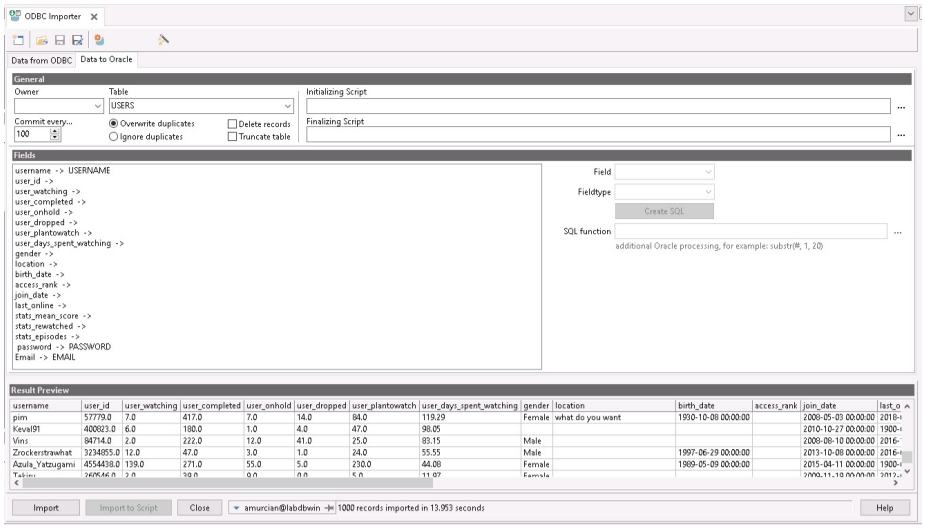

Figure 3: A screenshot showing how the ODBC importer imported our data

1.7.2 Populating Posts

Since this table only contains one column, and that column's value was always generated by the database, this simple script was enough to insert the required number of post IDs into the post table. We originally had inserted over 200,000 but due to severe space constraints we reduced this to 3,000.

`/sql/insert/posts.sql`

```
BEGIN
    FOR v_LoopCounter IN 1..3000 LOOP
        INSERT INTO POSTS VALUES(DEFAULT);
        COMMIT;
    END LOOP;
END;
```

1.7.3 Populating Questions

We were able to obtain a CSV file containing all of the questions which were asked on LaTeX Stack Exchange between January 2010 and September 2020. Since we were restricted on space, we manipulated the CSV file using spreadsheet software to keep only the 1,000 shortest questions. After appending an ID from those which we inserted into the `posts` table to each row, and removing all columns which we do not need, we ran the text importer tool in PL/SQL Developer to add all of the data in the CSV file into the `questions` table.

Figure 4 shows the configurations for the text importer which was used to import the questions from our CSV file.

1.7.4 Populating Answers

We used the data generator tool in PL/SQL developer, as shown in figure 5, to populate this table with 1,000 random records.

1.7.5 Populating Comments

We used the data generator tool to generate 1,000 random comments. The settings used to generate these can be seen in figure 6.

1.7.6 Populating Topics

We were able to find a large list with 1,547 topics which Quora uses to organise their questions. We obtained it in the form of a plain text file with one line for each topic name. Since it contains duplicates, we can must first remove these. Then we generate some randomised description using the python module called `lorem`. This python code shows exactly how the insert statements were generated. The generated SQL code can be found at `/sql/insert/topics.sql`

`/data/topics/topics_to_insert.py`

```
from lorem.text import TextLorem
```

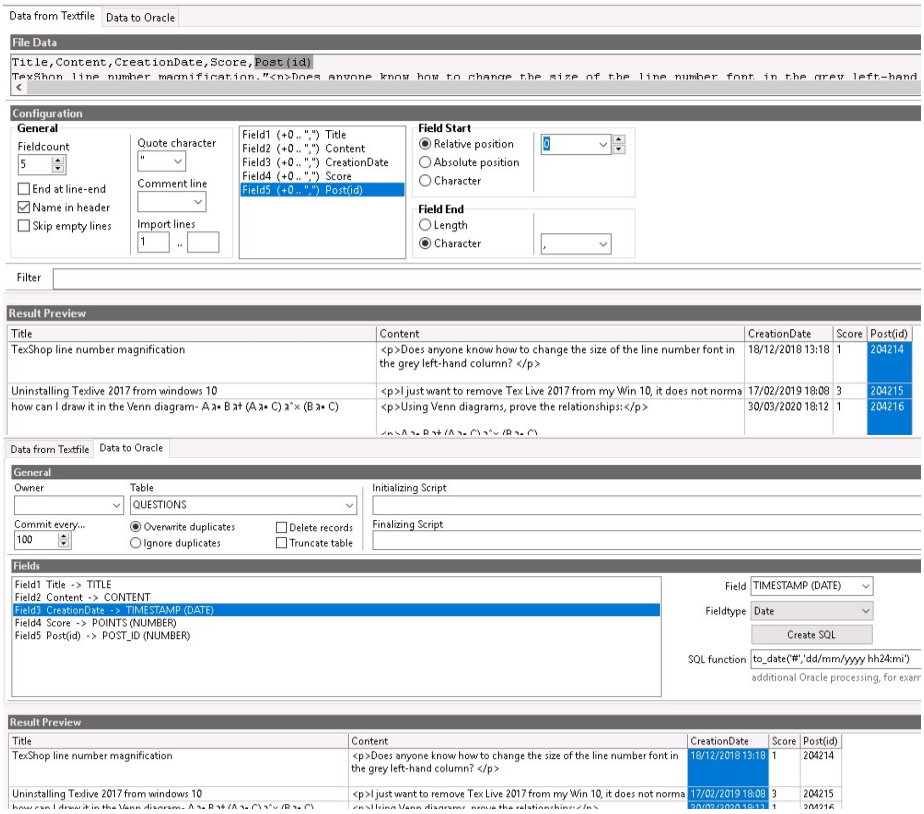

Figure 4: The configurations in the text importer for the questions table

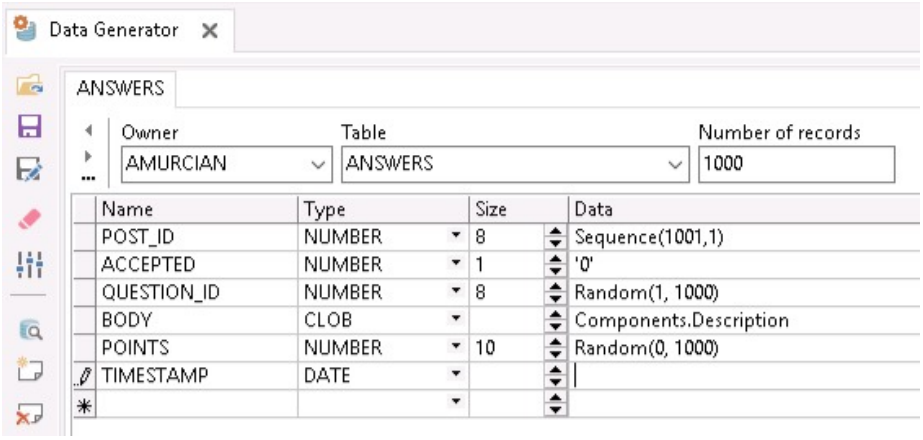

Figure 5: The configurations in the data generator tool for the answers table

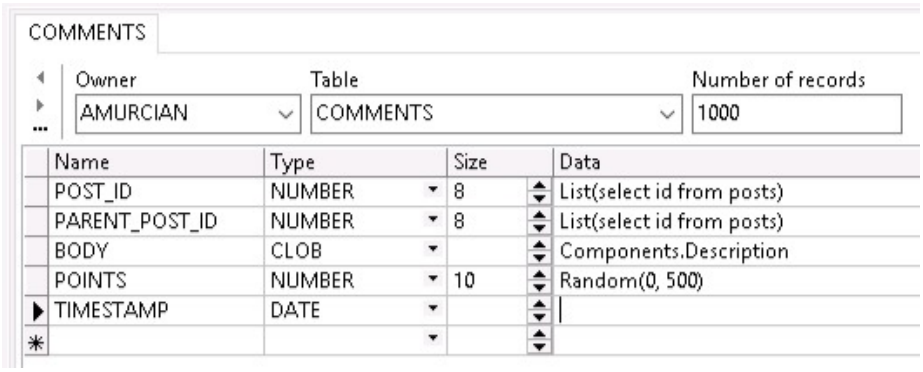

Figure 6: The configurations in the data generator tool for the comments table

```
# remove duplicate entries from the list of topics
with open("topics.txt", "w") as topics:
        lines_seen = set() # holds lines already seen
        for line in open("topics_w_dups.txt", "r"):
                if line not in lines_seen: # not a duplicate
                        topics.write(line)
                        lines_seen.add(line)

# make an insert statement for each topic
with open("topics.txt", "r") as topics:
        sql = open("../../sql/insert/topics.sql", "w")
        # 4-16 words/sentence, 2-16 sentences/paragraph
        lorem = TextLorem(srange=(4,16), prange=(2,16))
        for topic in topics:
                while True:
                        description = lorem.paragraph()
                        # ensure the description fits in db
                        if len(description) <= 255:
                                break
                # remove trailing \n and escape single quotes
                label = topic[:-1].replace("'", "''")
                sql.write(
"""
insert into topics
        (label, description)
values
        ('""" + label + "', '" + description + """');
"""
                )
        sql.close()
```

1.7.7 Populating Votes

We used the data generator to populate this table. The settings we used to generate the random data is practically identical to those seen in figure 7.

1.7.8 Populating Follows

This table was populated using the data generator tool in PL/SQL Developer. We asked the system to generate a number from the users' ID column for the foreign key to that row, and similarly for the other foreign key. Figure 7 shows the settings we used to generate 1,000 rows.

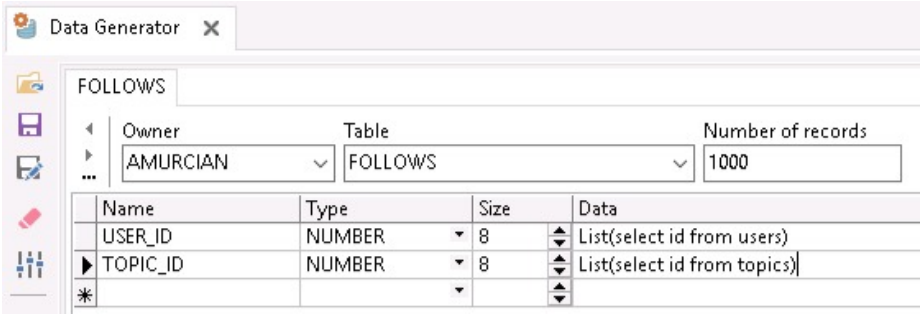

Figure 7: Settings used for generating the follows table data

1.7.9 Populating Relates To

This table also had its data generated in the same way as the `follows` and `relates_to` tables.

1.8 Validating the Data

1.8.1 Negative Points for Certain Questions

Since the data we used for populating the questions table was obtained from Stack Exchange, some of the questions had a negative number for the points field. However, unlike Stack Overflow, we do not allow negative points, since we only provide users with the ability to up-vote posts, not to down-vote them.

We therefore ran the following SQL statement to negate all the negative values.

```
UPDATE questions SET points = -points WHERE points < 0;
```

1.8.2 Circular References in Comment Hierarchy

Since we randomly generated the comments, and each comment has a field `post_id`, we ended up with some circular references where a comment was its own parent, or a comment's descendant was also an ancestor of that same comment.

In order to solve this, we exported the data as a CSV which we then modified using spreadsheet software and formulae so that each comment referenced a post with an ID smaller its own. We then truncated the table to remove the random data, and used the text importer to import the processed CSV in the same way which we imported the questions. This CSV file can be found at `/data/comments/comments.csv`

1.9 Backup and Restore

To test that the data is all in place, we are tasked with creating backups of all the data, then deleting all the data from the database and finally importing it all again.

The exported backup files can be found in the directory `/sql/backup/`, as well as the script which we use to delete all the data, which is called `delete_all.sql`.

`/sql/backup/delete_all.sql`

```
delete from relates_to;
delete from follows;
delete from votes;
delete from comments;
delete from answers;
delete from questions;
delete from posts;
delete from users;
delete from topics;
```

2 Second Stage