Mini Project in Database Systems

Abraham Murciano, Elad Harizy, and Ezra Dweck

October 15, 2020

Contents

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.

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.

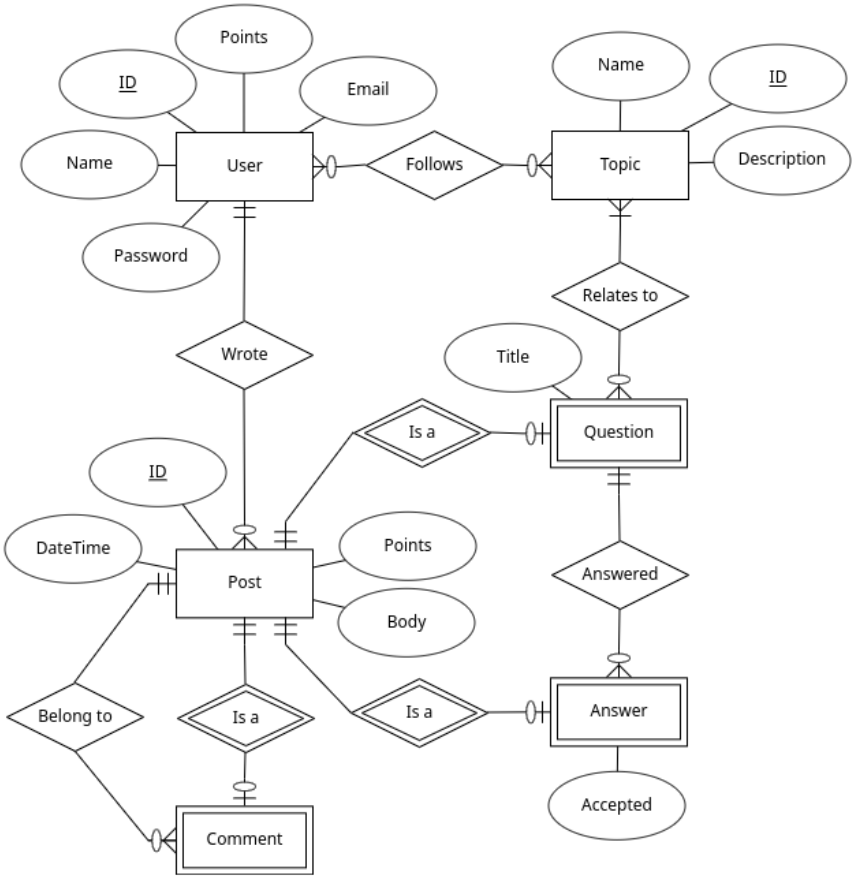

Figure 1: The first design of the ERD

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.

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)

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

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

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

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.

$$\text{ID} \rightarrow (\text{Name, Email, Password, Points})$$
$$\text{Email} \rightarrow (\text{ID, Name, Password, Points})$$

These are the functional dependencies for the Topic table.

$$\text{ID} \rightarrow (\text{Name, Description})$$

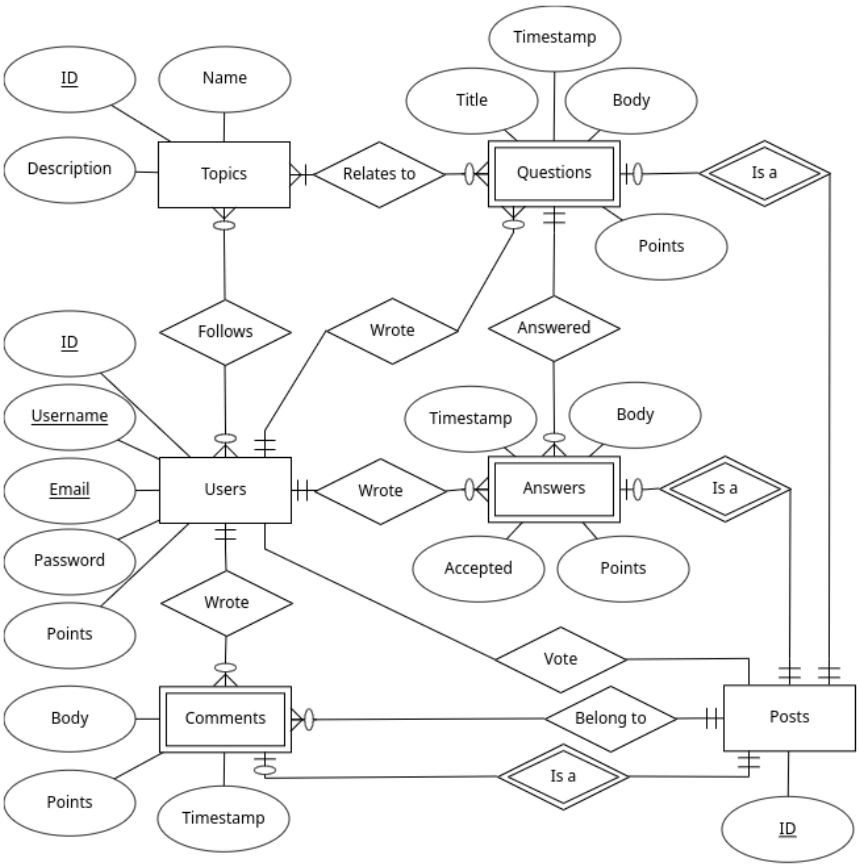

Figure 2: The final ERD for the database for Ask Us

These are the functional dependencies for the QUESTION table.

$$\text{Post ID} \rightarrow (\text{Title, Body, Points, TimeStamp, Author ID})$$

These are the functional dependencies for the ANSWER table.

$$\text{Post ID} \rightarrow (\text{Accepted, Question ID, Body, Points, TimeStamp, Author ID})$$

These are the functional dependencies for the COMMENT table.

$$\text{Post ID} \rightarrow (\text{Parent Post ID, Body, Points, TimeStamp, Author ID})$$

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 $X \rightarrow Y$ 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 43 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 varchar(255) not null,   -- base64 sha256 hash of password
        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. The `author_id` is the ID of the user who wrote this question.

/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,
```

```
        author_id number(8) references users(id) 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,
        author_id number(8) references users(id) 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,
        author_id number(8) references users(id) 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(
        question_id references questions(post_id),
        topic_id references topics(id),
        primary key (question_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/tables/user_list.xlsx`.

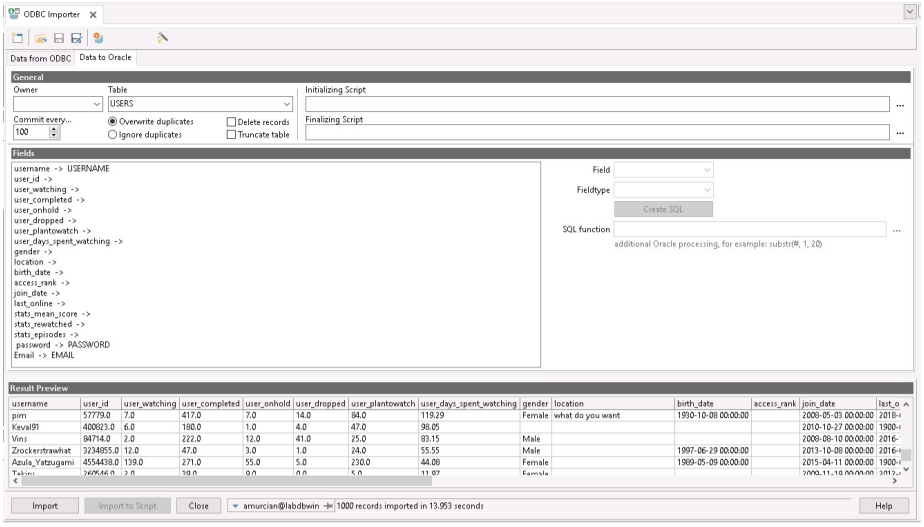

Figure 3: A screenshot showing how the ODBC importer imported our data

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/tables/topics/topics_to_insert.py`

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

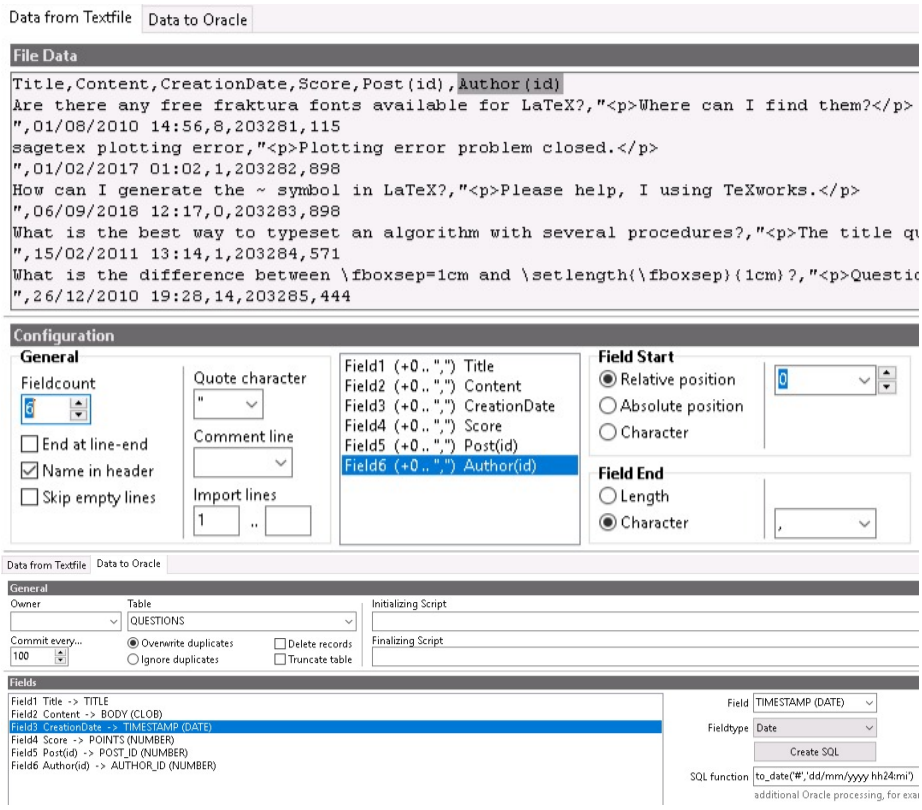

Figure 4: The configurations in the text importer for the questions table

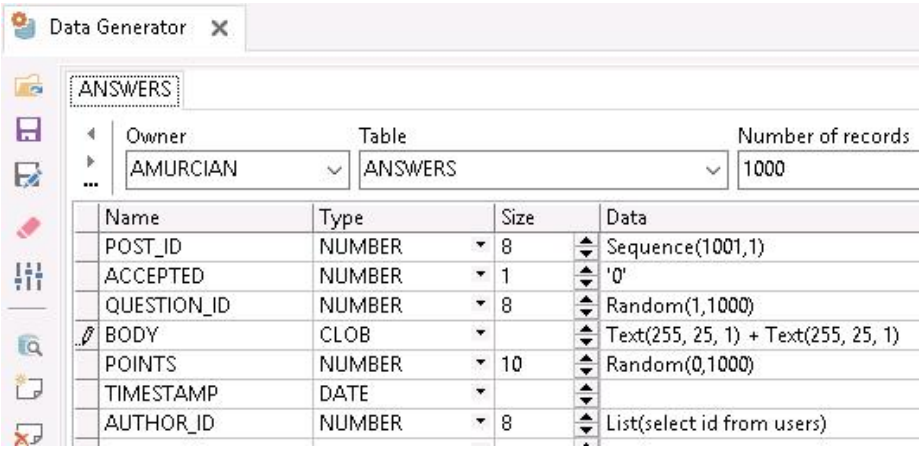

Figure 5: The configurations in the data generator tool for the answers table

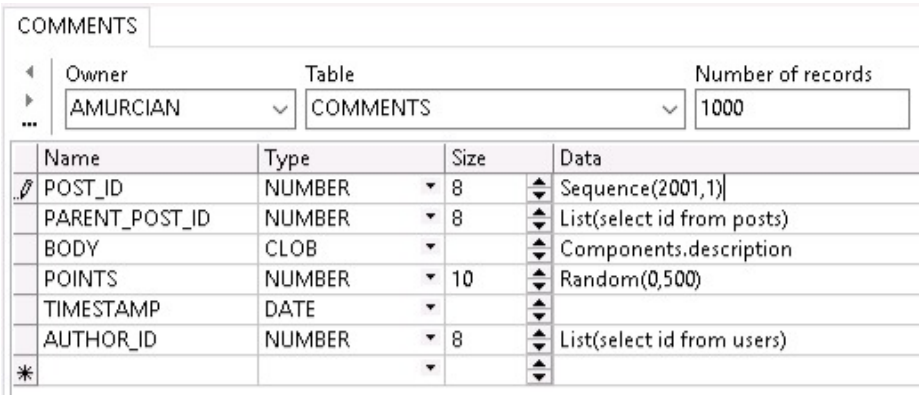

Figure 6: The configurations in the data generator tool for the comments table

```
# 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.

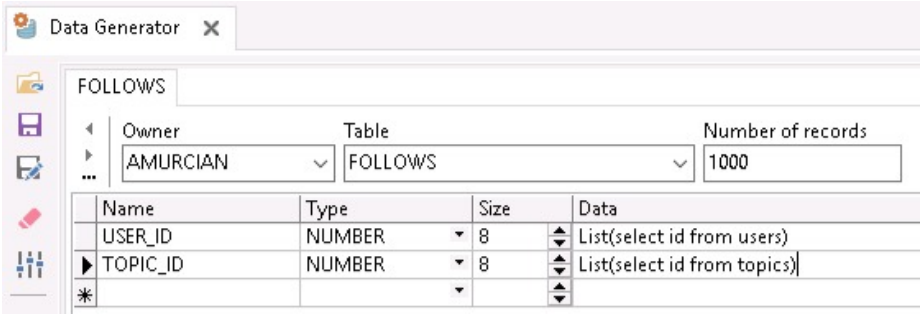

Figure 7: Settings used for generating the follows table data

1.7.9 Populating Relates To

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

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/tables/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

2.1 Queries

Now that we have built and populated our database, it is time that we use it to retrieve useful information for various users. Therefore we need to create some queries that will perform this task.

2.1.1 Query Descriptions

What follows is a description of some queries that will need to be retrieved from the database.

1. Perhaps the most frequent query that will be used is a search query. In essence, the query would be *obtain all questions whose title or body contain the provided words.*
2. Another query which will be often requested is one that retrieves questions to display on a user's home page. We can describe it as follows. *Retrieve the most popular questions which relate to a topic which a certain user follows.*
3. An alternative query to the previous one with a similar purpose can be to obtain new questions which have not yet been answered instead of popular ones. We can describe this one as follows. *Retrieve the newest unanswered questions which relate to a topic which a certain user follows.*
4. This query is one that will have to be performed every time a question is loaded. *Find all answers which answer the question with a particular ID.*
5. Similar to the previous one, whenever any post is loaded, we would need to query for all the comments on that post. *Select all comments whose parent post has a certain ID.*
6. A user may be interested in viewing all the answers that have been given to the questions they ask, together with any comments on any of their posts, in an 'inbox' fashion. *Obtain all answers to questions as well as all comments to posts which were posted by a certain user, sorted from newest to oldest.*
7. Ask Us is interested in sending out a weekly newsletter containing the most popular questions with one of their answers each from the previous week. A query which would obtain these questions would look like this. *Obtain a certain number of questions, as well as each of their highest voted answers, which were posted between two weeks ago and one week ago, and received the most votes within a week of their posting.*
8. Ask Us would like to display on a user's profile page all the questions, answers, and comments which they posted. *Retrieve all questions, answers, and comments whose author is a specific user, sorted by newest first.*

2.1.2 Writing the Queries in SQL

Next we must convert these queries into SQL code. Wherever user input would have been appropriate, we used a literal instead. The results obtained are logged in CSV files in the directory `/data/queries/`.

The first query searches for a certain search phrase which is provided by the user. Here it is in SQL. This query took 0.071 seconds to execute.

/sql/queries/search.sql

```
select *
from questions
where lower(questions.title) like lower('%word%')
        or lower(questions.body) like lower('%word%')
order by points desc;
```

The next queries retrieve information to display on a user's home page, sorted to show the most popular or the newest answers. They are displayed below in SQL. They took 0.015 seconds and 0.012 seconds to execute, respectively.

/sql/queries/home_page_popularity.sql

```
select * from (
        select title, substr(body, 1, 100), points, timestamp
        from questions q
        where exists (
                        select topic_id from follows
                        where user_id = 66
                        intersect
                        select topic_id from relates_to
                        where question_id = q.post_id
                )
        order by points desc
) where rownum <= 10;
```

/sql/queries/home_page_new.sql

```
select * from (
        select title, substr(body, 1, 100), points, timestamp
        from questions q
        where exists (
                        select topic_id from follows
                        where user_id = 66
                        intersect
                        select topic_id from relates_to
                        where question_id = q.post_id
                ) and not exists (
                        select post_id from answers
                        where question_id = q.post_id
                )
        order by timestamp desc
) where rownum <= 10;
```

Next we have the query that retrieves all the answers to a question. This one took 0.015 seconds to retrieve the results.

/sql/queries/answers.sql

```
select body, points, accepted, timestamp
from answers
where question_id = 1
order by points desc;
```

After that, we created the query that obtains the direct child comments of a particular post. This query took 0.01 seconds.

/sql/queries/comments.sql

```
select body, points, timestamp
from comments
where parent_post_id = 100
order by points desc;
```

The next query we have is the one that finds all answers for a particular user's questions. This one took 0.041 seconds.

/sql/queries/inbox.sql

```
select * from (
        (
                select
                        q.title subject,
                        to_char(substr(a.body, 1, 100)) preview,
                        a.timestamp
                from
                        answers a
                        inner join questions q
                        on a.question_id = q.post_id
                where
                        q.author_id = 11
        )
        union
        (
                select
                        q.title,
                        to_char(substr(c.body, 1, 100)) preview,
                        c.timestamp
                from
                        comments c
                        inner join questions q
```

```
                        on c.parent_post_id = q.post_id
                where
                        q.author_id = 11
        )
        union
        (
                select
                        to_char(substr(a.body, 1, 20)) subject,
                        to_char(substr(c.body, 1, 100)) preview,
                        c.timestamp
                from
                        comments c
                        inner join answers a
                        on c.parent_post_id=a.post_id
                where
                        a.author_id = 11
        )
        union
        (
                select
                        to_char(substr(p.body, 1, 20)) subject,
                        to_char(substr(c.body, 1, 100)) preview,
                        c.timestamp
                from
                        comments c
                        inner join comments p
                        on c.parent_post_id=p.post_id
                where
                        p.author_id = 11
        )
) order by timestamp desc;
```

The query which retrieves content for the newsletter is written in SQL here. This query took 0.44 seconds to run. Since at the time which we ran the query there was no data with a timestamp within the past seven days, we ran the query to retrieve data from the past year only for this test.

/sql/queries/newsletter.sql

```
select * from (
        select q.title, q.body question, a.body answer, a.points
        from questions q
        inner join answers a
        on q.post_id = a.question_id
        where a.post_id = (
                select * from (
                        select a2.post_id from answers a2
                        where a2.question_id = q.post_id
                        order by a2.points desc
                ) where rownum = 1
        )
        and q.timestamp >= sysdate - 7
```

```
        order by q.points desc
) where rownum <= 15
```

The last query we have is one that retrieves the content to display on a user's profile page. This query took 0.019 seconds.

/sql/queries/profile.sql

```
select * from (
        select title, points, timestamp
        from questions
        where author_id = 11
        union
        select to_char(substr(body, 1, 4000)), points, timestamp
        from answers
        where author_id = 11
        union
        select to_char(substr(body, 1, 4000)), points, timestamp
        from comments
        where author_id = 11
) order by timestamp desc;
```

2.2 Indexed Structures

2.2.1 Creating Indices

Our task is to create three indices on our tables in order to improve the time efficiency of the queries we wrote in the previous section. We have decided to create an index on the `points`, `timestamp`, and `author_id` columns of each of the posts' tables. This is because these columns are often the ones ordered by or the ones found in where clauses of the queries above.

Below are the SQL commands which we used to create the indices on all these columns.

/sql/indices/indices.sql

```
create index index_questions_points on questions(points);
create index index_answers_points on answers(points);
create index index_comments_points on comments(points);

create index index_questions_timestamp on questions(timestamp);
create index index_answers_timestamp on answers(timestamp);
create index index_comments_timestamp on comments(timestamp);
```

```
create index index_questions_author_id on questions(author_id);
create index index_answers_author_id on answers(author_id);
create index index_comments_author_id on comments(author_id);
```

2.2.2 Evaluating Index Efficiency

Now that we have indices, we are to check if they actually improve efficiency. Table 1 compares the average run time of five executions of each of the queries before and after the indices were added.

Query	Pre-Index	Post-Index
search	0.0702s	0.0616s
home_page_popularity	0.0240s	0.0198s
home_page_new	0.0136s	0.0130s
answers	0.0234s	0.0235s
comments	0.0232s	0.0214s
inbox	0.0864s	0.0810s
newsletter	0.0648s	0.0642s
profile	0.0220s	0.0214s

Table 1: Comparison of execution times before and after indices were created

As the table shows, adding the indices did in fact improve the efficiency of the queries, although the difference is not always significant.

2.3 Updating the Data

We now have to write some SQL commands that will manipulate the data we have stored.

2.3.1 Updating Two Records

We have written an SQL query that will update the email addresses of users 'watchfuleye' and 'werlwend'. This command is shown below.

/sql/update/emails.sql

```
update users
set email = username||'@yahoo.com'
where username = 'watchfuleye' or username = 'werlwend';
commit;
```

2.3.2 Deleting Two Records

Here is an SQL command that will delete two records from the follows table. This represents the fact that some user does not want to follow some topic any more.

/sql/update/unfollow.sql

```
delete from follows where
(user_id = 265 and topic_id = 2120)
or
(user_id = 267 and topic_id = 2327);
commit;
```

2.3.3 Updating Multiple Records With a Condition

When we populated the users table, we manually calculated the base 64 encoded SHA256 hash of the string 'password' and stored that as all the users' passwords. Now, we will use an SQL command to update the passwords of half of the users (those with an odd ID number) to the first part of their email (before the '@' symbol). Here we present said SQL command.

/sql/update/passwords.sql

```
update users
set password = substr(
        UTL_RAW.CAST_TO_VARCHAR2(
                UTL_ENCODE.BASE64_ENCODE(
                        STANDARD_HASH(
                                substr(
                                        email,
                                        1,
                                        instr(email, '@') - 1
                                ),
                                'SHA256'
                        )
                )
        ), 1, 43
)
where mod(id, 2) != 0;
commit;
```

2.4 Rollbacks

We are to write an SQL script which selects a subset of the data from one of our tables, then updates and displays the updated records. Then the script should

perform a rollback and display the results of the query again.

We have written a script, which is shown below, which selects all the users who have a username starting with the letter A. Thirty records were found which satisfy this condition. Then the script changes their email address to end in '@aol.com'.

/sql/rollback/rollback.sql

```
-- 1. Select the data
select * from users where username like 'a%';

-- 2. Update email endings
update users
set email = substr(email, 1, instr(email, '@') - 1) || '@aol.com'
where username like 'a%';

-- 3. Select the updated data
select * from users where username like 'a%';

-- 4. Undo the changes
rollback;

-- 5. Select the rolled back data
select * from users where username like 'a%';
```

The results of the first select statement before the update (statement 1) can be found in `/data/rollback/pre-update.csv` and the results of the last select statement after the rollback (statement 5) can be found in `/data/rollback/post-rollback.csv`. As expected, they are identical. The reason for this is that when we run the update statement, it is in some temporary state until the changes are committed. Since they are then rolled back, they are never committed, so by the end of the script the data has not changed at all.

Next, we are to run the same script, but instead of the rollback command, we are to commit the changes. This time, the statement five did show the email addresses all ending in '@aol.com'. These results can be seen in the file `/data/rollback/post-commit.csv`. (The results of statement one were identical to the results of statement one from the rollback script.) The reason that this time we do in fact see the updated records with the last query is because we committed the changes, so they have been permanently applied to the table.

Now we are told to grant access to the user `oracle00` and run the rollback script and the commit script as that user.

```
grant select on users to oracle00;
```

When we ran both scripts, statement two failed in both scripts since that user does not have permission to update records. We were shown an error message saying "insufficient privileges".

2.5 Constraints

2.5.1 Adding Constraints

We are tasked with creating and applying any constraints we deem appropriate to our database. We have already applied all appropriate unique constraints when creating the table, now we will go over all our tables and add check constraints where necessary.

For the users table, we must check that the email address provided is well formed. Meaning that it contains some text, followed by an @ symbol, followed by a domain. We also must ensure that the points are always positive. Here is the alter table command for the users table.

/sql/constraints/users.sql

```
alter table users add (
        constraint check_users_email
                check (email like '_%@_%._%'),
        constraint check_users_points
                check (points >= 0)
);
```

When looking at the questions, answers, and comments table, the only field which needed to be checked was once again the points field. Below are the alter table commands for all three.

/sql/constraints/questions.sql

```
alter table questions add
constraint check_questions_points
        check (points >= 0);
```

/sql/constraints/answers.sql

```
alter table answers add
constraint check_answers_points
        check (points >= 0);
```

/sql/constraints/comments.sql

```
alter table comments add
constraint check_comments_points
        check (points >= 0);
```

2.5.2 Testing the Constraints

To test that all our constraints are working correctly, we ran the following commands which would, if successful, violate some of our constraints. First we tried to add a user with an invalid email address.

/sql/constraints/tests/users.email.sql

```
insert into users (
        username, email, password
) values (
        'obi-wan-k3nobi',
        'kenobi@jedi-council',
        'XohImNooBHFR0OVvjcYpJ3NgPQ1qq73WKhHvch0VQtg'
);
```

This command was unsuccessful and gave the following error message. The reason for this is because the email address provided does not have a valid domain, since there is no dot (.) character.

```
ORA-02290: check constraint (AMURCIAN.CHECK_USERS_EMAIL) violated
```

Subsequently, we ran an update query to ensure that the points field was not able to become negative.

/sql/constraints/tests/questions.points.sql

```
update questions set points = -66 where post_id = 1;
```

Similarly, we obtained an error message as follows, since we do not allow negative values in the points field.

```
ORA-02290: check constraint (AMURCIAN.CHECK_QUESTIONS_POINTS)
violated
```

We also checked that unique constraints could not be violated by attempting to update a user's username to an already existing one.

/sql/constraints/tests/users.username.sql

```
update users set username = 'werlwend' where username = 'watchfuleye';
```

This time, we received a slightly different error message. The reason for the error was because there was already an existing user with the username 'werlwend'.

```
ORA-00001: unique constraint (AMURCIAN.SYS_C00574442) violated
```

Finally, we tested a foreign key constraint by deleting a post from the post table, which had both a child comment and its question record referencing it.

/sql/constraints/tests/posts.id.sql

```
delete from posts where id = 369;
```

Once again, as expected, we got an error message, because if the record would have been deleted, there would be a question whose post id would not be in the post table, and there would be a comment whose parent post would not exist.

```
ORA-02292: integrity constraint (AMURCIAN.SYS_C00574679) violated
 - child record found
```

2.6 Views

We are tasked with creating a script that creates two views, with two `select`, `insert`, `update`, and `delete` statements for each of the views.

2.6.1 Users view without passwords

The first view that we made is called `users_passwordless` and it is essentially the users table, but without the hashed passwords column. Its purpose is to maintain this sensitive information away from some of the less trusted employees at Ask Us who have no need to access it.

/sql/views/users_passwordless/users_passwordless.sql

```
create view users_passwordless as
        select id, username, email, points from users;
```

Below are two `select` statements which retrieve data from the view.

/sql/views/users_passwordless/select.sql

```
select * from users_passwordless where username = 'werlwend';
select * from users_passwordless where points >= 10;
```

Then we ran these two insert statements on the view. As expected, they did not succeed, and yielded the error message below.

/sql/views/users_passwordless/insert.sql

```
insert into users_passwordless (username, email)
values ('user1', 'user1@yahoo.com');
values ('user2', 'user2@yahoo.com');
```

```
ORA-01400: cannot insert NULL into ("AMURCIAN","USERS",
"PASSWORD")
```

Next, we tried to update the username of one user and the email of another with the following commands. These completed successfully.

/sql/views/users_passwordless/update.sql

```
update users_passwordless
set username = 'zeran1'
where email = 'zeran154335@gmail.com';

update users_passwordless
set email = 'werlwend1@yahoo.com'
where username = 'werlwend';
```

Finally we tried to remove two users from the view. We first had to find some users who were not referenced by a foreign key. Here are the commands we used to delete them.

/sql/views/users_passwordless/delete.sql

```
delete users_passwordless where id = 537;
delete users_passwordless where username = 'Rukawa';
```

2.6.2 Combination of all different post types

The second view is intended to combine all the posts, with their content into one 'table' `all_posts`.

/sql/views/all_posts/all_posts.sql

```
create view all_posts
        (id, body, points, timestamp, author_id, type)
as (
        select post_id, to_char(body), points, timestamp, author_id, 'Q' from questions
        union
        select post_id, to_char(body), points, timestamp, author_id, 'A' from answers
        union
        select post_id, to_char(body), points, timestamp, author_id, 'C' from comments
);
```

We have written the following two `select` statements to test this view.

/sql/views/all_posts/select.sql

```
select * from all_posts where points > 100;
select * from all_posts where body like '%you%';
```

We also ran a couple of insert statements on this view, expecting it to fail. Indeed we saw this error message.

/sql/views/all_posts/insert.sql

```
insert into all_posts (body) values        ('Test 1');
insert into all_posts (body) values        ('Test 2');
```

```
ORA-01732: data manipulation operation not legal on this view
```

Next we ran two update commands, as shown below, but of course, we received the same error.

/sql/views/all_posts/update.sql

```
update all_posts set points = 1 where points <= 0;
update all_posts set timestamp = sysdate where id = 66;
```

Finally, we tried to delete some rows from the view, but once again, we received the same error as above.

/sql/views/all_posts/delete.sql

```
delete all_posts where id = 1;
delete all_posts where author_id = 2;
```

3 Third Stage

3.1 Extending SQL

This stage is about using PL/SQL to extend the SQL we have used so far. We have two tasks for this section. Firstly, we must write a script that uses PL/SQL and demonstrates a cursor loop, and before and after triggers.

First of all, our script uses a cursor loop to display all the topics that each user follows.

/sql/plsql/loop.sql

```
declare cursor c is
        select username, label
        from follows
                inner join users on follows.user_id = users.id
                inner join topics on follows.topic_id = topics.id
        order by username;

begin
        prev_user varchar(20) := '';
        dbms_output.enable(1000000);
        for row in c
        loop
                if row.username = prev_user then
                        dbms_output.put(', ' || row.label);
```

```
                else
                        dbms_output.new_line;
                        dbms_output.put(row.username || ': ' || row.label);
                        prev_user := row.username;
                end if;
        end loop;
end;
```

This script gives us an output like this. (Only the first ten lines are shown here.)

/sql/plsql/loop_output.txt

```
-Sphinx-: Student Exchanges
-otaku-sama-: Disease Transmission, Drug Patents, Word Lists
-vania-: Fields Medal
Omarakagami: Sports Clubs
5hortie: Locations, Family Interdisciplinary Topics, Medicine and Healthcare in
Mexico
AAA: Preventive Medicine
AX3M: Sport Organizations
AbandonAccount2: Korean Food, Medical Protocols
Acation: Family Interdisciplinary Topics, International Conflicts
AcidHearts: Specific Types of Schools
```

Then our script uses before insert and update triggers to store the given password as the base 64 encoded SHA256 hash, rather than as the given plain text.

/sql/plsql/triggers/hash_password.sql

```
create trigger hash_password
before insert or update
   on users
   for each row
declare
        hashed_pass varchar(43);
begin
        if inserting or :old.password != :new.password then
                select substr(
                        utl_raw.cast_to_varchar2(
                                utl_encode.base64_encode(
                                        standard_hash(
                                                :new.password,'SHA256'
                                        )
                                )
                        ), 1, 43
                )
```

```
                into hashed_pass from dual;
                :new.password := hashed_pass;
        end if;
end;
```

Following this, we tested the trigger with an insert and an update statement (shown below), and they both resulted in storing the correct hashed password.

/sql/plsql/triggers/test_hash_password.sql

```
insert into users
        (username, email, password)
values
        ('spongebob', 'bob@krustykrab.bb', 'password');

update users
set password = 'spongebob'
where username = 'spongebob';
```

For our 'after' trigger, we found that whenever someone votes on a post, as well as inserting a record into the votes table, we also need to increment the points of the post that was voted on, as well as the points of the author of the post. We therefore created a trigger on the votes table that would do this automatically.

/sql/plsql/triggers/points_distributer.sql

```
create or replace trigger points_distributer
after insert on votes
for each row
begin
        update users
        set points = points + 1
        where id = (
                select author_id
                from all_posts
                where id = :new.post_id
        );

        update questions
        set points = points + 1
        where post_id = :new.post_id;

        update answers
        set points = points + 1
        where post_id = :new.post_id;
```

```
        update comments
        set points = points + 1
        where post_id = :new.post_id;
end;
```

We then tested it with the insert statement shown below, and indeed, the post and the user both received an additional point.

/sql/plsql/triggers/test_points_distributer.sql

```
insert into votes (user_id, post_id)
values (755, 2072);
```

3.2 Writing a Full PL/SQL Program

In this section we are tasked with creating a PL/SQL package. We have chosen to create a package which will contain some procedures to manage some user data. Below are the package specification and the body.

In our package we have a function `hash_password` which returns a base 64 encoded SHA256 hash of the input string, capable of converting any user's password into the form stored in the database, both for comparison and storage purposes.

Additionally we have a `check_password` function, which uses the aforementioned function to check if an input password matches the hash stored in the database for a particular user.

Finally, we have a `change_password` procedure, which takes a user ID, their old password, and a new password. First it checks that the old password is correct using the previous function, then it uses a cursor to loop through all users who use the same password, and print a warning for each of these users saying that said user uses the same password (which is a catastrophic idea regarding security, but for the purposes of demonstrating a cursor loop, it splendidly suffices).

/sql/plsql/package/specification.sql

```
create package user_management as
        function hash_password(
                password in users.password%type
        ) return users.password%type;

        function check_password(
```

```
                uid in users.id%type,
                password in varchar
        ) return boolean;

        procedure change_password(
                id users.id%type,
                old_pass varchar,
                new_pass varchar
        );
end user_management;
```

/sql/plsql/package/body.sql

```
create package body user_management as
        function hash_password(password in users.password%type)
        return users.password%type is hashed_pass users.password%type;
        begin
                select substr(
                        utl_raw.cast_to_varchar2(
                                utl_encode.base64_encode(
                                        standard_hash(
                                                password, 'SHA256'
                                        )
                                )
                        ), 1, 43
                )
                into hashed_pass from dual;
                return hashed_pass;
        end;

        function check_password(
                uid in users.id%type,
                password in varchar
        ) return boolean is
                hashed_pass users.password%type;
        begin
                select password into hashed_pass
                from users where id = uid;

                if hash_password(password) = hashed_pass then
                        return true;
                else
                        return false;
                end if;
        end;

        procedure change_password(
                id users.id%type,
                old_pass varchar,
                new_pass varchar
        ) is
                hashed_pass users.password%type := hash_password(new_pass);
                cursor c_users is
                        select username from users
```

```
                        where password = hashed_pass;
        begin
                if check_password(id, old_pass) = false then
                        dbms_output.put_line('Error: Invalid password.');
                        return;
                end if;

                for r_user in c_users
                loop
                        dbms_output.put_line(
                                'Warning: User '
                                || r_user.username
                                || ' already has this password.'
                        );
                end loop;

                update users u set password = hashed_pass
                where u.id = id;
        end;
end user_management;
```

4 Fourth Stage

In the fourth stage we must create an application which will access the database. We considered creating a web based application, but decided that due to time constraints, a command line application would be more manageable.

We have created one using Python, which can be found in the directory `/app`. Since we were unable to connect to the Oracle database remotely, we had to port over the database to another DBMS. We chose SQLite3 for this, since it is by far the easiest to set up. The database can also be found in the same directory.