Project Report for Database Systems
Vanessa Job and Xioameng Li

1.1 - Description of database.

Our database facilitates the management of a small company hosting a game called the Song Game[1]. During the Song Game, users take turns entering names of songs. Each player must match at least one word of the prior player's song to keep playing. Words match if they are the same word, a word that is the same up to the addition or removal of a suffix, or homophones. For instance 'which" matches "witch". Sample game play follows:

Player 1: Elephant Talk

Player 2: Talking Dreams ("talking" matches "talk")

Player 1: Beautiful Dreamer ("dreamer" matches "dream")

Player 2: Scars to your Beautiful ("beautiful" matches "beautiful")

Player 1: What makes you beautiful ("beautiful" matches "beautiful")

Player 2: What does the fox say? ("what" matches "what")

Player 1: Foxy lady ("foxy" matches "fox")

Player 2: Lady in Red ("lady matches "lady")

Player 1: Red solo cup ("red matches red")

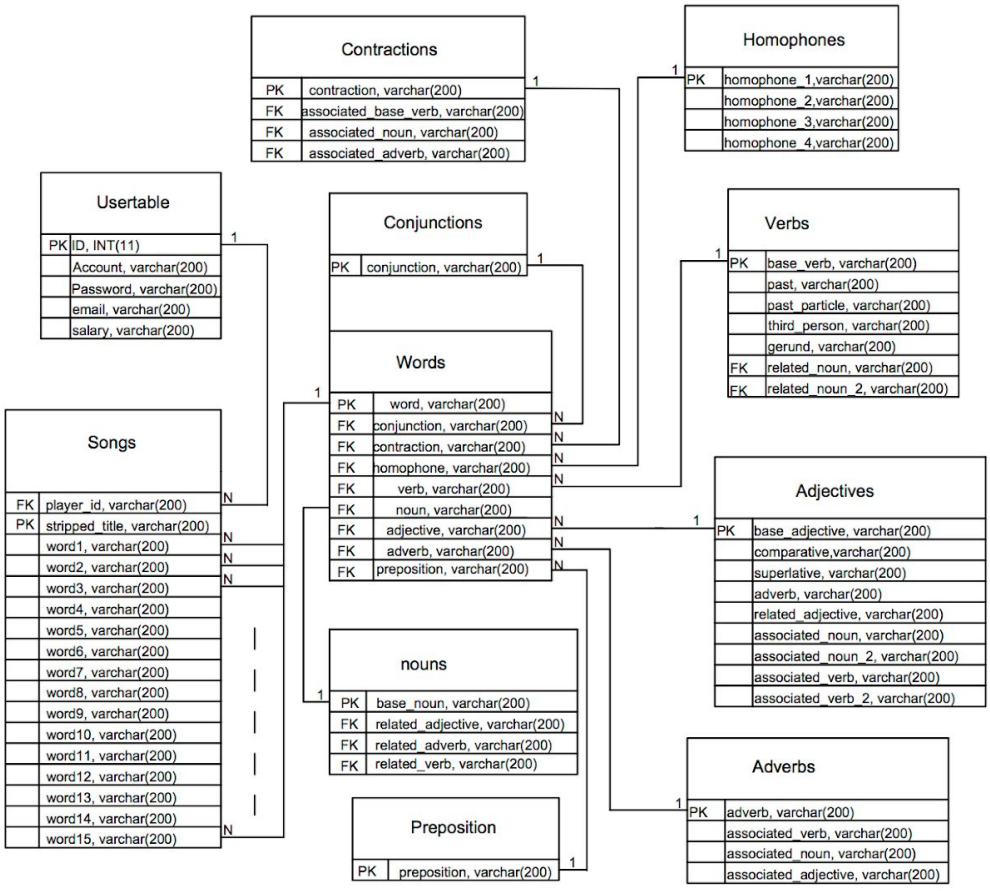
Player 2: Red lights ("red matches red")

The company has a supervisor (called the Boss) and employees. Tables record basic information on all employees, including salary. Tables also record the course of the game to insure that duplicate songs are not entered.

1.2 and 1.4 - ER diagram and Relational Schema

Two diagrams follow: ER diagram with data types and a relational schema.

[1] The Song Game was invented by Vanessa's sister, Melissa Job Perry.


Contractions
PK contraction, varchar(200)
FK associated_base_verb, varchar(200)
FK associated_noun, varchar(200)
FK associated_adverb, varchar(200)
Homophones
PK homophone_1,varchar(200)
homophone_2,varchar(200)
homophone_3,varchar(200)
homophone_4,varchar(200)
Usertable
PK ID, INT(11)
Account, varchar(200)
Password, varchar(200)
email, varchar(200)
salary, varchar(200)
Conjunctions
PK conjunction, varchar(200)
Verbs
PK base_verb, varchar(200)
past, varchar(200)
past_particle, varchar(200)
third_person, varchar(200)
gerund, varchar(200)
FK related_noun, varchar(200)
FK related_noun_2, varchar(200)
Words
PK word, varchar(200)
FK conjunction, varchar(200)
FK contraction, varchar(200)
FK homophone, varchar(200)
FK verb, varchar(200)
FK noun, varchar(200)
FK adjective, varchar(200)
FK adverb, varchar(200)
FK preposition, varchar(200)
Songs
FK player_id, varchar(200)
PK stripped_title, varchar(200)
word1, varchar(200)
word2, varchar(200)
word3, varchar(200)
word4, varchar(200)
word5, varchar(200)
word6, varchar(200)
word7, varchar(200)
word8, varchar(200)
word9, varchar(200)
word10, varchar(200)
word11, varchar(200)
word12, varchar(200)
word13, varchar(200)
word14, varchar(200)
word15, varchar(200)
Adjectives
PK base_adjective, varchar(200)
comparative,varchar(200)
superlative, varchar(200)
adverb, varchar(200)
related_adjective, varchar(200)
associated_noun, varchar(200)
associated_noun_2, varchar(200)
associated_verb, varchar(200)
associated_verb_2, varchar(200)
nouns
PK base_noun, varchar(200)
FK related_adjective, varchar(200)
FK related_adverb, varchar(200)
FK related_verb, varchar(200)
Adverbs
PK adverb, varchar(200)
associated_verb, varchar(200)
associated_noun, varchar(200)
associated_adjective, varchar(200)
Preposition
PK preposition, varchar(200)

2.3 - Indices

We chose to build an index on the `usertable` because if the game became popular, the `usertable` could grow large. All of the tables containing parts of speech, nouns, verbs, conjunctions, etc. are relatively small tables - definitely fewer than 10,000 entries. We investigated building an index on the email column of `usertable` because one might want to look up a user account by email if a user had forgotten their account name or password. We built a table of 20000 fake users for the `usertable` to test this out. We then tested the query

SELECT * FROM usertable WHERE email like '%user5008@gmail.com'

before adding an index. The query took 0.0028 seconds:

Then we built an index on the email column and ran the same query. The query took 0.0224 seconds, slightly less time.

There was not much difference in the time the query took with and without the index. We theorize that if the user table contained 100,000 users, an index would be really worth the effort. Also, we wonder if the performance of the index was affected by the email addresses being similar. Perhaps the index would yield a greater speed up if there was more variety among the email addresses.

We tried this experiment again looking for an email address that was not in the table:

SELECT * FROM usertable WHERE email like 'bozo@theclown.com'

This time we got a little more encouraging results. Without the index, the query took 0.0195 seconds.

With the index, the query took 0.0208 seconds:

Since MySQL builds a B-tree to construct an index, we can rationalize these results. Depending on how the B-tree is constructed, it may be fairly quick to figure out that an item is not in the table, whereas finding an item that is in the table might require recursing all the way down the tree.

It would make sense to build an index on the `Account` names of the `usertable` as well. Since one would expect similar results to above, we decided to experiment on the `songtable` for variety. We built a table of 20,000 dummy songs and did the following query to look at the first word of the songs:

```
SELECT * FROM songtable WHERE word1 LIKE 'buffalo";
```

Without an index on `word1`, this query took 0.256 seconds:

When we added an index on `word1`, this query took 0.0003 seconds:

After an index was added, the query took far less time. Is this because the word 'buffalo' was not in the table? When time permits, we would like to do a more extensive investigation of indices.

PHP PART 3

3.1.a - You must allow users to access and update all appropriate tables. This requirement is vague because each database will have unique requirements. For all tables, you must write a statement as to why you allowed or did not allow access.

In what follows, we describe who has access to each table and the type of access they have. Please note that players do not have direct access to any tables.

Adjective Table

Boss: access to select, update, insert and delete table

Employees: access to insert and update

Bosses and employees should have the capacity to add new words to the adjective table because a song may be found that contains words not in the table. Players should not have this capacity. We do not want players added fictitious or offensive words.

Bosses could use select to see everything in adjective table, update the `base_adjective` column, insert the `base_adjective` column, and delete any row. Employees could update the `base_adjective` column or insert into `base_adjective` column.

Adverbs Table

Boss: access to select, update, insert and delete table

Employees: access to insert and update

Similar to above, we want the bosses and employees to have the ability to modify these tables. Boss could select to see adverb column in adverb table, update the adverb column, insert the adverb column, and delete any row. Employee could update the adverb column or insert into adverb column.

Conjunctions, Contractions, Homophones, Prepositions

Bosses: have no access

Employees: have no access

These parts of speech are stable in English. No one should need to change these tables and to prevent inadvertent errors, we do not allow any users access to these tables.

Nouns Table

Boss: access to select, update, insert and delete table

Employees: access to insert and update

Boss could select to see everything in base_noun column, update base_noun column, insert into base_noun column or delete any row. Employee could update the base_noun column or insert into base_noun column.

Verb Table

Boss: access to select, update, insert and delete

Employee: access to insert and update

Boss could select to see `base_verb`, past and past_participle three columns of verb table, update any `base_verb` column, insert new word into `base_verb` column, or delete any row. Employees could update the `base_verb` column or insert into `base_verb` column.

Song Table

Boss: have no access

Statements: The song table is a table used for the game. Neither the boss nor he employee needs to be able to affect the outcome of the game. Note: every time a game ends, we will truncate the song table so that it is ready for the next time the game is played.

Users Table

Boss: access to select, update, insert, and delete

Employee: access to insert and update

Boss could select to see everything in the user's information. He or she can update anyone's salary including his or her own, insert any new person with all of their information, and delete anyone from user table.

Employees could only update their own email in the user table. They may not update their own salary. Employees could insert into user table a new member with all the new information in (account, password, email, salary) columns.

Words Table

Boss: has no access

Employee: has no access

Statements: The word table is a table used for the game. No one needs to be able to alter the course of the game. Note: at the end of the game, this table gets truncated so that next game, players have a fresh start.

3.1.a.1 - Groups will be required to implement multi-user access and transaction-safe

database interaction.

In this project, there are three levels of user access: Boss, Employee and Player. Each of these may have a different level access to each table. The various levels of access are described above.

3.1.b - You must implement security features to prevent unauthorized access and unauthorized changes to your database.

The command injection prevention we did is simple. First, for every input delivered by PHP, we use the following code to "kill" all characters from string except A-Z as well as "0-9" and "@".

$string_input = preg_replace("/[^a-z0-9@.]+/i", " ", $string_input);

Second, whatever command we used for the connection between SQL and PHP, we add a **'** to make every message be and only be a harmless string.

```
INSERT INTO table (xx) VALUES ' $string_input'; OR:

UPDATE table SET xx = ' $string_input' where ,,,,,;
```

With these two lines of codes, all the dangerous input "the one with <drop> sentence" will be treated as a normal string input.

Another function to do this:

```
function inputfilter($input){

  $input = str_replace("", "\'", $input);

  return $input;

}
```

We used both methods during the project.