ECE 356: Database Systems

Group Assignment

Final Report

By:

Dhruv Chopra (dchopra)

James Urquhart (jaurquha)

Tong Jia (tjia)

**Schema** DHRUV

*DROP TABLE IF EXISTS FriendRequest;*

*DROP TABLE IF EXISTS Friend;*

*DROP TABLE IF EXISTS gName;*

*DROP TABLE IF EXISTS WorkAddress;*

*DROP TABLE IF EXISTS Review;*

*DROP TABLE IF EXISTS DoctorSpecialization;*

*DROP TABLE IF EXISTS Doctor;*

*DROP TABLE IF EXISTS Patient;*

*DROP TABLE IF EXISTS gUser;*

*DROP VIEW IF EXISTS FullPatientView;*

*DROP VIEW IF EXISTS DoctorReviewsMetaData;*

*DROP VIEW IF EXISTS FullDoctorView;*

*---------------------------Reinsert all tables------------------------------------*

*DROP TABLE IF EXISTS gUser;*

*CREATE TABLE gUser(*

*gAlias VARCHAR(34) PRIMARY KEY,*

*EmailAddress VARCHAR(66),*

*PasswordHash VARCHAR(66),*

*PasswordSalt VARCHAR(66)*

*);*

*DROP TABLE IF EXISTS Doctor;*

*CREATE TABLE Doctor(*

*gAlias VARCHAR(34) PRIMARY KEY,*

*Gender VARCHAR(8),*

*YearMedicalLicense INT,*

*FOREIGN KEY (gAlias) REFERENCES gUser(gAlias)*

*);*

*DROP TABLE IF EXISTS Patient;*

*CREATE TABLE Patient(*

*gAlias VARCHAR(34) PRIMARY KEY,*

*City VARCHAR(34),*

*Province VARCHAR(34),*

*FOREIGN KEY (gAlias) REFERENCES gUser(gAlias)*

*);*

*DROP TABLE IF EXISTS WorkAddress;*

*CREATE TABLE WorkAddress(*

*gAlias VARCHAR(34),*

*Street\_Name VARCHAR(34),*

*Street\_Number VARCHAR(10),*

*Apt\_Number VARCHAR(8),*

*PostalCode VARCHAR(8),*

*City VARCHAR(34),*

*Province VARCHAR(34),*

*FOREIGN KEY (gAlias) REFERENCES Doctor(gAlias)*

*);*

*DROP TABLE IF EXISTS DoctorSpecialization;*

*CREATE TABLE DoctorSpecialization(*

*gAlias VARCHAR(34),*

*SpecializationName VARCHAR(66),*

*FOREIGN KEY (gAlias) REFERENCES Doctor(gAlias),*

*PRIMARY KEY(gAlias , SpecializationName )*

*);*

*DROP TABLE IF EXISTS FriendRequest;*

*CREATE TABLE FriendRequest(*

*RequestergAlias VARCHAR(34),*

*RequesteegAlias VARCHAR(34),*

*FOREIGN KEY (RequestergAlias) REFERENCES Patient(gAlias),*

*FOREIGN KEY (RequesteegAlias) REFERENCES Patient(gAlias),*

*PRIMARY KEY(RequestergAlias , RequesteegAlias )*

*);*

*DROP TABLE IF EXISTS Friend;*

*CREATE TABLE Friend(*

*PatientgAliasA VARCHAR(34),*

*PatientgAliasB VARCHAR(34),*

*FOREIGN KEY (PatientgAliasA) REFERENCES Patient(gAlias),*

*FOREIGN KEY (PatientgAliasB) REFERENCES Patient(gAlias),*

*PRIMARY KEY(PatientgAliasA , PatientgAliasB )*

*);*

*DROP TABLE IF EXISTS Review;*

*CREATE TABLE Review(*

*ReviewContentID MEDIUMINT PRIMARY KEY AUTO\_INCREMENT,*

*PatientgAlias VARCHAR(34),*

*DoctorgAlias VARCHAR(34),*

*StarRating FLOAT,*

*gDate DATETIME,*

*Text VARCHAR(1002),*

*FOREIGN KEY (PatientgAlias) REFERENCES Patient(gAlias),*

*FOREIGN KEY (DoctorgAlias) REFERENCES Doctor(gAlias)*

*);*

*DROP TABLE IF EXISTS gName;*

*CREATE TABLE gName(*

*gAlias VARCHAR(34) PRIMARY KEY,*

*FirstName VARCHAR(34),*

*MiddleName VARCHAR(34),*

*LastName VARCHAR(34),*

*FOREIGN KEY (gAlias) REFERENCES gUser(gAlias)*

*);*

*DROP VIEW IF EXISTS FullDoctorView;*

*CREATE VIEW FullDoctorView AS*

*SELECT gUser.gAlias, Gender, YearMedicalLicense,EmailAddress,FirstName,MiddleName,LastName from Doctor INNER JOIN gUser ON (Doctor.gAlias=gUser.gAlias) INNER JOIN gName ON (gName.gAlias=gUser.gAlias);*

*DROP VIEW IF EXISTS DoctorReviewsMetaData;*

*CREATE VIEW DoctorReviewsMetaData AS*

*SELECT COUNT(DoctorgAlias) as ReviewCount, AVG(StarRating) as AvgStarRating, DoctorgAlias FROM Review GROUP BY DoctorgAlias;*

*DROP VIEW IF EXISTS FullPatientView;*

*CREATE VIEW FullPatientView AS*

*SELECT gUser.gAlias, EmailAddress,FirstName,MiddleName,LastName,Province,City from Patient INNER JOIN gUser ON (Patient.gAlias=gUser.gAlias) INNER JOIN gName ON (gName.gAlias=gUser.gAlias);*

**Changes to design since deliverable 1 (initial report)**

We did not make any changes to our ER diagram. However, we did make a slight change to the WorkAddress table. We were first using a WorkAddress ID to make every ID unique. However, now we are simply using a doctor’s alias to refer to their work address ID. In effect we do not have a primary key for that table, just a foreign key (doctor’s alias). This is because the work address ID was not necessary since multiple doctors can have the same work address. Also, we only ever need to filter that table on a doctor’s alias and working location. An ID would be of no benefit in such situations. Apart from that our constraints remain the exact same.

We did however make a lot of changes to our queries and Views. This is because our queries were not working completely before (there were some syntax errors and it was not picking distinct data in all cases). Hence we decided to redesign our queries and view so that all operations are completed properly.

As far as views go, we have defined three of them that we will use for all of our queries. The first is the FullDoctorView and this view will list all of the statistics for all the doctors except for their specializations, work addresses and reviews. This is because we felt that this view is used throughout several queries and so it will be beneficial to have it in one place for convenience when writing the queries.

Similar to the DoctorFullView, we have the PatientFullView for all the same reasons.

The next view is the DoctorReviewsMetaData view. This view will give information such as a doctor’s average review score and the number of reviews they have. This view is also used in multiple locations (while searching for a doctor with a specific review rating or higher, and while loading a doctor’s profile information).

The syntax for creating all of these views can be found in the SQL DDL presented in the previous section.

Finally, since we did not use the views that we set described in the first deliverable, we had to rewrite all of our operations. All operations occur in stored procedures and are almost all executed in one SELECT statement (if the nature of the operation is search). The SQL DDL for the operations is not provided here, but the names of the stored procedures are. These stored procedures are included along with the code that is submitted on LEARN.

**Operation 1** is done using stored procedure ‘patientToPatientSearch’.

**Operation 2** is done using stored procedure ‘Friendship’.

**Operation 3** is done using stored procedure ‘viewMyFriendRequests’.

**Operation 4** is done using stored procedure ‘superDoctorSearch’.

**Operation 5** is done using stored procedures ‘getDoctorProfileInfo’, ‘getDoctorSpecialization’, ‘getDoctorWorkAddresses’, and getDoctorReviewStats’ (the application side controls the visibility of any columns … for example emailAddress column won’t show on the page if the user is a patient) .

**Operation 6** is done using stored procedure ‘getReviewAndNeighbors’.

**Operation 7** is done using stored procedure ‘writeDoctorReview’.

**Operation 8** is done using stored procedure ‘getDoctorProfileInfo’, ‘getDoctorSpecialization’, ‘getDoctorWorkAddresses’, and getDoctorReviewStats’ (the application side controls the visibility of any columns … for example emailAddress column won’t show on the page if the user is a patient).

**Third-party software**

The project was completed using only the NetBeans IDE, with direct use of MySQL and Java Web technologies. Limiting the amount of technologies used to this was done to reduce complexity, allowing focus to be placed on the core components, rather than clutter the project with unnecessary additions. The more technologies added to a project, the less portable/maintainable the project becomes due to the ramp-up time required for new developers/team members to become familiar with the technology. This alternative does result in a higher setup time since a lot of components need to be created manually before the core functionality can be developed.

Although no additional technologies were used, one type of technology that would have been helpful in the development of the project is Object Relational Mappers or ORMs. An ORM would have allowed the Data Transfer Object classes to be automatically generated, based on the table definitions in the Database. In other words, the ORM would have generated a Java class for each table in the DB, with field names and types corresponding to columns in each table. Due to unfamiliarity with ORM in the Java environment, it was deemed that implementing such a technology would be too time consuming relative to the number of business entities/tables in the database. In larger or more complex projects, an ORM would have paid dividends over the long run.

**Interface design**

In our GUI implementation, we designed it so that the model-view-controller (MVC) pattern was followed. Our strategy was to separate the web application into three distinct layers, from top to bottom: the view, the controller, and the model. Each layer serves its own purpose: the view layer determines how the data is laid out, and how it is displayed; the controller decides what the user’s input was, how the model needs to change as a result of that input, and which resulting view should be used; the model is the database that is used to store and manipulate data.

The view layer mainly consists of the JSP (JavaServer Pages) and HTML (HyperText Markup Language) files. We typically have either a JSP or an HTML page for each part of the GUI. They provide an interface to clearly display for and accept data from the user. They only interact a subset of the data from the database, with logic usually only consisting of for loops for displaying information in a repetitive manner. We use a combination of simple prints, lists, and tables to display data, and use forms, drop down menus (using HTML’s ‘SELECT’ element) and buttons to interact with the user and retrieve data.

The controller layer consists of the JAVA files. Here, we defined various classes and servlets that contain most of the logic in our GUI implementation. The servlets contain the flow of traffic in the application. We also define business entities as separate objects, for example a Name object consists of the firstName, middleName, and lastName fields. Using this, we were able to control exactly how data is transferred (and not transferred) between the view layer and the model layer. With the use of Java interfaces, a seam was created between the controller layer, and the data provider. This interface defined methods for fetching business entities, where the controller layer was agnostic of the source of the information. This seam was called DBAO, or Database Access Object. Once the controller layer pulled the data from the DBAO, the information was passed to the view layer using standard web technologies such as request attributes. The controller layer was also responsible for the flow of traffic, and user restrictions. Further detail on these responsibilities is outlined in the Access Control section of this report.

The model layer was comprised entirely of the DBAO contract, and its implementations. During development, we had two implementations of the DBAO. In order to develop both the front end and the database component simultaneously, a mock DBAO was used to provide static testing data. This mock DBAO abided by the DBAO contract required in the final product. This allowed the controller class to be independent of the DBAO implementation, and allowed seamless transition to the real DBAO implementation once the SQL DB backend was developed and tested. The final DBAO implementation contained all the calls to the SQL DB. This contained all the connection code, as well as executing the stored procedures on the DB.

**Security and access control** James

**SQL injection prevention** James

SQL injection was prevented by limiting interaction with the database to stored procedures. Using Java’s SQL package, all database calls were made using the java.sql.CallableStatement object. This class populates arguments for the stored procedure at run time, and sanitizes user input. Since all interactions are limited to stored procedures, there is no exposed entry point for SQL injection attacks. Additionally, all database calls were limited to a single class, the final DBAO implementation.

**Password hashing and salting** James

The password hashing function implemented for this lab was built upon SHA2, with 256 bit hash output space, or 64 characters. SHA2 provides sufficiently distributed outputs from marginally different inputs. In order to prevent rainbow table attacks, where the attacker has pre-computed hash values, the hash function was performed on the raw password concatenated with a stored salt. This results in distinct password hashes for users even if they all use the same original password. This prevents users with the same password from having identically hashed output values stored in the database. We wrote custom function to calculate the password hash. We decided on a custom function over other cryptographic functions that use password salting in order to reduce complexity and prevent performance costs. Cryptographic functions such as bcrypt and scrypt provide additional security against rainbow table and brute force attacks. Since the system being designed was for academic purposes only, we decided that the cost of implementing a more complex algorithm was orthogonal to the learning objective of the lab. In other words, the system would never be exposed to complex attacks. Again we decided to use a simple solution for generating salts. Salts were a combination of the term “salt\_” and the user’s alias. Since the user’s alias was guaranteed to be unique as defined in our DB, our salts where likewise guaranteed to be unique. A random number generator would also produce sufficient uniqueness for any two given salts, but required additional complexity which provided little to no additional benefits.

**Access control**

We used various methods to ensure that a particular user is authorized to perform a particular action in the GUI.

First, we implemented our JSP and HTML files in a way so that they only work with data given to them. Since these files are used to format and display information to the user, by controlling what data is given to them controls what data is served to the user. In other words, JSPs provided with no data do not contain any revealing information about the state of the system or information in our database.

Next, in order to make sure that the correct data is passed on to the JSP and HTML files, we created servlets corresponding to each webpage. The servlets contain the logic to retrieve data from the database, send the data to the JSP and HTML files, and most importantly, logic to gate the data from unauthorized users.

Since servlets are defined in JAVA, we were free to use any logical expressions to determine if a given user was authorized to view a particular page. In order to prevent having to login for each action, the user’s alias and user type was stored on the server side using session IDs. These unique IDs were stored client side as cookies. Session state was controlled using Java web’s HTTPSession class. Since the gating process was used between several servlets, a separate static class was defined to allow us to quickly assert a user’s alias or UserType, the SessionAssert class.

When a user is first logged in successfully, we store their user alias and user type in the session in the following way:

                       UserType  userType = DBAOSingleton.getInstance().validateLogin(alias, password);

                        //Cache succesful login to session

                       if(userType != UserType.Invalid){

                           SessionAssert.SetLoggedInSession(request.getSession(), alias, userType);

                    }

When a servlet was given a request, it could freely query the SessionAssert to ensure that authorized user’s and usertypes were abided by. In most cases if access was deemed unauthorized, the servlet responded by redirecting the request to the LoginServlet:

       if(!SessionAssert.AssertUserType(request.getSession(), DBAO.UserType.Patient)){

            url = "/LoginServlet";

       } else {

                       // protected actions

If the user failed the validation, the user will be sent back to the LoginServlet, which handles the user authentication process. There will be two cases in this situation: if the user was logged in (session info was found), but performed an action not allowed by the user’s userType, for example patient pasted a view-doctor-full-profile url into the browser, the user would be redirected to the initial landing page after a login; if the user was never logged into the system (no session info was found), then the user would be redirected to the login page with a friendly warning. An example of this redirection can be seen below:

           String loggedinAlias = SessionAssert.getLoggedInUserAlias(request.getSession());

               if(loggedinAlias != null){

                   UserType userType = SessionAssert.getLoggedInUserType(request.getSession());

               url = RedirectByType(userType, request, loggedinAlias );

            } else {

                   //pull the login values from req and validate

                   String alias = request.getParameter("alias");

                   String password = request.getParameter("password");

                   if(alias == null || password == null){

                                       url = "/index.jsp";

                                       request.setAttribute("errorMessage", "Invalid Login Credentials. Try Again");

                   } else {

                                      // normal login procedure

                   }

            }

Finally, when the user logs out of the system, we clear the session info so that no one else can erroneously have access to any data they shouldn’t have access to.

With this access control implementation in place, we are able to confidently block actions such as patients accessing the interface for doctor by copying and pasting a URL and vice versa, and patients accessing a full doctor profile with the doctors email, etc.

**Indexing strategy**

We used InnoDB as our database engine for this lab because our application is not time critical (since it is a regular web application). Hence using an in-memory DBMS did not make sense from a design perspective because our application requirements did not need a particularly fast application. The benefit from using a disk based storage engine allowed us more efficient use of the server’s memory.

Due to our storage engine choice, we did not have a choice on the type of index we could use. Between HASH and BTREE, we could only pick BTREE because HASH is not supported by InnoDB.

The only indices provided to us automatically were the primary keys of the tables involved (for each query). A large portion of the application was based around alias’ of users. Alias’ were stored in our database as primary keys, thus we did not require explicit indexing for a fair number of queries.

For indexing, we did not have to explicitly declare a lot of indices. This is because we designed our major queries to run on primary keys. Hence, we get indexing “for free” on all of those queries. It is possible to further speed up the queries by further indexing a query. However, there is no guarantee on how many of those indices will be beneficial to the query on any given run. For example, our flexible doctor search query executes and returns all data in one long query. We could index all the columns that the user specifies to speed up the query. However, there is no way of knowing how many columns the user will specify on any given run and so the extra indices may or may not be pointless.

Overall, we are satisfied with our indexing strategy because we designed our queries to run on primary and foreign keys (mainly the user alias) and so we guaranteed ourselves fast queries and automatic optimization by the database. If this system were a real world application, business analytics could be used to find the search fields used in the Doctor and Patient search that are most commonly used. This would allow targeted use of indexes and provide reasonable performance gains well limiting index space and maintenance time.

**Concurrency**

Transactions were required anywhere multiple statements are intrinsically tied to one another. This commonly occurred in our database when an update and delete needed to occur simultaneously. In other words, whenever more than one database altering statement is executed and they are both tied to one another, then we must execute all those statements in one transaction. For example, we use a stored procedure called add\_doctor. This procedure takes in all of the data required for the doctor as a user and as a doctor. These values are inserted into multiple tables in the database. For example, the doctor’s name is entered into the Name table and the doctor’s gender is inserted into the Doctor table. We do not want a case where the doctor’s name is inserted, but the gender fails to insert. Hence we must execute both of these inserts in one table so that either all of a doctor’s information is inserted, or none of it is inserted.

Thus, we only ever need transaction when we are writing or updating the database. So any SELECT statements do not require to be executed in a transaction. Moreover, any SELECT statements executed in one line do not need to be executed in a transaction either because MYSQL already treats them like a transaction.

The doctor and patient search operations are executed in single SELECT statements and so are read only statements. Moreover, they are already single statements and so MYSQL treats them as a transaction by default.

Below are a few examples of situations where we needed transactions.

1. Add\_doctor: This is a procedure that accepts a doctor’s user information such as alias and gender, and also accepts their profession related information such as “year of medical license.” We have multiple tables that store a doctor’s information. For example the User table will store the Doctor’s alias, and the Name table will store the first, middle and last name of the doctor (indexed on the doctor’s alias which is a foreign key referencing the User table). Hence, this stored procedure will require multiple insert statements. To avoid erroneous data, we must execute all of these statements in a SQL transaction. If we don’t and only part of the statements execute, then we may end up in a situation where a doctor’s name is stored but their gender is not available in the appropriate destination.
2. Add\_patient: this procedure behaves the same way as add\_doctor, but for patients.
3. Friendship: This stored procedure will potentially delete from a table and insert into another table. For that case we must use a transaction so that one cannot happen without the other. The case we are referring to is simple. If a user accepts a friend request, then that friend request data must be deleted from the FriendRequest table, and then a new friendship relation must be inserted into the Friend table. We cannot have a situation where a friend request is deleted, but the friend relation does not exist in the Friend table. Hence these two statements must be executed in a transaction.

Since our stored procedures only take one transaction each, the only time we process a request using multiple transactions is if we call multiple stored procedures. The maximum number of times we call a set of stored procedures for one request is when we try to get a doctor’s profile statistics (as requested by a patient). Each such request from the patient requires all of the doctor’s details including name, year of medical license, work addresses, specializations, and review statistics. We had an option of transferring all of this data in one result table (using one really long SQL query) or to transfer it back to the front end using multiple queries. The tradeoff of picking the former option would be a very large result set. Instead we opted for the latter option. This method requires less time to write the queries and will transfer the data back in smaller packages. Hence the transfer will be faster. Moreover, the queries are simpler and so will not take as much time to execute. Overall, we had to make a design decision on how we want to pull the data and we opted for the method that will be easier to implement since the speed tradeoff was not significant enough for our purposes.

We did not explicitly set the transaction isolation level, thus InnoDB's Repeatable Read transaction isolation level was used. This mode prevents non-repeatable reads from occurring. However phantom reads could occur. An example of a phantom read occurring is when a user writes a doctor review while another user simultaneous requests a doctor profile. The resulting page would incorrectly display metadata pertaining to that doctor's review, i.e. review count as the metadata was fetched before a review was added, and the review list was added after a review was added. Thus the newly added review would not be encountered in the review statistics. If metadata integrity is required, the Serializable isolation mode could be enforced, otherwise the user can re-request the page to get the correct updated information.

**Novel Features**

Although we do not think we had any novel features in our codebase, we do feel that the way we handled our access control was cool. This is explained in more detail in the Access Control section (for example our mechanism to kick the user out to the LoginPage if they aren’t the right user using HTTPSession).