

# **ECE 456 Project Phase 2**

ECE 456 - Database Systems  
Group #: ECE.456.W12-G007

Date: April 2nd, 2012

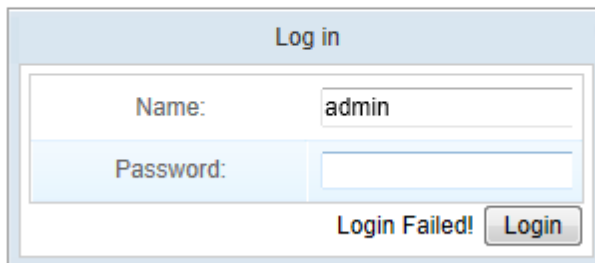
Craig Gurnik, cgurnik, 20280219  
Andrei Tulai, atulai, 20248881  
Kanishka Raajkumar, kraajkum, 20264989  
Preethi Sivachidambaram, psivachi, 20284095

# Table of Contents

- [1. Final Design](#)
  - [a\) Login System](#)
  - [b\) Administrator](#)
  - [c\) Patient](#)
  - [d\) Doctor](#)
  - [e\) Staff](#)
  - [f\) Finance](#)
  - [g\) Changes from Initial Design](#)
- [2. Portability](#)
- [3. Concurrency](#)
- [4. Testing](#)
- [5. Features](#)
- [6. Appendix A - Phase I](#)

# 1 Final Design

## a) Login System


A screenshot of a web-based login window titled "Log in". The window has a light blue header bar. Below the header, there are two input fields: "Name:" with the text "admin" entered, and "Password:" which is empty. Below these fields, there is a "Login Failed!" message in red text and a "Login" button. The button is rectangular with a light blue gradient and a thin border.

The login window as seen in the figure above is accessible to all users and forms the base of the system. It prompts the user for a username and password. Only users who have registered access to the system can login successfully. This is accomplished by querying the *users* table with the provided user information. Other users will have to get prior access information from an administrator or a staff member. This ensures security of the system so that invalid users cannot login to the system. Depending on the user's role type: administrator, patient, doctor, staff or finance, one is redirected promptly to their respective pages. This further improves security so that users of a particular role type do not have the rights and access to another role type's pages.

## b) Administrator

**LOGIN SYSTEM**

On successful login

Welcome Andrei Tulai [Logout](#)

User Management Visits

New Patient New Doctor New Staff New Finance New Admin

User ID	Name	Username	Role	Active
1	Andrei Tulai	atulai	ADMIN	1
2	Craiginder Singh	csingh	DOCTOR	1
3	Kanishh Raajkumar	kraajkumari	PATIENT	1
6	YAY Test	staff	STAFF	1
47	Craig Gurnik	cgurnik	PATIENT	1
48	Nima Dehnashi	ndehnash	PATIENT	1
49	Alex Sin	asin	STAFF	1
50	Craig Gurnik	sadf	PATIENT	1

User ID	Name	Address	Province	SIN	Health Card	Phone Num	Current He
3	Kanishh Raajkumar	123 Four Dr.	ON	123456789	123456789	214748364	null
47	Craig Gurnik	4 Wagler Ave.	ON	123123123	234234234	511231231	null
50	Craig Gurnik	234	ON	238840206	809283046	234092384	null

Clicking new doctor, staff, finance or admin

New Doctor - Andrei

First Name:

Last Name:

Username:


New Password:

Role:

Status:

Save

Clicking the visits tab

Welcome Andrei Tulai [Logout](#)

User Management Visits

Date

Patient Name

Diagnosis

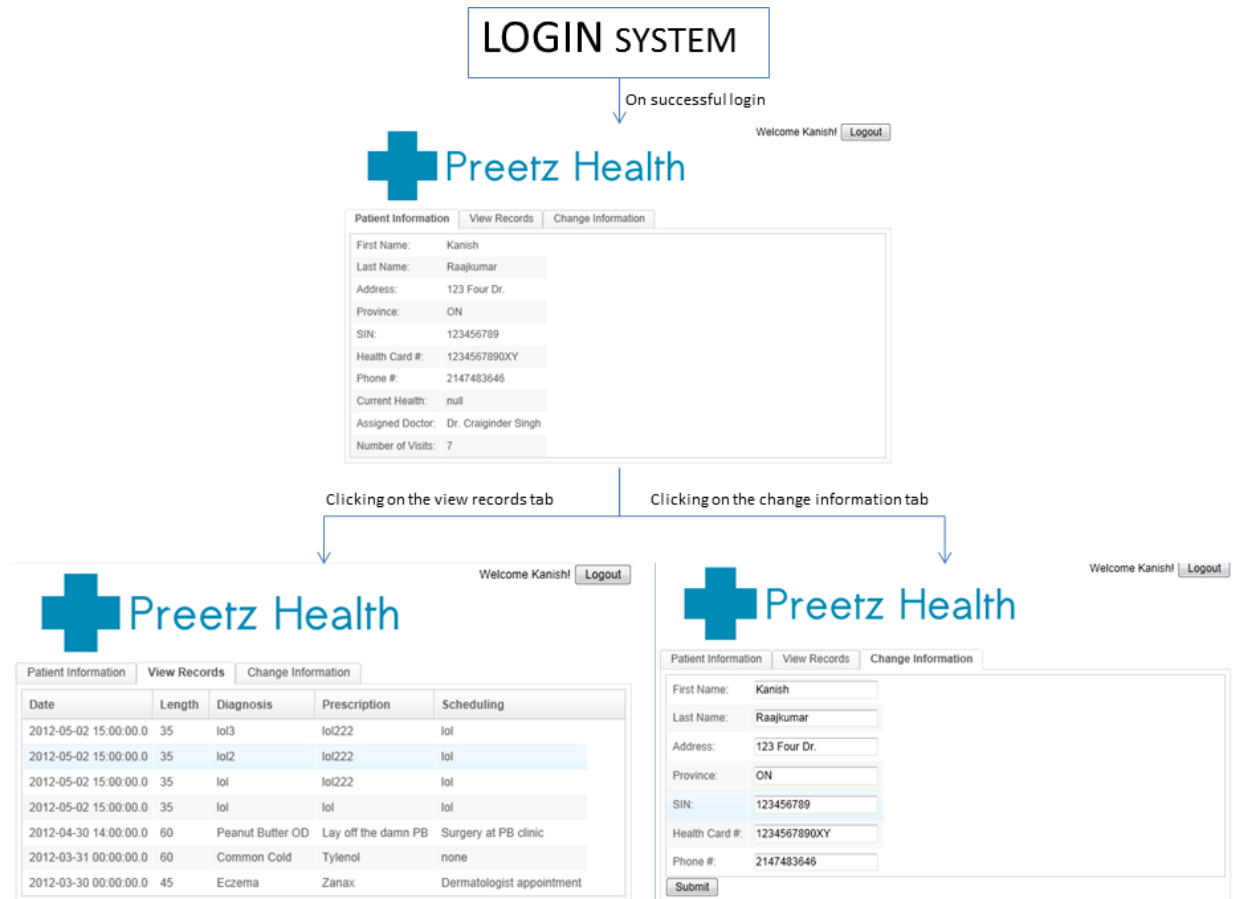
Prescription

Comments

No items match your search

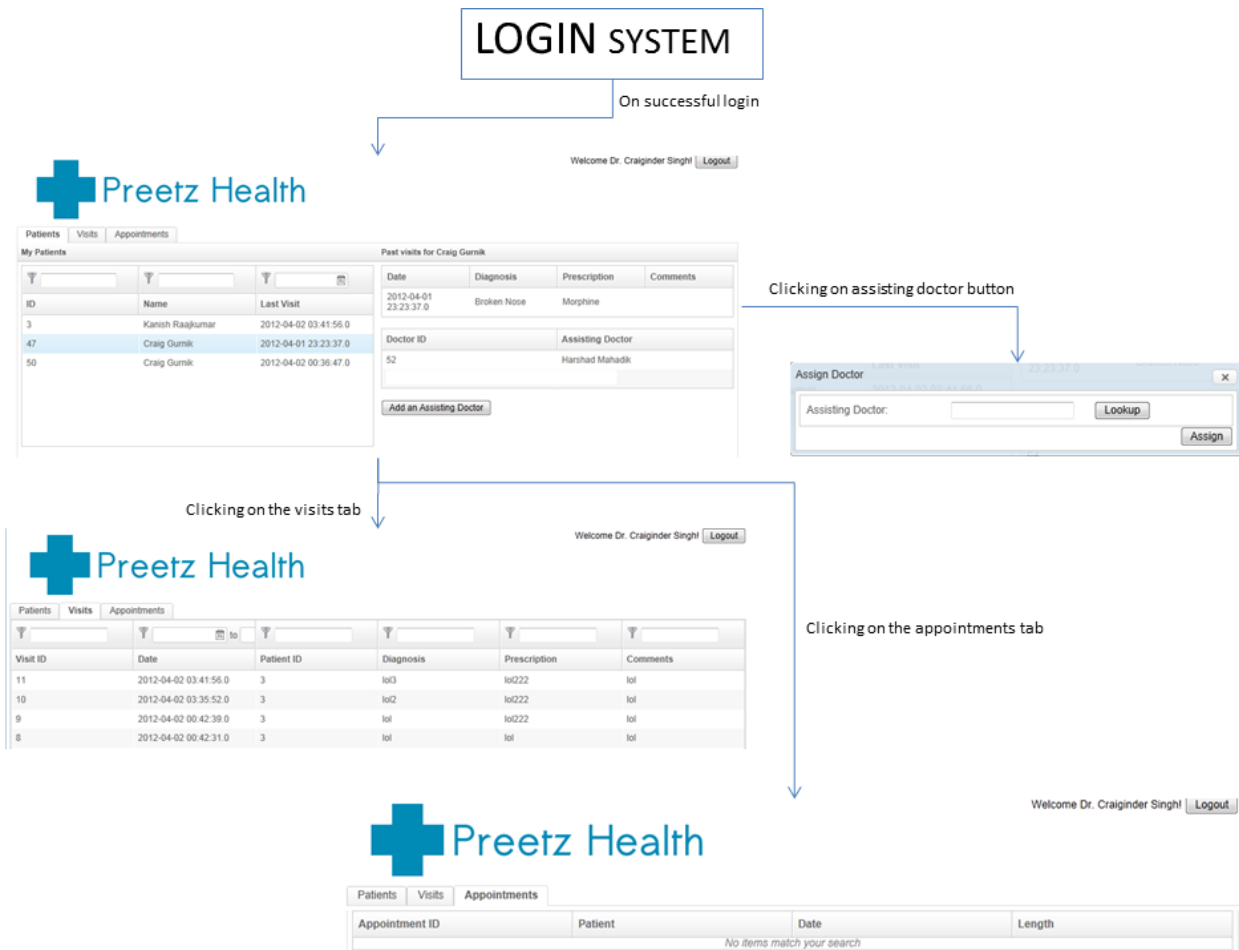
If the logged in user has administrator privileges, the user is redirected to the respective administrator page. The administrator has the rights to create new, edit and remove existing patients, doctors, staff and finance members. To create a new user, a record is inserted into the *users* table with user information and the role of the user is specified in the *role\_type* table. If the new user is a staff member, the user is assigned to a doctor. To edit an existing user, the appropriate user record is updated in the *users* table with the new information. To remove a user, the *active* column of the user record in the *users* table is set to inactive. This way, the record isn't completely deleted and if the user needs access to the system in the future, the *active* column is modified appropriately.

## c) Patient



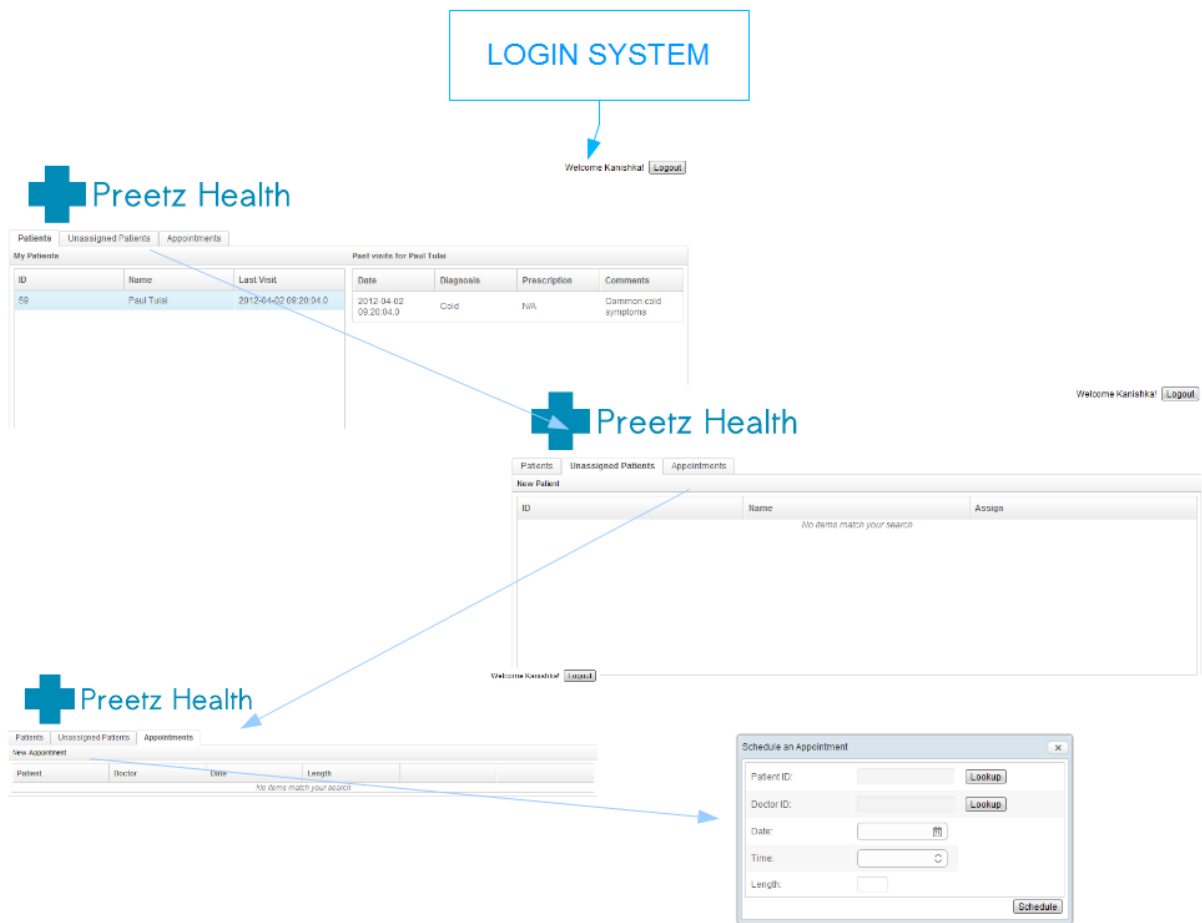
When a patient logs into the system, the user is redirected to the patient page. Here, the patient can view personal information, the name of the doctor that was assigned to the patient and the number of visits to the doctor. The *patients* and *users* table are queried to retrieve user information and the *relationships* table is queried to get the default doctor of the patient. The patient can also view all the details such as the diagnosis and prescription of every past visit to the doctor by querying the *appointments* and *visits* tables. In the third tab, the patient can modify basic personal information such as name, address, SIN, etc. The appropriate records in the *users* and *patients* tables are updated.

## d) Doctor



Once logged in, a doctor is redirected to a page that displays all the patients assigned to that doctor. The doctor can click on any patient and the table on the right side is populated with the details of all the visits of that particular patient. Here, the doctor can also grant and revoke the rights to an additional doctor to view the record of the selected patient. The doctor can search for a patient using their id, name or by specifying a particular date. By clicking on the *Visits* tab, the doctor can view all the past visits of all patients which and can filter the results using parameters such as the date, patient name, diagnosis, prescription or comments. The *Appointments* tab allows a doctor to view future appointments of patients. Here, the doctor can also create a visitation record where the doctor can save the details of that particular visit.

## e) Staff



A staff member can view all the patients and records of the doctor that the person was assigned to. Clicking on a patient's name shows all the past visits of that patient. Clicking on *Unassigned Patients* allows the staff to assign a new patient to one of the doctors. Here, the staff also have the privileges to create a new patient record. The new patient information is inserted into the *patients* and *users* tables. The *Appointments* tab allows the staff to view existing appointments and schedule new appointments through a modal window..

## f) Finance

LOGIN SYSTEM

Visitation Records

Doctors

ID	Name
56	Craig Gurnik

Visits

Patient ID	Date	Diagnosis	Prescription	Comments
No items match your search				

Welcome Preeti Chidambaram! [Logout](#)

Visitation Records

Doctors

ID	Name
56	Craig Gurnik

Visits

Patient ID	Date	Diagnosis	Prescription	Comments
59	2012-04-02 09:20:04.0	Cold	N/A	Common cold symptoms

Welcome Preeti Chidambaram! [Logout](#)

A member of the finance department can view visitation records for all doctors, and can also search the records using a variety of filters. Clicking on the doctor's name, will show all past visitations that doctor has recorded, with the patient name omitted for privacy purposes.

## g) Changes from Initial Design

The database schema and SQL statements used in the final system were the same as the ones provided in the initial design of the Phase I document. There were no changes. The Phase I document has been attached in Appendix A.

## 2 Portability

The system was developed using a web based Java framework called ZK. Since Java was the



base of the development environment, source code portability is very high. Using C or C++ based frameworks would have provided various opportunities for developing non portable code. However, using a Java based framework allows source code to be easily transferred from one system to another. Another level of portability is provided due to the web based nature of the system. The system should be accessible and run safely without hassle on any computer as long as the user has a web browser to access the system. There is no need for additional software or plugins to be installed to use the system.

### 3 Concurrency

There is some concurrency implemented in the system. If there are two or more SQL statements that have to be executed in succession, a SQL transaction is utilized to perform this action. The atomic nature of transactions allow ensure that all the required queries are performed or none of them are. As a result, consistent data is maintained in the database system. This is accomplished by using the SQL statement *START TRANSACTION* followed by multiple queries that are to be executed. If no errors were encountered during the execution of the queries, *COMMIT* statement is used to save all the changes made to the database permanently. However, if some errors resulted from the execution of these queries, then *ROLLBACK* command is used to undo all the changes made to the database. The database is returned to its initial state before the execution of the transaction.

### 4 Testing

The system was thoroughly tested for bugs and errors using a detailed test plan. The test plan consisted of various test cases built based on the functional requirements of the system. The table below provides a sample of five test cases to showcase the testing procedure used.

#	Test Case	Intended Outcome	Result	Comments
1	When a patient changes personal information, all fields should be validated for appropriate format	An error message should be shown to user regarding the validation problem and the entered data should not be saved in the database	Success	
2	When a user enters invalid information in the login window	The system should indicate to the user	Success	

		to enter valid information		
3	When a staff member schedules an appointment for a doctor, there is a conflict	The user should be an error that there is a conflict and re-prompt for another time	Success	
4	When a staff member creates a new patient, all fields should be validated properly	A message should be shown to the user stating that there are errors and they need to be fixed before saving to the database	Success	
5	When a patient modifies personal information	Both the users and patients table need to be updated for that specific patient	Success	

## 5 Features

A major feature of our system is the dynamic nature of the web interface. The ZK framework allowed us to build a data-driven system that allows a user of a specific role to stay on one page, and perform all of their required activities without having to be redirected to other pages. This is done through a variety of forms and data-bound grids that use AJAX to communicate with the server.

## **6 Appendix A - Phase I**

### **A. Introduction**

#### **a) Objective**

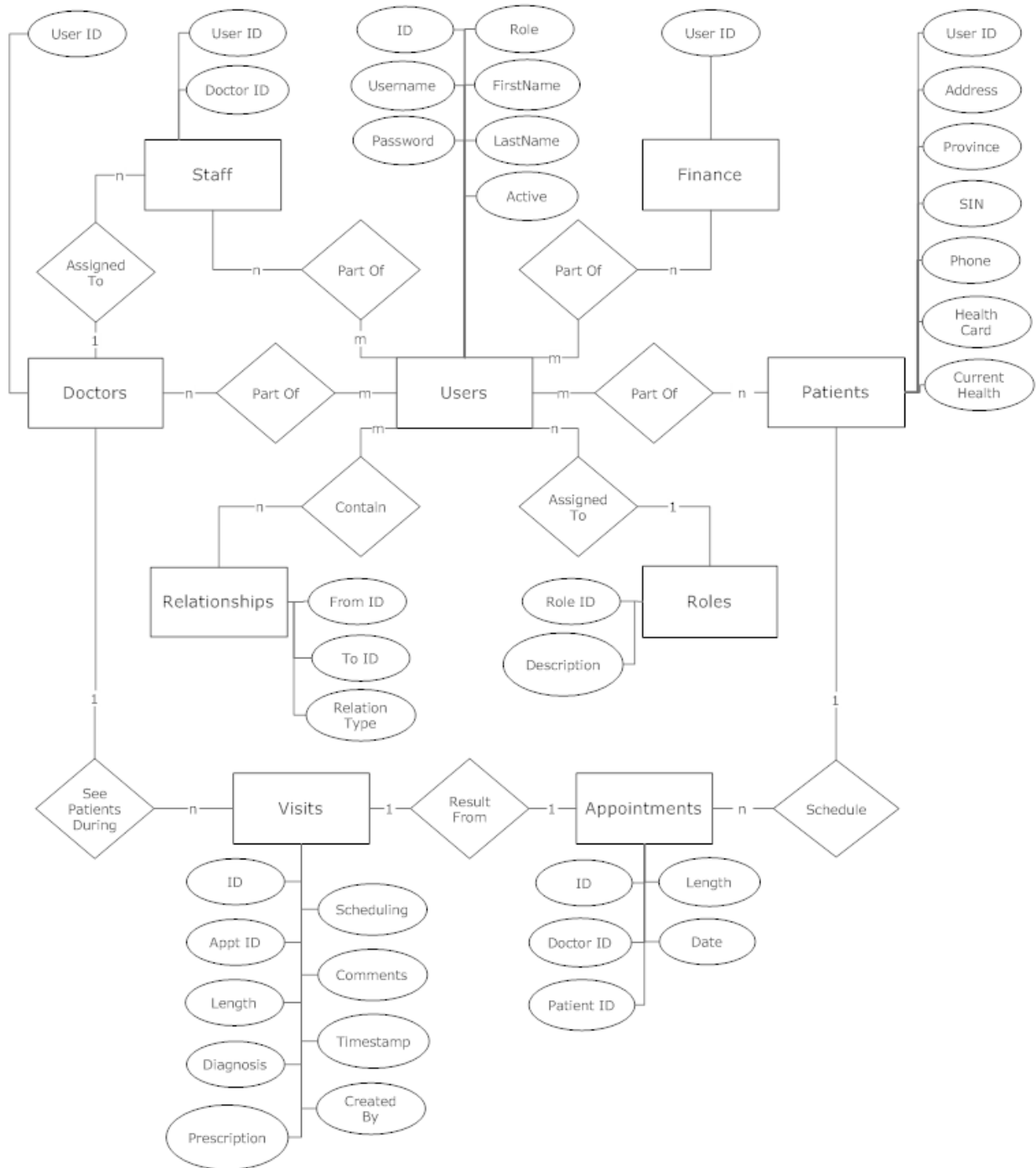
Health care institutions such as hospitals are required to deal with various patients, doctors, staff and other stakeholders so that they can function properly. As a result, there is a vast amount of data that has to be managed in an efficient manner to ensure that all the different parties are satisfied. The objective of this project is to develop a database information system that can be used to manage the patient records according to the given functional requirements. This database management system should be efficient, user-friendly and highly secure. It should also take into account of the different needs and requirements of the diverse kinds of users such as doctors, staff, patients, finance and legal departments. The purpose of this report is to provide the preliminary design details on the system that will be developed.

#### **b) Accomplishments**

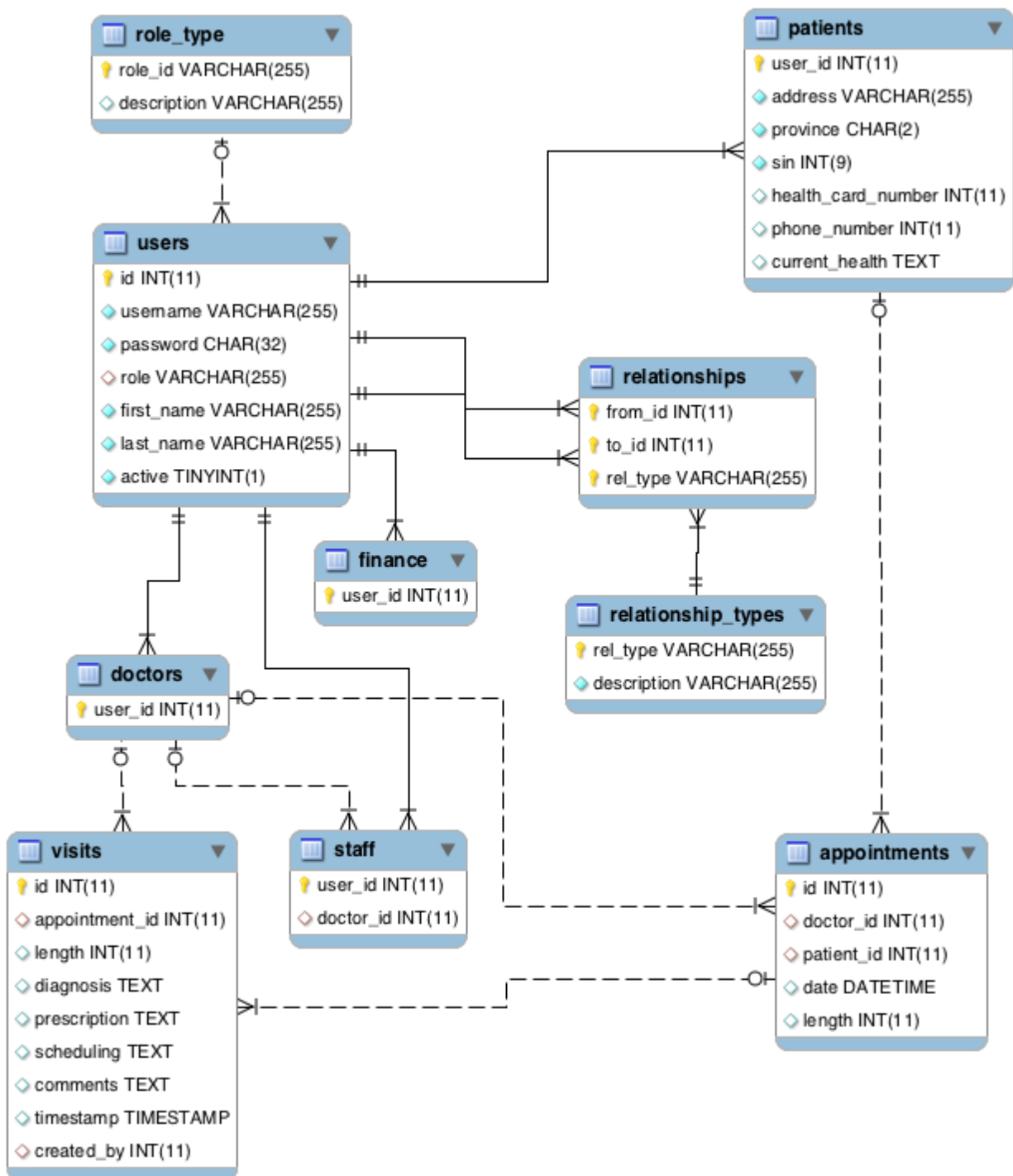
This document outlines the main database structure that will form the backbone of the hospital management information system. This is accomplished by providing the entity-relationship diagram and a detailed database schema. Furthermore, the integrity constraints that have to be maintained in the database and the normalized database schemes are also provided. The security portion of the database is also detailed using the access rights for different users. Finally, some sample SQL queries and the development environment that will be used to build the system is explained in detail.

## B. Initial Design

### a) ER Diagram



## b) Initial Relational Database Schema



## C. Integrity Constraints

### a) Candidate Keys, Primary Keys, and Super Keys

Table	Primary Keys	Candidate Keys	Super Keys
Appointments	id	id	{id, doctor_id, date}; {id, patient_id, length}
Doctors	user_id	user_id	user_id
Finance	user_id	user_id	user_id
Patients	user_id	user_id; SIN; health_card_number	{user_id, SIN, address, province, phone_number}; {user_id, health_card_number, current_health}
Relationships	{from_id, to_id, rel_type}	{from_id, to_id, rel_type}	{from_id, to_id, rel_type}
Relationship_types	rel_type	rel_type; description	{rel_type, description}
Role_type	role_id	role_id; description	{role_id, description}
Staff	user_id	user_id; doctor_id	{user_id, doctor_id}
Users	id	id; username; {first_name, last_name, role}	{id, username, password}; {first_name, last_name, role, active}
Visits	id	id; appointment_id	{id, length, diagnosis, timestamp}; {appointment_id, prescription, scheduling, created_by}

### b) Functional Dependencies and Multi Valued Dependencies

Users

Dependency	Description
id $\rightarrow$ password, role, first_name, last_name, active	each user only has one password, role, first name, last name, and active status

#### Patients

Dependency	Description
$user\_id \rightarrow address, province, sin, health\_card\_number, phone\_number, current\_health$	a patient only has one address, province, sin, health card number, phone number, and current health status

#### Appointments

Dependency	Description
$id \rightarrow doctor\_id, patient\_id, date, length$	an appointment links a doctor and a patient at a specific date and time for a specific length of time

#### Visits

Dependency	Description
$id \rightarrow appointment\_id, length, diagnosis, prescription, scheduling, comments, timestamp, created\_by$	a visit is a record of the manifestation of an appointment, and includes a diagnosis, prescription, scheduling, comments, creation timestamp, and created by fields

#### Staff

Dependency	Description
$user\_id \rightarrow doctor\_id$	a staff member is assigned to a doctor

No multi-valued dependencies were found in the initial schema.

### c) Foreign Key Constraints

Foreign key constraints are extremely important for this project as they are essentially required to ensure referential integrity between data in two tables. For example, the *doctors* table maintains a list of doctors with a foreign key on *user\_id* that references the user table. If the row in the *users* table is updated or deleted, the row in the *doctors* table should be handled appropriately to avoid any “dangling pointers”. A similar condition exists between the *appointments* and *visits* tables, because an appointment is required to exist before a visit can be created referencing that appointment’s unique identifier. Additionally, we must ensure that deleting specific rows does not cause any information to be lost. For example, deleting a doctor should not delete that doctor’s staff, previous appointments, or visitation records. Additionally, certain information pertaining to that doctor should still be available, such as their name when

viewed on historical visitation records, even after the doctor has been deleted. These are not the only cases where foreign keys are utilized (see section 5 for complete details), but it does represent many of the cases which need to be considered through the application.

In order to ensure that the aforementioned requirements are met, all foreign keys will be set to “restrict” for both updates, and deletes. To understand why this constraint was chosen, it is important to understand why the other options were not chosen. The first alternative is “cascade”, which deletes/updates the row in the parent table, and then alters the matching rows in the children tables accordingly. The advantage of this constraint is that it ensures that there are no dangling references to non-existent rows, since all associated rows are altered (deleted or updated) appropriately. However, this option is not suitable because in the case of a delete, it would result in the loss of information in child tables. As per the requirements stated by the legal department, and along an assumption that losing information is bad, cascading changes is not a suitable solution. Utilizing the cascade constraint on updates is not useful, since in practice a user should never be given the option to change their user id. Similarly, appointments and visitation records also do not require that their identifiers be changed.

The second unsuitable option is “set null”, which allows the delete/update to occur in the parent table, and then proceeds to update the value of the foreign key field to null. This is unsuitable for a similar reason to aforementioned cascade option, since it results in data loss. For example, if the doctor’s id is nulled on an appointment or visitation record, it will become much less useful, as we will no longer know which doctor treated the patient. This also violates the legal department’s requirements that data should be able to be audited.

Finally, we are left with the “restrict” constraint, which throws an error if the foreign key in the parent table is updated or deleted. This constraint is the default for MySQL, so if a constraint is not specified for “on update”, or “on delete”, restrict is assumed. The benefit of using this constraint is that it ensures referential integrity regardless of what operation is performed. It also forces developers to have a greater understanding of how data is interconnected before blindly updating and deleting rows. Our project will utilize transactions and handle all of the updating and deleting of foreign keys manually. This will ensure that the data is manipulated exactly as we would like it, and also allows for greater flexibility in the future if the client’s requirements change.

## **D. Normalized Database Schemes**

### **a) 3NF, 4NF, BCNF**

For the identified functional dependencies in part 3b we know that the ids on the left hand side of every dependency are candidate keys, as they are unique auto-generated ids. Given that we could not identify any functional dependencies within the rest of the columns of each table, and our functional dependencies in the form of  $X \rightarrow A$  hold in R where  $A \notin X$ , we know that X is a superkey (and also because X functionally determines all of the other columns in the relation). And since X is a superkey, for each FD, we know that the relations are in BCNF.

Also, we do not need to go into performing 3NF or 4NF, since the lack of decomposition



occurring prevented the introduction of any dependency problems.

## **b) Lossless Decomposition**

Since no decomposition occurred, it is useless to talk about lossless decomposition in this case.

## **c) Dependency Preserving Decomposition**

Since no decomposition occurred, it is useless to talk about dependency preservation in this case.

## **d) Candidate Keys, Primary Keys and Integrity Constraints**

Since no decomposition occurred, this means that the candidate keys, primary keys, and integrity constraints did not change.

# **E. SQL Create Statements**

Note: By default, MySQL uses the RESTRICT constraint on foreign keys for both ON UPDATE and ON DELETE. Since all of our foreign keys utilize this constraint, it has been omitted from the following queries.

```
CREATE TABLE `appointments` (  
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
  `doctor_id` int(11) unsigned DEFAULT NULL,  
  `patient_id` int(11) unsigned DEFAULT NULL,  
  `date` datetime DEFAULT NULL,  
  `length` int(11) DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `patient_id` (`patient_id`),  
  KEY `doctor_id` (`doctor_id`),  
  CONSTRAINT `appointments_ibfk_2` FOREIGN KEY (`patient_id`) REFERENCES  
  `patients` (`user_id`),  
  CONSTRAINT `appointments_ibfk_3` FOREIGN KEY (`doctor_id`) REFERENCES  
  `doctors` (`user_id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
CREATE TABLE `doctors` (  
  `user_id` int(11) unsigned NOT NULL,  
  PRIMARY KEY (`user_id`),  
  CONSTRAINT `doctors_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users`  
  (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
CREATE TABLE `finance` (  
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
  `date` datetime DEFAULT NULL,  
  `amount` int(11) DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  KEY `date` (`date`),  
  KEY `amount` (`amount`)
```

```

    `user_id` int(11) unsigned NOT NULL,
    PRIMARY KEY (`user_id`),
    CONSTRAINT `finance_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users`
    (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `patients` (
    `user_id` int(11) unsigned NOT NULL,
    `address` varchar(255) NOT NULL DEFAULT '',
    `province` char(2) NOT NULL DEFAULT '',
    `sin` int(9) unsigned NOT NULL,
    `health_card_number` int(11) DEFAULT NULL,
    `phone_number` int(11) DEFAULT NULL,
    `current_health` text,
    PRIMARY KEY (`user_id`),
    CONSTRAINT `patients_ibfk_3` FOREIGN KEY (`user_id`) REFERENCES `users`
    (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `relationship_types` (
    `rel_type` varchar(255) NOT NULL DEFAULT '',
    `description` varchar(255) NOT NULL DEFAULT '',
    PRIMARY KEY (`rel_type`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `relationships` (
    `from_id` int(11) unsigned NOT NULL,
    `to_id` int(11) unsigned NOT NULL DEFAULT '0',
    `rel_type` varchar(255) NOT NULL DEFAULT '',
    PRIMARY KEY (`from_id`, `to_id`, `rel_type`),
    KEY `to_id` (`to_id`),
    KEY `rel_type` (`rel_type`),
    CONSTRAINT `relationships_ibfk_3` FOREIGN KEY (`rel_type`) REFERENCES
    `relationship_types` (`rel_type`),
    CONSTRAINT `relationships_ibfk_1` FOREIGN KEY (`from_id`) REFERENCES `users`
    (`id`),
    CONSTRAINT `relationships_ibfk_2` FOREIGN KEY (`to_id`) REFERENCES `users`
    (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `role_type` (
    `role_id` varchar(255) NOT NULL DEFAULT '',
    `description` varchar(255) DEFAULT '',
    PRIMARY KEY (`role_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `staff` (
    `user_id` int(11) unsigned NOT NULL,
    `doctor_id` int(11) unsigned DEFAULT NULL,
    PRIMARY KEY (`user_id`),
    KEY `doctor_id` (`doctor_id`),
    CONSTRAINT `staff_ibfk_2` FOREIGN KEY (`user_id`) REFERENCES `users` (`id`),
    CONSTRAINT `staff_ibfk_1` FOREIGN KEY (`doctor_id`) REFERENCES `doctors`
    (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `users` (
    `id` int(11) unsigned NOT NULL AUTO_INCREMENT,

```

```

`username` varchar(255) NOT NULL DEFAULT '',
`password` char(32) NOT NULL DEFAULT '',
`role` varchar(255) DEFAULT '',
`first_name` varchar(255) NOT NULL DEFAULT '',
`last_name` varchar(255) NOT NULL DEFAULT '',
`active` tinyint(1) unsigned NOT NULL DEFAULT '1',
PRIMARY KEY (`id`),
KEY `role` (`role`),
CONSTRAINT `users_ibfk_1` FOREIGN KEY (`role`) REFERENCES `role_type`
(`role_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `visits` (
`id` int(11) unsigned NOT NULL AUTO_INCREMENT,
`appointment_id` int(11) unsigned DEFAULT NULL,
`length` int(11) DEFAULT NULL,
`diagnosis` text,
`prescription` text,
`scheduling` text,
`comments` text,
`timestamp` timestamp NULL DEFAULT NULL ON UPDATE CURRENT_TIMESTAMP,
`created_by` int(11) unsigned DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `appointment_id` (`appointment_id`),
KEY `created_by` (`created_by`),
CONSTRAINT `visits_ibfk_1` FOREIGN KEY (`appointment_id`) REFERENCES
`appointments` (`id`),
CONSTRAINT `visits_ibfk_2` FOREIGN KEY (`created_by`) REFERENCES `doctors`
(`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

## F. Design for Audit Trail

The design for keeping an audit trail for this project is relatively simple as everything is contained within a single table. Within the *visits* table, there are two fields for ids. The first field, *id*, is an auto-incrementing primary key that is (by definition of a primary key) unique. The second field, *appointment\_id*, is not unique, but instead is a foreign key referencing the *id* field in the *appointments* table. When a visitation record is first inserted into the *visits* table, it is assigned a unique *id*, and the *appointment\_id* is set based upon the appointment that the visitation record is associated with. If the visitation record needs to be altered, an insert to the table is performed instead, rather than an update. This new insert will again receive a unique *id*, but the *appointment\_id* will be set to the same as the previous row. This will allow us to maintain a complete history of every change made to a visitation record, while still allowing us to easily retrieve the most up to date copy of the record. For the legal department's purposes, the entire history of visitation records can be retrieve using a query that does not impose a limit. For doctors and staff (who are probably only concerned with the most update to date copy of the record), the rows can simply be sorted in descending order by *id*, with a limit of one applied. This approach satisfies all of the legal department's criteria, while still maintaining a simple implementation.

## G. Access Rights

Every user will have to go through a login authentication system to access the management information system using a username and a password. The passwords for the login system are protected using the SHA-1 hash function for maximum security. There is a *role\_type* table which is used to assign a specific role (doctor, patient, staff, finance or legal department) to the user. Therefore, every user will only have certain rights depending on the role that has been assigned to the user. For example, a user whose role is a patient can only view personal records and nobody else's. This user may not have the rights to make appointments, for example, since only the staff members have the rights do so. There are also *relationships* and *relationship\_types* tables which are used to map relationships between doctors, patients and the staff. There are three different relationship types specified: 'Default Doctor', 'Assisting Doctor' and 'Doctor to Staff'. 'Default Doctor' relationship type is used to assign a doctor to a patient. 'Assisting Doctor' is used by the default doctors to grant/revoke permissions to another doctor for accessing on of their patient's records. 'Doctor to Staff' is used to assign a staff member to a particular doctor. In this manner, every user is granted the appropriate permissions and rights when accessing the system.

## H. Sample SQL Queries

The following queries are examples of various queries that system features will allow the user to perform:

Query	Description
SELECT * FROM visits	Get all visitation records.
INSERT INTO visits (appointment_id, length, diagnosis, prescription, scheduling, comments, created_by) VALUES (1111, '163', 'cough, running nose', 'tylenol', 'xyz', 'none', '2222')	Create a new visitation record.
SELECT * FROM visits WHERE timestamp < '2012-03-14 11:00:00' AND timestamp > '2012-03-07 11:00:00'	Get visitation records for a specific period of time.
SELECT * FROM visits WHERE prescription='xyz' AND diagnosis='abc' AND comments LIKE '%fever%'	Find past records based on various search criteria.
DELETE appointments.* FROM appointments LEFT JOIN visits ON appointments.id = visits.appointment_id WHERE appointments.id='1234' AND visits.id IS NULL	Delete appointments that have no corresponding

	visitation records.
UPDATE appointments SET date= '2012-03-14 11:00:00' WHERE id='1234' AND date = '2012-03-13 10:00:00'	Update appointments.
INSERT INTO patients (user_id, address, province, sin, health_card_number, phone_number, current_health) VALUES (1111, '163 University Ave West', 'Ontario', '123456', '12345678', '2268086750', 'none')	Create new patients.
UPDATE patients SET address='163 University Ave West', phone_number='2268086750' WHERE user_id='1234'	Update patient information.
INSERT INTO relationships (from_id, to_id, rel_type) VALUES (1111, 2222, 'DEFAULT_DOCTOR')	Assign a patient to a doctor.
SELECT * FROM staff INNER JOIN visits ON staff.doctor_id = visits.created_by WHERE staff.user_id = 1234	Review patient records for a doctor that is served by this staff person.

## I. Development Environment

The group will use MySQL as our database server of choice. MySQL was chosen for several reasons, including its cost (free), great documentation, and ease of setup. Also, there is a very strong community surrounding MySQL, which provides a great source of third-party tutorials for group members who have not used the product before.

For our backend language, the group will use PHP for similar reasons. It is very easy to setup, cross-platform (Windows, OSX, Linux, etc), and it also has a thriving community. Additionally, several of the group members have previous experience using PHP, which will allow additional time to be given to data modeling issues rather than learning another language's syntax. Since PHP does not impose any sort of restrictions on file structure or architecture, we will use the CodeIgniter (<http://www.codeigniter.com/>) framework to ensure that the code is properly structured, easy to understand, and maintainable. CodeIgniter specifically provides an MVC architecture and requires that controllers, models and views are located in certain locations on the file system to ensure proper separation of logic. PHP also includes a native MySQL driver *mysqlnd* (<http://dev.mysql.com/downloads/connector/php-mysqlnd/>), so no additional components are required for PHP to connect to a MySQL server.

On the frontend, the group will use HTML, CSS and JavaScript to create web-based user interface. This will allow support across multiple platforms, including basic support for mobile devices. These technologies were chosen because they are industry standards for producing quality websites. Although alternatives exist such as Flash and Adobe AIR, their support among

mobile devices is non-existent or limited at best.

The group will use Apache to serve both dynamic (PHP) and static (images, CSS, HTML) content. Specifically, XAMPP (<http://www.apachefriends.org/en/xampp.html>) will be used to develop and demonstrate our product. XAMPP was chosen because it is extremely easy to setup, and it includes all of the components mentioned above, making it well suited for this project. It is also cross platform, which is important since our group members use a combination of Windows and OSX. This also satisfies the portability requirements described in the project description.