# Medical Records Database

A SQLite Implementation

**Ryan Bomalaski & Stuart Hernandez**

CSE 5290/4020

Dr. Marius Silaghi

12/01/2017

## Introduction

Electronic medical records are an organized collection of the medical information pertaining to a hospital or insurance systems patients and the details of their care.[1] Health Information Management is a quickly growing field, and in order to meet the primary needs of security and speed, great strides have been made in both database technology and database design for EMR.[2] It is the goal of this project to emulate a simple EMR database in order to high light proper database design, implementation and testing methods using freely available tools such as SQLite3 and Python.

## Design

The main considerations when designing the database was that it must store information about the doctors, their patients, and their illnesses. Since those were the three important aspects of the database each one was created as an entity, the relevant attributes were given to them, and the proper relations between the entities were established.

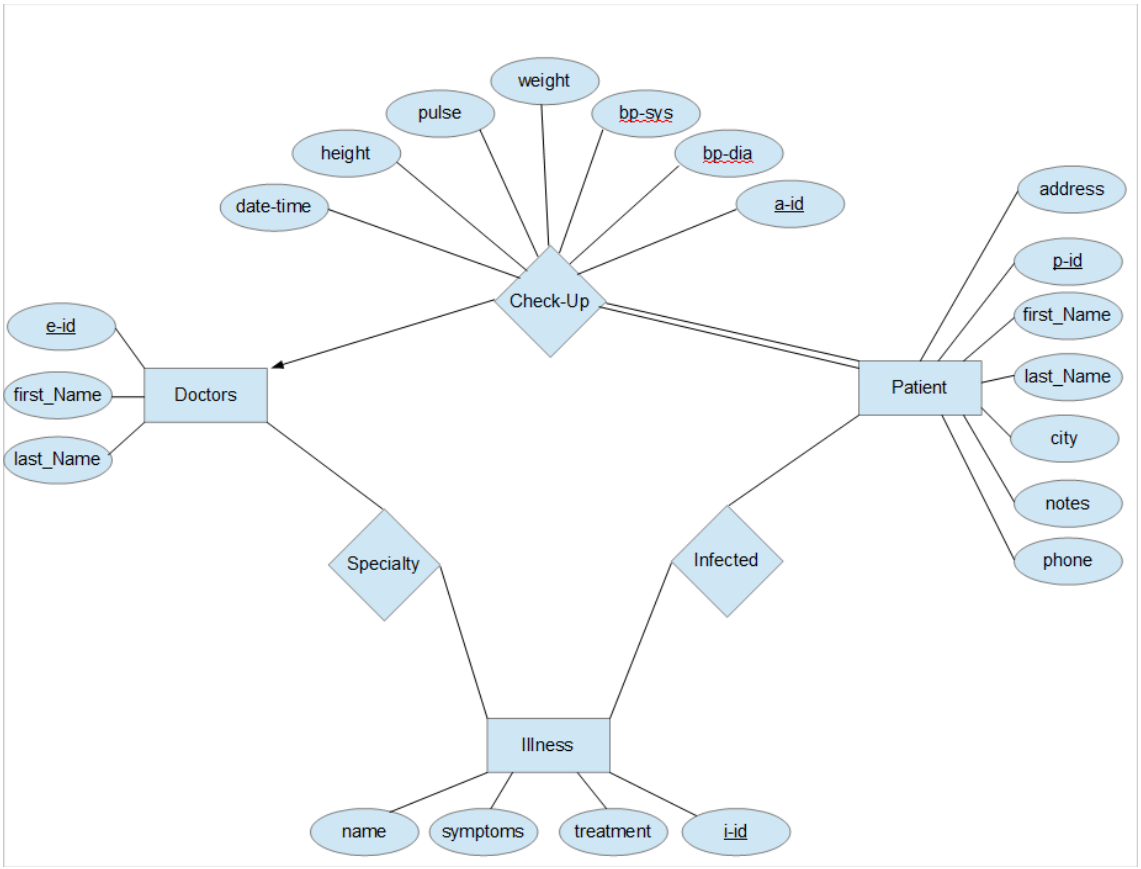


***Fig. 1:*** *ER Representation*

The database is composed of the following relations:
$Doctors\{\underline{e-id}, first-Name, last-Name\}$
$Patient\{\underline{p-id}, address, first-Name, last-Name, city, notes, phone\}$
$Illness\{\underline{i-id}, name, symptoms, treatment\}$
$Check-Up\{\underline{a-id}, \underline{e-id}, \underline{p-id}, date-time, height, pulse, weight, by-sys, bp-dia\}$
$Specialty\{\underline{e-id}, \underline{i=id}\}$
$Infected\{\underline{p-id}, \underline{i-id}\}$

The database contains the following relational dependencies:
$i-id \rightarrow name, symptoms, treatment$
$e-id \rightarrow Doctors.first-Name, Doctors.last-Name$
$p-id \rightarrow address, city, notes, phone, Patient.first-Name, patient.last-Name$
$a-id \rightarrow p-id, e-id, date-time, height, pulse, weight, bp-sys, bp-dia$

For the first decomposition the check-up relation was decomposed to add a vitals relation as follows:
$Check-Up\{\underline{a-id}, \underline{e-id}, \underline{p-id}, date-time\}$
$Vitals\{\underline{a-id}, height, pulse, weight, by-sys, bp-dia\}$

This decomposition is both dependency preserving and has a lossless join. For a decom-

position to have a lossless join $(R1 \cap R2) \rightarrow R1$ or $(R1 \cap R2) \rightarrow R2$. This is the case for our decomposition which can be shown as follows:
$(R1 \cap R2) \rightarrow R1$
$\{a-id, e-id, p-id, date-time\} \cap \{a-id, height, pulse, weight, by-sys, bp-dia\} \rightarrow \{a-id, e-id, p-id, date-time\}$
$a-id \rightarrow \{a-id, e-id, p-id, date-time\}$

This is valid according to our fourth functional dependency. To prove a decomposition is dependency preserving if $\{F_1 \cup F_2 \cup ... \cup F_n\} = F^+$ where $F^+$ is the closure of the original functional dependencies and $F_i$ is a functional dependency from the decomposed database that relates to the same attributes as the closure of the original database. This decomposition is dependency preserving which can be shown as follows:
$\{F_1 \cup F_2 \cup ... \cup F_n\} = F^+$
$\{a-id \rightarrow e-id, p-id, date-time \cup a-id \rightarrow height, pulse, weight, by-sys, bp-dia\} = a-id \rightarrow p-id, e-id, date-time, height, pulse, weight, bp-sys, bp-dia$
$\{a-id \rightarrow e-id, p-id, date-time, height, pulse, weight, by-sys, bp-dia\} = a-id \rightarrow p-id, e-id, date-time, height, pulse, weight, bp-sys, bp-dia$

This shows that all the functional dependencies of the related attributes are retained in the decomposition, and therefore this composition is both lossless and dependency preserving.

For the second decomposition the illness relation was decomposed to add a symptoms relation as follows:
$Illness\{\underline{i-id}, name, treatment\}$
$Symptom\{\underline{i-id}, symptoms\}$

This decomposition is both dependency preserving and has a lossless join. For a decomposition to have a lossless join $(R1 \cap R2) \rightarrow R1$ or $(R1 \cap R2) \rightarrow R2$. This is the case for our decomposition which can be shown as follows:
$(R1 \cap R2) \rightarrow R1$
$\{i-id, name, treatment\} \cap \{i-id, symptoms\} \rightarrow \{i-id, name, treatment\}$
$i-id \rightarrow \{i-id, name, treatment\}$

This is valid according to our third functional dependency. To prove a decomposition is dependency preserving if $\{F_1 \cup F_2 \cup ... \cup F_n\} = F^+$ where $F^+$ is the closure of the original functional dependencies and $F_i$ is a functional dependency from the decomposed database that relates to the same attributes as the closure of the original database. This decomposition is dependency preserving which can be shown as follows:
$\{F_1 \cup F_2 \cup ... \cup F_n\} = F^+$
$\{i-id \rightarrow name, treatment \cup i-id \rightarrow symptoms\} = i-id \rightarrow name, treatment, symptoms$
$\{i-id \rightarrow name, treatment, symptoms\} = i-id \rightarrow name, treatment, symptoms$

This shows that all the functional dependencies of the related attributes are retained in the decomposition, and therefore this composition is both lossless and dependency preserving.

# Data Creation

In order for the database to better represent what a hospital or insurance network might utilize, a random record generator was developed in the Python language. As this was just a tool to fill in data for our database, it did not need to be secure nor did it need to be extremely quick. Python was chosen because it was seen to be good enough.

```
>>> create_vitals()
INSERT INTO vitals VALUES(1,44,173,121,91,67,100);
INSERT INTO vitals VALUES(2,29,122,159,129,70,125);
INSERT INTO vitals VALUES(3,51,199,139,109,69,104);
INSERT INTO vitals VALUES(4,53,181,75,45,61,103);
INSERT INTO vitals VALUES(5,7,208,118,88,64,72);
INSERT INTO vitals VALUES(6,71,234,60,30,75,80);
INSERT INTO vitals VALUES(7,5,162,137,107,63,84);
INSERT INTO vitals VALUES(8,43,226,145,115,63,111);
INSERT INTO vitals VALUES(9,61,180,70,40,69,87);
INSERT INTO vitals VALUES(10,69,168,151,121,62,130);
INSERT INTO vitals VALUES(11,40,253,107,77,64,72);
```

***Fig. 2:*** *Example output of the create_vitals() function.*

The generator, called *insert_generator_db2.py*, creates SQL insert commands for seven of the database's tables: *doctors, patients, vitals, infected, specialty, illness and check_up*. Each table has its own method for generating an output of the sql commands. They are named as follows:*create_doctors, create_patients, create_vitals, create_infected, create_specialty, create_checkup, and create_illness*. Vitals was included as a separately generated table because at the time of the generator program's creation, the *vitals* table was still separate from the *check_up* table. This design became database 2, and as a result the generator was named for the fact that it generates information for database 2. The source code for the generator is attached and can also be found at the project's Github page.[3]

# Implementation

SQLite3 was chosen for the database engine because it is a simple to implement but still robust and powerful SQL implementation. The project fit perfectly into SQLite's ideal usage scenarios, both as a piece of educational software and as a means of demonstrating an enterprise database for testing.[4] A client/server model would have been overly complicated, though it may have better represented the implementation of a production level medical records database. Still, the goal of this project was to demonstrate database design, testing and implementation, not as an overview of database engines. The specifics of the engine should not change the design or the testing results.

The contents of the DDL file used to create Database_1, *create_1.sql*, can be found below:

```
--Create Doctors Table
CREATE TABLE doctors (
  id INTEGER PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL
);

-- Create Patients Table
CREATE TABLE patients (
  id INTEGER PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  phone INTEGER NOT NULL,
  address TEXT NOT NULL,
  city TEXT NOT NULL,
  notes TEXT
);

-- Create illnesses Table
CREATE TABLE illnesses (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  treatment TEXT NOT NULL,
  symptoms TEXT NOT NULL
);

-- Create check_up Table
CREATE TABLE check_up (
  id INTEGER PRIMARY KEY,
  date_time TEXT NOT NULL,
  patient_id INTEGER,
  doctor_id INTEGER,
  weight INTEGER,
  bp_sys INTEGER,
  bp_dia INTEGER,
  height INTEGER,
  pulse INTEGER,
  FOREIGN KEY(patient_id) REFERENCES patients(id),
  FOREIGN KEY(doctor_id) REFERENCES doctors(id)
);

-- Create specialty Table
CREATE TABLE specialty (
  id INTEGER PRIMARY KEY,
  doctor_id INTEGER,
  illness_id INTEGER,
  FOREIGN KEY(doctor_id) REFERENCES doctors(id),
  FOREIGN KEY(illness_id) REFERENCES illnesses(id)
);
```

```
50
51 -- Create infected Table
52 CREATE TABLE infected (
53   id INTEGER PRIMARY KEY,
54   patient_id INTEGER,
55   illness_id INTEGER,
56   FOREIGN KEY(patient_id) REFERENCES patients(id),
57   FOREIGN KEY(illness_id) REFERENCES illnesses(id)
58 );
```

Databases 2 and 3 were created by using the DML files *update_to_db_2.sql* and *update_to_db_3.sql*. Each of the queries from the Queries section was run on each database to ensure proper data migration.

# Queries

In order to test the database, three queries were proposed. In plain English, the queries are as follows:

1. Find the full patient name, the full doctor name, the name of the patient's illness, and the patient's blood pressure for any patient who had an appointment on September 30th, 2008 at 9:00 AM.

2. Find the list of potential doctors that can most effectively work on any disease infecting a patient with the first name Samantha.

3. Find the names of all patients who have been treated for Malaria by a Dr. Harmon and have a BMI over 25.

**Relation Algebra - Query 1**

$r1 \leftarrow \sigma_{date-time=09/30/200809:00AM} Check - Up$
$r2 \leftarrow \rho_{first-Name=d.firstName,last-Name=d.lastName} Doctors$
$r3 \leftarrow \rho_{first-Name=d.firstName,last-Name=p.lastName} Patients$
$r4 \leftarrow r2 \bowtie r1 \bowtie r3 \bowtie Infected \bowtie Illness \bowtie Check - Up$
$\Pi_{d.firstName,d.lastName,p.firstName,p.lastName,name,bp-sys,bp-dia} r4$

**Relation Algebra - Query 2**

$r1 \leftarrow \rho_{first-Name=d.firstName,last-Name=d.lastName} Doctors$
$r2 \leftarrow \rho_{first-Name=d.firstName,last-Name=p.lastName} Patients$
$r3 \leftarrow \sigma_{firstName=Samatha} r2 \bowtie Infected \bowtie Illness$
$r4 \leftarrow r1 \bowtie Specialty \bowtie r2$
$\Pi_{d.firstName,d.lastName} r4$

**Relational Algebra - Query 3**

$r1 \leftarrow (\sigma_{weight/height^2>25} Checkup)$

$r2 \leftarrow (\sigma_{name=Malaria} Illness)$
$r3 \leftarrow (\sigma_{first_name=Francisandlast_name=Harmon} Doctors)$
$\Pi_{name}((r1 \bowtie Patients \bowtie Infected \bowtie r2 \bowtie Specialty \bowtie r3))$

**SQL**

Below are SQL representations of the above queries as pertaining to Database 1. The queries for Databases 2 and 3 are attached to the project and can also be found on the project's github page.[3]

### SQL- Database 1 - Query 1

```
SELECT
    patients.first_name AS patient_first_name, patients.last_name AS
    patient_last_name, doctors.first_name AS doctor_first_name, doctors.
    last_name AS doctor_last_name, illnesses.name, check_up.bp_sys, check_up.
    bp_dia
FROM
    patients
    INNER JOIN
        check_up
ON
        patients.id = check_up.patient_id
    INNER JOIN
        doctors
ON
        doctors.id = check_up.doctor_id
    INNER JOIN
        infected
on
        infected.patient_id = patients.id
    INNER JOIN
        illnesses
on
        infected.illness_id = illnesses.id
WHERE
    check_up.date_time = '09/30/2008 09:00 AM';
```

### SQL- Database 1 - Query 2

```
SELECT
    doctors.first_name, doctors.last_name
FROM
    doctors
    INNER JOIN
        specialty
ON
    specialty.doctor_id = doctors.id
    INNER JOIN
        infected
ON
    infected.illness_id = specialty.illness_id
```

```
    INNER JOIN
        patients
ON
    patients.id = infected.patient_id
WHERE
    patients.first_name = 'SAMANTHA';
```

**SQL- Database 1 - Query 3**

```
SELECT
    patients.first_name, patients.last_name
FROM
    patients
    INNER JOIN
        check_up
ON
    check_up.patient_id = patients.id
    INNER JOIN
        infected
ON
    patients.id = infected.patient_id
    INNER JOIN
        illnesses
ON
    infected.illness_id = illnesses.id
    INNER JOIN
        doctors
ON
    check_up.doctor_id = doctors.id
WHERE
    doctors.last_name = "HARMON"
    AND illnesses.name = "MALARIA"
    AND (check_up.weight*.45) / ((check_up.height*.025) * (check_up.height
    *.025)) > 25;
```

# Testing

Each of out database implementations were tested both to ensure data preservation as well as query expediency. For each database, each query was run 1000 times and was timed during the execution of the queries. To facilitate this testing, bash scripts were written with the following layout.

```
#!/bin/bash
cd ../
cd sql_files
cd database_1
sqlite3 db_1.db ".read sql_queries_1-1.sql"
...
sqlite3 db_1.db ".read sql_queries_1-1.sql"
```

**test_script_1-1.sh:**Where line 6 represents 998 other calls identical to line 5

In order to time the execution of these scripts, the Unix command *time* was used. This command returns information on the real time taken to execute a command, the time the command was in user space, and the time it was in system space.

```
time test_script_1 -1.sh
```

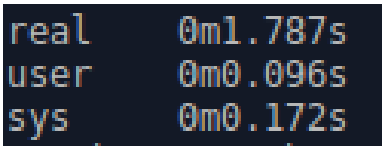


***Fig. NUM:*** *Example output of time Unix command*

A script containing just the first 4 lines of each test script was also run, and these times were subtracted from the full test script values. This was done in order to account for any time that may have been used for the navigation commands. However, it was found that the time tool was not exact enough to measure their execution. As such, it was assumed that the navigation time was negligible in comparison to query time.

BASH was also used for database creation scripts, allowing the creation and testing of the databases to be fulling scripted and workable from the command line.

# Results

After testing it was found that the queries performed quite quickly, with most taking only small fractions of a second to complete their queries in system and user space.

| | REAL | SYS | USER |
|---|---|---|---|
| DB 1 - Q 1 | 2.022 | .0492 | .052 |
| DB 1 - Q 2 | 2.172 | 0.400 | 0.600 |
| DB 1 - Q 3 | 1.845 | 0.088 | 0.152 |
| DB 2 - Q 1 | 1.733 | 0.116 | 0.128 |
| DB 2 - Q 2 | 2.387 | 0.228 | 0.356 |
| DB 2 - Q 3 | 2.361 | 0.316 | 0.324 |
| DB 3 - Q 1 | 1.690 | 0.136 | 0.180 |
| DB 3 - Q 2 | 2.155 | 0.104 | 0.224 |
| DB 3 - Q 3 | 1.626 | 0.060 | 0.1040 |

***Table 1:***Results for 1000 executions of each query

| | REAL | SYS | USER |
|---|---|---|---|
| DB 1 - Q 1 | 0.002022 | 0.000492 | 0.00052 |
| DB 1 - Q 2 | 0.002172 | 0.0004 | 0.0006 |
| DB 1 - Q 3 | 0.001845 | 0.000088 | 0.000152 |
| DB 2 - Q 1 | 0.001733 | 0.000116 | 0.000128 |
| DB 2 - Q 2 | 0.002387 | 0.000228 | 0.000356 |
| DB 2 - Q 3 | 0.002361 | 0.000316 | 0.000324 |
| DB 3 - Q 1 | 0.00169 | 0.000136 | 0.00018 |
| DB 3 - Q 2 | 0.002155 | 0.000104 | 0.000224 |
| DB 3 - Q 3 | 0.001626 | 0.00006 | 0.000104 |

***Table 2:***Results for 1 executions of each query

# Conclusion

In summation, it has been shown that a simple but effective medical records database can be designed, implemented and tested using free to use tools. SQLite3 was shown to be effective and quick in performing selection queries and would perform admirably as a platform for creating quick proofs of concept.

Further testing could be done to look into the speed of UPDATE queries, though making dynamic update can be problematic. Likewise, server/client layouts should be explored for the EMR realm as well.

# Bibliography

1 *Electronic Medical Records*, Luo. J, *Primary Psychiatry*, Feb. 1, 2006

2 *Embracing the Future: New Times, New Opportunities for Health Information Managers*,http://library.ahima.org/doc?oid=58258#.WiIPqvZOlqs *American Health Information Management Association*

3 https://github.com/ryanbomo/5260-Project

4*Appropriate Uses For SQLite* https://sqlite.org/whentouse.html