Use of Traditonal Database Methodoloy for a Vaccine Passports Database and Scaling it for Real World Application

Jacob Baggs
Graduate College of Computer Science
Kennesaw State University
Marietta, GA, USA
jbaggs2@students.kennesaw.edu

Abstract*— There are many ways to store data in the modern-day including blob storage such as Amazon web service's S3 or Microsoft Azure's Blob Storage, NOSQL database such as columnar, wide column, object orientated, key-value databases, and relational databases. When it comes to maintaining relationship-based data, SQL has been the industry standard for decades. In current times COVID 19 has plagued the world as one of the world's fastest growing pandemics. To fight against the spread, vaccines have been administered to millions of individuals. All this information about vaccine administration has to be stored and potentially made available as a "vaccine passport". We propose to use a full stack solution of MYSQL, a relational database, and python to implement a proof-of-concept solution to store, create, update, and maintain information records about the shots administered and who the shots were administered to as a vaccine passport.

***Keywords*—API, COVID-19, Databases, MYSQL, Python**

I. Introduction

This paper discusses the theory and application of applying current traditional database methodologies to build a proof-of-concept database for vaccine passports for the SARS-Co-V-2 virus. This paper will describe the beginning of the process of designing a schema and mapping out the relationships and the theory behind the decisions made in tables. It will then follow up and cover the limitations and dependencies of those tables and relationships from a practical and theoretical stand point. Next the paper will cover implementation of this schema in both MYSQL and a python application program interface (API) to interact with the database. Further the paper will outline the permission design for different users, and the reason behind these decisions. Finally, the paper will describe how the proof of concept could be scaled up to serve an industrial tool and ways the project could be improved by new research and technologies.

A. Problem Statement

I want to design a database that at scale follows the best practices and procedures as described by industry experts. The information that needs to be stored in a successful vaccine passport date base includes patient information, volunteer information, clinic information, manufacturer information, and vaccine administration information. We want to establish a schema and design that is high performance, saleable, maintainable and makes the information readily available for end users.

II. Related work

Currently there are several widely known information sources being used to obtain and store SARS-CoV-2 vaccine information including VSAFE (1), electronic health record data, or immunization registries. None of these are meant for widespread public use and do not solve our problem statement.

III. Abbreivations

API	Application Programmable Interface
SQL	Standard Query Language
AWS	Amazon Web Services
EKS	Elastic Kubernetes Service
VPC	Virtual Private Cloud
ETL	Extract Transform Load
ELT	Extract Load Transform
CI/CD	continuous integrations and continuous deployment/delivery
DAG	Direct Acrylic Graph
EMR	Elastic Map Reducer

IV. Database Design and Implmentation

A. Schema Design

When designing our database, I set out to build a database with following the Boyce–Codd normal form of normalization. This form of normalization focuses on minimizing functional dependency such that all dependencies have the following conditions met are met in (1) or (2) [2]. This was done in order to follow best practices and increase the performance of database. The overall design scheme can be viewed in Appendix A. The center of our star schema was Patient. I chose patient as the center point because everything can be related to the patient. Also, our super key, as defined by (2), is the primary keys from the Manufacturer, Volunteer, Patient, Dose, and Clinic tables. The super key can be defined as X(Manufactuer.ID, Volunteer.ID, Patient.ID, Clinic.ID, Dose.DoseNumber) which is a super key for schema R, with our R being defined as the schema in Appendix A. This was designed in this way to follow the conditions shown in (2).

B. Tables

The first table at the center of schemes as show in Appendix A and Table I, was Patient. Patient has the primary key of Patient.ID, a nonrepeating autogenerating value generated at time of insertion of a new patient by MYSQL. The rest of the columns represent features of a patient. These columns were Date of Birth (DOB), first name, last name, street number, street name, city, state, zip, and gender. Some design decisions were made such as the choice to not separate out address into its own table. This choice to not separate our address may theoretically break the Boyce-Codd rules of normal form normalization. However, from a practical stand up, it did not make sense to break up the address to a different table and create a possibly transitive dependency if addresses cannot be verified by a separate database and rely on manual entry. Also, address was separated into multiple fields in order to reduce the number of string values being used and allow for greater control of what was being inputted into the columns. This in the long term can increase data quality and reduce the need to fix incorrect string inputs using methods like machine learning or auto filling information. The functional dependency can be described as Patient ID -> DOB, FirstName, LastName, StreetNum, StreetName, City, State, Zip and Gender. See Code I for SQL code to generate the Patient table.

Code I
Patient SQL Code

Code:
```
create table Patient(
   ID              INT NOT NUll PRIMARY KEY,
   DOB               DATE,
   FirstName           VARCHAR(255),
   LastName            VARCHAR(255),
   StreetNum           INT,
   StreetName          VARCHAR(255),
   City             VARCHAR(255),
   State            VARCHAR(255),
   Zip              VARCHAR(10),
   Gender             VARCHAR(15) ;
```

Table I
Patient Database Table

Column Name	*Column Type*
ID	ID
DOB – Data of Birth	Date
FirstName	VARCHAR
LastName	VARCHAR
StreetNum	Integer
StreetName	VARCHAR
City	VARCHAR
State	VARCHAR
Zip	Integer
Gender	VARCHAR

The next table designed was Clinic, which is shown in Appendix A and Table II. The primary key for this table was Clinic.ID which was another a nonrepeating autogenerating value generated at time of insertion of a new clinic by MYSQL. The table's columns as listed below were: ID, clinic (CName), StreetNum, StreetName, City, State, Zip. This table also followed the same design decision for address as the Patient table followed. There were not any transitive dependencies in this table. The functional dependency can be described as Clinic(ID)-> (CName, StreetNum, StreetName, City, State, Zip). See Code II for SQL code to generate the Clinic table.

Code II
Clinic Table SQL Code

Code:
```
CREATE TABLE Clinic(
  ID               INT NOT NULL PRIMARY KEY,
  CName               VARCHAR(255),
  StreetNum          INT,
  StreetName          VARCHAR(255),
  City             VARCHAR(255),
  State            VARCHAR(255),
  Zip              VARCHAR(10)
);
```

Table II
Client Database Table

Column Name	*Column Type*
ID	ID
CName	VARCHAR
StreetNum	Integer
StreetName	VARCHAR
City	VARCHAR
State	VARCHAR
Zip	Integer

The Manufacturer table follows the clinic table. This table is also described in Appendix A and Table III, This table had the primary key of Manufacturer.ID. Again, Manufacturer.ID was a nonrepeating autogenerating value generated at time of insertion of a new manufacturer by MYSQL similar to the Clinic and Patient tables. The columns in this table were ID, manufacturer (Mname), StreetNum, StreetName, City, Zip, State. We follow the same strategy for address as the previous tables. Finally the functional dependencies for this table can be expressed as Manufacture(ID) -> Mname, StreetNum, StreetName, City, Zip, State with zero transitive dependencies. See Code III for the SQL code to generate the manufacturer table.

Code III
Manufacturer Table SQL Code

Code:
```
create table Manufacturer(
   ID         INT NOT NUll PRIMARY KEY,
   MName       VARCHAR(255),
   StreetNum   INT,
   StreetName  VARCHAR(255),
   Zip         VARCHAR(10),
   City        VARCHAR(255),
   State       VARCHAR(255)
   );
```

Table III
Manufacturer Database Table

Column Name	*Column Type*
ID	ID
MName	VARCHAR
StreetNum	Integer
StreetName	VARCHAR
City	VARCHAR
State	VARCHAR
Zip	Integer

The Volunteer table was then designed to be able to refer to the clinic table for a foreign key. The primary key for this table was Volunteer.ID which again borrowed from the MYSQL functionality to be a nonrepeating autogenerated value that is never null. The columns for this table are described in Appendix a and Table IV as follows: ID, FirstName, LastName, Age, and ClinicID. ClinicID is a foreign key that is important to reference what clinic the volunteer is associated with. The foreign key is used tos maintain referential integrity. Hence, a volunteer cannot be entered without first adding the clinic. Volunteer can also be associated with different clinics. The functional dependencies of this table can be expressed as Volunteer(ID)-> FirstName, LastName, Age, and ClinicID. There were not any transitive dependencies in this table. In the future, as this is a N:N relationship where the volunteer may and could work at multiple clinics, this one table could be replaced with two table. The first table would include a row for each volunteer. A second table would then be needed representing the relationships between each Volunteer.ID and Clinic.ID. A real-life example of when this may come up is traveling nurse or pharmacist that work on multiple pharmacy location such as Publix. However, this was thought to be rare at time of original design and such a design would be an improvement over the current design. The SQL code for generating the Volunteer table is represented in Code IV.

Code IV
Volunteer Table SQL Code

Code:
```
CREATE TABLE Volunteer(
   ID         INT NOT NULL PRIMARY KEY,
   FirstName   VARCHAR(255),
   LastName    VARCHAR(255),
   Age        Int(3),
   ClinicID    INT, FOREIGN KEY (ClinicID) REFERENCES
Clinic(ID)
);
```

Table IV
Volunteer Database Table

Column Name	*Column Type*
ID	ID
FirstName	VARCHAR
LastName	VARCHAR
Age	Integer
ClinicID	ID

The final table created Dose. Dose was created last due to all the referential integrities needed. This table has several unique qualities to it. The columns in this table were DoseNumber, PatientID, ManufacturerID, LotID, DateReceived, VolunteerID, ClinicID. First the primary key was a combination key of ManufactuerID, PatientID, and DoseNumber. This was chosen as it provides a unique key without making an additional ID variable for each specific dose. Also, this table's primary key was a combination of the two foreign keys and one created column, DoseNumber. LOTID does not provide a unique key and is an attribute of the dose. LotID could not be used as thousands of vaccine can be in a lot, and a patient potentially could receive multiple doses from the Lot. However, LotID can be used to trace certain vaccine doses in case of a bad batch that needs to be investigated or patients need to be alerted to. ClinicID, ManufacturerID, VolunteerID, and PatientID were all put into the dose table to create referential integrity. Without these there may need to be extraneous joins created that didn't allow the queries to be answered as need. There were zero transitive dependencies and the functional dependencies can be expressed as Dose(DoseNumber,PatientID, ManufacturerID)->LotID, DateReceived, VolunteerID, ClinicID. As mentioned, this table has to be built last as it references foreign keys from all the other tables. The SQL code for the Dose table is described in Code V.

Code V
Dose Table SQL Code

Code:
```
create table Dose (
   ManufacturerID      INT, FOREIGN KEY (ManufacturerID)
REFERENCES Manufacturer(ID),
   LotID             VARCHAR(255),
   DoseNumber         INT,
   DateReceived       DATE,
   VolunteerID        INT, FOREIGN KEY (VolunteerID)
REFERENCES Volunteer(ID),
   PatientID          INT, FOREIGN KEY (PatientID)
REFERENCES Patient(ID),
   ClinicID           INT, FOREIGN KEY (ClinicID) REFERENCES
Clinic(ID),
   PRIMARY KEY        (DoseNumber,PatientID,ManufacturerID)
);
```

Table V
Dose Database Table

Column Name	*Column Type*
ManufacturerID	ID
LotID	VARCHAR
DataRecieved	Integer
VolunteerID	VARCHAR
PatientID	VARCHAR
ClinicID	VARCHAR

C. *Relationships*

The super key for this table as defined previously is the combination of the primary keys above Manufactuer.ID, Volunteer.ID, Patient.ID, Clinic.ID, Dose.DoseNumber), which here as be reference as super key X. This is important because with the super key you are able to maintain a relationship between all tables. For example, to determine who administers a specific dose to a patient, I can use ID variable in the Patient table to join Dose table on the PatientID, and then further join the Volunteer table using the VolunteerID from the Dose table and ID variable from the Volunteer table. See an example of this query using SQL in Code VI. Our super key allows us to express the relationshis in our database as defined in (2). This allows us to traverse from any table to another table via joins.

Code VI
Identifying Volunteer who Administered a Specific Dose to a Patient

Code:
```
Select Volunteer.FirstName, Volunteer.LastName
From volunteer as Volunteer
Inner join Dose as Dose on Dose.VolunteerID = Volunter.ID
Inner Join Patient.ID as Patient on Dose.PatientID = Patient.ID
Where Patient.ID=1234 and Dose.DoseNumber = 1;
```

D. Users

After building out the tables as describe above, I then begun to work on the design to establish users for the database. In the original development, I developed two types of users. The first user was a power user to perform crucial functions on the database without having full root privileges. At scale, this user would be locked down and only assumed either by nonhuman accounts or data base administrators when non-root access was needed. This is a best practice as root access, if over granted, if obtained by a bad actor could cause devastating effects on our productional version of the database[3]. The second user type created was a test user. This user would be created via the web form registration kick off a SQL query to create a new user who only had access to edit their data. This would allow us to lock down their queries to the specific user after creation, which could be done by putting in a where condition, on each query to make it so that the patient ID had to equal their user ID that was created. Another way to develop this functionality would be to create a user map to patient ID table and use that instead of making the patient ID and user ID variables the same. We also could add an additional layer of security by only granting the ability to access the Patient and Dose tables, where as the power user has the ability to reference all our tables. Also, to create efficiencies and be able to create users at scale, Python was used to create a script to grant privilege calling the API of MYSQL, see Code VII. This adds the ability for better logging and error handling and scales better. Potential another user could be created with the sole ability to only create users following the practice of least privileges [3]. Overall, my strategy when creating user was focus on only providing the minimum access required.

Code VII
Generate and Grant Privileges SQL and Python Code

Code:
```
import mysql.connector
from mysql.connector import Error

connection = mysql.connector.connect(
    host="localhost",
    database="vaccineDB",
    user="root",
    password="password"
)


# Function to create user

def createUser(cursor, userName, password):
    try:
        sqlCreateUser = "CREATE USER IF NOT EXISTS
'%s'@'localhost' IDENTIFIED BY '%s';" % (userName, password)
        cursor.execute(sqlCreateUser)
    except Exception as e:
        print("Error creating MySQL User: %s" % (e))


cursor = connection.cursor();

# create users one power and one test user to represent a user of our
application
createUser(cursor, "powerUser", "pass123")
createUser(cursor, 'testUser1', "pass456")
createUser(cursor, 'jacobbaggs1', 'password')
# make sure we have the users
mySqlListUsers = "select host, user from mysql.user;";
cursor.execute(mySqlListUsers)

# Fetch all the rows
userList = cursor.fetchall()
# Print all the users
print("List of users:")
for user in userList:
    print(user)


def grantPrivs(cursor, privs, dataBase, userName):
    try:
        sqlGrantPowers = "GRANT %s on %s.* TO %s@localhost;" %
(privs, dataBase, userName)
        print(sqlGrantPowers)
        cursor.execute(sqlGrantPowers)
    except Exception as e:
        print("Error Grant Privileges to : %s" % (e))


grantPrivs(cursor, "SELECT, INSERT, DELETE, UPDATE",
"vaccineDB", "powerUser")
grantPrivs(cursor, "SELECT, INSERT, DELETE, UPDATE",
"vaccineDB", "jacobbaggs")


if connection.is_connected():
    cursor.close()
    connection.close()
    print("Mysql connection is closed")
```

E. Python API and Routes

When I started with developing the backend API, I chose python as the language. I then chose to use the Flask library for the API library and pymysql to integrate with the database. I then designed my routes. I initial started with a route per test query, creating multiple routes that had a singular function. But I ended up designing only 4 routes. The first route was a select route; this route was designated Select Route. As seen in Code VIII, we have several additional benefits of using this versus command line. First when deployed anyone with the address could call it without having to login into our database. Next, I designed a password validation to determine whether the user had the proper permission to perform the query. I also added error handling into this so the user would be able to see error codes in case of errors. I then created 2 more endpoints the Delete and the AdminQuery endpoints. The reason delete was separated out was in case of SQL injections [4] and because the select query relies on the cursor.fetchall function. But, delete relies on the cursor.commit function from pymsql in order for the code to run properly as seen in Code IX. The final route AdminQuery was designed to be used by power or root users to complete any administrator based tasks It has a more freeform input that allows user to enter in whole query string; however it cannot select data as it uses the cursor.commit function as seen in Code X.

Code VIII
Select Route

```
@app.route('/SELECT')
def Select():
try:
_json = request.json
app.config['MYSQL_DATABASE_USER'] = _json['UserName']
app.config['MYSQL_DATABASE_PASSWORD'] =
_json['Password'] _query = _json['query']
sqlQuery = f"{_query}" conn = mysql.connect()
cursor = conn.cursor(pymysql.cursors.DictCursor) query =
(sqlQuery)
cursor.execute(query)
row = cursor.fetchall()
value = str(row)
resp = jsonify(value) resp.status_code = 200 return resp
except Exception as e: print(e)
```

Code IX
Delete Route python API

```
@app.route('/DELETE')
def Delete():
try:
_json = request.json _table = _json['table'] _where =
_json['where'] app.config['MYSQL_DATABASE_USER'] =
_json['UserName']
app.config['MYSQL_DATABASE_PASSWORD'] =
_json['Password'] sqlQuery = f"DELETE From {_table} where
{_where}" conn = mysql.connect() cursor =
conn.cursor(pymysql.cursors.DictCursor)
query = (sqlQuery) cursor.execute(query) conn.commit() resp =
jsonify(query) resp.status_code = 200
eturn resp except Exception as e:
print(e)
```

Code X
AdminQuery Route

```
@app.route('/AdminQuery')
def Query(): try:
_json = request.json
_query = _json['query']
app.config['MYSQL_DATABASE_USER'] = _json['UserName']
app.config['MYSQL_DATABASE_PASSWORD'] =
_json['Password'] sqlQuery = f"{_query}"
conn = mysql.connect()
cursor = conn.cursor(pymysql.cursors.DictCursor)
query = (sqlQuery)
cursor.execute(query) conn.commit()
resp = jsonify(query) resp.status_code = 200 return resp
except Exception as e: print(e)
```

F. *Web Page Form*

A further enhancement would be to design a webpage form using html and css. I would then use a module like Flask to further pass in the data from the front end of the customer response, to our API route built on the back end. The query passed could be highly focused and reduce the chances of SQL injections [4]. Each field would have a corresponding query that limits the ability for user to input or update info that was not their data. Also, this would allow the queries to be performed by a nonhuman account and simplify the administration of multiple SQL users. A web form would also greatly increase the customer experience.

V. Real World Application

A. *Scaling up the database*

Scaling up the database is something I explored but did not complete. There would be several problems to tackle in order to move this database into a production database. The first would be access. A more robust strategy that follows least privilege practices would be needed. This would include exposing specific ports and multitenant users. If I was to move this database to a cloud provider to host IAM Policies and more network policies would need to be defined. Measure of throughput to decided utilization in order to design an efficient scale up and scale down policy. A few other concerns and technologies would need to be considered like whether data base caching would be used. In order to serve and design a database solution that would serve, we would need to look into enacting a caching service like Redis. This can decrease the network cost and help increase the application performance by using the cache technology to write to the database[5]. This technology allows for faster reading of the database as the most current and needed data is available in the caching technology. Twitter uses this philosphy to increase their performance of returning use tweet and reduce the load on their backend[6]. The next concern to would be to address data sharding, as the would be a national database we would need to shard the database so users can be physical closer to the data. For that reason I would either use a cloud provider to deploy the application using a service like AWS RDS and setup cross account replication[7]. This adds additional complication for deployment, however will bring the data closer to my end user for faster response time. This allows us horizontal scaling for both the API's and the database as the compute resources would be spread across multiple data center and availability zones [7]. The combination of using cross account replication and memory caching at the edge nodes would greatly increase our performance and reduce CPU utilization on the database. This is one of the strengths of using a cloud provider to host our database.

B. *Scaling up and deploying API's and Webpage*

To serve the data programmatically I developed the python API. To deploy the API at scale we could use a technology like EKS and API Gateways. The API Gateways could read the inputs from the web page previously discussed above and serve the info to our API deployment on EKS. First the API Gateway would handle all the authentication, removing the need to build it in front end single webpage, and using an outside based authorization method like OAuth 2.0[8]. This reduces the attack vectors in which our application could be hacked. Next the API gateway would handle the network request to a vpc Link [9]. This isolates our network from the internet further reducing the attack vector on our database. The information that can be passed to a load balancer which can finally pass the info to our API to make calls to the database. All this traffic will be routed automatically without manual intervention from the user. All these steps will give our users a more seamless experience and provide added security. Finally using infrastructure as code and ci/cd pipeline would provide easily deployed applications and databases by using code to build the infrastructure around our code.

The continuous integration and pipeline allow for rapid deployments.

C. *Moving the Data to the Data warehouse*

There are two strategies when moving data from ETL and ELT. To move the vaccine database the primary strategy could be either. As our use case is around private health care data, ETL make more sense. ETL will allow us to scrub and mask any sensitive data before it reaches the broader audience of the data lake [10]. The first decision to make when moving the data base is where it needs to be landed and will there be real time replication. Since we have chosen AWS and ETL, Redshift would be the prominent choice. There is not a need for staging in S3 as we are performing the transformation already being performed before the data is loaded [10]. Next depending on our down stream consumer will determine if the use case would demand real time replication or batch replication. For real time application, change data capture is one of the main methods of moving data for replication [11]. Tools like Kafka are popular real time streaming tools to perform crud operations on the secondary data source [11]. These tools would allow us to move the data in real time and transform it to fit our needs and land it at data warehouse technology like Redshift. Now for a batch process, Spark is one of the most popular modern ETL tools. The Spark APIs allow us to quickly move large amounts of over the data from one data source at one time. These jobs are often orchestrated by tools like airflow which generates DAGs to control and govern the jobs as they move data into the data sink. On the AWS platform, EMR would be used as the technology of choice to implement the Spark jobs. Both options are possible and would be determined by the use case.

D. *Create Data Marts*

Creating Data Marts would be the other logical step especially for non-real-time based reporting and analysis. The data from data warehouse could be moved to cheaper storage like S3. This allows less throughput requests on our data warehouse and gives specific business areas or department access to own their data. This is additional layer of security and horizontal scaling. This would often be done for things like analytical reporting and discovery process for machine learning.

VI. Equations

X-Y is a trivial function dependency ($Y \subseteq X$)[2] (1)

X is a super key for schema R [2] (2)

VII. Conclusions and Future work

We demonstrated our main goal of this project to develop a database for storing and retrieving data related to a vaccine passport following known best practices. We accomplished our goal by developing a MYSQL database and python API that can access the database and retrieve information from it. The possible future development is covered in depth in the real world application where I discuss the required work needed to deploy this database at scale.

VIII. References

[1] “V-safe After Vaccination Health Checker,” Centers for Disease Control and Prevention. [Online]. Available: https://www.cdc.gov/coronavirus/2019-ncov/vaccines/safety/vsafe.html. [Accessed: 18-Jul-2021].

[2] A. Silberschatz, H. F. Korth, and S. Sudarshan, Database system concepts. New York, NY: McGraw-Hill, 2011.

[3] “Least Privilege,” *Cybersecurity and Infrastructure Security Agency CISA*. [Online]. Available: https://us-cert.cisa.gov/bsi/articles/knowledge/principles/least-privilege. [Accessed: 19-Jul-2021].

[4] “SQL Injection,” *OWASP*. [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection. [Accessed: 20-Jul-2021].

[5] S. Behara, “Designing Highly Scalable Database Architectures,” Simple Talk, 03-Jun-2021. [Online]. Available: https://www.red-gate.com/simple-talk/databases/sql-server/performance-sql-server/designing-highly-scalable-database-architectures/. [Accessed: 21-Jul-2021].

[6] “The Infrastructure Behind Twitter: Scale,” Twitter. [Online]. Available: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale. [Accessed: 21-Jul-2021].

[7] “Creating a read replica in a different AWS Region,” AWS Docs. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_ReadRepl.XRgn.html. [Accessed: 21-Jul-2021].

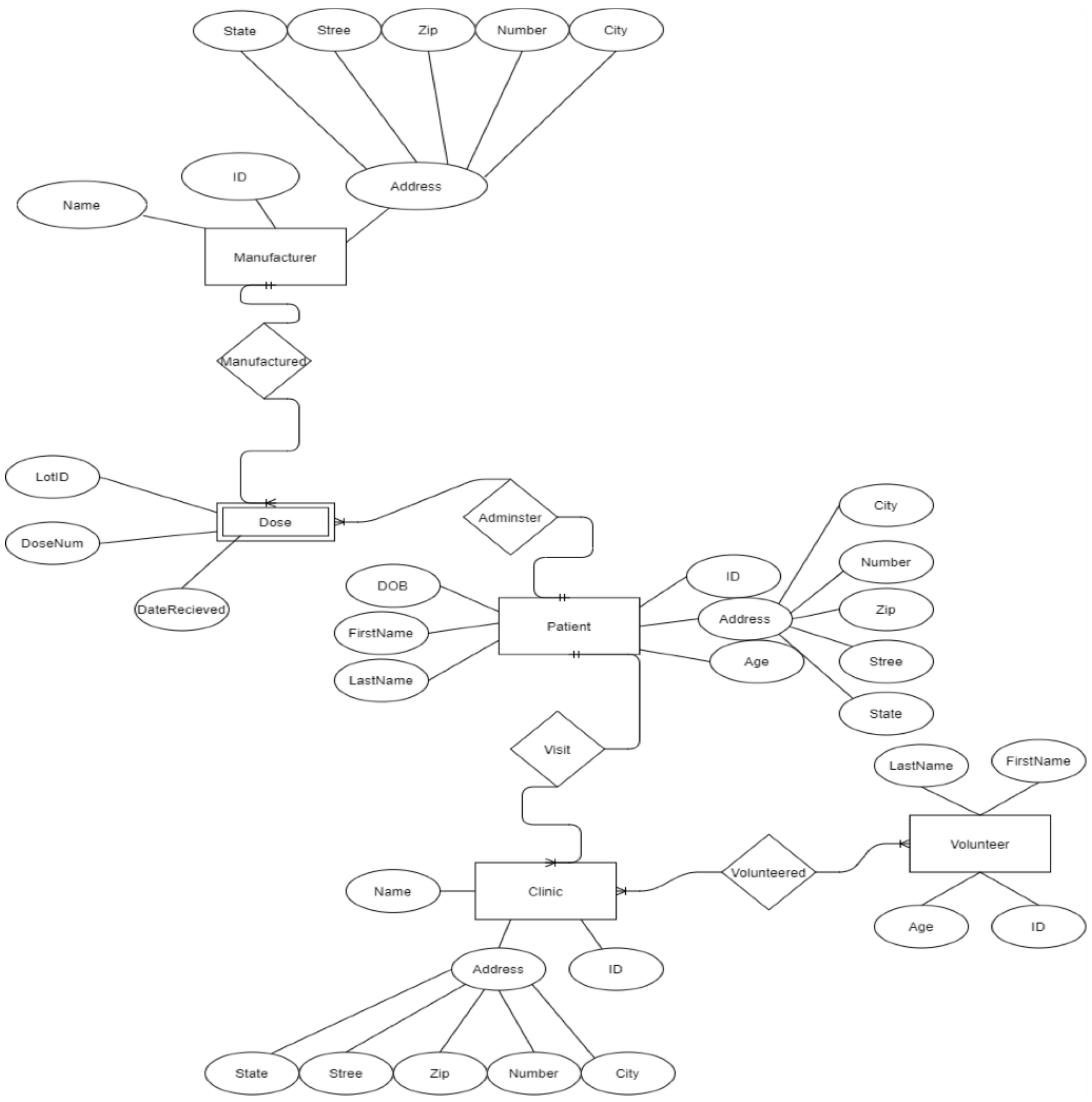
[8] “Single Page Applications: Guide to a Secure Login Pathway,” LoginRadius. [Online]. Available: https://www.loginradius.com/blog/start-with-identity/single-page-applications/. [Accessed: 21-Jul-2021].

[9] S. Engdahl, “Blogs,” Amazon, 2008. [Online]. Available: https://aws.amazon.com/blogs/containers/integrate-amazon-api-gateway-with-amazon-eks/. [Accessed: 21-Jul-2021].

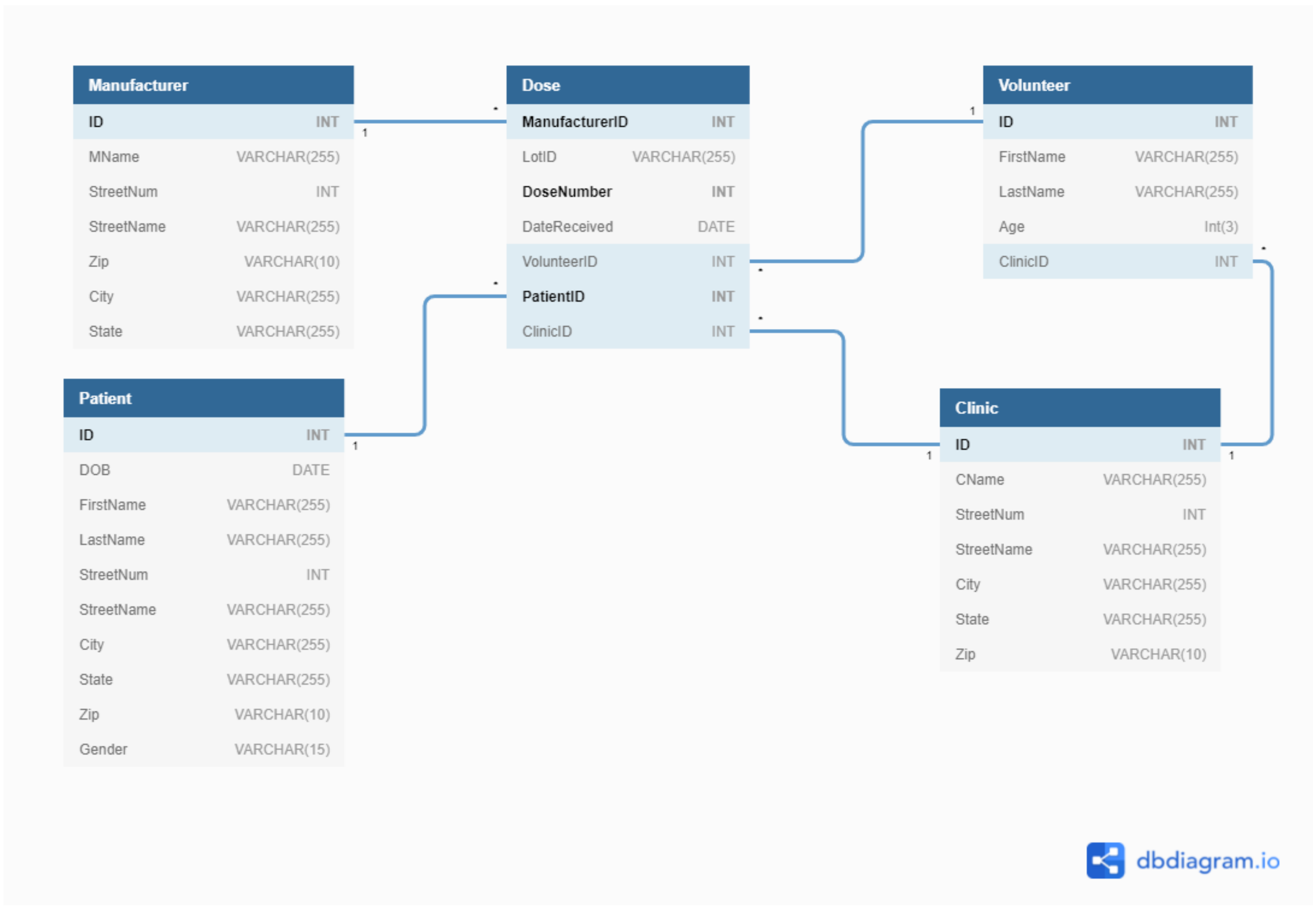
[10] M. Smallcombe, “ETL vs ELT: 5 Critical Differences,” Xplenty, 29-Nov-2018. [Online]. Available: https://www.xplenty.com/blog/etl-vs-elt/. [Accessed: 22-Jul-2021].

[11] O. E. on D. Integration, Nicholas Samuel on Data Driven Strategies, and S. A. on A. Software, “How to Create MySQL CDC Kafka Pipeline: Easy Steps,” Hevo, 14-Jul-2021. [Online]. Available: https://hevodata.com/learn/mysql-cdc-kafka/. [Accessed: 22-Jul-2021].

Appendix A


State
Stree
Zip
Number
City
ID
Name
Address
Manufacturer
Manufactured
LotID
Dose
DoseNum
DateRecieved
Adminster
City
Number
ID
DOB
Address
Zip
Patient
FirstName
LastName
Age
Stree
State
Visit
LastName
FirstName
Volunteer
Volunteered
Name
Clinic
Age
ID
Address
ID
State
Stree
Zip
Number
City

Appendix B



Manufacturer	
ID	INT
MName	VARCHAR(255)
StreetNum	INT
StreetName	VARCHAR(255)
Zip	VARCHAR(10)
City	VARCHAR(255)
State	VARCHAR(255)

Dose	
ManufacturerID	INT
LotID	VARCHAR(255)
DoseNumber	INT
DateReceived	DATE
VolunteerID	INT
PatientID	INT
ClinicID	INT

Volunteer	
ID	INT
FirstName	VARCHAR(255)
LastName	VARCHAR(255)
Age	Int(3)
ClinicID	INT

Patient	
ID	INT
DOB	DATE
FirstName	VARCHAR(255)
LastName	VARCHAR(255)
StreetNum	INT
StreetName	VARCHAR(255)
City	VARCHAR(255)
State	VARCHAR(255)
Zip	VARCHAR(10)
Gender	VARCHAR(15)

Clinic	
ID	INT
CName	VARCHAR(255)
StreetNum	INT
StreetName	VARCHAR(255)
City	VARCHAR(255)
State	VARCHAR(255)
Zip	VARCHAR(10)