Title: Big Data Byte Real Estate Database Design

Subtitle: Project Phase Three, ERD, Sample Data, and Storyboard

Group - Big Data Byte:

- ❏ Jason St. George
- ❏ Justin Lad
- ❏ Qiaoran Li
- ❏ Robert Kubiniec

Due Date: 7/27/2018

Professor: Jeremy Brown

Class: CSCI 620 Introduction to Big Data

School: Rochester Institute of Technology

Table of Content

Description of Project

The BigDataBytes Real Estate application is a prototype application for an enterprise real estate management firm. It provides comprehensive data storage and retrieval for the various needs of a real estate firm. The application provides storage for all properties on the market including associated info, a history of all sold properties by the firm, agent information, and buyer and seller information. This prototype is implemented via specific commands the user can enter, which are listed in detail on the User Commands document (see appendix 1). The next step is building a graphical interface to interact with the database, to make it a little more user friendly for the typical user. And finally, the finished product would connect the available properties to a webUI interface, so potential buyers can easily browse homes on the market.

The application is split into various sections, developed for each particular role in the real estate organization. The Sale History view is geared towards the brokers, who may want to keep tabs on which agents are bringing in the most money, and which ones are struggling. It will also tell the broker how much money the office is making per month, and other various administrative queries related to sales. The Available Properties view is constructed primarily for potential buyers, who would like to view homes are on the market and use various filtering, like properties in a range of cost, or homes in certain cities or states. Next, the agent view is built agents, so they can keep track of how much sales they are generating, how many properties they manage, and other features that would be useful to agents. The sold properties view will be useful for record keeping of which homes were sold when, which agent sold, and who bought the home. Lastly, the client list is geared towards agents, who need to access contact information for a buyer or seller.

A developer will be hired on retainer to handle database management, including updating data and inserting new data into the database. And more advanced users with appropriate credentials may enter any custom SQL commands for any functionality not supported by standard commands.

Database Design

The ER diagram is designed to make data easy to understand and query. We found that our initial design had serious flaws that made information redundant, difficult to obtain, or lost entirely. We revised our design to address the issues we found. We feel that the current design addresses all our customers' needs.

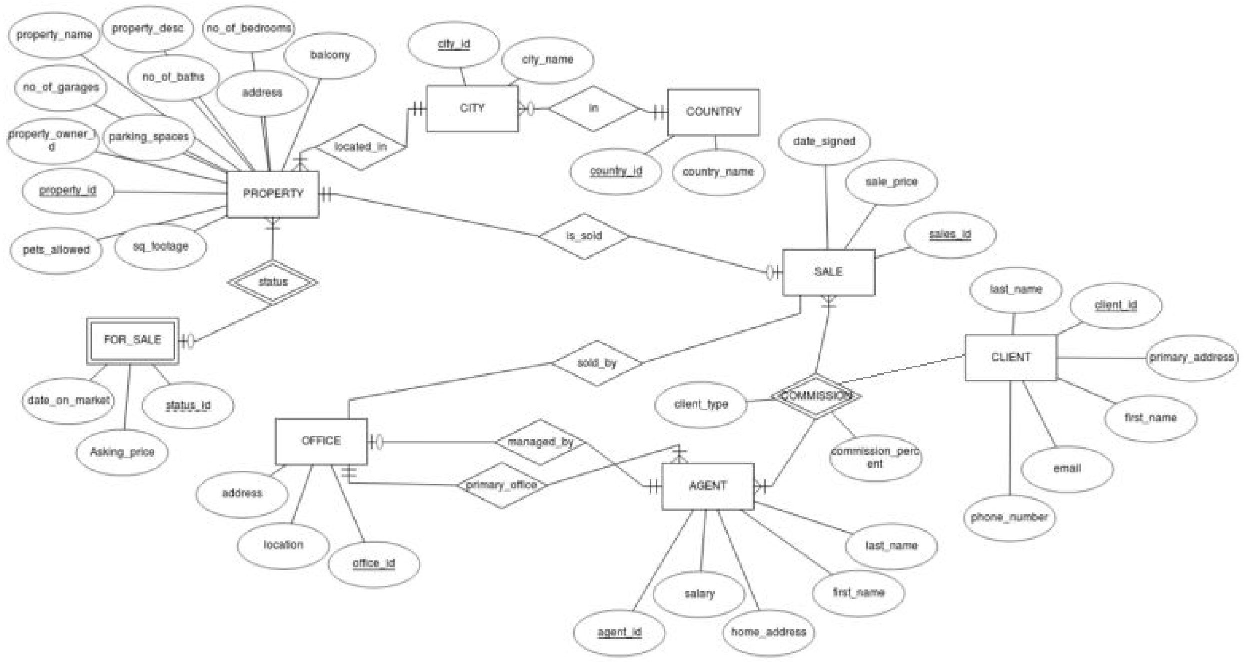


Figure 1. ER Diagram of the Database Design

The first and most important entity is PROPERTY, which includes information such as address, owner, square footage, and more. The location of a PROPERTY is stored via the relationship located_in, to connect to the PROPERTY address to CITY and COUNTRY entities.

PROPERTIES on the market are listed in FOR_SALE. FOR_SALE is a weak entity set because information from PROPERTY is needed to uniquely identify a PROPERTY's status (FOR_SALE.status_id and PROPERTY.property_id), so the uid is split into partial keys.

PROPERTY sales information is stored in the SALE entity, connected with the goes_on relationship. Each SALE is managed at one OFFICE, connected by the sold_by relationship. The SALE entity describes all associated information such as price and date_signed.

A three-way relationship called commission between SALE, AGENT, and CLIENT describes the contract between a CLIENT and an AGENT for a single SALE. This relationship includes an agreed commission percentage and whether the client is the buyer or seller.

AGENTS are described by the AGENT entity. AGENTS have a name, a salary, and basic contact information.

The OFFICE entity represents a real estate company's OFFICES, identified by the office_id key. Each OFFICE has an AGENT "manager," depicted in the managed_by relationship. Each AGENT has a one-to-one primary_office relationship with an OFFICE.

Lastly, the CLIENT entity describes CLIENTS who participate in SALES as either buyers or sellers. The CLIENT entity holds contact information for the buyers and sellers.

The result after converting this ER structure into relations is one relation for each entity and an additional table for the commission relationship which has foreign keys to SALE, AGENT, and CLIENT.

Database Design Choices

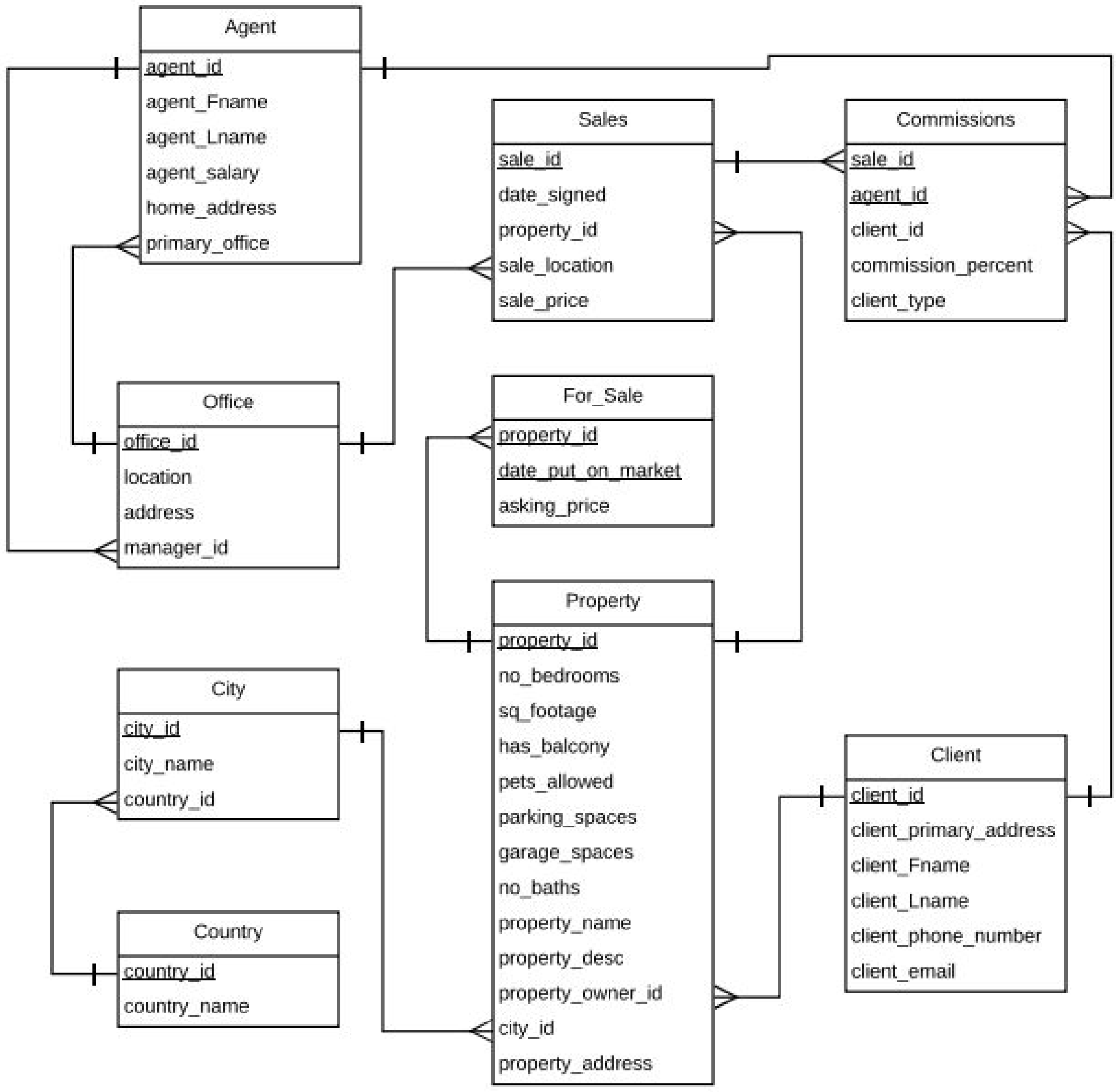


Figure 2. Relation Scheme of Underlying Database Design

This new design differs from previous versions significantly. As mentioned above, there were some unforeseen issues with our original design that we discovered when designing views and queries to fulfill desired use cases.

One change was consolidating the BUYER and SELLER tables into the CLIENT table. The old design meant that if a CLIENT buys and sells property then they would have been listed in each table with redundant information. If their information changed between SALES, the tables might be given contradicting data. To simplify the design and remove the potential for unreliable data, we put all CLIENTS in the same table and used a "status" attribute in the commission contract table to differentiate between buyers and sellers.

The commission contract table itself also went through a significant overhaul. We found that storing buyer, seller, buyer's agent, and seller's agent in the SALE table seemed clunky, and it made it hard to query information about relationships between AGENTS and CLIENTS because their relationship was not explicit. A three-way relationship between AGENT, SALE, and CLIENT might be complicated, but we feel that this structure is a better fit for the situation, and most importantly it allows queries about the relationships between all of AGENTS and their SALES, CLIENTS and their SALES, and CLIENTS and their AGENTS.

One last change we made was to remove the RECENTLY_SOLD table. This type of table is difficult to maintain because entries would have to be removed automatically on a regular schedule based on some definition of "recently sold." Also, when designing queries a user might need relating to recent SALES, we found that querying the SALE table directly with a specific range on date_signed was more intuitive and more likely to give the desired results.

SQL Table Creation

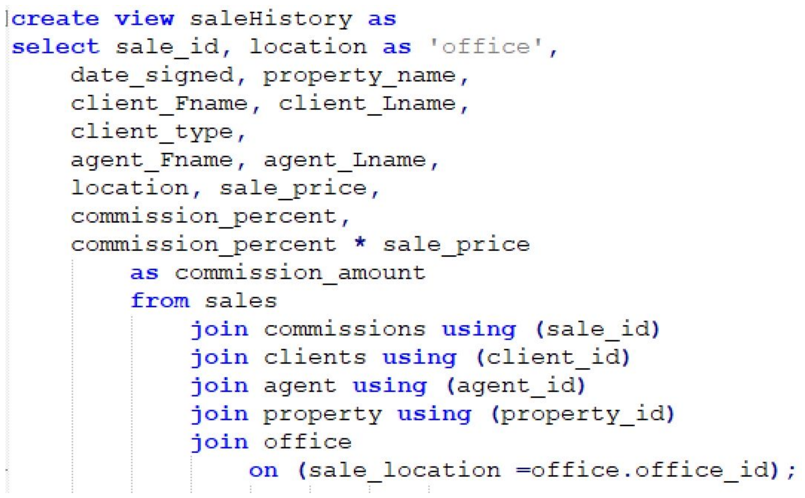
```
# Qiaoran Li
# this file creates the tables needed for the project
# note: tinyint is boolean
# 7/3/2018

CREATE database bigdatabyte;
USE bigdatabyte;

CREATE TABLE `country` (
  `country_id` varchar(10) NOT NULL,
  `country_name` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`country_id`)
);

CREATE TABLE `city` (
  `city_id` int(11) NOT NULL,
  `city_name` varchar(50) DEFAULT NULL,
  `country_id` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`city_id`),
  KEY `city_ibfk_1` (`country_id`),
  CONSTRAINT `city_ibfk_1` FOREIGN KEY (`country_id`) REFERENCES `country`
(`country_id`)
);

CREATE TABLE `clients` (
  `client_id` int(11) NOT NULL,
  `client_primary_address` varchar(50) DEFAULT NULL,
  `client_Fname` varchar(20) DEFAULT NULL,
  `client_Lname` varchar(20) DEFAULT NULL,
  `client_phone_number` varchar(12) DEFAULT NULL,
  `client_email` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`client_id`)
);

CREATE TABLE `property` (
  `property_id` int(11) NOT NULL,
  `no_bedrooms` tinyint(1) DEFAULT NULL,
  `sq_footage` int(11) DEFAULT NULL,
  `has_balcony` tinyint(1) DEFAULT NULL,
  `pets_allowed` tinyint(1) DEFAULT NULL,
  `parking_spaces` int(11) DEFAULT NULL,
  `garage_spaces` tinyint(1) DEFAULT NULL,
  `no_baths` int(11) DEFAULT NULL,
  `property_name` varchar(20) DEFAULT NULL,
  `property_description` varchar(200) DEFAULT NULL,
  `property_owner_id` int(11) DEFAULT NULL,
  `city_id` int(11) DEFAULT NULL,
  `property_address` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`property_id`),
  KEY `city_id` (`city_id`),
  KEY `property_owner_id` (`property_owner_id`),
  CONSTRAINT `property_ibfk_1` FOREIGN KEY (`city_id`) REFERENCES `city` (`city_id`),
  CONSTRAINT `property_ibfk_2` FOREIGN KEY (`property_owner_id`) REFERENCES
`clients` (`client_id`)
);

CREATE TABLE `for_sale` (
  `property_id` int(11) NOT NULL,
  `date_put_on_market` date NOT NULL,
  `asking_price` decimal(15,2) DEFAULT NULL,
  PRIMARY KEY (`property_id`,`date_put_on_market`),
  CONSTRAINT `for_sale_ibfk_1` FOREIGN KEY (`property_id`) REFERENCES `property`
(`property_id`)
);
```

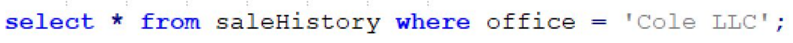
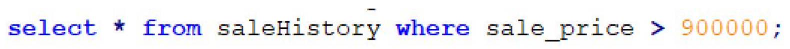
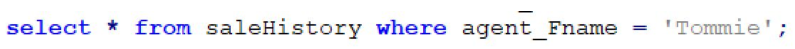
```
CREATE TABLE `office` (
  `office_id` int(11) NOT NULL,
  `location` varchar(50) DEFAULT NULL,
  `address` varchar(50) DEFAULT NULL,
  `manager_id` int(11) DEFAULT NULL,
  PRIMARY KEY (`office_id`)
);

CREATE TABLE `agent` (
  `agent_id` int(11) NOT NULL,
  `agent_Fname` varchar(20) DEFAULT NULL,
  `agent_Lname` varchar(20) DEFAULT NULL,
  `agent_salary` decimal(10,2) DEFAULT NULL,
  `home_address` varchar(50) DEFAULT NULL,
  `primary_office` int(11) DEFAULT NULL,
  PRIMARY KEY (`agent_id`),
  KEY `primary_office` (`primary_office`),
  CONSTRAINT `agent_ibfk_1` FOREIGN KEY (`primary_office`) REFERENCES `office`
(`office_id`)
);


ALTER TABLE office
        ADD FOREIGN KEY (manager_id) REFERENCES agent(agent_id);

CREATE TABLE `sales` (
  `sale_id` int(11) NOT NULL,
  `date_signed` date DEFAULT NULL,
  `property_id` int(11) DEFAULT NULL,
  `sale_location` int(11) DEFAULT NULL,
  `sale_price` decimal(10,2) DEFAULT NULL,
  PRIMARY KEY (`sale_id`),
  KEY `property_id` (`property_id`),
  KEY `sales_ibfk_5` (`sale_location`),
  CONSTRAINT `sales_ibfk_1` FOREIGN KEY (`property_id`) REFERENCES `property`
(`property_id`),
  CONSTRAINT `sales_ibfk_5` FOREIGN KEY (`sale_location`) REFERENCES `office`
(`office_id`)
);

CREATE TABLE `commissions` (
  `sale_id` int(11) NOT NULL,
  `agent_id` int(11) NOT NULL,
  `client_id` int(11) DEFAULT NULL,
  `commission_percent` decimal(2,2) DEFAULT NULL,
  `client_type` tinyint(4) DEFAULT NULL,
  PRIMARY KEY (`sale_id`,`agent_id`),
  KEY `agent_id` (`agent_id`),
  KEY `client_id` (`client_id`),
  CONSTRAINT `commissions_ibfk_1` FOREIGN KEY (`sale_id`) REFERENCES `sales`
(`sale_id`),
  CONSTRAINT `commissions_ibfk_2` FOREIGN KEY (`agent_id`) REFERENCES `agent`
(`agent_id`),
  CONSTRAINT `commissions_ibfk_3` FOREIGN KEY (`client_id`) REFERENCES `clients`
(`client_id`)
);
```

SQL Views and Scripts

The following table provides a sample overview of the views and scripts used in this project, the original code can be found in createDB.sql or viewsAndScript.sql. **For more scripts, please refer to userCommand.pdf**.

Query	**Who uses this?**	**What information does it reveal?**
The following section includes views and script that illustrates the details of sales history. For example, the view saleHistory includes the date, office, buyer, seller, buyer agent, seller agent, sale_price, and commission for each property sold. Certain users, can add special filter to the result. For example, viewing a list of sales historys by office location, sale_price, or agent names, and etc.		

 `create view saleHistory as`
`select sale_id, location as 'office',`
`    date_signed, property_name,`
`    client_Fname, client_Lname,`
`    client_type,`
`    agent_Fname, agent_Lname,`
`    location, sale_price,`
`    commission_percent,`
`    commission_percent * sale_price`
`        as commission_amount`
`        from sales`
`            join commissions using (sale_id)`
`            join clients using (client_id)`
`            join agent using (agent_id)`
`            join property using (property_id)`
`            join office`
`                on (sale_location =office.office_id);`
	Agent, client, payroll	A list of sales historys

 `select * from saleHistory where office = 'Cole LLC';`	Agent, client, payroll	A list of sales history by a particular office location
 `select * from saleHistory where sale_price > 900000;`	Agent, client, payroll	A list of sales history within a sale price range
 `select * from saleHistory where agent_Fname = 'Tommie';`	Agent, Client, payroll	A list of sale history by a particular agent via bio information

The following section includes views and script that illustrates the details of available properties. For example, the property id, asking price, number of bedrooms, square footage, whether it has balcony, whether pets are allowed, number of parking spaces, number of garage spaces, number of bathrooms, name of property, description of property, client name, city name, last date put on market are shown in the output. Certain user can add special filters to the output such as by agent, asking price, or owners.

`create view availableProperty as`
`select property_id, location as 'office',`
`    asking_price, no_bedrooms,`
`    sq_footage, has_balcony, pets_allowed,`
`    parking_spaces, garage_spaces, no_baths,`
`    property_name, property_description,`
`    client_Fname as 'owner_Fname',`
`    client_Lname as 'owner_Lname',`
`    property_address, city_name,`
`    max(date_put_on_market) as 'last_posting_date',`
`    max(date_put_on_market) > max(date_signed)`
`    as on_the_market`
`    from for_sale`
`        join sales using (property_id)`
`        join property using (property_id)`
`        join office on (sales.sale_location = office.office_id)`
`        join clients on (property_owner_id = clients.client_id)`
`        join city using (city_id)`
`            group by property_id`
`                having on_the_market = 1;`
	Agent,
	
	Client
		A list of available properties

`select * from availableProperty where last_posting_date > '2018-05-01';`	Agent, Client	A list available properties by posting date
`select * from availableProperty where asking_price < 100000;`	Agent, Client	A list of available properties within a asking price range
`select * from availableProperty where owner_Fname = 'Tiffi';`	Agent	A list of available properties with a particular owner/seller

The following section includes views and script that illustrates the details of sold properties. For example, the property id, office, sale price, number of bedrooms, square footage, whether it has balcony, whether pets are allowed, number of parking spaces, number of garage spaces, number of bathrooms, name of property, description of property, client name, city name, date signed are shown in the output. Certain user can add special filters to the output such as by date signed, sale price, or property info.

`create view soldProperty as select property_id, location as 'office', sale_price, no_bedrooms, sq_footage, has_balcony, pets_allowed, parking_spaces, garage_spaces, no_baths, property_name, property_description, client_Fname as 'owner_Fname', client_Lname as 'owner Lname', agent_Fname as 'agent_Fname', agent_Lname as 'agent_Lname', property_address, city_name, date_signed, max(date_put_on_market) < max(date_signed) as on_the_market from for_sale join sales using (property_id) join commissions using (sale_id) join agent using (agent_id) join property using (property_id) join office on (sales.sale_location = office.office_id) join clients on (property_owner_id = clients.client_id) join city using (city_id) group by property_id having on_the_market = 1;`	Agent, Client	A list of sold properties
`select * from soldProperty where date_signed < '2018-05-01';`	Agent, Client	A list of sold properties before 2018-5-1
`select * from soldProperty where sale_price < 500000;`	Agent, Client	A list of sold properties within a specific sale price range
`select * from soldProperty where has_balcony = 1;`	Agent, client	A list of sold properties with balconies
`select * from availableProperty where propery name like 'Otcom';`	Agent, Client	A list of available properties with a particular name

The following section includes views and script that illustrates the details of client, which can be a buyer, seller, or both depending on the particular sale record. For example, information such as the client id, office, his/her primary address (the place he/her can be contacted), client type, client name, client phone, client email are presented in the query result. Certain users can

special filters such as viewing the results by the sale office, client, and etc.		

`create view clientList as`
`select client_id,`
`    location as 'office',`
`    client_primary_address,`
`    client_type,`
`    client_Fname, client_Lname,`
`    client_phone_number,`
`    client_email`
`    from clients`
`        join commissions using (client_id)`
`        join sales using (sale_id)`
`        join office on (sale_location = office.office_id);`
	Agent	A list of customers

`select * from clientList where office = 'Cremin Group';`	Agent	A list of customers by a particular office
`select * from clientList where client_Fname = 'Rad';`	Agent	A list of customers by bio information
The following section includes views and script that illustrates the details of the agents. For example, information such as agent id, office, agent name, agent salary, agent home address are included in the query result. Certain user, like the payroll, can view the agents by their office, or bio information such as names, phone, or address.		

`create view agentList as`
`select agent_id, location as 'office',`
`    agent_Fname, agent_Lname,`
`    agent_salary, home_address`
`        from agent join office`
`            on (primary_office = office_id);`
	Payroll	A list of agents

`select * from agentList where office = 'Cole LLC';`	Payroll	A list of agent by a particular office
`select * from agentList where agent_Fname = 'Tommie';`	Payroll	A list of agent by agent bio

		information

`######## SAMPLE MOD #########################`
`# client modify his information`
`update client set Fname = 'Qiaoran';`
`update client set phone_number = '585-000-0000';`
 	
`# seller modify his information`
`update seller set Fname = 'James';`
`update seller set phone_number = '585-111-1111';`
 	
`# payroll change commission amount - NOT CORRECT`
`update commissions`
`    join agent on (commissions.agent_id = agent.agent_id)`
`        set commission_percent = 0.28`
`        where agent.agent_id = 101;`
	As seen in comments	A few update sample queries. Users can modify their personal information or perform office duty (payroll changing commission).

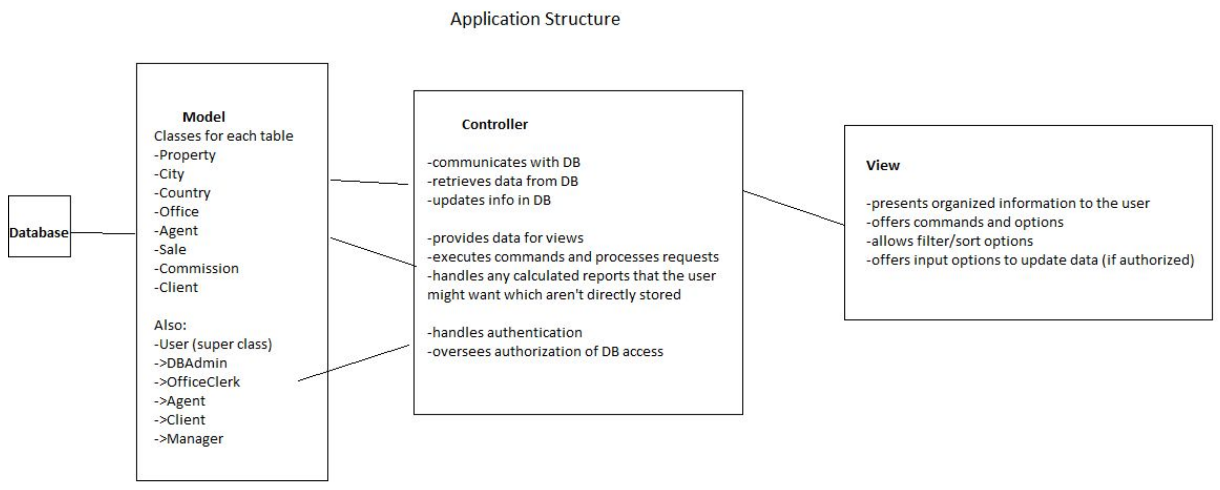
`############### manage users ######################`
`create role clients;`
`grant select on availableProperty to clients;`
`grant clients to Robert;`
	DBA	Create and manage users

Application Design

The Java app is connected to the database with Java's built in libraries. The basic structure is that the user identifies a view they would like to access (ideally this would be determined by their role), and then from the view they can query specific information they would like to see. Each command is essentially a unique set of keywords defined in our manual followed by optional "specific" parameters that narrow the search by certain fields (ex. filtering properties by a specific city).

We implemented a Model-View-Controller paradigm to organize the code and separate concerns. The Model is the database and the low-level Java/SQL code that hooks up the data to the app. Our View is a simple command line experience that gives the user their formatted data. In between, a controller turns the raw data of the tables and SQL views into different formats for different users, issues commands including updates to the database, and handles user authorization/authentication and role management.

The below diagram outlines our plan. The development team decided that formalities like implementing classes for each table were excessive for the scope of this project, so the MVC design is more implicit than originally planned, but the Model, View, and Controller can be identified as separate entities when inspecting the code.


Application Structure
Database
Model
Classes for each table
-Property
-City
-Country
-Office
-Agent
-Sale
-Commission
-Client
Also:
-User (super class)
->DBAdmin
->OfficeClerk
->Agent
->Client
->Manager
Controller
-communicates with DB
-retrieves data from DB
-updates info in DB
-provides data for views
-executes commands and processes requests
-handles any calculated reports that the user might want which aren't directly stored
-handles authentication
-oversees authorization of DB access
View
-presents organized information to the user
-offers commands and options
-allows filter/sort options
-offers input options to update data (if authorized)

Use Cases

Our app focuses on the needs of three types of user: Real estate agents, clients, and administrative employees of the real estate office. We prioritized delivering a great experience for agents and clients as the main users of this database application, but we didn't want to leave out other important employees such as payroll who need access to information about the real estate agents of the office to process commission.

Agents are the primary users of our program. Agents will likely focus more on the buyers/sellers and will need to access information about them. They will primarily use available properties, client list, and agent list views. They have the following use cases:

- -List contact information for specific seller
- -List contact information for specific buyer or buyers who are linked to a specific office
- -Finds the average price of properties sold by that office
- -List agents with the most commission globally or by office (ascending or descending)

Clients are the other main users of our database. For the purposes of use cases, we can divide Clients into Buyers and Sellers:

Buyers will be able to view all the properties for sale at a certain office, by state, or by city. Preferably, this will feed a webUI display, where the user can scroll through pictures and descriptions of properties, using various filters like office, city, state, zip. This information is fed by the available properties view. Their use cases include:

- -List all properties in a city / state / zip code
- -List all properties in a city / state /zip code under a certain price, over a price, or within a range or by office

Sellers will be able to look at the recent sales records and get a feel for sales prices in the area. To do so, they will use the available properties and sales history views.

Office employees with duties like payroll need to use the application as more of an administrator and pull information about agents and sales data. They will use sales history, available property, sold property, and agent list views to resolve the following queries. Their use cases include:

- -List all sales for their office
- -List agents with the most commission for their office (ascending or descending)
- -Review the workload of the agents by checking how many properties each agent manages
- -Find out sales data for the office
- -Lists all the property sales (and resulting commission) by the office or within that last x days.

Last but not least, database administrators are able to handle update/insert commands, and modify the database if needed.

Team Contributions and Project Experience

For a team with 4 members, we did a remarkable job of dividing the work evenly. Everyone contributed significantly to the project, and everyone had their moment to take charge of the situation and make sure something important got delivered.

Phase 0 was all Jason's work. Jason started the team by taking the initiative ahead of the class start date and reaching out to classmates. By recruiting us early he assembled a team of four proactive and productive individuals. Jason wrote and submitted the phase 0 documents.

Starting with phase 1, we started coordinating our team efforts. We used a discord server to have weekly meetings. Jason started the database design and took ownership of the ER diagram. He was receptive to everyone's feedback, and the discussions we had online enhanced the design. Qiaoran found a website to generate fake data to fit our needs, and she also handled the task of document organization in our team google drive, assembling diagrams, data, and report fragments into a formatted submission. Robbie and Justin split the written portion of this phase evenly. Robbie performed the final zipping and submitting of phase 1.

Phase 2 went as smoothly as phase 1. We decided early on that having four developers on this relatively small Java app was impractical. Jason and Justin were designated coders. Qiaoran implemented the database in MySQL. Robbie helped with the high-level MVC app design, made the UML diagram, updated the database design and ER diagram as needed, and wrote most of the report with help from Qiaoran on the sample data and Justin on use cases. On the day of the deadline Qiaoran put the finishing touches on everything and submitted. Again, our weekly online meetings kept everyone in the loop, and no one delivered less than they promised.

By the time phase 3 began most of us were working in addition to this class, so we got a bit sloppy about organization. The team failed to meet for a couple of weeks. Finally Justin kept everyone accountable and got us back on track with an in-person meeting on RIT's campus to iron out the remaining issues. After that, Jason and Justin finished the Java code. Qiaoran handled database/Java app integration testing. Robbie wrote this report.

The overall experience was very positive. Everyone on the team went above and beyond to ensure the project would exceed expectations. There were just a few bumps in the road. On the phase 2 submission day, everyone said they finished their part and Qiaoran volunteered to submit. While checking over everything she found the report was missing screenshots of the app in use. Robbie and Justin weren't online so she handled it herself, finished the report, and submitted. The whole team was grateful for her attentiveness and problem-solving. As mentioned above, phase 3 was a bit messy due to the team's other commitments, but everyone found the time to do their part right. On the technical side, MySQL was a pain to work with. Qiaoran was the only team member to successfully get it running the database, so she had to do all the testing. Maintaining the updated Java code was difficult because we didn't use source control, but fortunately there were no issues resulting from manually merging the file. Besides those small hiccups, this project went as well as a team project could possibly go.