Hassan Najjar, Stephania Abdallah, Khaled Hasbini

LAU

**Cedar Real Estate Consortium Database Report**

**Table of Contents**

[**Introduction:** 2](#_Toc153310606)

[**Requirements:** 3](#_Toc153310607)

[**Table Creation:** 5](#_Toc153310608)

[**Relations Description:** 13](#_Toc153310609)

[**Basic SQL:** 15](#_Toc153310610)

[**Advanced SQL:** 30](#_Toc153310611)

[**Conclusion:** 38](#_Toc153310612)

(ctrl + click to go to specified section)

# 

# **Introduction:**

The ***Cedar Real Estate Consortium***, a collaboration among multiple real estate agencies in Lebanon, has built an advanced database system. This system is analogous to a major organizer, keeping track of the significant features and functions of the consortium such as agencies, branches, departments, property deals, client connections, and money matters. The goal of this consortium is to achieve smoother work, keep clients satisfied, and aid in better and united workplace in Lebanon's real estate world.

This database holds a lot of information: details about agencies and their branches, different departments, who manages what, property sales and rents, client's interaction, payments, and feedback. In this report, we'll explain how this database is set up, what it contains, and how all the parts connect.

# **Requirements:**

The ***Cedar Real Estate Consortium***, an alliance of various real estate agencies operating in Lebanon, has implemented an extensive database system to streamline their operations effectively. This database structure is similar to a well-structured library with distinct sections allocated for various aspects vital to their functioning.

Within this organized framework, there exists a designated section devoted to "**Agencies**." This segment holds critical information concerning each agency, including essential details such as email addresses, names, and physical locations. Each agency oversees multiple "**Branches**", these branches possess their individual sections, containing information such as contact numbers and addresses.

Embedded within these branches are diverse "**Departments**" administered by appointed "**Managers**". Data concerning these managerial roles, including their names and assigned departments, is well organized within separate sections.

Collaborating closely with the agencies are "**Agents**" who facilitate property transactions on behalf of "**Owners**". Agents have dedicated sections within the database, cataloging relative details such as their names, email addresses, and affiliations with respective agencies. Revenue generated from property transactions is divided between the agents and the property "**Owners**," with the latter's information being recorded distinctly.

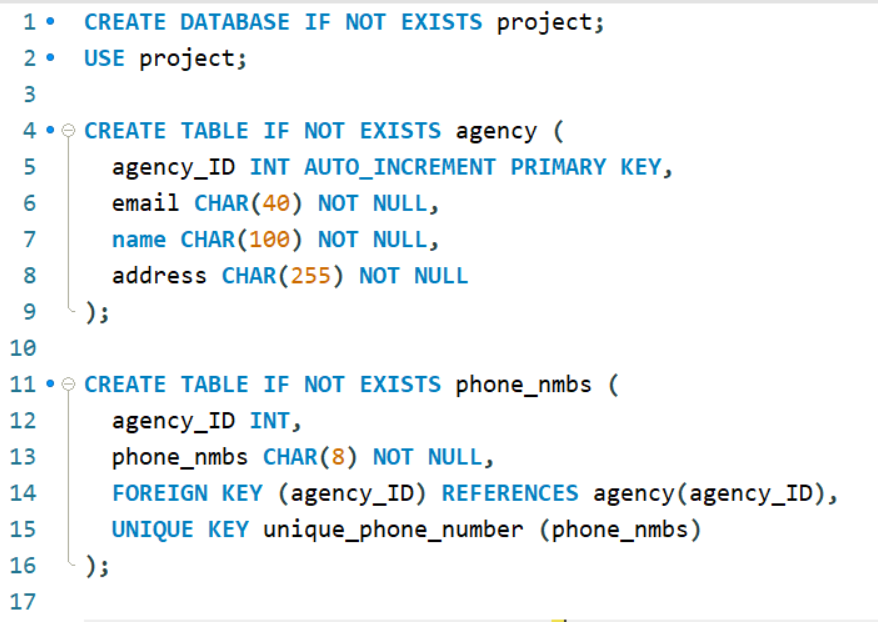
All properties, whether for sale or rent, are archived systematically within a segment labeled "**Property**". This section encompasses critical details regarding each property, including specifics like addresses, sizes, prices, and transaction types. Additionally, there exists a specialized area for "**Maintenance Companies**" entrusted with the upkeep of these properties.

The consortium engages with two categories of clients: "**Old Clients**", with a history of residing in their properties, and "**New Clients**" seeking properties presently. These distinct client groups are allocated individual segments within the database, housing comprehensive details encompassing their names, contact information, financial constraints, and property preferences.

Furthermore, financial transactions between clients and agents are recorded within a designated segment termed "**Payment**". This section comprehensively documents the specifics of each transaction, including payment amounts and methods.

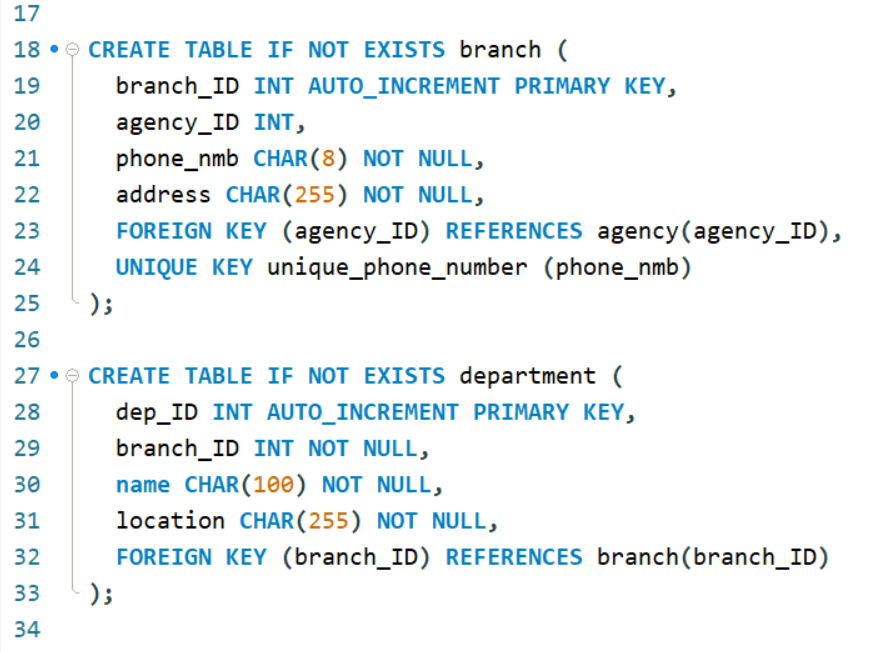
In essence, this systematically structured database system serves as a pivotal tool for the Cedar Real Estate Consortium, allowing efficient oversight of their office operations, property dealings, client engagements, and financial transactions in a well-ordered and proficient manner.

# **Table Creation:**



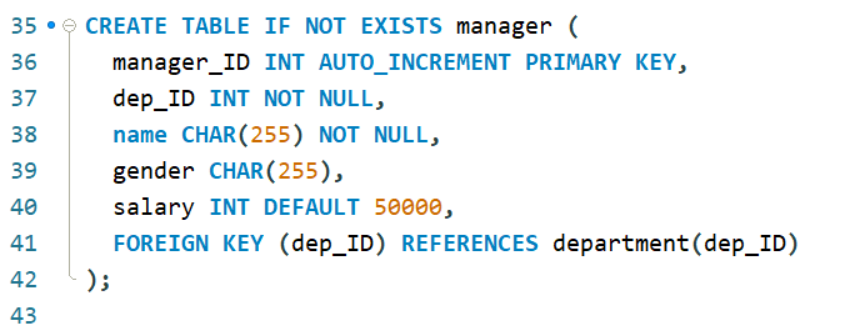
The code above deals with creating a database called “project” if no such database already exists, then picks the database to work in it.  
It then checks if the table already exists, if not, it creates a table for the agencies to be stored in, each agency has an agency\_ID that is auto-incremented (doesn’t get inserted and is incremented automatically) , which will be a recurring theme is this database, as a way to make the database easier to work with avoiding the issue of conflicting IDs, it also has an email, a name, and an address which have to be specified due to the importance of them.

The agency also has multiple phone numbers (multivalued attribute) which need to be presented in a table alone, each phone number has an agency\_ID value which refers to which agency it belongs to, as well as the actual phone number. Here the agency\_ID is a foreign key from the agency table so we referenced it, and each phone number should be unique because no 2 phone numbers are the same.

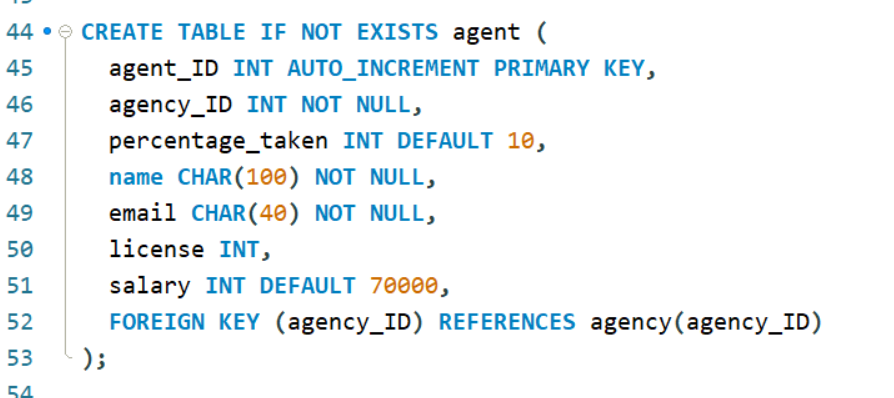


This code creates 2 tables, branch and department. The branch table stores all the branches that each agency has, also having an auto-incremented branch\_ID. The branch also has an agency\_ID foreign key which refers to what agency the branch is part of. Each branch has 1 phone number which is unique and represented in the phone\_nmb value, as well as having an address.

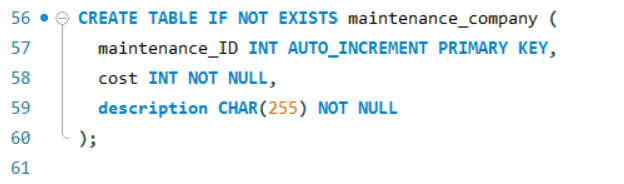
Then Each branch has multiple departments so a table for departments contains an automized dep\_ID and foreign key branch\_ID to link the department to the branch it is part of. As well as having a department name and location within the branch complex.



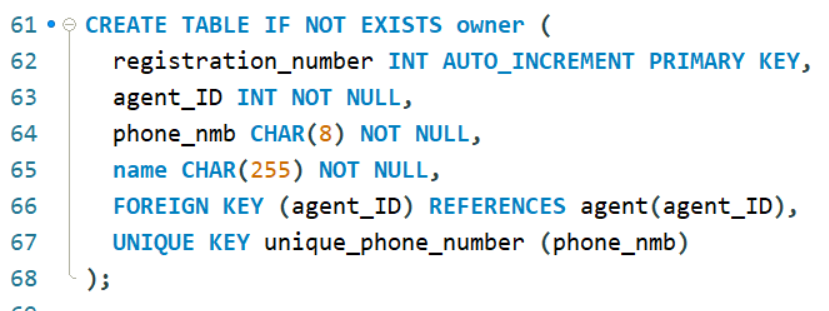
This code provides a data table for the managers of the departments, the manager IDs are automated, the id of the department the manager is responsible for is a foreign key referring to the dep\_ID value in the departments table, also having a name, gender, and a salary of default value 50000.



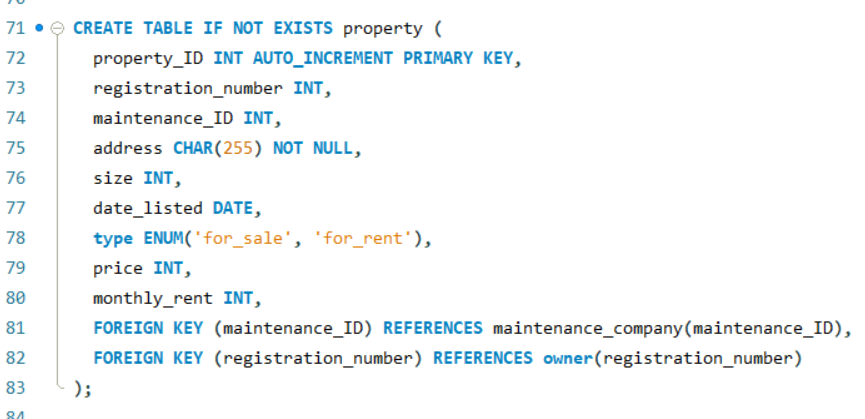
The agent table stores information about the agents such as an automated agent\_ID primary key, a foreign key of agency\_ID referencing the agency that the agent works in, a percentage\_taken attribute which refers to the percentage of the money earned from the selling of a property that an agent gets to keep as a reward for actually selling the property, the default value is 10%, a name for the agent, an email, a license that lends them credibility, and their salary is set to 70000 by default if no value is inserted.



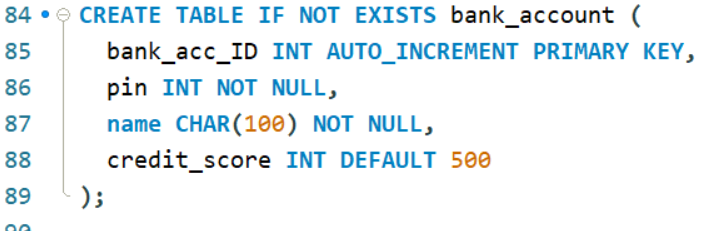
The database also stores data about the maintenance companies that work with the agencies to keep the properties clean and in good shape, the maintenance company table has an automated maintenance\_ID for the company, a cost which they set for their work, and a description for what they do and specialize in so agencies can know what they need and pick for each property.



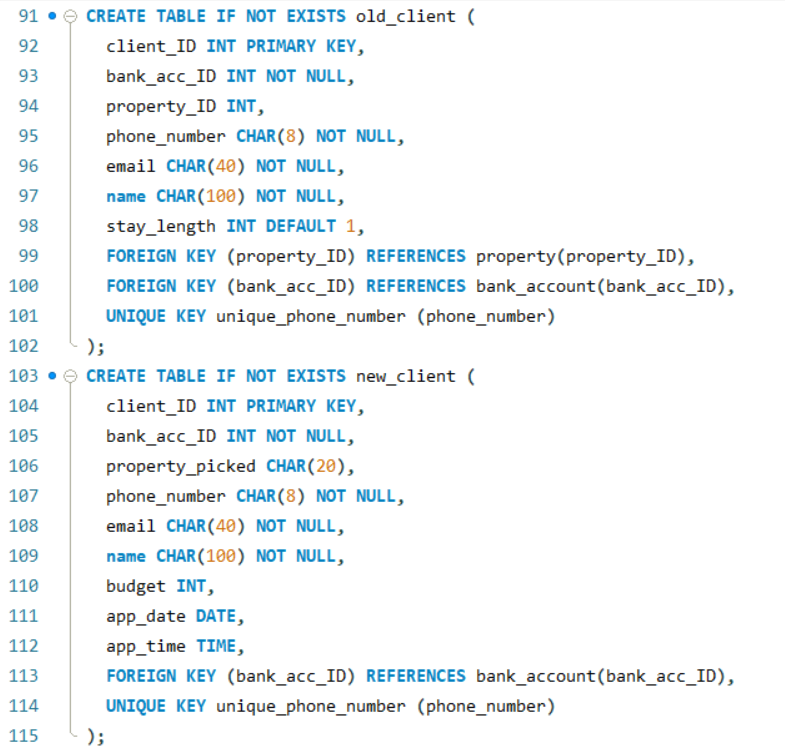
The owner table has a “registration\_number” value which refers to the number at which the owner registered in the database at, it is automated and a primary key. There is also an agent\_ID value which is a foreign key that refers to the agent in which the property owner is working with to sell the property they own. There also is a unique phone number for each owner and a name for them.



The property table stores information about each property added to the database, an automated property\_ID keeps track of them and ensures easy storing. The registration\_number is a foreign key referring to the primary key of the owner that owns this property, since an owner can own multiple properties. Each property should also have another foreign key referring to the maintenance company that is working on keeping this property in good conditions.

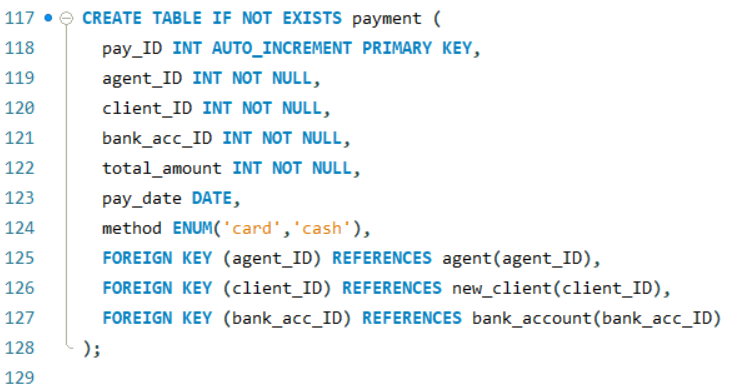


The bank account table stores information about all the bank accounts that go around and are being used in the agencies by the clients for ease of use with loyal customers. An automated bank\_acc\_ID keeps track of the bank accounts, a pin to access the bank account which cannot be set to null, the name of the bank that the account is made in, and the credit score for the client so that the agents can know who to trust and sell to, because a low credit score would mean a risk of not being able to commit to financial deadlines.

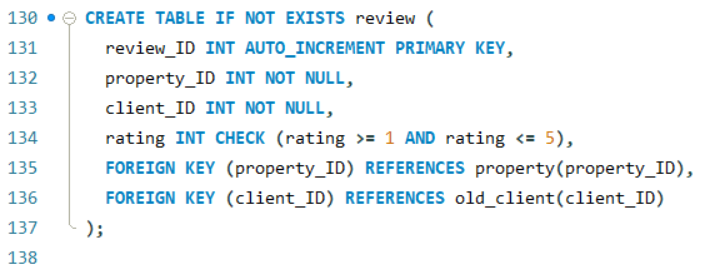


These 2 tables deal with the 2 types of clients we have, new clients that are looking to buy/rent, and old clients that used to be residents at some of the properties before. A manual client\_ID to avoid conflict between the 2 tables, the ID of the bank account, a unique phone number, an email and a name for both tables.  
The new client table has specific values to it such as a budget attribute that keeps track of how much the client is willing to spend so the program knows what to show to the client. An appointment date and time “app\_date” and ”app\_time” which specify the appointment/ house touring that the client is going to do with the client to figure out whether they like the property or not.

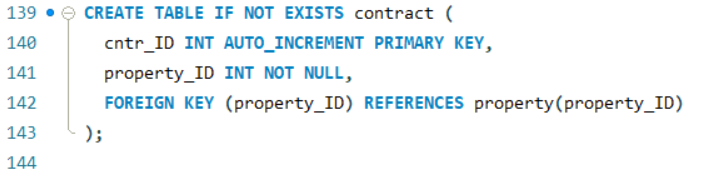
The old clients table has the value of “stay\_length” in which it specifies the time that the old client stayed for.



The payment (which is a weak entity) table has an automated pay\_ID, as well as foreign keys of agent\_ID to refer to who sold properties and who didn’t so the company keeps track of their employees. As well as a client\_ID and a bank\_acc\_ID that refer to the client and the bank account they used to make the payment.  
There is also a “total\_amount” attribute that tells the amount paid by the client to buy a property, and a pay\_date that keeps track of what days the payment was executed. And a method value that can only take 2 values, “card” and “cash” so that the agency knows what to do.



The review table has an automated ID system and a property\_ID that refers to the ID of the property being reviewed, and a client\_ID that references the client who used to live In the property.



The contract table stores values on the database, the contract has an automated ID system and a property\_ID foreign key that references the property that is being reviewed.

# **Relations Description:**

This section deals with explaining what the relations in the database do, and what entities they are connected to.

**Works\_in:**

Establishes the employment relationship between an agent and an agency. Indicates that an agent is employed by an agency, allowing multiple agents to work within a single agency, while each agent is exclusively associated with only one agency.

**Has (agency – branch):**

Represents the ownership relationship between an agency and its branches. Specifies that an agency can own multiple branches across various locations, while each branch is affiliated with only one agency.

**Divided:**

Denotes a weak entity relationship between a branch and its departments. Ensures that each branch is divided into distinct departments, and each department is linked to only one branch.

**Managed\_by:**

Establishes the managerial association between departments and their managers. Ensures a one-to-one relationship where each department has precisely one manager, and each manager oversees a single department.

**Receives:**

Represents the transactional link between an agent and client payments. Specifies that an agent receives payment from a client for a property sale, ensuring that each payment is directed to only one agent.

**Monetary\_transaction:**

Indicates the financial transactions occurring from a client's bank account. Specifies that each payment originates from a single bank account while allowing multiple payments from the same account for various services.

**Payment\_record:**

Establishes a record-keeping relationship between client payments and their activities. Represents a linkage between a weak entity (payment) and its parent entity (client), providing a comprehensive overview of payment-related activities.

**Assigns:**

Ensures that each client possesses a bank account, facilitating cashless payment options for transactions within the service.

**Writes:**

Allows old clients to provide feedback in the form of reviews based on their property stay experiences. Ensures that reviews originate from old clients who have previously resided in the property.

**Reads:**

Enables new clients to access and review feedback left by old clients, aiding their decision-making process regarding property selection.

**Gives:**

Represents the commission distribution relationship between agents and property owners. Specifies that agents receive a percentage from the property sale price as earnings, and the remaining balance is transferred to the respective owner. Each agent can earn from multiple owners but can only earn from one transaction at a time.

**Owns:**

Indicates the direct ownership relationship between property owners and their properties. Ensures that each property is owned by one owner while allowing owners to possess multiple properties.

**Has (propery-contract):**

Establishes the contractual relationship required for property sales or rentals, facilitating agreements between clients and property owners.

**Regards:**

Represents property-specific attributes related to urgency and patience, contributing to the property's characteristics or features.

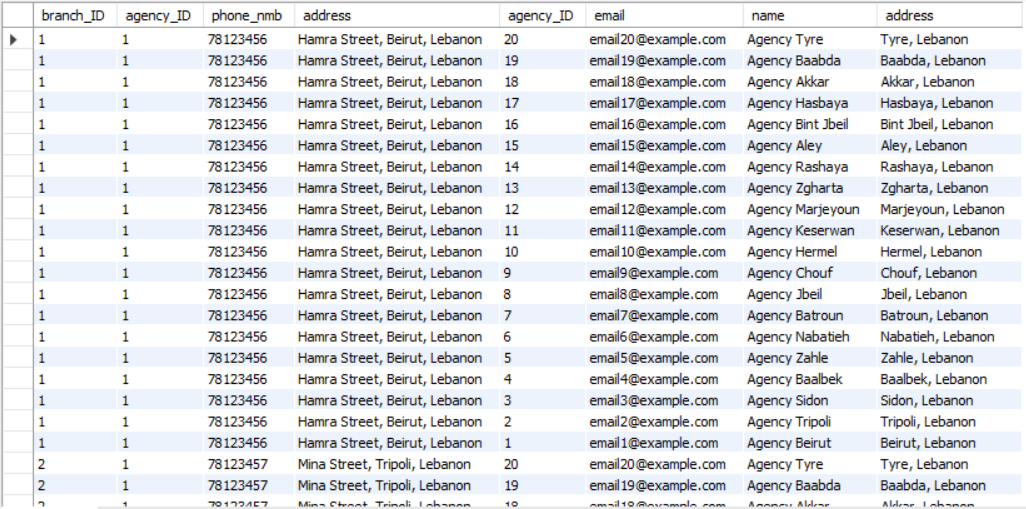
**Lived\_in:**

Ensures that old clients have a history of residing in at least one property, reflecting their experience in the real estate managed by the agency.

# **Basic SQL:**

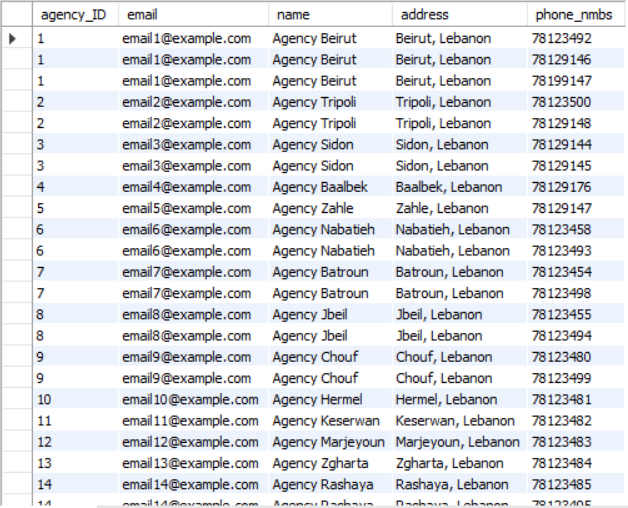
1. **SELECT** \* **FROM** branch, agency;

This Query gives us the cartesian product of the combination of the 2 tables (branch and agency), this could help visualize what branch belongs to what agency and so on, it is very inefficient to do though.

The result:  


1. **SELECT** \* **FROM** agency **NATURAL** **JOIN** phone\_nmbs;

This query matches each agency with the phone number it possesses using natural join, this can be very helpful if a user wants to obtain the phone numbers of the agencies so they can contact them and perform a purchase.

The result:  


1. **SELECT** \* **FROM** branch **JOIN** department **USING** (branch\_ID);

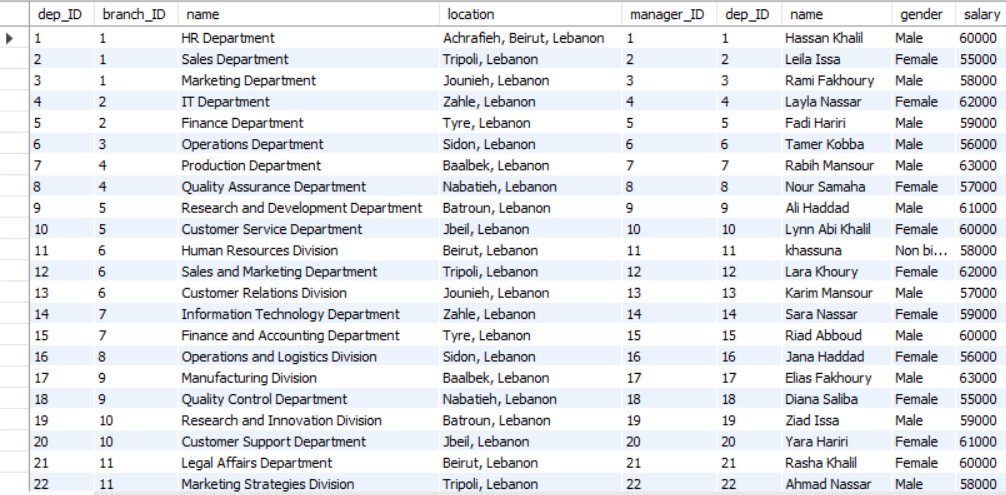
This Query uses theta join using “**JOIN**” to report all the branches in the database, as well as their respective departments, places in a way easy for the user to visualize and understand, and shows all the information about the 2 entities too.

The result:



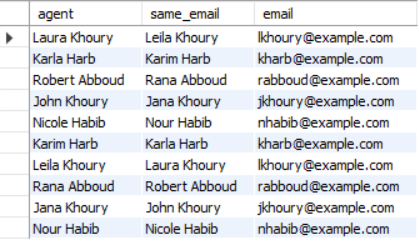
1. **SELECT** \* **FROM** department **JOIN** manager **ON** department.dep\_ID = manager.dep\_ID;

This Query uses theta join with “**ON**” to find all the departments and matching each department with its respective manager, this is hugely beneficial for a database to be able to retrieve the data about the departments and their managers without having to manually go through each table and match the IDs. This is done by matching the foreign key “dep\_ID” in the managers table, which refers to the ID of the department that the manager manages, to the primary key “dep\_ID” in the departments table.

The result:  


1. **SELECT** a.name **AS** agent, b.name **AS** same\_email, a.email  
   **FROM** agent a, agent b  
   **WHERE** a.email = b.email **AND** a.agent\_ID <> b.agent\_ID;

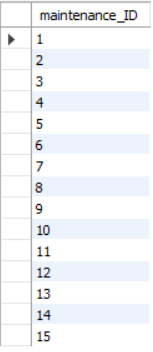
This query runs through the agents table, and gives back all the agent pairs that have the same email, as well as returning the actual common email between them. This can prove useful so that the administrator can get all the matching emails and issue a report to change those emails to avoid confusion. The matching email instance could happen due to agents having the same family name and the same first letter of their first name since it is a common practice to use your first name’s first letter joined with your last name to make your email.  
This query works by self-join, taking 2 instances of the agents table and checking if they have the same email as well as checking that those 2 agents are not the same agent by checking the agent ID since each agent has their own ID, this is done to avoid unnecessary results in the resulting table.

The result:  


1. **SELECT** **DISTINCT** maintenance\_ID **FROM** property;

This query results in the ID of each maintenance company that was contracted to work within one of the properties of the agency, each ID is present once in the resulting table to avoid clusters of the same data.

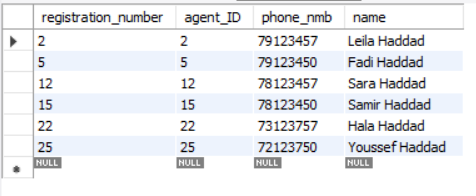
This query used the **DISTINCT** keyword to get each ID once, to increase efficiency in obtaining the data.

The result:  


1. **SELECT** \* **FROM** owner **WHERE** name **LIKE** '%Haddad';

This query gives us all the owners whose name contains a certain word, in this example we used “Haddad” to provide results. This can be very useful if an agency had a bad experience with a certain owner and wants to avoid working with people from his family so they can easily retrieve all the data they need about the owners with a name they specify.

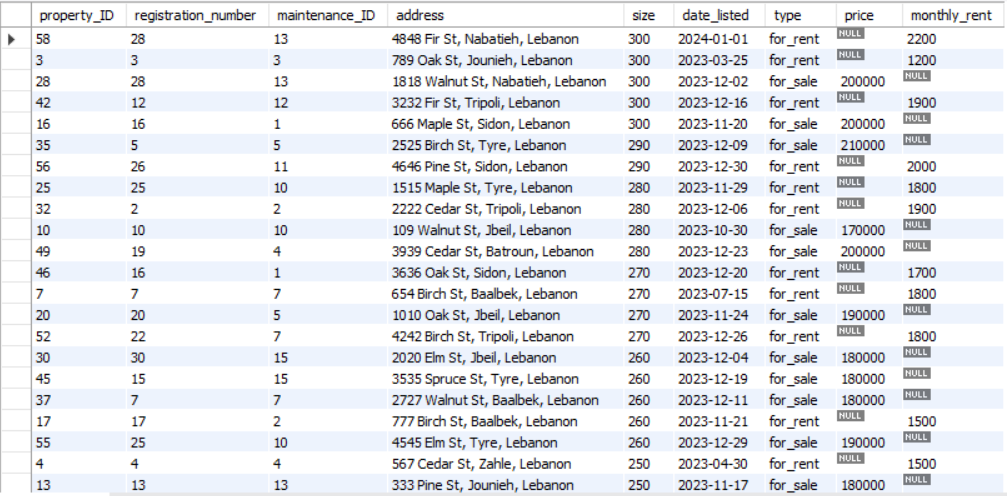
This query works using the “**LIKE**” keyword which is used to match strings of the same values together, the % is used here to indicate that the name “Haddad” should be the last name of the person since a % before a word indicates that there could be other values present in the string before the word we are looking for. However we didn’t use a & after the name because we are looking for the last name, so nothing would come after the last name, hence we eliminated the possibility of having owners named “Haddad” come up with our search since they are not related to the owner we are trying to avoid.

The result:  


1. **SELECT** \* **FROM** property **ORDER** **BY** size **DESC**;

This query results in a table containing all the properties in descending size order, this query can be very useful if a new client wants to go all out and get a fancy mansion so they are looking at the biggest properties by size, hence showing the properties they are looking for at the top, with their complete information.

This query is implemented by the “**ORDER BY**” keyword and the “**DESC**” keyword which displays the values in descending order of size.

The result:  


1. **SELECT** name **FROM** old\_client  
   **UNION**  
   **SELECT** name **FROM** new\_client;

This query can prove helpful in a database if agencies want to compile data about all the clients (old and new) in the database and want all their names reported.

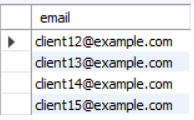
This query combines the names of all the clients (old and new) and displays them elegantly for the agency to utilize.

The result:  


1. **SELECT** **DISTINCT** old\_client.email  
   **FROM** old\_client  
   **INNER** **JOIN** new\_client **ON** old\_client.email = new\_client.email;

This query was put as a replacement for the query that utilizes the **INTERSECT** keyword, but due to MySQL not having the keyword we created a substitute for it using the code above.   
  
Here is what the code with **INTERSECT** would have been:  
**SELECT** email **FROM** old\_client **INTERSECT** **SELECT** email **FROM** new\_client;

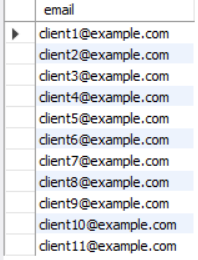
This query can be very helpful in a database since it gives us what emails are present in the 2 tables (old clients and new clients). By this, an agency could find out if a new client was an old client of any of the other agencies and serve them to their pleasures and needs.  
  
The query works here by getting the distinct emails from the old clients and doing an inner join on them with the emails from the new clients table to check which of them match.

The result:  


1. **SELECT** **DISTINCT** old\_client.email  
   **FROM** old\_client  
   **LEFT** **JOIN** new\_client **ON** old\_client.email = new\_client.email  
   **WHERE** new\_client.email **IS** **NULL**;

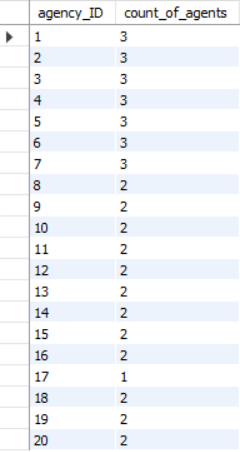
Similarly to the intersect keyword, the except keyword is not present in MySQL however, the same result can be achieved by the code above.  
  
Here is what the code using the **EXCEPT** keyword would’ve been like:  
**SELECT** name **FROM** old\_client **EXCEPT** **SELECT** name **FROM** new\_client;

The query selects unique email addresses from the 'old\_client' table that do not exist in the 'new\_client' table by comparing email addresses between the two tables using a LEFT JOIN and checking for NULL values in the 'new\_client' table.  
  
This is implemented by the “**LEFT JOIN**” keyword that retrieves records from the table that is mentioned before the join with the matching records from the right table.

The result:  


1. **SELECT** agency\_ID, **COUNT**(\*) **AS** count\_of\_agents **FROM** agent **GROUP** **BY** agency\_ID;

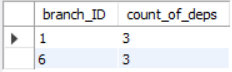
This query uses the count to get the number of rows, and the group by keyword to get the count for each agency in the agents table.  
  
This query is helpful to help agencies and administrators to keep track of how many agents each agency has working for it since every agency has a separate ID for it and here we are counting the count of presence of each agency\_ID value.  
(count Grouping aggregate function (with GROUP BY))

The result:  


1. **SELECT** branch\_ID, **COUNT**(\*) **AS** count\_of\_deps **FROM** department **GROUP** **BY** branch\_ID **HAVING** **COUNT**(\*) > **2**;

This query gives us the number of departments each branch has, but only the branches that have more than 2 departments. This can be helpful to help filter out which agency branches operate on a bigger scale than others by having more departments.

This is implemented by having count(\*) >2 which checks the instances of the branch\_ID and only displays the ones that are present 2 times in the departments table.  
Grouping aggregate function with condition (with GROUP BY and HAVING)

The result:  


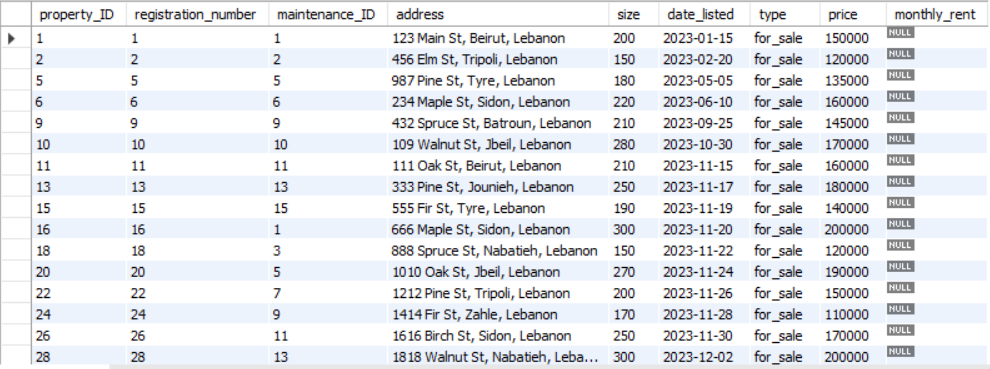
1. **SELECT** **COUNT**(\*) **AS** row\_count **FROM** property;

This query returns the count of the properties that are present in the database using the COUNT(\*) function that returns the number of rows, and each row here represents a property.  
  
This can be very helpful for agencies to keep track of the amount of properties present in the database.  
(count Aggregate function without GROUP BY)

The result:  

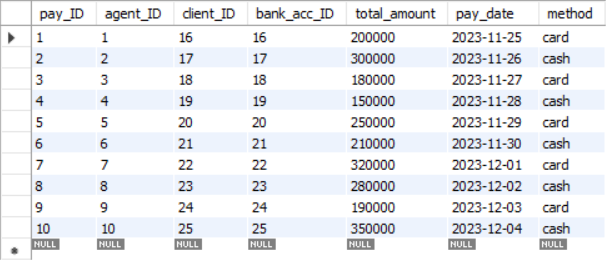

1. **SELECT** \* **FROM** property **WHERE** type **LIKE** '**for\_sale**';

This query gives us all the properties that are listed for sale, this is a very important query if a client wants to look at only the buyable properties excluding the rent-listed ones.  
  
This query works by checking the type of the property if it matches “for\_sale” which is a mandatory value for each property to have either for\_sale or for\_rent.

The result:  


1. **SELECT** \* **FROM** payment **LIMIT** **10**;

This query gives us the first 10 rows appended to the database in the payment table using the **LIMIT** keyword, this can be used by agencies to check the first customers that engaged in purchase in the database.

The result:  


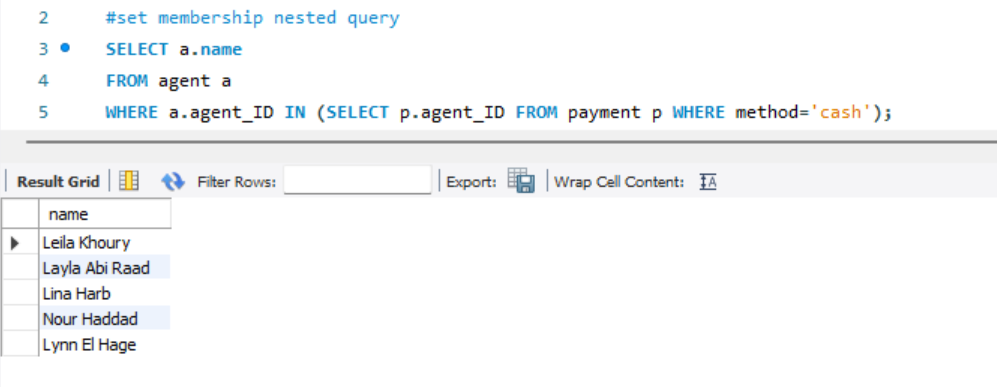
1. **SELECT** **CURDATE**() **AS** date;

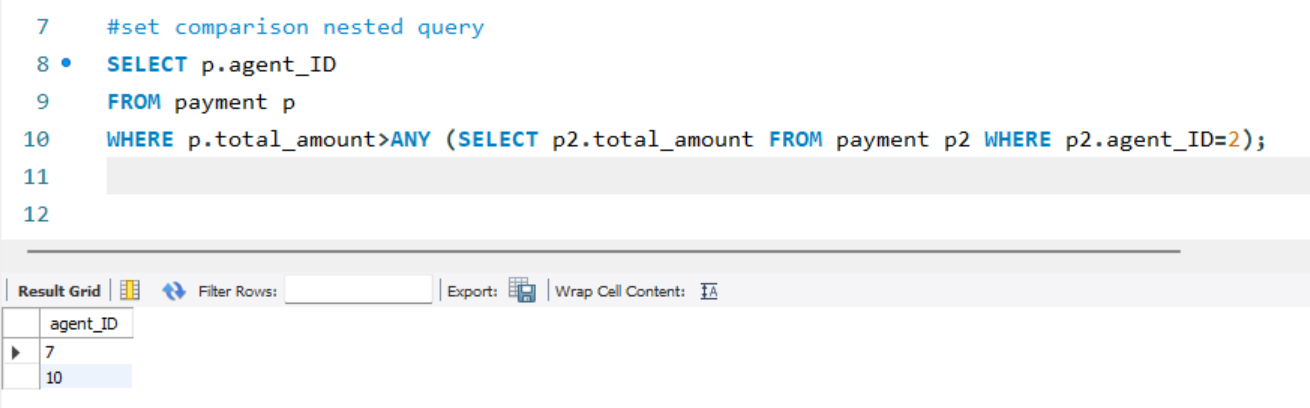
This query gives us the current day, this can be helpful in a database if they need to create some papers and documents automatically so they can just call this function automatically within the database.

The result:  


# **Advanced SQL:**

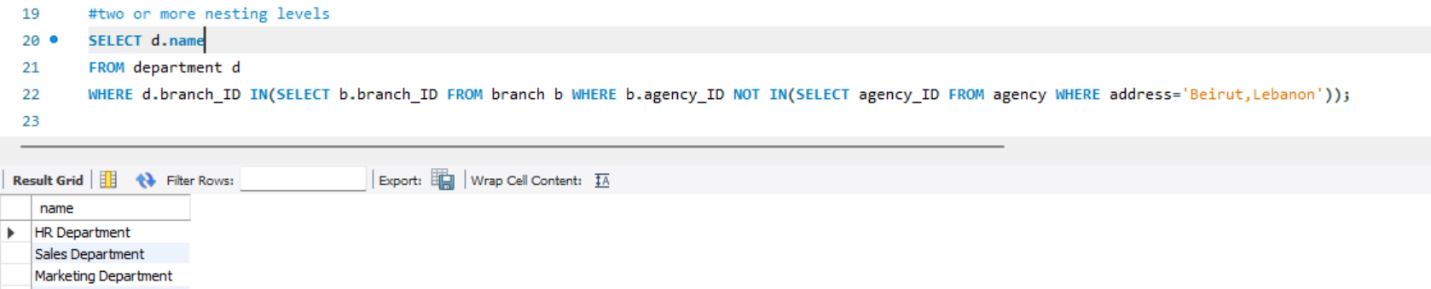
1. Membership nested query helps retrieve the name of agents who get payment in cash (since there are two types of methods) by having access from one table to another through a common column which is agent\_ID. This query offers insights into cash transactions involving agents.



1.  Comparison nested query retrieves data by comparing it to a subquery. It helps find agents who have performed better or made larger transactions compared to a specific agent (in this case, agent ID 2).
2. Cardinality nested queries refers to the number of rows or elements returned by the inner subquery within the outer query. This query helps in counting the number of properties associated with a specific owner.



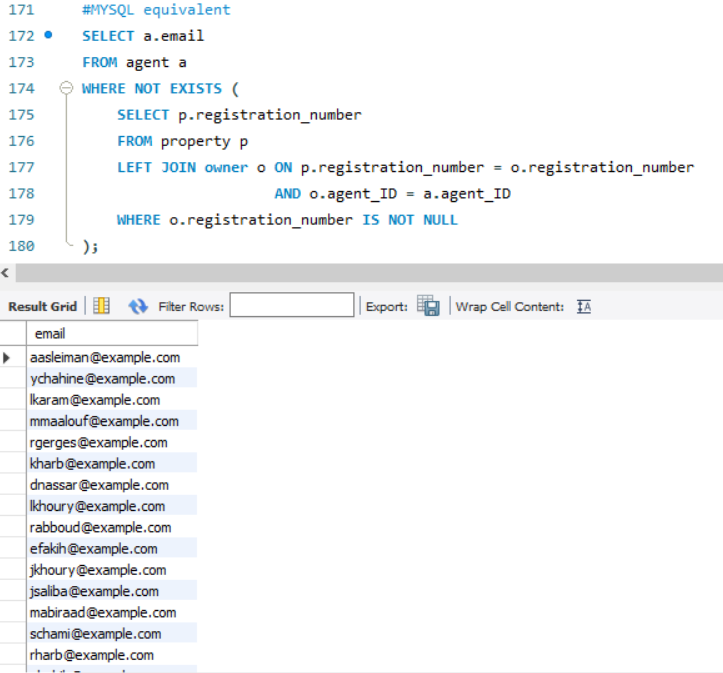
1. 2 or more nesting query is having a subquery inside another one.This query finds department names that are part of branches not linked to agencies located in Beirut, Lebanon. It's for out departments in branches that aren’t managed by agencies filtering located in Beirut.

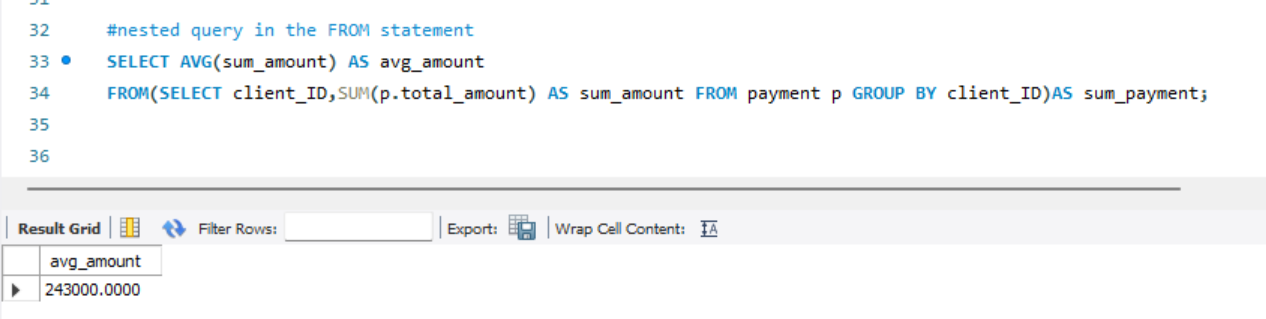
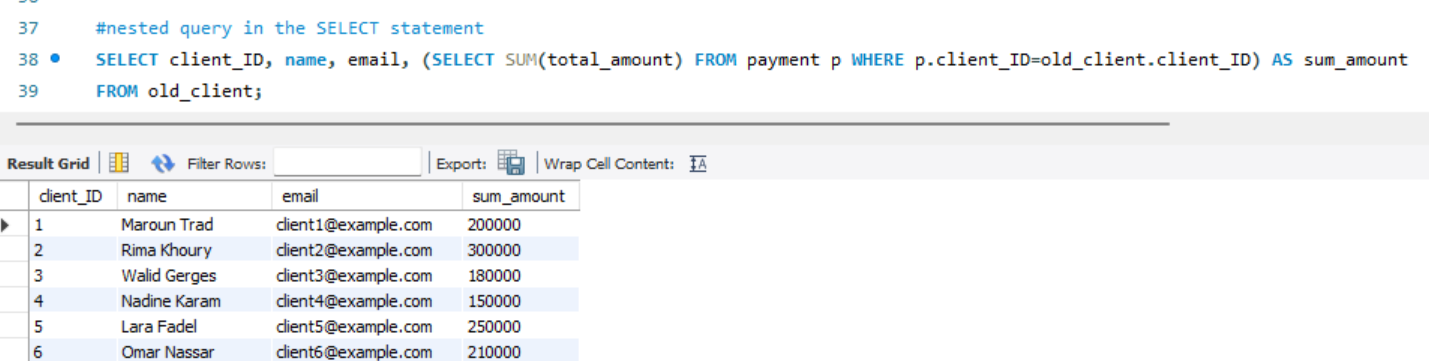
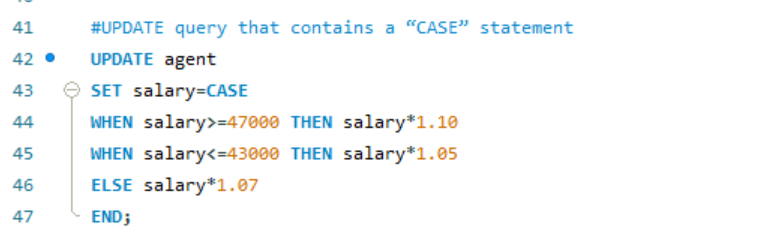
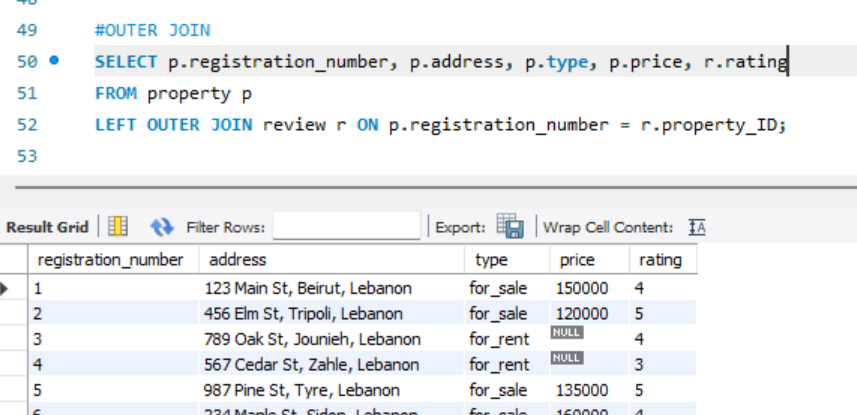


1. Division operation is not supported by MySQL. It is used to find records in a table that are related to all values in another table. This query verifies agents who exclusively own all the properties they are associated with. This validation can be crucial in ensuring that agents have complete control or responsibility for the properties they manage.

A close-up of a computer screen

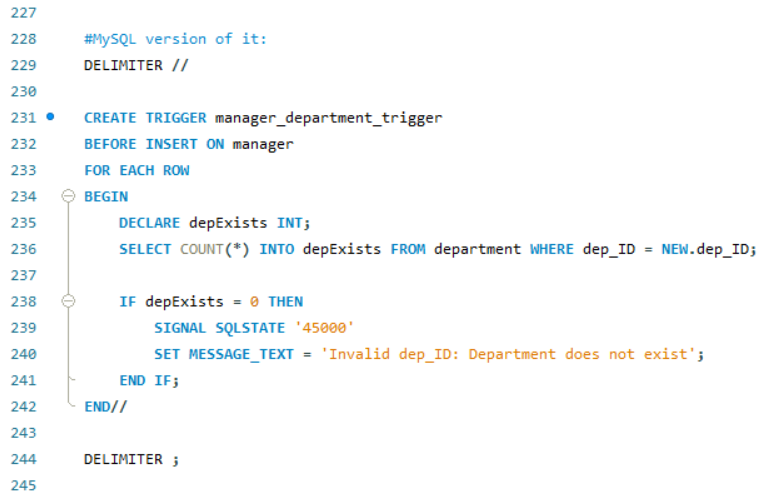
Description automatically generated

The MySQL equivalent of this operation is:  


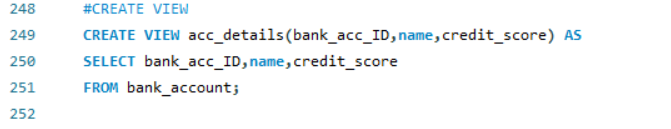
1. Nested query in the FROM statement involves using a subquery inside the FROM clause to create a temporary result set that can be further processed in the outer query. This query helps in understanding the average spending pattern per client based on their total payment amounts. It's beneficial for analyzing customer behavior and average payment trends, providing insights into overall client expenditure.
2. A nested query in the SELECT statement refers to using a subquery within the SELECT clause to retrieve or compute values based on the results of that subquery. This query is beneficial for fetching specific client-related details alongside their total payment amounts.
3. The UPDATE statement is used to modify or update existing records in a table. This query provides a conditional approach to adjusting salaries based on promoting fairness, performance-based rewards, and potentially enhancing overall employee motivation.
4. NATURAL OUTER JOIN is of 3 types LEFT, RIGHT, FULL. This query will retrieve all properties from the property table, along with any reviews they might have from the review table. If a property doesn’t have a corresponding review, NULL values will be shown in the columns fetched from the review table. This type of outer join is beneficial as it provides a comprehensive overview, allowing real estate managers or agents to see properties and their reviews.
5. CREATE ASSERTION is not supported by MySQL. It is used to enforce specific conditions or rules on a database to maintain data integrity. This assertion is beneficial because it enforces a rule or constraint within the database, maintaining data consistency. It prevents situations where a manager is assigned to a department that doesn't exist in the system, which could lead to incorrect or invalid data relationships.

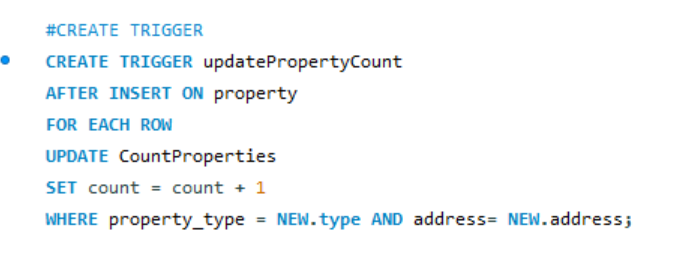
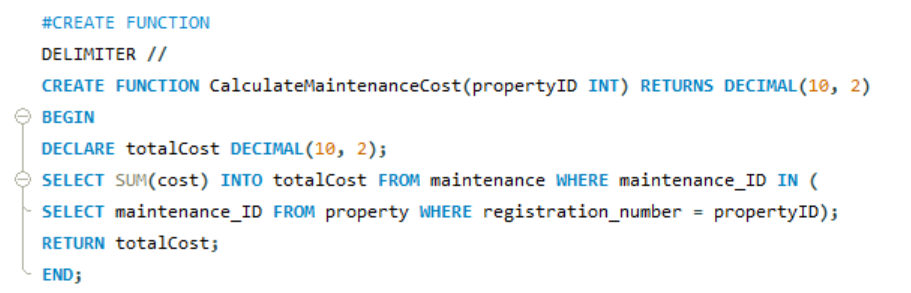
A close-up of a computer screen

Description automatically generated

MySQL equivalent:  


1. CREATE VIEW creates a virtual table based on the result set of a SELECT query. It does not produce any output in the sense of returning data when it's executed. Users needing information about bank accounts can simply query this view instead of specifying the columns and conditions each time.



1. CREATES TRIGGER fires automatically after an **INSERT** operation on the **property** table. For each new row inserted into the **property** table, this trigger executes. It does not produce any output in the sense of returning data when it's executed. This trigger automates the process of updating property counts in real-time upon new property additions without manual intervention
2. CREATE FUNCTION is a Data Definition Language statement used to create user-defined functions. It doesn't return any output indicating success, instead, it creates a new function in the database based on the provided specifications. This function takes a property ID as input and calculates the total cost of maintenance associated with that property. It retrieves the sum of the maintenance costs related to the specified property by joining the property and maintenance tables.
3. A computer screen shot of text

   Description automatically generatedCREATE PROCEDURE is a set of SQL statements that are stored in the database and can be executed multiple times, unlike functions. This procedure retrieves the count of properties available for sale and for rent separately by querying the property table based on the 'for\_sale' and 'for\_rent' types. Then, it returns the counts of properties for sale and for rent as a result set.

# **Conclusion:**

The database created by the Cedar Real Estate Consortium marks a significant advancement in Lebanon's real estate industry. This well-structured database efficiently organizes essential information such as agencies, properties, clients, and payments. It functions as a well organized library, simplifying real estate operations and improving overall organization.

Looking forward, there are avenues for enhancing this database. For instance, incorporating features to track property trends and market values over time could empower agents and owners to make more informed decisions. Additionally, implementing a client satisfaction survey system could provide valuable feedback for the consortium to enhance its services based on direct client experiences. Furthermore, introducing digital signatures for contracts could streamline and expedite transaction processes, benefiting all involved parties.

In summary, the database stands as a pivotal tool for the Cedar Real Estate Consortium, offering efficiency and organization. By implementing these methods, the database can be significantly improved, offering greater value and utility to stakeholders and users.

Thank you.