

Real Estate Property Management System

Database Design and Implementation

Group 8

Abigia Gashahun Edossa (NF1021072)

Dhruv Alpeshkumar Patel (NF1019537)

Keerthi Sadanand (NF1015200)

Oludeji Fashoro (NF1003542)

Professor: PhD Hany Osman

CPSC 500-7 : SQL Databases

University of Niagara Falls Canada

March 24th, 2025

Table of Contents	
ABSTRACT	2
INTRODUCTION	3
Purpose of the Database	4
DATABASE DESIGN: ER DIAGRAM EXPLANATION	5
Entity-Relationship Diagram (ERD)	7
Key Entities and Their Relationships	7
Diagram Representation	8
Importance of the ERD	9
Data Definition Statements (DDL)	10
1. Database Creation	10
2. Table Creation	10
3. Entity Relationships and Foreign Keys	15
(3.1) Relationship Overview	15
(3.2) Foreign Keys and Referential Integrity	17
4. Indexing	19
5. ALTER Command	20
6. DROP Command	20
Data Manipulation Statements (DML)	21
1. INSERT Command	21
2. Update Command	22
3. DELETE Command	22
Data Retrieval Statements (DQL)	23
1. Rank Properties by Price within Each City	23
2. Calculate the Running Total of Sales by Month	24
3. Identify Agents with Above-Average Sales	25
4. Properties with No Transactions in the Last 6 Months	26
5. Identify Tenants with Late Payments	27
6. Monthly Rental Income per City	28
7. Agent Commission Analysis	29
8. Average Selling Price Per Property Type	30

9. Yearly Growth Rate of Sales	31
CONCLUSIONS	32
REFERENCES.....	33

ABSTRACT

The **Real Estate Property Management System** is a robust database designed to streamline and manage all aspects of real estate operations. It handles various entities such as property ownership, transactions, rentals, maintenance requests, and financial activities. The system is built with 16 interconnected tables, including key components like Properties, Transactions, Rentals, Payments, MaintenanceRequests, and Agents. These tables are designed to maintain data integrity and optimize performance, supporting efficient data management across the platform.

This system allows for the effective management of property listings, sales transactions, rental agreements, and commissions, providing a comprehensive solution for both real estate sales and rental operations. The Payments and Commissions tables track financial transactions, ensuring accurate reporting of income and agent earnings. Additionally, it supports property maintenance management through the MaintenanceRequests and Inspections tables, ensuring that property upkeep is handled efficiently.

To enhance performance, the system employs advanced indexing strategies, including composite indexes, to improve the speed of query processing, especially when handling large volumes of data. Foreign key relationships, unique constraints, and validation checks ensure data consistency and accuracy. By maintaining referential integrity and optimizing data retrieval, the system provides a scalable and secure platform for property managers, agents, and other stakeholders. This centralized solution improves workflow, enhances decision-making, and boosts efficiency, transforming how real estate businesses operate.

INTRODUCTION

The real estate industry is a complex and dynamic sector that involves the management of properties, transactions, rentals, and relationships between various stakeholders, including property owners, buyers, tenants, agents, and agencies. Efficient management of these operations requires a robust and well-organized system capable of handling large volumes of data while ensuring accuracy, consistency, and accessibility. In today's digital age, a centralized database system is essential for streamlining processes, improving decision-making, and enhancing customer satisfaction.

This report presents the design and implementation of a Real Estate Management Database, a comprehensive solution tailored to meet the needs of real estate businesses. The database is structured to manage critical aspects of real estate operations, including property listings, sales, rentals, maintenance, legal documentation, and financial transactions. By leveraging relational database principles, the system ensures data integrity, scalability, and efficient query performance, enabling stakeholders to access and analyze information seamlessly.

The database schema is designed with a focus on modularity and flexibility, allowing it to adapt to the evolving needs of real estate businesses. Key entities such as Properties, Owners, Agents, Buyers, Tenants, and Agencies are interconnected through well-defined relationships, ensuring that all relevant data is linked logically. For example, properties are associated with owners and locations, while transactions and rentals are tied to buyers, tenants, and agents. This relational structure not only simplifies data management but also provides a holistic view of real estate operations.

To optimize performance, the database incorporates strategic indexing on frequently queried columns, such as property status, transaction dates, and agent IDs. Composite indexes are also implemented for complex queries involving multiple criteria, such as filtering transactions by agent and date or searching for rentals by property and rent amount. These optimizations ensure that the system can handle large datasets and deliver fast, accurate results, even under heavy usage.

The Real Estate Management Database also addresses the need for compliance and accountability by maintaining detailed records of legal documents, inspections, maintenance requests, and financial transactions. This ensures transparency and facilitates audit processes, which are critical in the real estate industry.

In conclusion, this database system is a powerful tool for real estate businesses seeking to modernize their operations, improve efficiency, and deliver exceptional service to clients. By centralizing data and automating key processes, the system empowers stakeholders to make informed decisions, reduce operational overhead, and focus on growing their business. The following sections of this report provide a detailed overview of the database schema, its components, and the rationale behind its design.

Purpose of the Database

The primary objective of the **Real Estate Management System database** is to provide an efficient and scalable solution for managing real estate properties. This database is designed to:

1. **Store and Manage Property Listings:** Maintain records of properties, including ownership details, locations, types, and status (available, sold, or rented).

2. **Facilitate Transactions:** Handle sales and rental transactions between property owners, buyers, and tenants.
3. **Support Agent and Agency Operations:** Track agent performance, commissions, and agency relationships.
4. **Automate Financial Records:** Manage rental payments, commissions, and financial transactions securely.
5. **Maintain Legal and Maintenance Records** – Store lease agreements, maintenance requests, and legal documents related to properties.

DATABASE DESIGN: ER DIAGRAM EXPLANATION

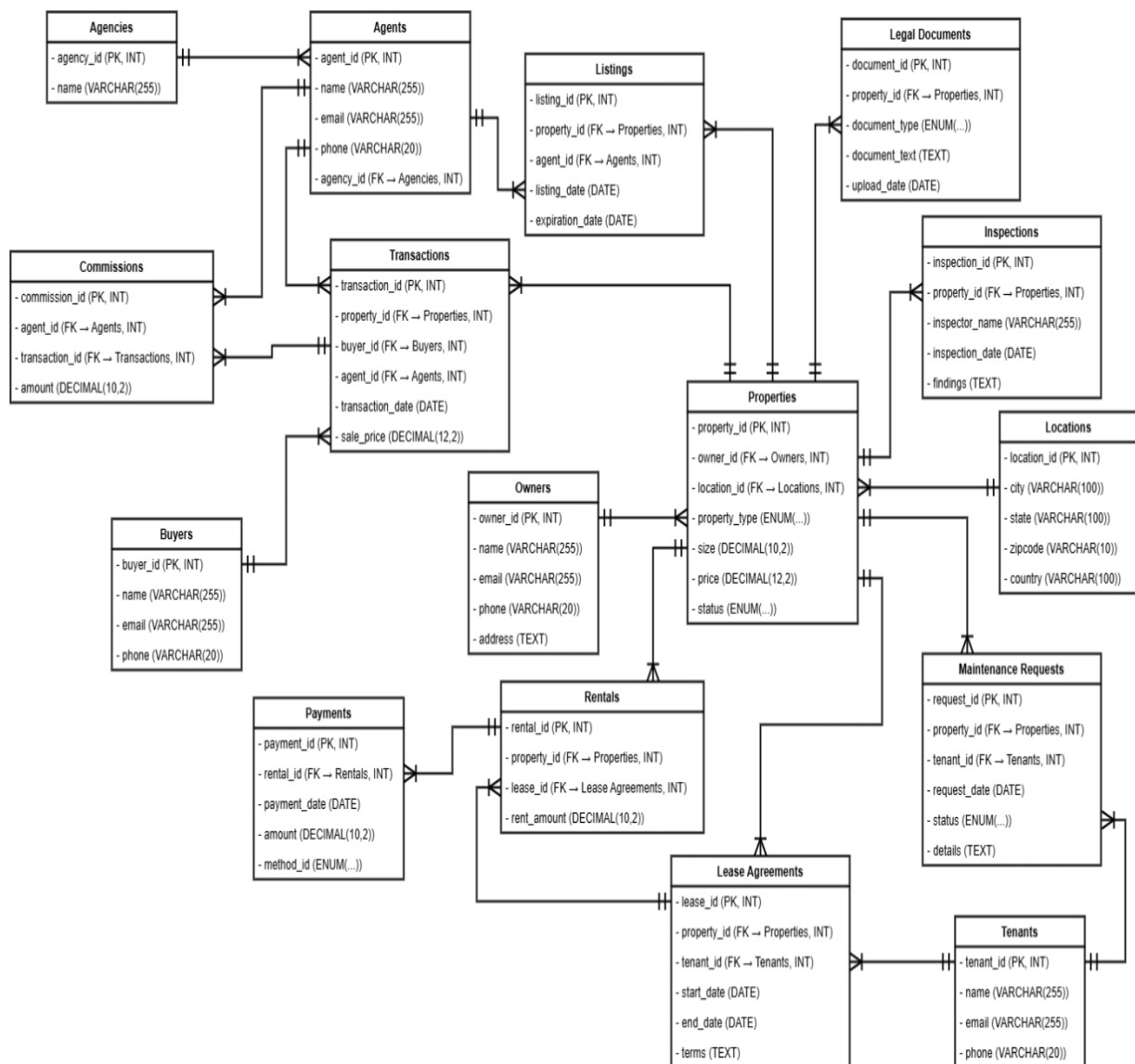
Database Design Overview

The REMS database is structured to encompass various aspects of real estate operations with 15+ interrelated tables, ensuring data consistency and efficient querying. The key components include:

- **Owners, Agents, and Agencies:** Tracks property owners and real estate professionals.
- **Properties and Locations:** Stores property details and geographical data.
- **Listings and Transactions:** Manages property sales and rental listings.
- **Tenants and Lease Agreements:** Keeps records of rental agreements and tenant information.
- **Financial Transactions and Payments:** Logs financial exchanges, rent payments, and commissions.

- Maintenance and Legal Documents: Facilitates maintenance requests and legal compliance.

To optimize query performance, indexes are implemented on critical attributes such as property status, agent transactions, and rental payments, ensuring fast data retrieval and efficient management.



Entity-Relationship Diagram (ERD)

The **Entity-Relationship Diagram (ERD)** provides a **visual representation** of the relationships between different entities in the **Real Estate Management System database**.

This diagram illustrates how various tables interact with each other, ensuring a structured and organized database design that supports the system's functionality.

Key Entities and Their Relationships

The **real_estate database** consists of several key entities, each representing a crucial aspect of real estate management:

- **Owners** → Own one or more **Properties**.
- **Properties** → Belong to **Owners**, are in **Locations**, and can be listed for sale or rent.
- **Locations** → Contain information about where properties are situated.
- **Agencies** → Employ **Agents** who manage property listings and transactions.
- **Agents** → Are associated with an **Agency** and handle **Listings** and **Transactions**.
- **Listings** → Connect **Properties** with **Agents**, indicating properties available for sale or rent.
- **Buyers** → Can purchase properties through **Transactions**.
- **Transactions** → Record the sale of properties, involving **Buyers**, **Agents**, and **Properties**.
- **Tenants** → Lease properties under **Lease Agreements**.
- **Lease Agreements** → Define rental terms between **Tenants** and **Properties**.

- **Rentals** → Associate **Lease Agreements** with rental payments.
- **Payments** → Track financial transactions related to **Rentals**.
- **Inspections** → Record property inspections and findings.
- **Maintenance Requests** → Store maintenance issues reported by tenants or owners.
- **Legal Documents** → Maintain property-related legal paperwork.
- **Commissions** → Record commissions earned by **Agents** from **Transactions**.

Diagram Representation

The ERD visually depicts these entities along with their **primary keys (PK)**, **foreign keys (FK)**, and **cardinality** (one-to-many, many-to-many relationships). Below is a textual representation of key relationships:

- **Owners (1) → (M) Properties**
- **Locations (1) → (M) Properties**
- **Agencies (1) → (M) Agents**
- **Agents (1) → (M) Listings**
- **Listings (1) → (1) Properties**
- **Listings (1) → (1) Agents**
- **Buyers (1) → (M) Transactions**
- **Transactions (1) → (1) Properties**
- **Transactions (1) → (1) Agents**

- **Properties (1) → (M) Lease Agreements**
- **Tenants (1) → (M) Lease Agreements**
- **Lease Agreements (1) → (1) Rentals**
- **Rentals (1) → (M) Payments**
- **Properties (1) → (M) Inspections**
- **Properties (1) → (M) Maintenance Requests**
- **Agents (1) → (M) Commissions**
- **Transactions (1) → (M) Commissions**

Importance of the ERD

- **Improves Database Understanding:** Helps stakeholders visualize the structure and relationships within the database.
- **Ensures Data Integrity:** Defines constraints and relationships to avoid data anomalies.
- **Optimizes Query Performance:** Guides index placement and query structuring for efficiency.

The ERD serves as a foundation for database implementation and ensures the system is **well-organized, scalable, and functional**.

https://viewer.diagrams.net/?tags=%7B%7D&lightbox=1&highlight=0000ff&edit=_blank&layers=1&nav=1&title=ER_REAL_ESTATE.drawio&dark=0#Uhttps%3A%2F%2Fdrive.google.com%2Fuc%3Fid%3D1o-DTfqJoblko9a8_cfaL_VKrg0EyJK8n%26export%3Ddownload

Data Definition Statements (DDL)

Data Definition Language (DDL) is used to define and manage the structure of the database. In this project, the DDL statements are used to create tables, define relationships between them, and enforce constraints to ensure data integrity and consistency. The following DDL commands were implemented for the “real_estate” database:

1. Database Creation

The database is created with a structured schema to support real estate operations.

```
Drop database if exists real_estate;
CREATE DATABASE real_estate;
USE real_estate;
```

2. Table Creation

(2.1) Owner Table :

Stores information about property owners.

```
-- Owners Table
CREATE TABLE Owners (
    owner_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL CHECK (email REGEXP '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'),
    phone VARCHAR(20) NOT NULL CHECK (phone REGEXP '^[0-9]{10,15}$'),
    address TEXT
);
```

(2.2) Location Table :

Contains details about property locations.

```
-- Locations Table
CREATE TABLE Locations (
    location_id INT PRIMARY KEY AUTO_INCREMENT,
    city VARCHAR(100) NOT NULL,
    state VARCHAR(100) NOT NULL,
    zipcode VARCHAR(10) NOT NULL CHECK (zipcode REGEXP '^[0-9]{5}(-[0-9]{4})?$',
    country VARCHAR(100) NOT NULL
);
```

(2.3) Properties Table:

Stores property details and ownership information.

```
-- Properties Table
CREATE TABLE Properties (
    property_id INT PRIMARY KEY AUTO_INCREMENT,
    owner_id INT,
    location_id INT,
    property_type ENUM('Apartment', 'House', 'Condo', 'Land', 'Commercial') NOT NULL,
    size DECIMAL(10,2) CHECK (size > 0),
    price DECIMAL(12,2) NOT NULL CHECK (price > 0),
    status ENUM('Available', 'Sold', 'Rented') NOT NULL,
    FOREIGN KEY (owner_id) REFERENCES Owners(owner_id) ON DELETE SET NULL,
    FOREIGN KEY (location_id) REFERENCES Locations(location_id) ON DELETE SET NULL
);
```

(2.4) Agencies Table

Stores real estate agency details.

```
-- Agencies Table
CREATE TABLE Agencies (
    agency_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) UNIQUE NOT NULL
);
```

(2.5) Agents Table

Stores agent details and their association with agencies.

```
-- Agents Table
CREATE TABLE Agents (
    agent_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL CHECK (email REGEXP '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'),
    phone VARCHAR(20) NOT NULL CHECK (phone REGEXP '^[0-9]{10,15}$'),
    agency_id INT,
    FOREIGN KEY (agency_id) REFERENCES Agencies(agency_id) ON DELETE SET NULL
);
```

(2.6) Listing Table

Links properties with agents for sale or rent listings.

```
-- Listings Table
CREATE TABLE Listings (
    listing_id INT PRIMARY KEY AUTO_INCREMENT,
    property_id INT,
    agent_id INT,
    listing_date DATE NOT NULL,
    expiration_date DATE,
    FOREIGN KEY (property_id) REFERENCES Properties(property_id) ON DELETE CASCADE,
    FOREIGN KEY (agent_id) REFERENCES Agents(agent_id) ON DELETE SET NULL
);
```

(2.7) Buyers Table:

Stores information about buyers interested in purchasing properties.

```
-- Buyers Table
CREATE TABLE Buyers (
    buyer_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL CHECK (email REGEXP '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'),
    phone VARCHAR(20) NOT NULL CHECK (phone REGEXP '^[0-9]{10,15}$')
);
```

(2.8) Transaction Table:

Records sales transactions between buyers and sellers.

```
-- Transactions Table
CREATE TABLE Transactions (
    transaction_id INT PRIMARY KEY AUTO_INCREMENT,
    property_id INT,
    buyer_id INT,
    agent_id INT,
    transaction_date DATE NOT NULL,
    sale_price DECIMAL(12,2) NOT NULL CHECK (sale_price > 0),
    FOREIGN KEY (property_id) REFERENCES Properties(property_id) ON DELETE CASCADE,
    FOREIGN KEY (buyer_id) REFERENCES Buyers(buyer_id) ON DELETE SET NULL,
    FOREIGN KEY (agent_id) REFERENCES Agents(agent_id) ON DELETE SET NULL
);
```

(2.9) Tenants Table:

Stores information about tenants.

```
-- Tenants Table
CREATE TABLE Tenants (
    tenant_id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL CHECK (email REGEXP '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'),
    phone VARCHAR(20) NOT NULL CHECK (phone REGEXP '^[0-9]{10,15}$')
);
```

(2.10) LeaseAgreements Table:

Stores rental agreements between tenants and property owners.

```
-- Lease Agreements Table
CREATE TABLE LeaseAgreements (
    lease_id INT PRIMARY KEY AUTO_INCREMENT,
    property_id INT,
    tenant_id INT,
    start_date DATE NOT NULL,
    end_date DATE NOT NULL,
    terms TEXT,
    FOREIGN KEY (property_id) REFERENCES Properties(property_id) ON DELETE CASCADE,
    FOREIGN KEY (tenant_id) REFERENCES Tenants(tenant_id) ON DELETE CASCADE
);
```

(2.11) Rentals Table:

Manages rental properties and associated lease agreements.

```
-- Rentals Table
CREATE TABLE Rentals (
    rental_id INT PRIMARY KEY AUTO_INCREMENT,
    property_id INT,
    lease_id INT UNIQUE,
    rent_amount DECIMAL(10,2) NOT NULL CHECK (rent_amount > 0),
    FOREIGN KEY (property_id) REFERENCES Properties(property_id) ON DELETE CASCADE,
    FOREIGN KEY (lease_id) REFERENCES LeaseAgreements(lease_id) ON DELETE CASCADE
);
```


(2.12) Payments Table:

Stores rent payments made by tenants.

```
-- Payments Table
CREATE TABLE Payments (
    payment_id INT PRIMARY KEY AUTO_INCREMENT,
    rental_id INT,
    payment_date DATE NOT NULL,
    amount DECIMAL(10,2) NOT NULL CHECK (amount > 0),
    method ENUM('Credit Card', 'Bank Transfer', 'Cash', 'Cheque') NOT NULL,
    FOREIGN KEY (rental_id) REFERENCES Rentals(rental_id) ON DELETE CASCADE
);
```

(2.13) Inspection Table:

Records inspection details of properties.

```
-- Inspections Table
CREATE TABLE Inspections (
    inspection_id INT PRIMARY KEY AUTO_INCREMENT,
    property_id INT NOT NULL,
    inspector_name VARCHAR(255) NOT NULL,
    inspection_date DATE NOT NULL,
    findings TEXT,
    FOREIGN KEY (property_id) REFERENCES Properties(property_id) ON DELETE CASCADE
);
```

(2.14) Maintenance request Table:

Logs maintenance issues reported for properties.

```
-- MaintenanceRequests Table
CREATE TABLE MaintenanceRequests (
    request_id INT PRIMARY KEY AUTO_INCREMENT,
    property_id INT NOT NULL,
    tenant_id INT, -- Allow tenant_id to be NULL
    request_date DATE NOT NULL,
    status ENUM('Pending', 'In Progress', 'Completed') DEFAULT 'Pending',
    details TEXT,
    FOREIGN KEY (property_id) REFERENCES Properties(property_id) ON DELETE CASCADE,
    FOREIGN KEY (tenant_id) REFERENCES Tenants(tenant_id) ON DELETE SET NULL
);
```

(2.15) Legal Documents Table:

Stores legal documents related to properties.

```
-- LegalDocuments Table
CREATE TABLE LegalDocuments (
  document_id INT PRIMARY KEY AUTO_INCREMENT,
  property_id INT NOT NULL,
  document_type ENUM('Lease Agreement', 'Ownership Document', 'Inspection Report', 'Other') NOT NULL,
  document_text TEXT,
  upload_date DATE NOT NULL,
  FOREIGN KEY (property_id) REFERENCES Properties(property_id) ON DELETE CASCADE
);
```

(2.16) Commission Table:

Records commission amounts earned by agents.

```
-- Commission Table
CREATE TABLE Commissions (
  commission_id INT PRIMARY KEY AUTO_INCREMENT,
  agent_id INT NOT NULL,
  transaction_id INT NOT NULL,
  amount DECIMAL(10,2) NOT NULL CHECK (amount > 0),
  FOREIGN KEY (agent_id) REFERENCES Agents(agent_id) ON DELETE CASCADE,
  FOREIGN KEY (transaction_id) REFERENCES Transactions(transaction_id) ON DELETE CASCADE
);
```

3. Entity Relationships and Foreign Keys

The **Real Estate Management System (REMS)** database follows a structured **relational model**, ensuring data consistency through well-defined relationships and constraints. The **ERD** illustrates how different entities interact, with **foreign keys** enforcing referential integrity.

(3.1) Relationship Overview

The system consists of **multiple interrelated tables**, categorized based on their roles:

Key Relationships Between Entities

- **Property Ownership & Location**

- Each property is owned by a single **Owner** (owner_id in Properties references Owners).
- Each property is located in a specific **Location** (location_id in Properties references Locations).
- **Real Estate Agents & Agencies**
 - An **Agent** is associated with one **Agency** (agency_id in Agents references Agencies).
 - Agents **list properties** for sale or rent (agent_id in Listings references Agents).
- **Property Sales & Transactions**
 - A **Buyer** can purchase a **Property** (buyer_id in Transactions references Buyers).
 - Each transaction involves an **Agent** (agent_id in Transactions references Agents).
- **Rental & Lease Agreements**
 - A **Tenant** leases a **Property** (tenant_id in LeaseAgreements references Tenants).
 - Lease agreements are linked to **Rentals** (lease_id in Rentals references LeaseAgreements).
 - **Payments** are associated with **Rentals** (rental_id in Payments references Rentals).
- **Property Maintenance & Inspections**

- **Maintenance Requests** are tied to specific **Properties** and **Tenants** (property_id and tenant_id in MaintenanceRequests).
- **Inspections** are performed on **Properties** (property_id in Inspections references Properties).
- **Legal Documentation & Compliance**
 - **Legal Documents** belong to **Properties** (property_id in LegalDocuments references Properties).
 - **Commissions** are paid to **Agents** for successful **Transactions** (agent_id in Commissions references Agents).

(3.2) Foreign Keys and Referential Integrity

Each table enforces relationships using **foreign keys**.

Below is a summary of key constraints:

Child Table	Foreign Key	Parent Table	On Delete Action
Properties	owner_id	Owners	SET NULL
Properties	location_id	Locations	SET NULL
Agents	agency_id	Agencies	SET NULL
Listings	property_id	Properties	CASCADE
Listings	agent_id	Agents	SET NULL
Transactions	property_id	Properties	CASCADE

Child Table	Foreign Key	Parent Table	On Delete Action
Transactions	buyer_id	Buyers	SET NULL
Transactions	agent_id	Agents	SET NULL
LeaseAgreements	property_id	Properties	CASCADE
LeaseAgreements	tenant_id	Tenants	CASCADE
Rentals	property_id	Properties	CASCADE
Rentals	lease_id	LeaseAgreements	CASCADE
Payments	rental_id	Rentals	CASCADE
Inspections	property_id	Properties	CASCADE
MaintenanceRequests	property_id	Properties	CASCADE
MaintenanceRequests	tenant_id	Tenants	SET NULL
LegalDocuments	property_id	Properties	CASCADE
Commissions	agent_id	Agents	CASCADE
Commissions	transaction_id	Transactions	CASCADE

4. Indexing

Indexes are used to speed up data retrieval operations by creating a structured access path to the data. When large volumes of data are stored in a table, searching, updating, or deleting specific records can be slow. An index helps improve performance by providing a quick lookup for the data.

```
-- Properties Table
-- Index on owner_id and location_id to speed up lookups by owner or location:
CREATE INDEX idx_properties_owner ON Properties(owner_id);
CREATE INDEX idx_properties_location ON Properties(location_id);
-- Index on status if queries frequently filter by property status:
CREATE INDEX idx_properties_status ON Properties(status);

-- Listings Table
-- Index on property_id and agent_id to optimize searches for properties listed by agents:
CREATE INDEX idx_listings_property ON Listings(property_id);
CREATE INDEX idx_listings_agent ON Listings(agent_id);
-- Index on listing_date for filtering by recent listings:
CREATE INDEX idx_listings_date ON Listings(listing_date);

-- Transactions Table
-- Index on property_id, buyer_id, and agent_id to speed up sales history lookups:
CREATE INDEX idx_transactions_property ON Transactions(property_id);
CREATE INDEX idx_transactions_buyer ON Transactions(buyer_id);
CREATE INDEX idx_transactions_agent ON Transactions(agent_id);
-- Index on transaction_date for filtering by sale date:
CREATE INDEX idx_transactions_date ON Transactions(transaction_date);

-- Inspections Table
-- Index on property_id to speed up inspections search:
CREATE INDEX idx_inspections_property ON Inspections(property_id);
-- Index on inspection_date for sorting/filtering by inspection date:
CREATE INDEX idx_inspections_date ON Inspections(inspection_date);

-- Agents Table
-- Index on email and phone for quick lookups:
CREATE UNIQUE INDEX idx_agents_email ON Agents(email);
CREATE UNIQUE INDEX idx_agents_phone ON Agents(phone);
-- Index on agency_id for filtering agents by their agency:
CREATE INDEX idx_agents_agency ON Agents(agency_id);
```

```
-- Rentals Table
-- Index on property_id for faster rental queries:
CREATE INDEX idx_rentals_property ON Rentals(property_id);
-- Index on lease_id since it's unique and referenced often:
CREATE UNIQUE INDEX idx_rentals_lease ON Rentals(lease_id);

-- Payments Table
-- Index on rental_id for quicker payment lookups:
CREATE INDEX idx_payments_rental ON Payments(rental_id);
-- Index on payment_date for sorting and filtering payments by date:
CREATE INDEX idx_payments_date ON Payments(payment_date);

-- MaintenanceRequests Table
-- Index on property_id and tenant_id for quicker maintenance request lookups:
CREATE INDEX idx_maintenance_property ON MaintenanceRequests(property_id);
CREATE INDEX idx_maintenance_tenant ON MaintenanceRequests(tenant_id);
-- Index on status to speed up filtering by request status:
CREATE INDEX idx_maintenance_status ON MaintenanceRequests(status);
```

5. ALTER Command

The ALTER command is used to modify the structure of an existing table. We can add, modify, or delete columns, or even rename the table or columns.

```
-- Alter commands : Add column to owners table
ALTER TABLE owners
ADD COLUMN owners_address varchar(255);

-- Alter commands : Modify column to owners table
ALTER TABLE owners
MODIFY COLUMN owners_address varchar(100);

-- Alter commands : Drop or remove column to owners table
ALTER TABLE owners
DROP COLUMN owners_address ;
```

6. DROP Command

The DROP command is used to remove an entire database object (like a table, index, or view) permanently. This command completely deletes the object and all of its data.

```
-- Drop Command : Permanently remove table , column or index from the database
DROP TABLE agents_agencies;
DROP INDEX idx_rentals_property_rent ON Rentals;
```

Data Manipulation Statements (DML)

Data Manipulation Language (DML) statements are used to manage and manipulate the data within the tables of a database. DML includes operations such as INSERT, UPDATE, and DELETE, which allow for inserting new data, modifying existing data, and removing unwanted data. These operations are crucial for maintaining dynamic data management in the real_estate database. Below are examples of each DML statement:

1. INSERT Command

The INSERT statement is used to add new records to a table. In the context of the real_estate database, we populated tables with realistic data by inserting rows into different tables. We add all data into our schema using this command. For example, The given SQL INSERT INTO query adds multiple records to the **LegalDocuments** table. Here's what it does:

- **Inserts Legal Document Records** – The query adds different types of legal documents associated with various properties.
- **Columns Used:**
 - **property_id:** Associates each document with a specific property.
 - **document_type:** Specifies the type of document (e.g., Lease Agreement, Ownership Document, Inspection Report, or Other).
 - **document_text:** Provides a brief description of the document.
 - **upload_date:** Records the date when the document was uploaded.
- **Types of Documents Added:**
 - **Lease Agreements** (for rental contracts).
 - **Ownership Documents** (for property ownership records).

- **Inspection Reports** (for property inspections).
- **Other Documents** (such as legal notices or environmental reports).

```
INSERT INTO LegalDocuments (property_id, document_type, document_text, upload_date) VALUES
(3, 'Lease Agreement', 'Lease agreement for tenant Alice Williams, valid from 2025-01-01 to 2026-01-01.', '2025-01-01'),
(7, 'Lease Agreement', 'Rental contract for Bob Johnson, covering the period 2025-02-01 to 2026-02-01.', '2025-02-10'),
(12, 'Lease Agreement', 'Residential lease agreement signed by Charlie Brown for 12 months.', '2025-03-15'),
(18, 'Ownership Document', 'Title deed confirming ownership transfer to Daniel Stewart.', '2025-04-20'),
(22, 'Ownership Document', 'Property ownership certificate for Victor Martinez.', '2025-05-30'),
(28, 'Ownership Document', 'Official land deed issued for commercial property.', '2025-06-12'),
(5, 'Inspection Report', 'Inspection report for commercial space: No issues found.', '2025-07-08'),
(10, 'Inspection Report', 'Safety compliance inspection report for commercial unit.', '2025-08-14'),
(15, 'Inspection Report', 'Building inspection completed: minor electrical issues noted.', '2025-09-05'),
(20, 'Inspection Report', 'Inspection of structural integrity for property approved.', '2025-10-21'),
(25, 'Other', 'Legal notice regarding property tax assessment for the fiscal year.', '2025-11-11'),
(30, 'Other', 'Environmental impact assessment report submitted.', '2025-12-30');
```

2. Update Command

The UPDATE statement is used to modify existing records in a table. When business data or other dynamic details change, UPDATE ensures that the database reflects those changes. In below example we update the date of documentation

```
UPDATE LegalDocuments
SET upload_date= '2025-01-01'
WHERE document_id = 1;
```

3. DELETE Command

The DELETE statement is used to remove records from a table. It is important to use the DELETE statement carefully to avoid unintended loss of data.

```
DELETE FROM legaldocuments
WHERE document_id = 13;
```

Data Retrieval Statements (DQL)

1. Rank Properties by Price within Each City

Use case: Helps agencies analyse high-value properties in each city.

```

1  ### *1. Rank Properties by Price within Each City (Window Function)*
2  •  SELECT
3      p.property_id,
4      l.city,
5      p.price,
6      RANK() OVER (PARTITION BY l.city ORDER BY p.price DESC) AS price_rank
7  FROM real_estate.Properties p
8  JOIN real_estate.Locations l ON p.location_id = l.location_id
9  ORDER BY p.property_id DESC;

```

Result:

property_id	city	price	price_rank
30	Fresno	1600000.00	1
29	Tucson	275000.00	1
28	Albuquerque	420000.00	1
27	Milwaukee	1100000.00	1
26	Baltimore	550000.00	1
25	Louisville	1500000.00	1
24	Memphis	250000.00	1
23	Detroit	390000.00	1
22	Nashville	1000000.00	1
21	El Paso	520000.00	1
20	Boston	1400000.00	1
19	Washington	225000.00	1

This SQL query ranks properties based on their price within each city using the **RANK()** window function. It retrieves the **property_id**, **city**, and **price** from the **Properties** table and assigns a ranking to each property within its respective city, ordering them in descending order of price. The **PARTITION BY l.city** clause ensures that the ranking resets for each city, allowing agencies to analyze the most expensive properties in each location. This helps in identifying high-value properties and making informed pricing or marketing decisions.

2. Calculate the Running Total of Sales by Month

Use case: Tracks cumulative revenue growth over time.

```

11  ### *2. Calculate the Running Total of Sales by Month (Window Function)*
12  •  SELECT
13      DATE_FORMAT(transaction_date, '%Y-%m') AS month,
14      SUM(sale_price) AS monthly_sales,
15      SUM(SUM(sale_price)) OVER (ORDER BY DATE_FORMAT(transaction_date, '%Y-%m')) AS running_total_sales
16  FROM real_estate.Transactions
17  GROUP BY month
18  ORDER BY month DESC;

```

Result

Result Grid			
	month	monthly_sales	running_total_sales
▶	2025-04	6015000.00	82546641.65
	2025-03	29248000.00	76531641.65
	2025-02	9924000.00	47283641.65
	2025-01	1140000.00	37359641.65
	2024-12	1587101.45	36219641.65
	2024-11	305600.30	34632540.20
	2024-10	550300.00	34326939.90
	2024-09	185900.90	33776639.90
	2024-08	470800.40	33590739.00
	2024-07	290700.60	33119938.60
	2024-06	365400.80	32829238.00

This SQL query calculates the **monthly sales and running total of sales** over time using a window function. It first groups transactions by **month** (formatted as "YYYY-MM") and computes the total **sale_price** for each month. Then, using the **SUM() OVER (ORDER BY month)** window function, it calculates the cumulative sales total up to each month, allowing businesses to track revenue growth over time. This helps in identifying sales trends, forecasting future revenue, and making strategic financial decisions.

3. Identify Agents with Above-Average Sales

Use case: Identifies top-performing agents for commissions and bonuses.

```

20    ### *3. Identify Agents with Above-Average Sales (Window & Aggregate Function)*
21
22    ● WITH AgentSales AS (
23        SELECT
24            agent_id,
25            SUM(sale_price) AS total_sales
26        FROM real_estate.Transactions
27        GROUP BY agent_id
28    )
29    SELECT
30        a.agent_id,
31        a.name,
32        s.total_sales
33    FROM AgentSales s
34    JOIN real_estate.Agents a ON s.agent_id = a.agent_id
35    WHERE s.total_sales > (SELECT AVG(total_sales) FROM AgentSales);

```

Result:

Result Grid			
Filter Rows:			
	agent_id	name	total_sales
▶	1	John Adams	4490000.00
	3	Michael Williams	5345000.00
	7	James Walker	5489702.00
	8	Olivia Thompson	4611204.80
	10	Isabella White	5245403.20
	12	Charlotte Davis	6320900.40
	13	Daniel Miller	5216802.70
	14	Emma Robinson	4980600.60
	16	Ava Martinez	4707101.30
	18	Sophia Allen	4380000.20

This query identifies **real estate agents whose total sales exceed the average sales of all agents**. It first calculates each agent's total sales using a **common table expression (CTE)** and then filters agents whose sales are above the average, helping agencies recognize top performers for commissions and bonuses.

4. Properties with No Transactions in the Last 6 Months

Use case: Helps agencies focus on properties that need marketing efforts.

```

37  ### *4. Properties with No Transactions in the Last 6 Months (LEFT JOIN & Date Filtering)*
38  •  SELECT
39      p.property_id,
40      p.price,
41      p.status
42  FROM real_estate.Properties p
43  LEFT JOIN real_estate.Transactions t
44      ON p.property_id = t.property_id
45      AND t.transaction_date >= CURDATE() - INTERVAL 6 MONTH
46  WHERE t.transaction_id IS NULL;

```

Result:

Result Grid			
Filter Rows:			
	property_id	price	status
▶	1	450000.00	Available
	3	320000.00	Rented
	4	150000.00	Available
	5	950000.00	Available
	7	800000.00	Rented
	8	330000.00	Available
	9	200000.00	Available
	16	500000.00	Rented
	17	950000.00	Available
	19	225000.00	Available
	20	1400000.00	Rented
	21	520000.00	Available
	23	390000.00	Rented

This query identifies **properties that have not been sold or rented in the last six months** by using a **LEFT JOIN and date filtering**. It helps agencies focus their marketing efforts on properties that may need more promotion.

5. Identify Tenants with Late Payments

Use case: Detects tenants who consistently delay payments.

```

48  ### *5. Identify Tenants with Late Payments (CTE & Window Function)*
49  WITH PaymentStatus AS (
50      SELECT
51          r.property_id,
52          p.payment_date,
53          LEAD(p.payment_date) OVER (PARTITION BY r.property_id ORDER BY p.payment_date) AS next_payment
54      FROM real_estate.Payments p
55      JOIN real_estate.Rentals r ON p.rental_id = r.rental_id
56  )
57  SELECT * FROM PaymentStatus
58  WHERE DATEDIFF(next_payment, payment_date) > 30;

```

Result:

property_id	payment_date	next_payment
1	2025-01-05	2027-07-01
2	2025-02-05	2027-08-05
3	2025-03-10	2027-09-01
4	2025-04-15	2027-10-01
5	2025-05-01	2027-11-01
6	2025-06-01	2027-12-01
7	2025-07-01	2028-01-05
8	2025-08-01	2028-02-01
9	2025-09-05	2028-03-10
10	2025-10-01	2028-04-15

This query identifies **tenants who consistently make late payments** by using a **Common Table Expression (CTE)** and the **LEAD()** window function. It calculates the gap between consecutive payment dates for each rental property and filters cases where the difference exceeds 30 days. This helps property managers detect patterns of delayed payments and take necessary actions.

6. Monthly Rental Income per City

Use case: Helps agencies evaluate high-revenue locations.

```

60    ### *6. Monthly Rental Income per City (JOIN & Aggregation)*
61    •   SELECT
62        l.city,
63        SUM(r.rent_amount) AS total_rental_income
64    FROM real_estate.Rentals r
65    JOIN real_estate.Properties p ON r.property_id = p.property_id
66    JOIN real_estate.Locations l ON p.location_id = l.location_id
67    GROUP BY l.city
68    ORDER BY total_rental_income DESC;

```

Result:

Result Grid		Filter Rows:
	city	total_rental_income
▶	Louisville	4000.00
	Phoenix	3500.00
	Fresno	3500.00
	Charlotte	3200.00
	Nashville	3000.00
	Milwaukee	2900.00
	San Jose	2800.00
	Boston	2700.00
	Los Angeles	2500.00
	Jacksonville	2500.00
	Seattle	2500.00
	San Antonio	2200.00
	Baltimore	1900.00
	Indianapolis	1850.00
	Austin	1700.00

This query calculates the **total monthly rental income for each city** by aggregating rental payments from properties located in different areas. It joins the **Rentals, Properties, and Locations** tables to associate rental income with cities. The results are grouped by city and sorted in **descending order of total rental income**, helping agencies identify high-revenue locations for investment and marketing strategies.

7. Agent Commission Analysis

Use case: Identifies top-earning agents.


```
--
70     ### *7. Agent Commission Analysis (Window Function)*
71 •   SELECT
72       a.agent_id,
73       a.name,
74       SUM(c.amount) AS total_commission,
75       RANK() OVER (ORDER BY SUM(c.amount) DESC) AS commission_rank
76 FROM real_estate.Commissions c
77 JOIN real_estate.Agents a ON c.agent_id = a.agent_id
78 GROUP BY a.agent_id, a.name;
```

Result:

Result Grid



Filter Rows:



Export:

	agent_id	name	total_commission	commission_rank
▶	1	John Adams	134700.00	1
	3	Michael Williams	119700.00	2
	18	Sophia Allen	113700.00	3
	13	Daniel Miller	95850.00	4
	19	Mason Carter	89850.00	5
	12	Charlotte Davis	84300.00	6
	11	William Young	77850.00	7
	4	Emma Johnson	71850.00	8
	20	Noah Harris	70950.00	9
	7	James Walker	70260.00	10
	8	Olivia Thompson	69900.00	11

This query analyzes **agent commissions** by calculating the **total commission** earned by each agent and ranking them based on their earnings. It uses a **window function (RANK)** to assign a rank to each agent in descending order of total commission. The query joins the **Commissions** and **Agents** tables, grouping the results by **agent ID and name**. This helps identify top-earning agents for performance evaluation, rewards, or bonuses.

8. Average Selling Price Per Property Type

Use case: Helps understand pricing trends in different property categories.

```

80    ### *8. Average Selling Price Per Property Type (Aggregation)*
81    SELECT
82        p.property_type,
83        AVG(t.sale_price) AS avg_price
84    FROM real_estate.Transactions t
85    JOIN real_estate.Properties p ON t.property_id = p.property_id
86    GROUP BY p.property_type
87    ORDER BY p.property_type;

```

Result:

Result Grid		
	property_type	avg_price
▶	Apartment	315115.680769
	House	549463.147143
	Condo	364974.498387
	Land	321771.795238
	Commercial	903683.089024

This query calculates the **average selling price** for each **property type** by joining the **Transactions** and **Properties** tables. It groups the results by **property type** and computes the average sale price for each category. This helps identify pricing trends across different property types, providing insights for real estate agencies to analyze market behavior and set competitive prices.

9. Yearly Growth Rate of Sales

Use case: Analyses business performance trends over time.

```

99  WITH YearlySales AS (
100      SELECT
101          YEAR(transaction_date) AS years,
102          SUM(sale_price) AS total_sales
103      FROM real_estate.Transactions
104      GROUP BY YEAR(transaction_date)
105  )
106  SELECT
107      years,
108      total_sales,
109      LAG(total_sales) OVER (ORDER BY years) AS previous_year_sales,
110      (total_sales - LAG(total_sales) OVER (ORDER BY years)) / LAG(total_sales) OVER (ORDER BY years) * 100 AS growth_rate
111  FROM YearlySales
112  ORDER BY YearlySales.years DESC;

```

Result:

	years	total_sales	previous_year_sales	growth_rate
▶	2025	46327000.00	5453205.95	749.536959
	2024	5453205.95	5460005.95	-0.124542
	2023	5460005.95	5455005.95	0.091659
	2022	5455005.95	4505005.95	21.087653
	2021	4505005.95	5473205.95	-17.689815
	2020	5473205.95	5368205.95	1.955961
	2019	5368205.95	4505005.95	19.160907
	2010	4505005.95	NULL	NULL

This query calculates the **yearly growth rate of sales** by comparing each year's total sales with the previous year's sales using a **window function**. It helps analyze business performance trends and measure sales growth over time.

CONCLUSIONS

The Real Estate Property Management System (REMS) database successfully provides a structured, scalable, and efficient solution for managing various aspects of real estate transactions, including property ownership, sales, rentals, maintenance, and financial operations. By leveraging relational database principles and implementing indexing strategies, the system ensures data integrity, optimized query performance, and seamless data retrieval.

The ERD and database schema were carefully designed to establish logical relationships between key entities such as properties, owners, agents, tenants, buyers, transactions, and financial records. The integration of foreign key constraints and referential integrity enforces data consistency, reducing the likelihood of anomalies and redundant information.

Through the implementation of Data Definition Language (DDL) and Data Manipulation Language (DML) statements, the database supports core functionalities such as inserting, updating, and deleting records while ensuring security and accuracy. Additionally, the use of Data Query Language (DQL) enables complex queries for analytical insights, such as ranking properties by price, tracking agent performance, and identifying rental trends.

Overall, the REMS database modernizes and automates real estate operations, improving efficiency for property managers, agents, buyers, and tenants. The system facilitates real-time data tracking, enhances financial accountability, and supports data-driven decision-making. Future enhancements may include advanced analytics, integration with machine learning models for predictive analysis, and web-based interfaces to further improve usability.

REFERENCES

1. Coronel, C., & Morris, S. (2022). *Database Systems: Design, Implementation, & Management* (14th ed.). Cengage Learning.
2. Elmasri, R., & Navathe, S. B. (2020). *Fundamentals of Database Systems* (7th ed.). Pearson.
3. Ramakrishnan, R., & Gehrke, J. (2020). *Database Management Systems* (3rd ed.). McGraw-Hill.
4. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2019). *Database System Concepts* (7th ed.). McGraw-Hill.
5. Connolly, T., & Begg, C. (2021). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.
6. SQL Documentation. (n.d.). Retrieved from <https://www.sql.org/>
7. MySQL Documentation. (n.d.). Retrieved from <https://dev.mysql.com/doc/>
8. PostgreSQL Documentation. (n.d.). Retrieved from <https://www.postgresql.org/docs/>
9. IBM Cloud Docs. (n.d.). *Database Indexing Best Practices*. Retrieved from <https://www.ibm.com/cloud/blog/database-indexing>
10. Microsoft SQL Server Documentation. (n.d.). Retrieved from <https://docs.microsoft.com/en-us/sql/>