Lewis University  
CPSC 50900: Database Systems   
Spring 2024 Term Project

**Integrative Database Solution for SME Resource Management**

**Report Prepared by**

* Leelasrinivasraju Sarikonda leelasrinivasrajus@lewisu.edu
* Mohan Saty Surapa Raju Nadimpalli mohansatyasurapara@lewisu.edu

Github repository: [https://github.com/leelasrinivasrajusarikonda/CPSC50900DatabaseSystems](https://github.com/leelasrinivasrajusarikonda/CPSC-50900-Database-Systems)

**Table of Contents**

[Initial Proposal 2](#_Toc160685844)

[Data Sources 3](#_Toc160685845)

[Alternative Ways to Store the Data 3](#_Toc160685846)

[Conceptual and Logical Models 4](#_Toc160685847)

[Physical Model 6](#_Toc160685848)

[Populate the database with data 10](#_Toc160685849)

[Data Manipulation Language (DML) Scripts 15](#_Toc160685850)

[Indexes 20](#_Toc160685851)

[Views 21](#_Toc160685852)

[Stored Programs (Stored Procedures, Stored Functions, Triggers) 23](#_Toc160685853)

[Transactions 24](#_Toc160685854)

[Database Security 25](#_Toc160685855)

[Locking and Concurrent Access 28](#_Toc160685856)

[Backing Up Your Database 29](#_Toc160685857)

[Programming 30](#_Toc160685858)

[Suggested Future Work 32](#_Toc160685859)

[Activity Log 34](#_Toc160685860)

# **Initial Proposal**

Our project is designed to serve as a comprehensive data repository tailored for small to medium sized enterprises (SMEs), encompassing a wide range of information crucial to inventory management. This includes detailed product information, supplier contacts, sales transactions, customer profiles, order histories, and employee data. Such a diverse collection of data ensures a holistic approach to inventory control and overall business operations, addressing various factors essential for effective management.

The significance of this project lies in its focus on inventory management, a key area for SMEs. By providing detailed insights into stock levels, supplier relationships, and customer purchasing behaviors, our project directly contributes to enhancing the efficiency and profitability of retail and wholesale businesses. The importance of this data cannot be overstated; its the lifeline of an SMEs operational success, playing a crucial role in ensuring business processes are both effective and efficient.

Central to the effectiveness of this initiative is the principle of effective inventory management. Striking the right balance of stock is critical; overstocking can lead to unnecessary costs and product obsolescence, while insufficient stock can result in lost sales and customer dissatisfaction. Our project aims to empower businesses with a robust database they need to store their data and make informed decisions, minimize waste, and enhance customer satisfaction, all of which are key to maintaining a competitive edge.

To ensure the data relevance and practical applicability, we are sourcing it from simulated business operations that closely mirror real world scenarios. The inclusion of product catalogs, supplier data, sales records, and customer transaction histories offer a comprehensive view of business operations from multiple perspectives, enhancing the databases utility and scope.

The primary beneficiaries of this database are SME owners, inventory managers, and sales personnel. They will utilize this database for routine data management and leverage it for strategic planning and decision making. Beyond daily operations, the database is also a valuable resource for business analysts and software developers, who can use it to enhance system features and functionalities.

Our vision is to develop a fully-fledged database system for inventory management, incorporating features like stock level monitoring, order processing, sales tracking, and report generation. This system is not just a database; it’s a tool for gaining insights into inventory trends, promoting efficient stock management, and improving overall business operations.

To effectively meet these diverse needs, our system will include but not limited to tables such as Products, Suppliers, Sales, Customers, Categories, and Employees. Each table is designed with a specific purpose in mind, collectively creating a robust and comprehensive inventory management system database. This thoughtful design ensures that all aspects of inventory management are covered, providing SMEs with a powerful database to manage their operations more effectively.

# **Data Sources**

In our database project for the Inventory Management System, we’ve collected various data files, predominantly in CSV format, to populate our database effectively. These files cover different aspects of inventory management and are structured to facilitate integration into a relational database.

1. **Categories.csv**: Contains categories of products with fields like Category ID, Name, and Description. Its pivotal for classifying inventory items and will be linked to products in the database.
2. **Customers.csv:** Details customer profiles including Customer ID, Name, and Contact Information. This data will be connected to sales transactions to analyze customer purchasing behaviors.
3. **Employees.csv**: Lists employee information such as Employee ID, Name, Role, and Contact Details, essential for managing user roles and access within the system.
4. **Products.csv**: A comprehensive list of inventory items with details like Product ID, Name, Supplier ID, Category ID, Price, and Stock. This file is central to the database, interlinking with sales, categories, and suppliers.
5. **Sales.csv** and **Sales\_item.csv**: These files record sales transactions. **Sales.csv** includes overall sale data, while Sales\_item.csv details individual items sold, connecting inventory to customer purchases.
6. **Suppliers.csv**: Contains supplier information crucial for inventory procurement, including Supplier ID, Name, and Contact Details.
7. **Supply\_orders.csv**: Tracks inventory orders with details like Order ID, Date, Supplier ID, Product ID, and Quantity, vital for stock management.

Each file is structured with appropriate data types and constraints for integration into the database, ensuring data integrity and efficient management.

# **Alternative Ways to Store the Data**

In our exploration of alternative database storage methods for the Inventory Management System database, we came across two distinct models: the Hierarchical Database Model and the NoSQL document-oriented Database, each offering unique advantages and facing specific challenges.

The Hierarchical Database Model, one of the oldest forms of database structures, operates on a simple, treelike hierarchy. In this model, data is organized into levels, with a single parent node linked to various child nodes. For our Inventory Management System database, this could mean categorizing Products as a primary segment and branching it into subsegments like Categories, Suppliers, and Sales. Implementation involves establishing these parent-child relationships, which simplifies data navigation and ensures data integrity due to the clear hierarchical structure. However, this models’ major limitations lie in its inflexibility and inability to efficiently handle complex hierarchies or multiple relational links, only supporting one to many relationships.

Conversely, the NoSQL document-oriented Database, particularly ones utilizing JSON like formats, presents a more modern and flexible approach. In this model, data entities such as Products or Suppliers are stored as individual documents in a format similar to JSON, each containing key-value pairs. These documents are then grouped into collections for better organization. The primary advantage of this system lies in its flexibility; there’s no predefined schema, allowing for easy adjustments and additions. Additionally, its highly scalable, making it well suited for handling large datasets and high user traffic. Despite these benefits, the NoSQL model introduces complexities in aggregating data from various collections and may lead to consistency issues, as it typically ensures eventual consistency rather than immediate data accuracy.

Through these alternative models, our project explores different dimensions of database management. While the Hierarchical model provides a structured and easy to understand framework, it falls short in handling more complex, relational data scenarios. In contrast, the NoSQL approach, with its adaptable and scalable nature, may introduce challenges in data aggregation and real-time consistency. Each models’ adoption would depend on the specific needs and scale of the database application, highlighting the importance of aligning the database structure with the project requirements.

# **Conceptual and Logical Models**

The conceptual model focuses on identifying the entity sets and the relationships among them.

**Entity Sets:**

* Products
* Categories
* Suppliers
* Customers
* Employees
* Sales
* Sales Item
* Supply Orders.

**Relationships:**

* Products to Categories: One-to-Many (Mandatory for Product, Optional for Category)
* Sales to Customers: Many-to-One (Mandatory for Sale, Optional for Customer)
* Sales to Employees: Many-to-One (Mandatory for Sale, Optional for Employee)
* Sales Item to Products: Many-to-One (Mandatory for Sales Item, Optional for Product)
* Sales Item to Sales: Many-to-One (Mandatory for Sales Item, Optional for Sale)
* Supply Orders to Suppliers: Many-to-One (Mandatory for Supply Orders, Optional for Supplier)
* Supply Orders to Products: Many-to-One (Mandatory for Supply Orders, Optional for Product)

**Logical Model with Attributes and Functional Dependencies**

The logical model extends the conceptual model by adding attributes to each entity set and identifying functional dependencies.

* Products Table:

Functional Dependency: Product ID → name, description, category\_id, supplier\_id, price

* Categories Table:

Functional Dependency: Category ID → name, description

* Suppliers Table:

Functional Dependency: Supplier ID → name, phone\_number, email, address, account\_number

* Customers Table:

Functional Dependency: Customer ID → name, phone\_number, email, address, city, postal\_code, country

* Employees Table:

Functional Dependency: Employee ID → name, email, phone, address

* Sales Table:

Functional Dependency: Sale ID → employee\_id, customer\_id, amount, date

* Sales Item Table:

Functional Dependency: Sale ID, Product ID → amount, quantity

* Supply Orders Table:

Functional Dependency: Order ID → supplier\_id, product\_id, quantity, total\_cost, date

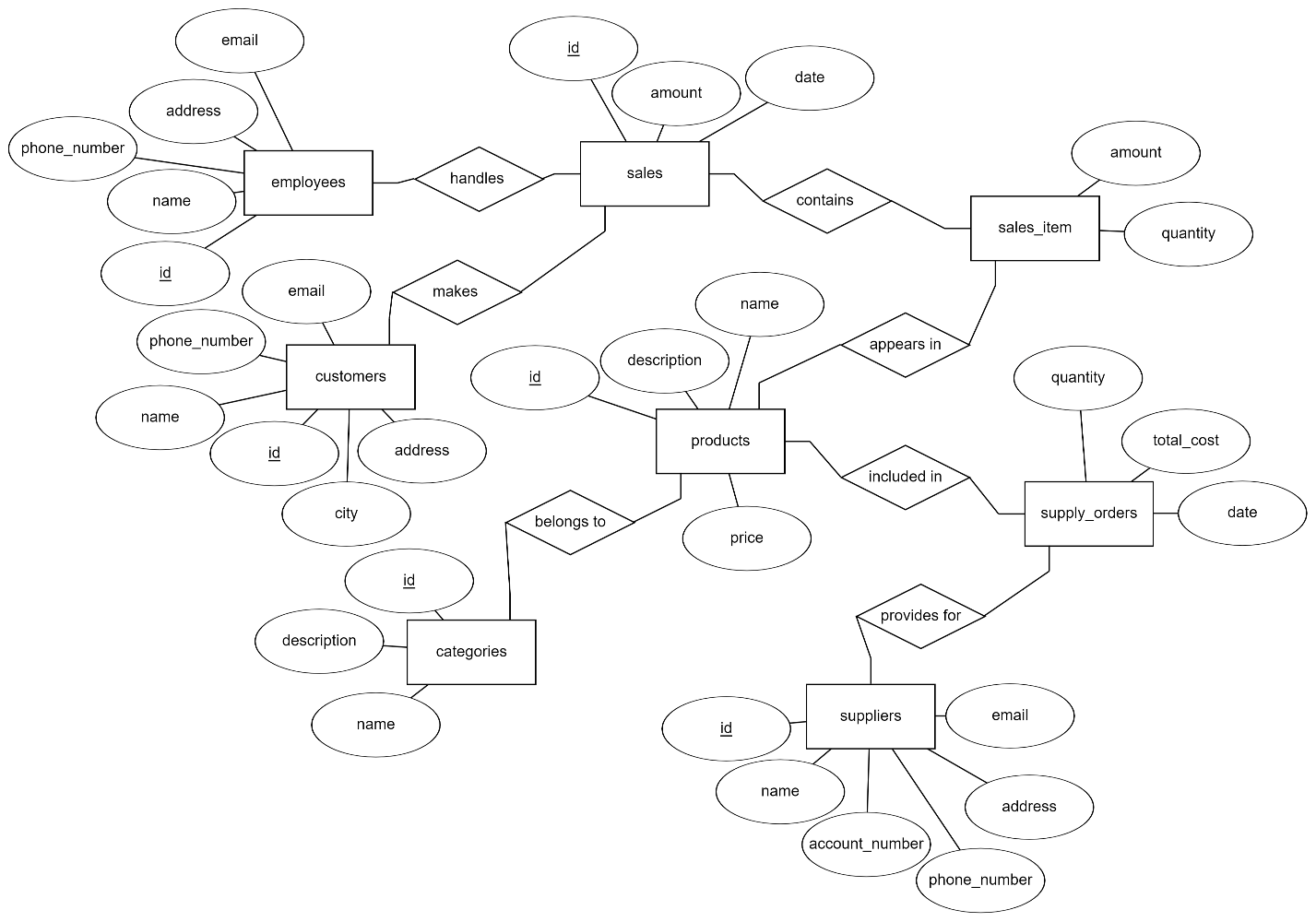
**Normalization**

The provided dependencies suggest that each table is already in 1st Normal Form (1NF) since all attributes are atomic.

The design appears to adhere to 2nd Normal Form (2NF) as well, each determinant is a candidate for a primary key and there are no partial dependencies on a composite primary key.

For 3rd Normal Form (3NF), the provided functional dependencies indicate that there are no transitive dependencies, as each non-key attribute is functionally dependent on the primary key alone.

**Entity-Relationship Diagram (ERD)**

****

The provided Logical Entity-Relationship Diagram (ERD) illustrates the structure of a database designed to manage a retail operation.

Each entity is represented by a rectangle containing the entity's name, with its attributes enclosed in ellipses connected to it. The ERD depicts not only the attributes of each entity but also the relationships between them, indicating how sales are handled by employees and made to customers, how products are categorized and included in sales as well as supply orders, and how suppliers provide products.

This diagram serves as a blueprint for creating a relational database, ensuring the integrity of data through the establishment of clear relationships, all while facilitating the easy retrieval, updating, and maintenance of the business's operational data.

# **Physical Model**

**Bridge Entity Sets**

Based on the logical ERD in the previous section, there are no direct M:N relationships because the sales\_item table already acts as a bridge entity for the sales and products, and supply\_orders does the same for suppliers and products.

**Adding Data Types and Constraints**

Below is a structured table format outlining the data types and constraints for each table based on the information provided. This format assumes typical data types used in SQL and includes common constraints such as primary keys, data type lengths, and not-null constraints.

* Products Table

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Constraints** |
| Product ID | int | PRIMARY KEY |
| Name | varchar(100) | NOT NULL |
| Description | text |  |
| Category ID | int | FOREIGN KEY (Category ID) |
| Selling price | decimal(8, 2) | FOREIGN KEY (Supplier ID) |
| Price | decimal(8, 2) | NOT NULL |
| Reorder level | int |  |
| Quantity | int |  |

* Categories Table

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Constraints** |
| Category ID | int | PRIMARY KEY |
| Name | varchar(100) | NOT NULL |
| Description | text |  |

* Suppliers Table

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Constraints** |
| Supplier ID | int | PRIMARY KEY |
| Name | varchar(50) | NOT NULL |
| Phone Number | varchar(20) |  |
| Email | varchar(50) |  |
| Address | varchar(100) |  |
| Account Number | varchar(50) |  |

* Customers Table

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Constraints** |
| Customer ID | int | PRIMARY KEY |
| Name | varchar(100) | NOT NULL |
| Phone Number | varchar(20) |  |
| Email | varchar(50) | NOT NULL |
| Address | varchar(100) |  |
| City | varchar(50) |  |
| Postal Code | varchar(10) |  |
| Country | varchar(50) |  |

* Employees Table

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Constraints** |
| Employee ID | int | PRIMARY KEY |
| Name | varchar(100) | NOT NULL |
| Email | varchar(50) | NOT NULL |
| Phone | varchar(20) |  |
| Address | varchar(100) |  |

* Sales Table

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Constraints** |
| Sale ID | int | PRIMARY KEY |
| Employee ID | int | FOREIGN KEY (Employee ID) |
| Customer ID | int | FOREIGN KEY (Customer ID) |
| Amount | decimal(8, 2) | NOT NULL |
| Date | date | NOT NULL |

* Sales Item Table (Bridge Table)

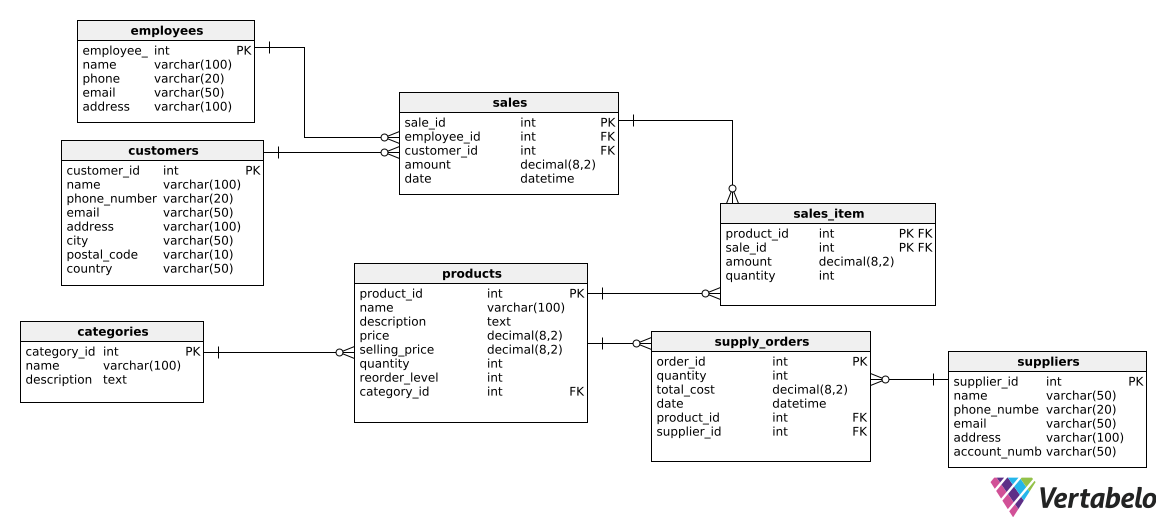
|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Constraints** |
| Sale ID | int | FOREIGN KEY (Sale ID), PART OF COMPOSITE PRIMARY KEY |
| Product ID | int | FOREIGN KEY (Product ID), PART OF COMPOSITE PRIMARY KEY |
| Amount | decimal(8, 2) | NOT NULL |
| Quantity | int | NOT NULL |

* Supply Orders Table

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Constraints** |
| Order ID | int | PRIMARY KEY |
| Supplier ID | int | FOREIGN KEY (Supplier ID) |
| Product ID | int | FOREIGN KEY (Product ID) |
| Quantity | int | NOT NULL |
| Total Cost | decimal(10, 2) | NOT NULL |
| Date | date | NOT NULL |

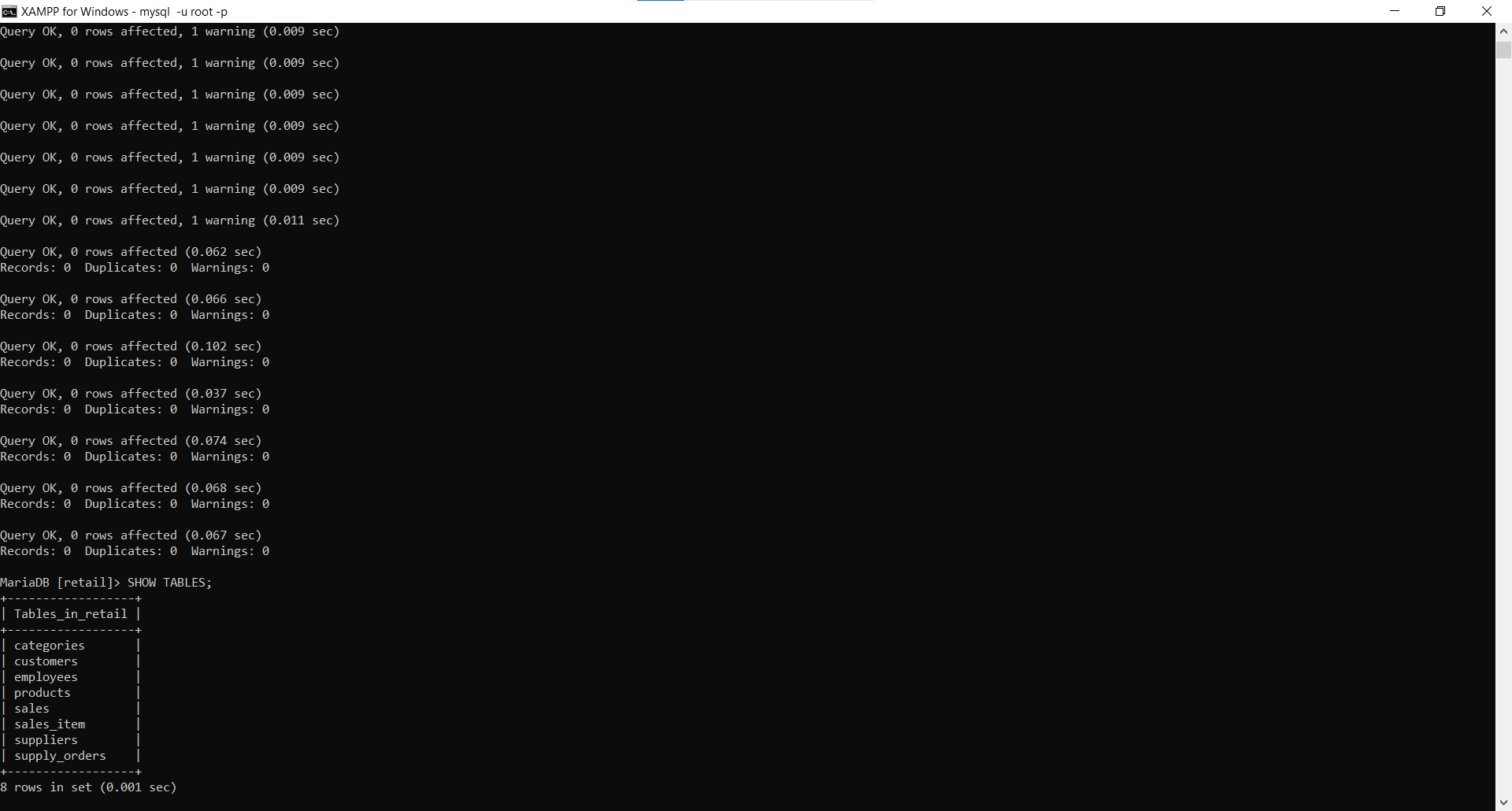
**Database Design – Entity Relational Model**

The provided Entity-Relationship Diagram (ERD) offers a comprehensive visual representation of the database structure. It depicts the interconnections between various entities such as employees, customers, products, sales, sales items, supply orders, categories, and suppliers. Each entity is clearly defined with its attributes, primary keys (PK), and foreign keys (FK), highlighting the relationships ranging from one-to-many to many-to-one. This diagram serves as a blueprint for understanding the database schema, showing how different tables relate to one another and laying the groundwork for the implementation of a relational database that efficiently manages the store's operations and data flow.

****

**Building database**

I utilized Vertabelo for the generation of the SQL script required to implement the database. The script for database creation has been made available on the project's GitHub, with the link provided [here](https://github.com/leelasrinivasrajusarikonda/CPSC-50900-Database-Systems/blob/main/sql-queries/create-database.sql). Below is a screenshot that shows execution of the database creation script using the MySQL shell, including a display of the table list.



# **Populate the database with data**

Following the implementation of the database structure, the next critical step involves populating these tables with data. To facilitate this, a Python script has been crafted to automate the conversion of previously discussed data files in CSV format into SQL INSERT statements. The excerpt of code provided below showcases a function named generate\_insert\_statements, which reads the contents of CSV files within a specified directory, dynamically constructs the corresponding INSERT statements for each row of data, and then compiles these statements into a .sql file. This automated approach streamlines the process of transferring bulk data into the newly created database tables, ensuring efficiency and accuracy in initializing the database with the required data.

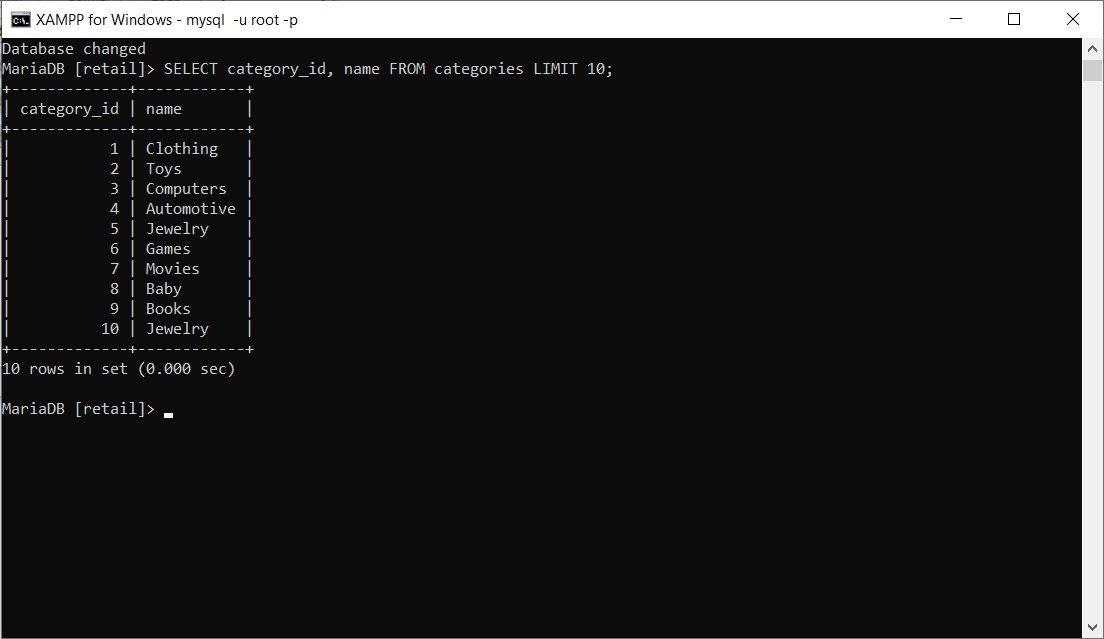


The scripts tasked with filling the database have been uploaded to our project's GitHub repository. As indicated in the preceding code snippet, the names of these files begin with ‘populate-‘. You can access the scripts via this link [here](https://github.com/leelasrinivasrajusarikonda/CPSC-50900-Database-Systems/tree/main/sql-queries).

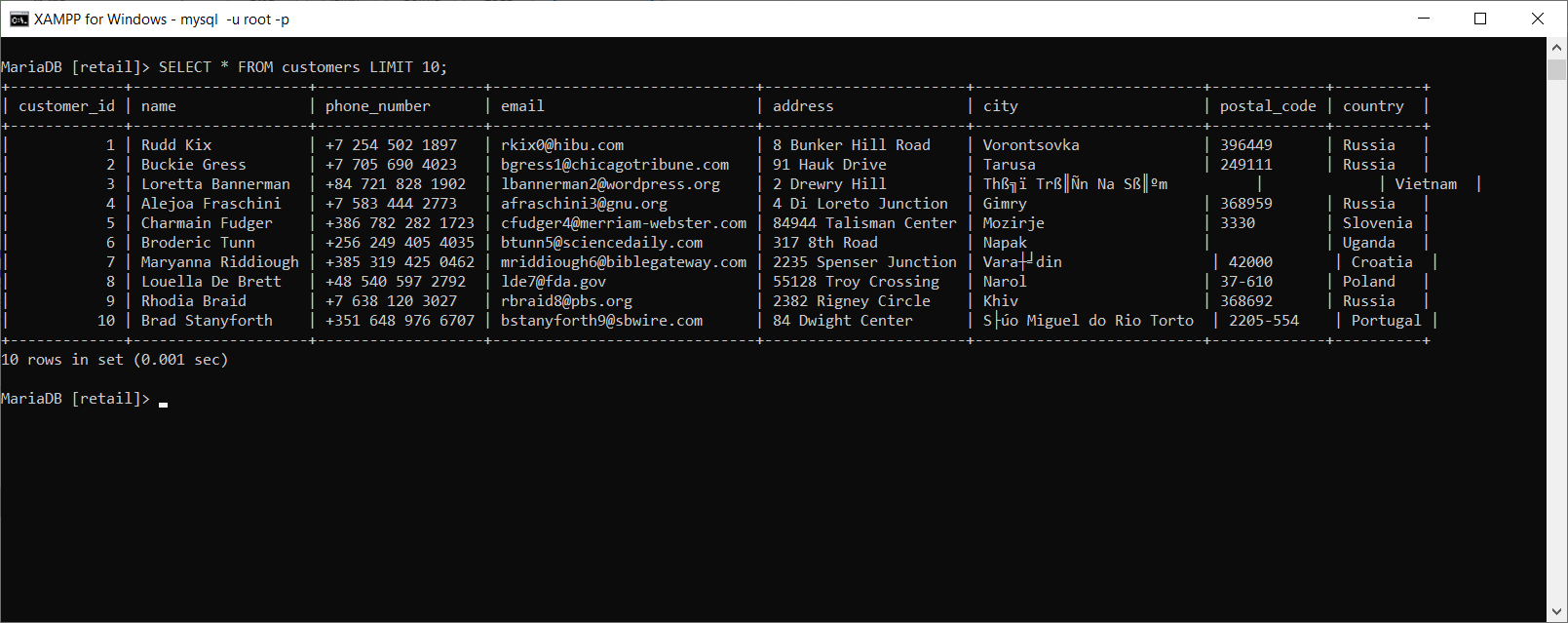
**Screenshots of data**

Below are screenshots of data:

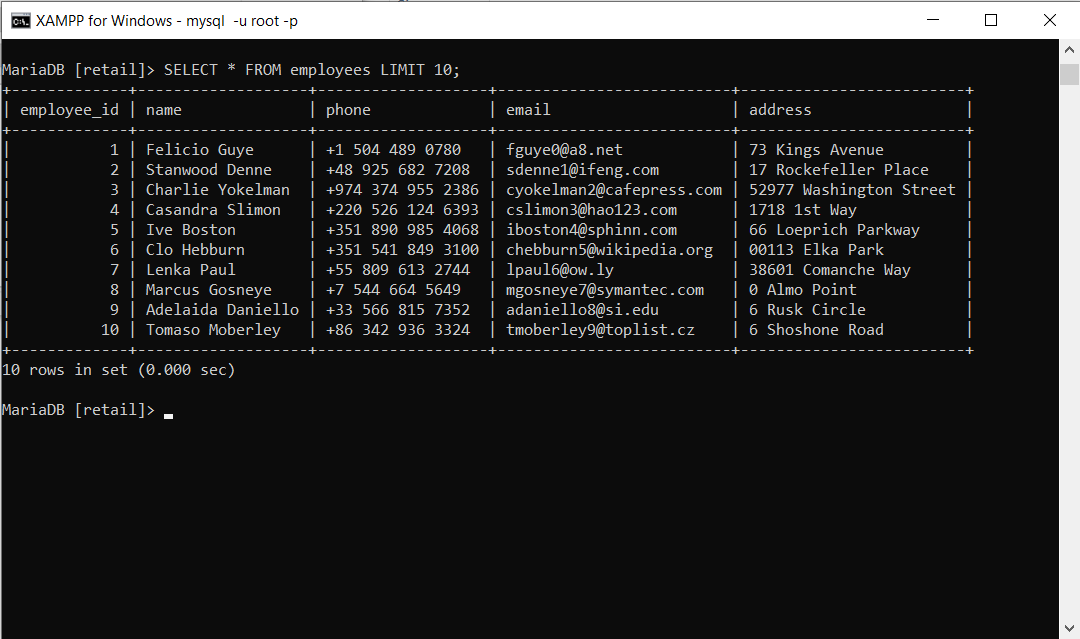
* categories table



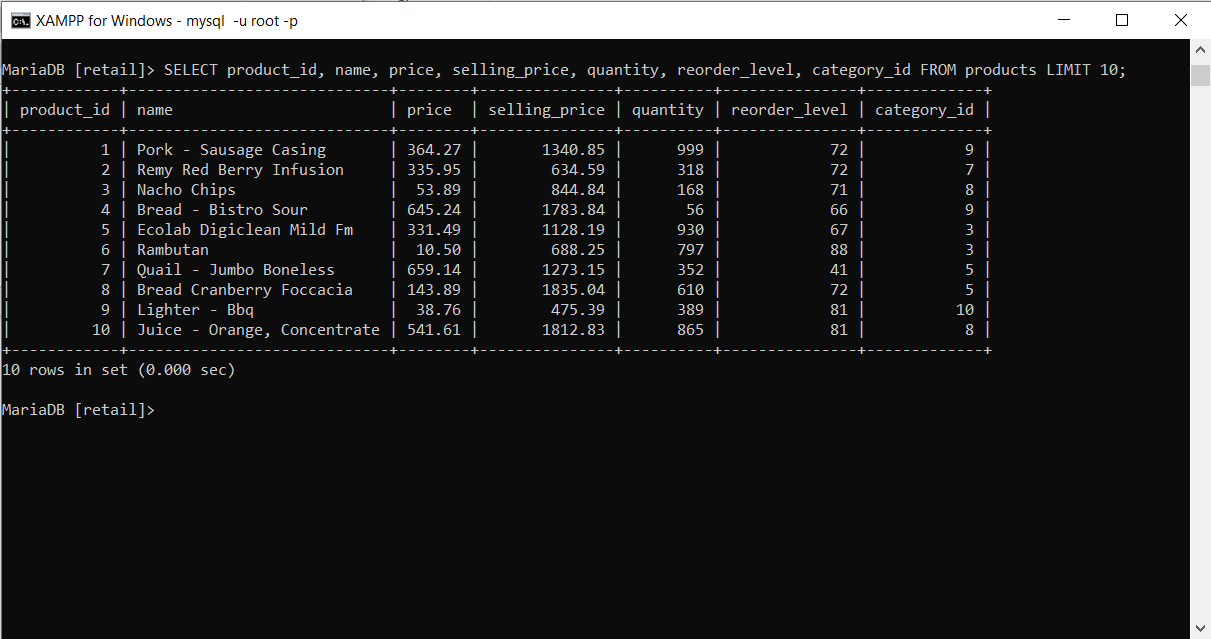
* customers table



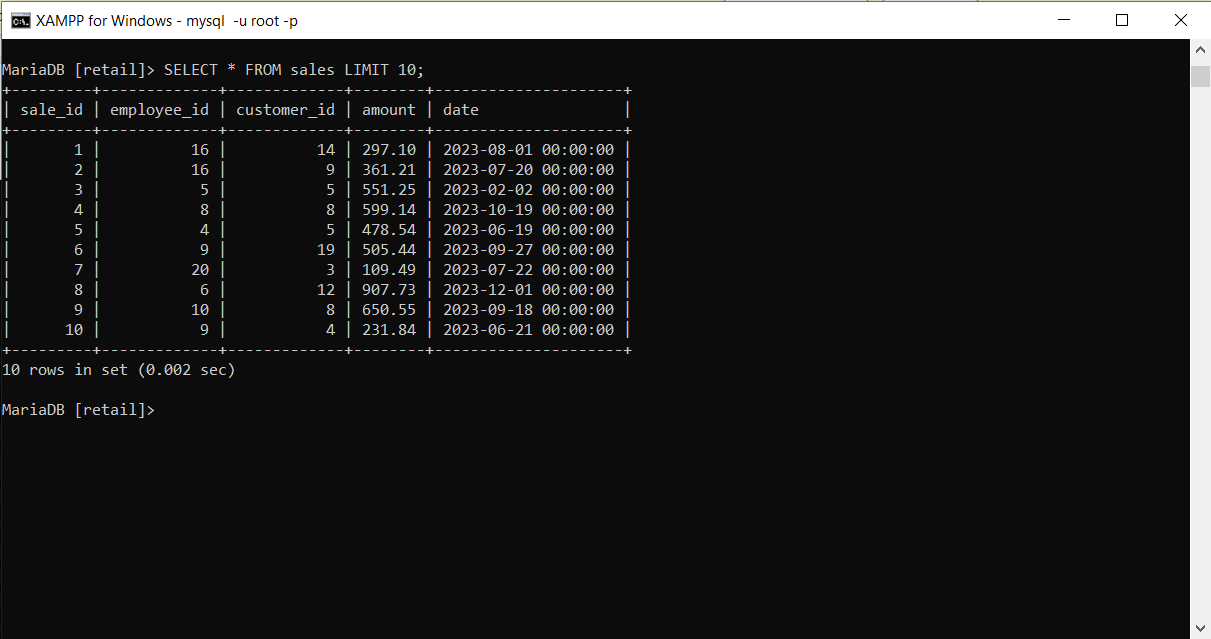
* employees table



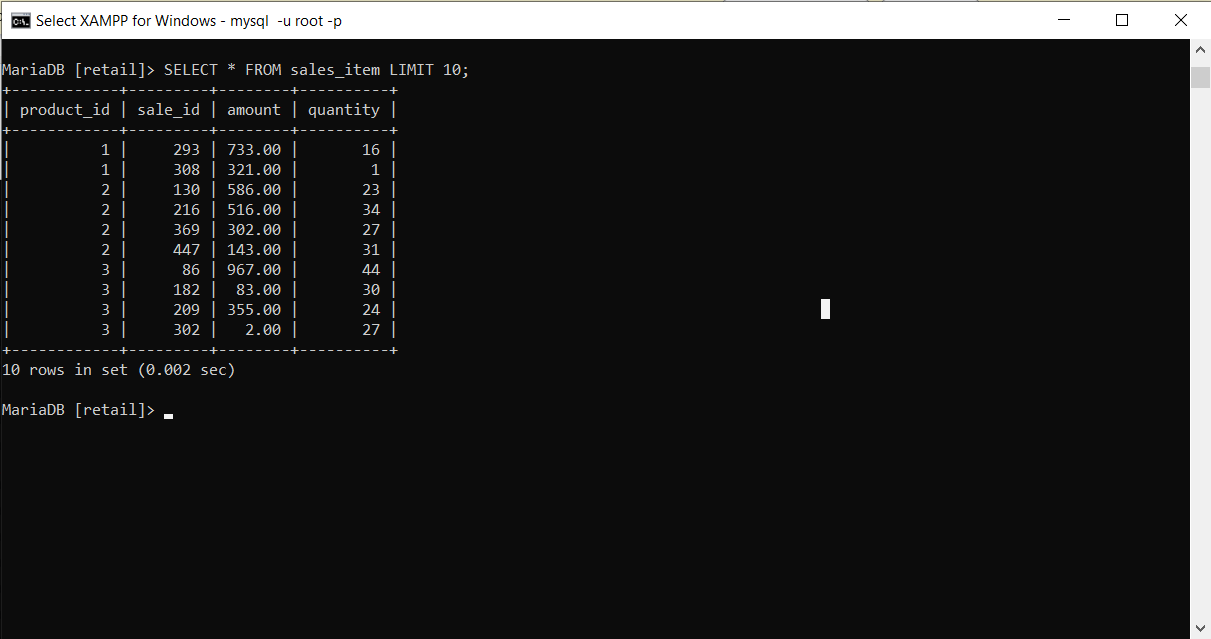
* products table



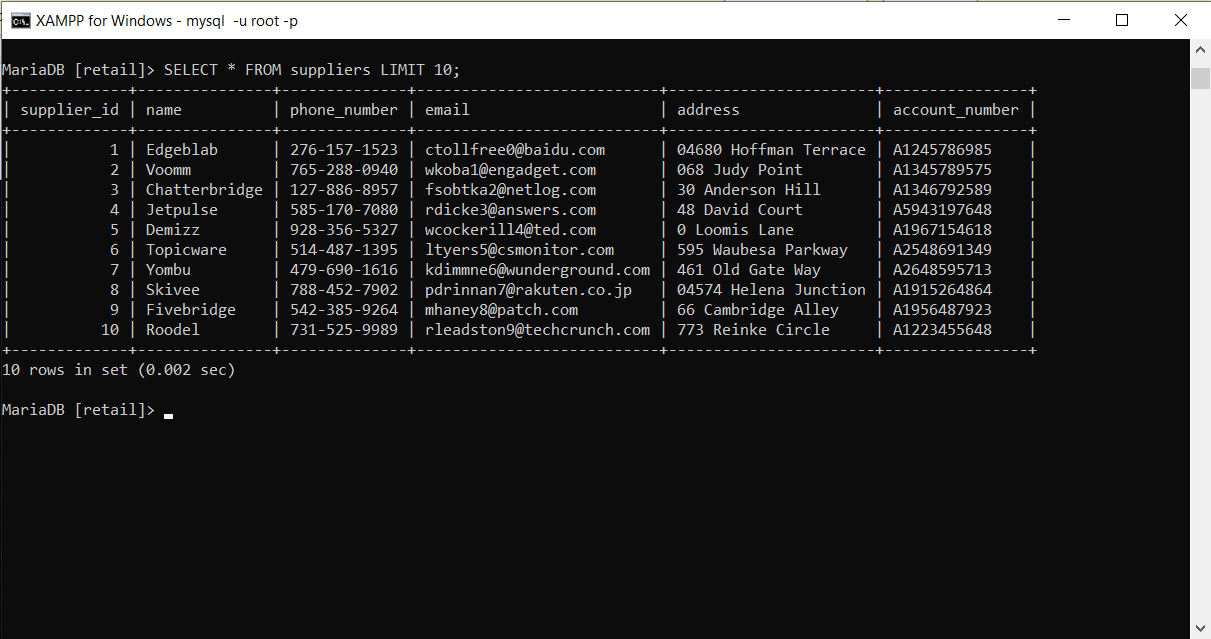
* sales table



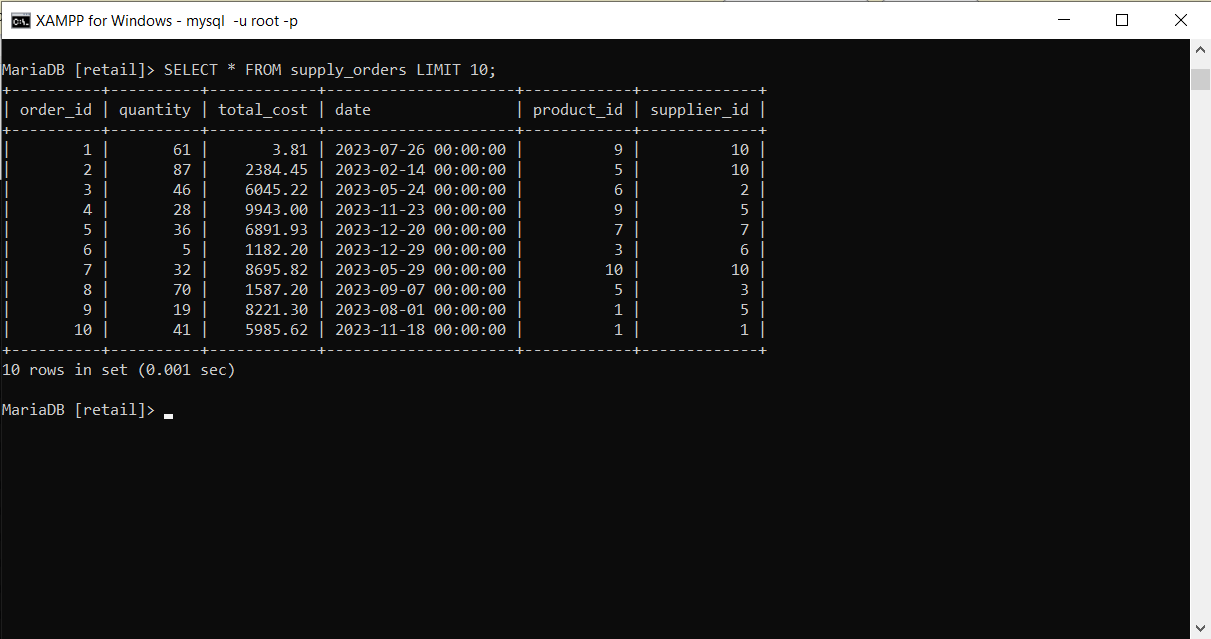
* sales\_item table



* suppliers table



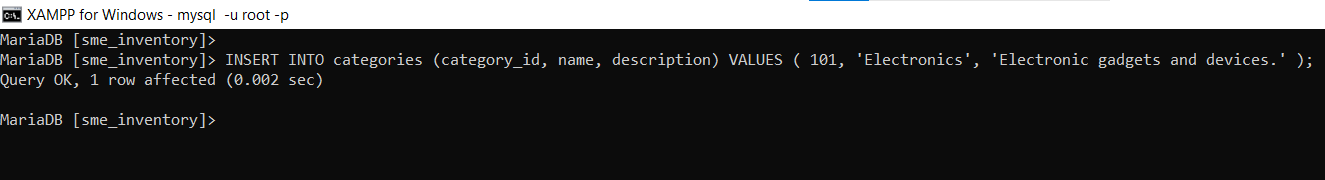
* supply\_orders table



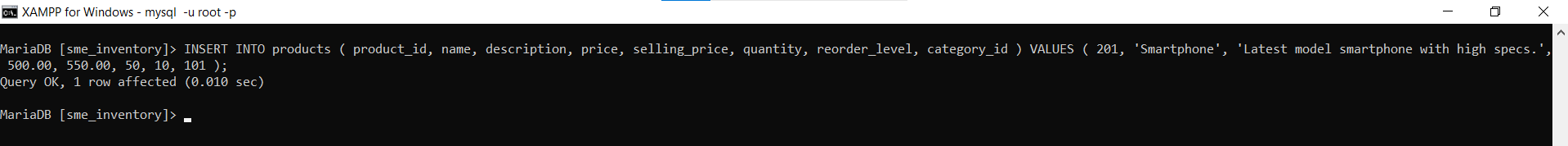
# **Data Manipulation Language (DML) Scripts**

1. **Insert Statements**

* **Insert a New Category**

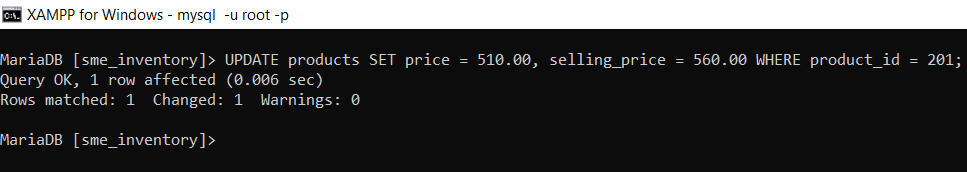
****

* **Insert a New Product**

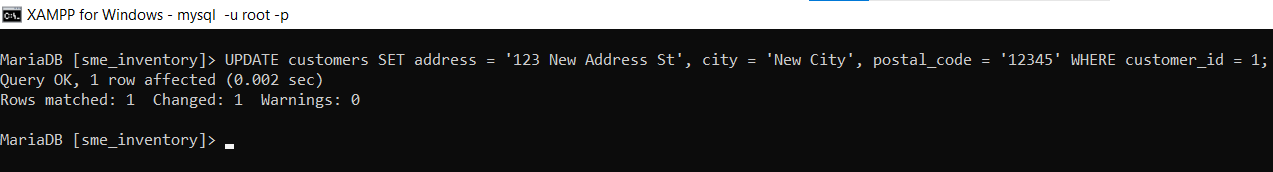
****

1. **Update Statements**

* **Update Product Price and Selling Price**

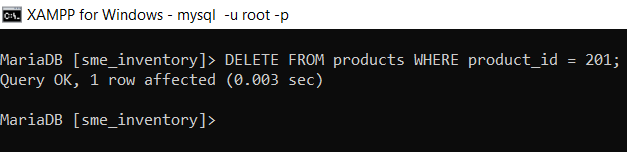
****

* **Update Customer Address**

****

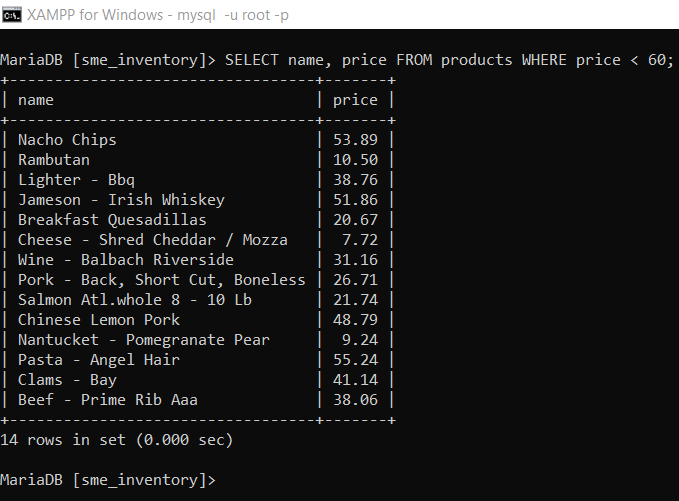
1. **Delete Statement**

* **Delete a Product**

****

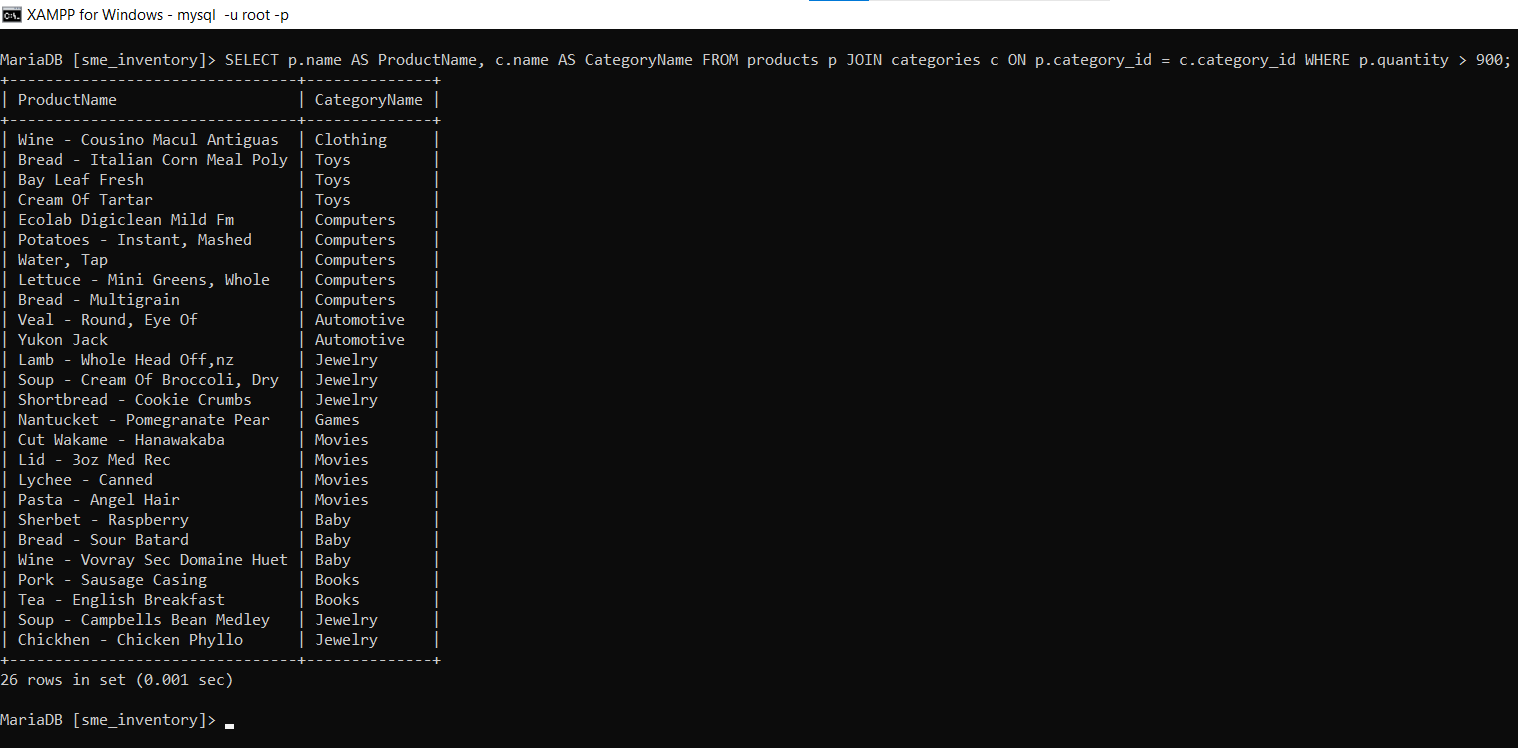
1. **Simple Select Statement**

* **Select Product Names and Prices**

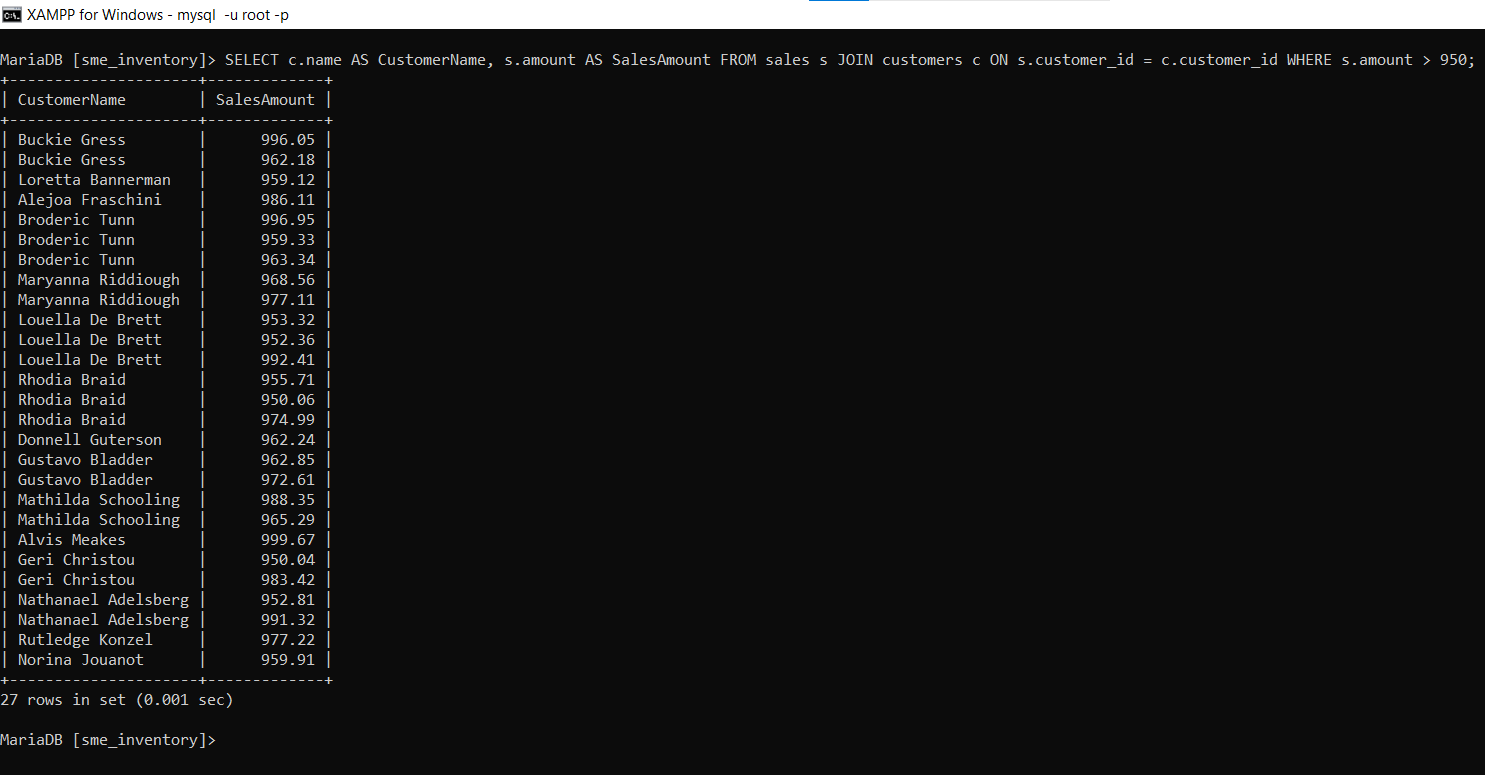
****

1. **Join Statements**

* **Product Names with Their Categories**

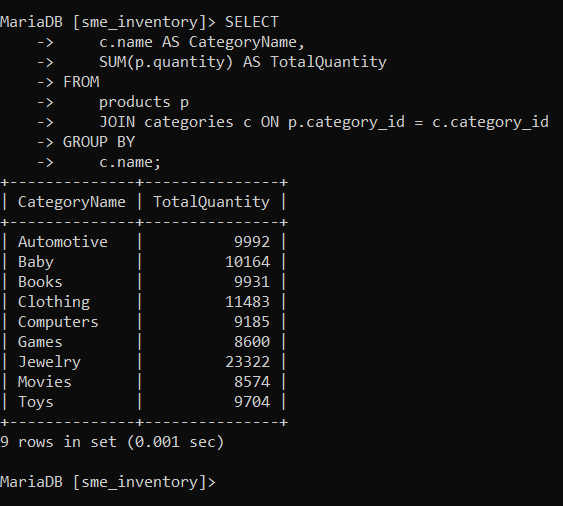
****

* **Sales Amounts and Customer Names for each sale order**

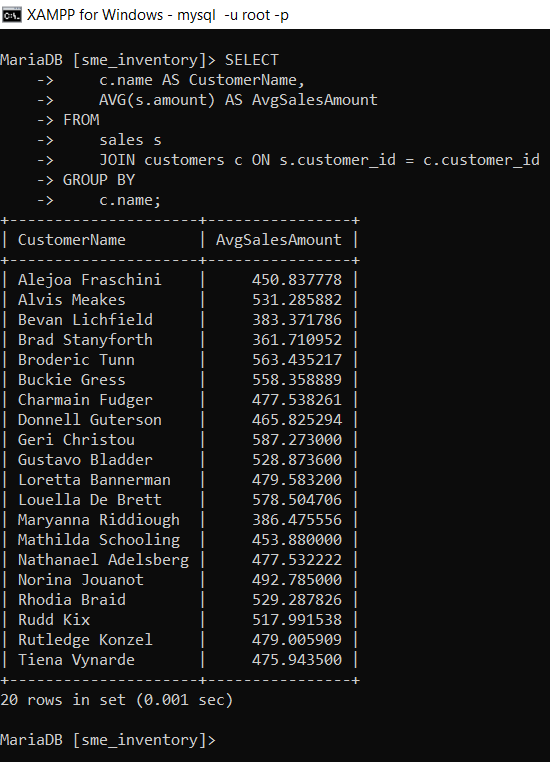
****

1. **Summary Functions**

* **Total Quantity of Products Grouped by Categories**

****

* **Average sales order amount for customers**

****

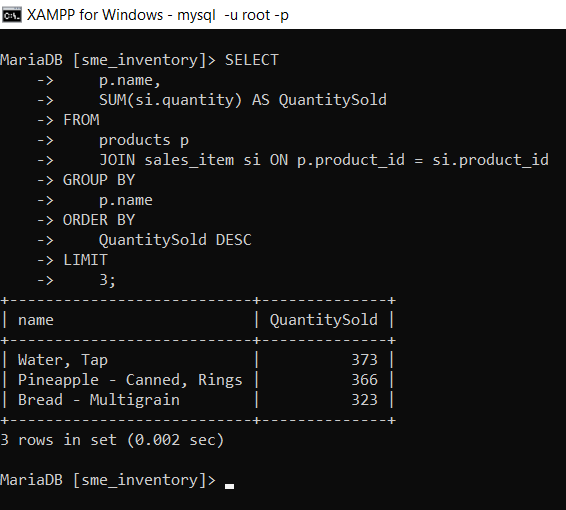
1. **Multi-table Query**

* **Products and Their Sales Quantities**

****

1. **Query of Choice**

* **Top 3 Best-Selling Products**

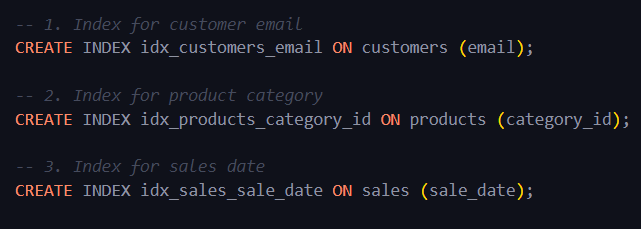
****

# **Indexes**

To improve the performance of the database design, we'll add indexes to the tables based on common query patterns, expected high-read areas, and foreign key relationships to speed up data retrieval and join operations. Below are the indexes:

* **Customer Email Index:** Since customer email addresses are unique and often used for customer lookup, an index on the email column in the customers table can significantly improve lookup times.
* **Product Category Index:** Products are frequently queried by category to list products of a specific type. An index on the category\_id foreign key in the products table will speed up these queries.
* **Sales Date Index:** Analyzing sales over time is a common operation, requiring efficient queries by date. An index on the sale\_date column in the sales table will facilitate faster retrieval of sales records within specific time frames.

**SQL to Generate the Indexes**



To demonstrate the performance improvement afforded by these indexes, we would conduct before-and-after benchmark tests on relevant queries. Here's how:

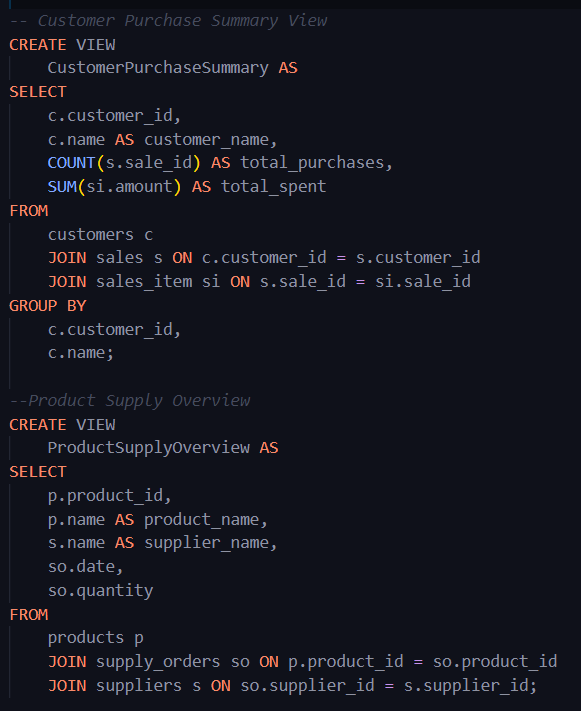
1. **Collect Baseline Performance Metrics:** Run a set of representative queries that would benefit from the indexes (such as looking up customers by email, listing products of a certain category, and retrieving sales records for a given period) and record their execution times and resource usage without the indexes in place.
2. **Implement Indexes and Gather New Metrics:** After creating the indexes, rerun the same queries and collect the same performance metrics.
3. **Compare Results:** Analyze the differences in execution times and resource usage between the baseline and post-index implementation. A significant reduction in query execution time and resource consumption would indicate a positive impact from the indexes.

This approach provides a clear, quantifiable measure of how the added indexes improve database performance, particularly for read-heavy operations.

# **Views**

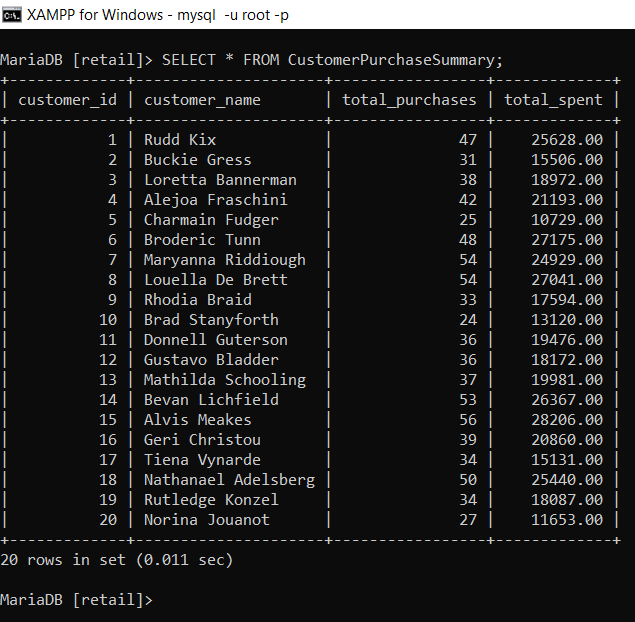
To enhance the accessibility and interpretation of data across multiple tables in the database, we introduced two views. These views aggregates and simplify the access to complex data relationships, providing streamlined data retrieval for common use cases.

Below is a screenshot of the SQL the that handles the creation of views.

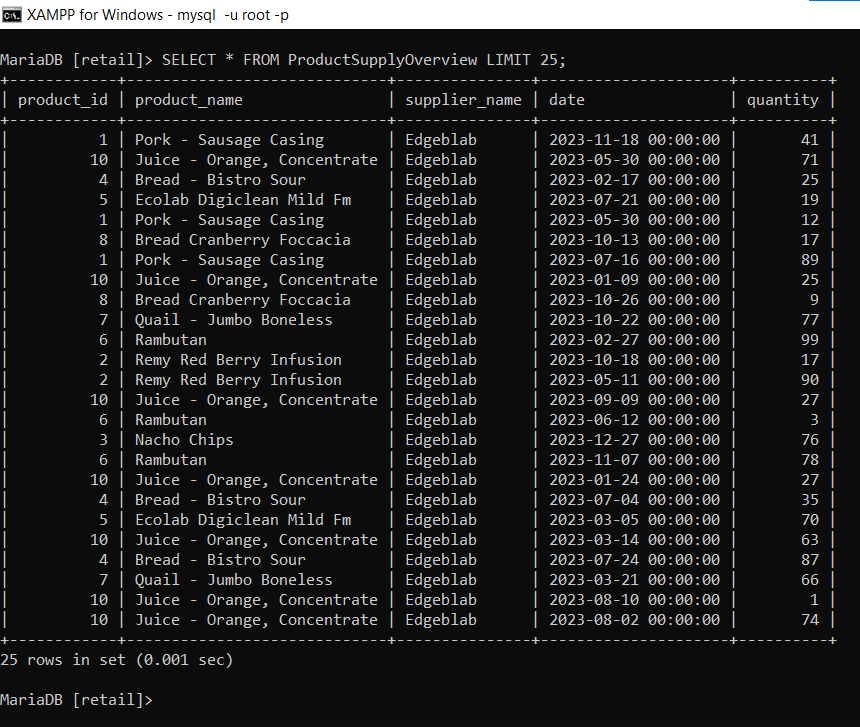


**Explanation and Value of Each View**

1. **Customer Purchase Summary:** This view is invaluable for customer relationship management and marketing efforts. It enables quick access to key metrics about customer behavior, such as frequency of purchases and money spent, which can be used to come up with marketing strategies, identify valuable customers, and enhance customer service.



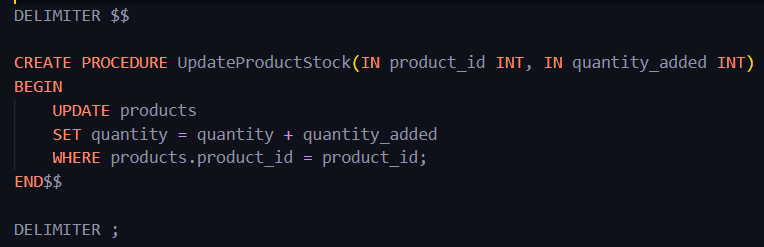
1. **Product Supply Overview:** This view aids in inventory management and supply chain oversight by providing a clear picture of the supply status of each product, including which suppliers are providing them and the details of current orders. It's crucial for ensuring stock levels are maintained and for planning future supply orders.



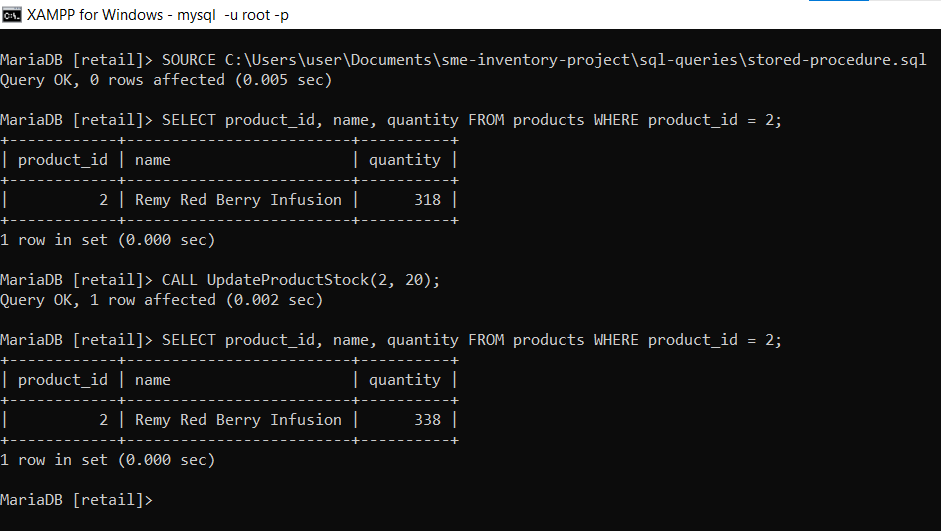
# **Stored Programs (Stored Procedures, Stored Functions, Triggers)**

To enhance the functionality and automation certain tasks within the database, it’s advised to introduce a stored procedure that serves a common operational need.

We created a stored procedure that updates the stock level of a specific product in the products table after a supply order has been received. It takes two parameters: the product id of the product to update and the quantity added, which is the number of items added to the stock.



Below is a screenshot that shows the creation and use of the stored procedure defined above. The procedure is used to update the stock of the product with id 2 from 318 to 338.



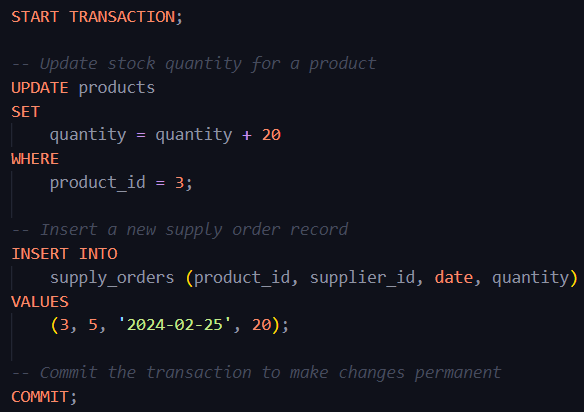
# **Transactions**

Transactions are a fundamental concept in database systems, crucial for maintaining data integrity and consistency, especially in environments where multiple operations need to be executed as a single, atomic unit. They are essential for ensuring ACID (Atomicity, Consistency, Isolation, Durability) behavior, which are four key properties that guarantee database transactions are processed reliably:

1. Atomicity: Ensures that all operations within a transaction are completed successfully. If any operation fails, the transaction is aborted, and the database state is rolled back to its state before the transaction began. This "all or nothing" approach prevents partial updates that could corrupt the database.
2. Consistency: Guarantees that a transaction transforms the database from one valid state to another valid state, maintaining all predefined rules, such as constraints, cascades, and triggers. This ensures that database integrity is preserved across transactions.
3. Isolation: Ensures that transactions are executed in isolation from each other, preventing concurrent transactions from affecting each other's execution and outcome. This isolation is critical in multi-user database systems, where concurrent transactions could lead to inconsistencies.
4. Durability: Ensures that once a transaction has been committed, it remains so, even in the event of a system failure. This property guarantees that the effects of the transaction are permanently recorded in the database.

**Demonstrating a MySQL Transaction**

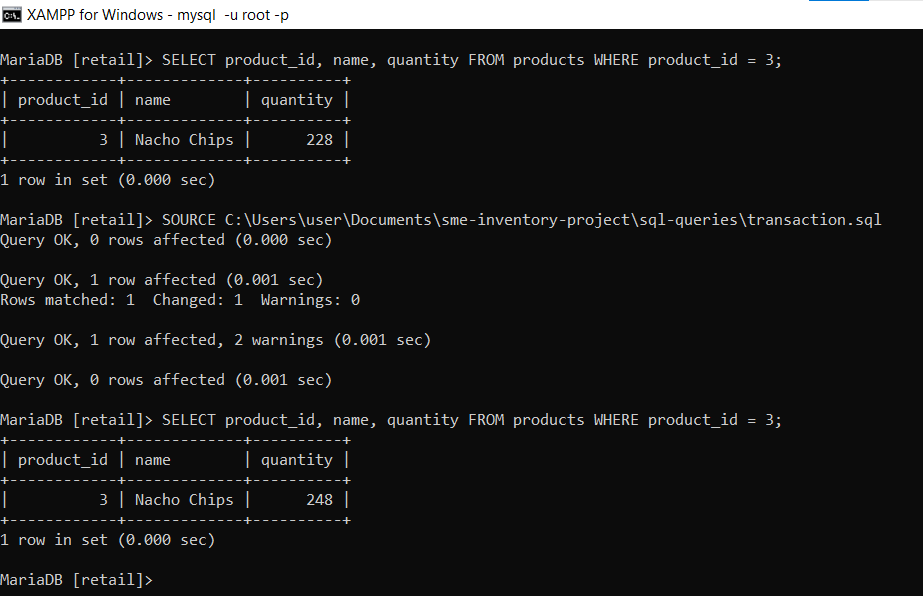
To demonstrate a transaction, we considered a scenario where we need to update the quantity in the products table and record the supply order in the supply\_orders table as part of a single transaction:



In this transaction:

* The START TRANSACTION command begins the transaction.
* Two SQL statements are executed: one to update the stock quantity of a product and another to insert a new supply order.
* The COMMIT command is used to permanently apply these changes. If either operation fails, a ROLLBACK command could be used instead of COMMIT to undo all changes made during the transaction, preserving atomicity.

Here is a screenshot that demonstrates the execution of the above defined transaction.



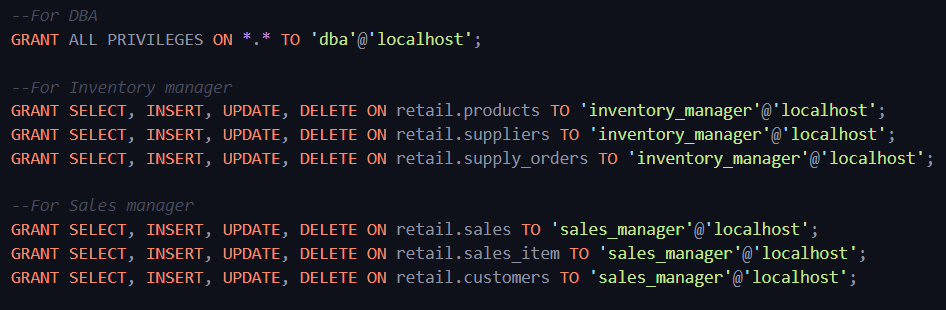
# **Database Security**

In any comprehensive database system, especially one managing resources for SMEs, it's crucial to delineate access privileges among various user roles to enhance security and operational efficiency. For our database, we can identify several key user roles:

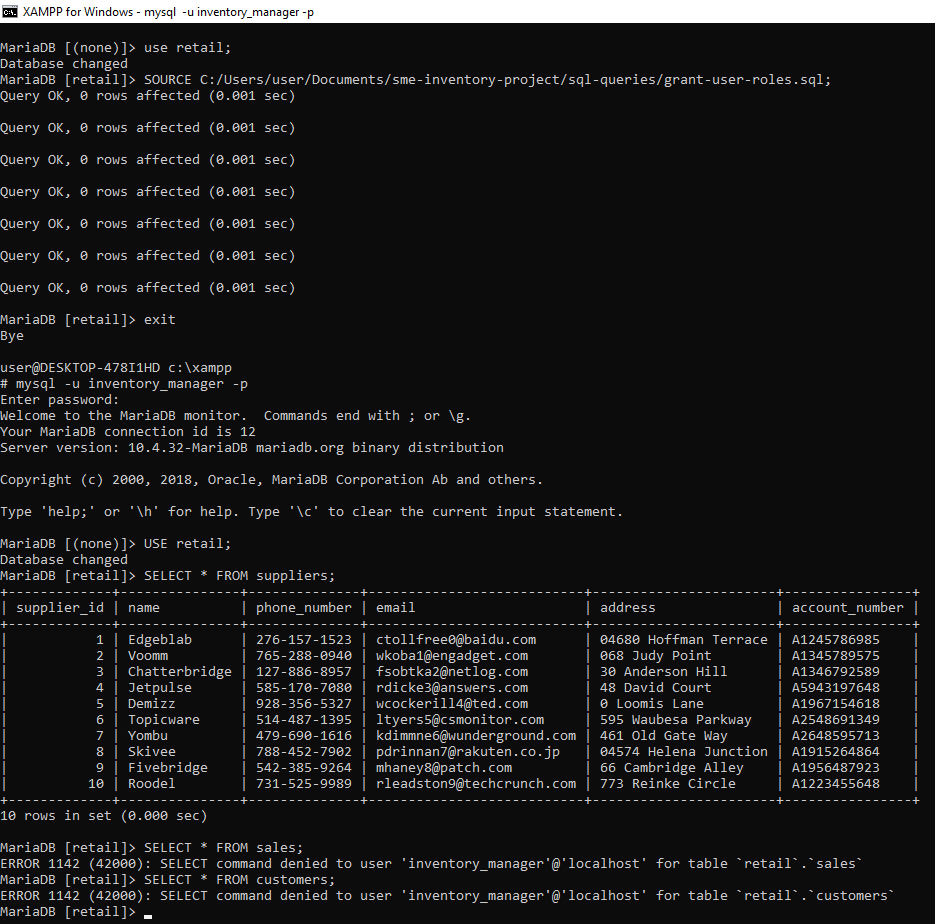
1. **Database Administrator (DBA):** Responsible for overall management, including schema creation, user management, and backup operations. They require full access.
2. **Inventory Manager:** Oversees inventory records, including products, suppliers, and supply orders. Needs access to read and write in specific tables.
3. **Sales Manager:** Manages sales data and customer information. Requires access to read and write sales-related and customer tables.

Below are the commands we used to assign privileges to the users we identified.

* **DBA:** Full access to all database operations.
* **Inventory Manager:** INSERT, UPDATE, DELETE on products, suppliers, supply\_orders.
* **Sales Manager:** INSERT, UPDATE, DELETE on sales, sales\_item, customers.



To illustrate the effective application of these permissions, following the execution of grant statements to allocate the necessary privileges, we proceeded to access the database under the role of inventory manager, as depicted in the screenshot below. We executed a select query on the authorized table and successfully retrieved the data. Nonetheless, when attempting to access data from a non-permitted table, we encountered an error message, which prohibited access to the data within the table.



# **Locking and Concurrent Access**

Locking tables in a database is crucial to maintaining data integrity when multiple transactions are trying to access the same data simultaneously. Without locking, concurrent transactions could lead to data anomalies such as lost updates, dirty reads, non-repeatable reads, and phantom reads.

**Example of the Need for Table Locking**

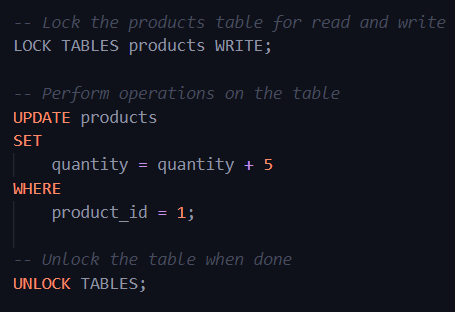
Consider a scenario in our database where two employees are attempting to update the stock quantity of the same product at the same time:

1. Employee A checks the current stock quantity of a product, sees there are 10 items, and decides to add 5 more.
2. Simultaneously, Employee B also checks the stock quantity, sees 10 items, and decides to add 10 more.
3. Employee A updates the stock to 15.
4. Shortly after, Employee B updates the stock to 20, based on the original quantity they observed.

In this case, Employee A's update is lost, and the stock quantity is incorrectly set to 20 instead of the correct total of 25. This is a classic "lost update" problem.

**Locking Tables to Prevent Inconsistencies**

To prevent such inconsistencies, we use table locking to ensure that when one transaction is reading or writing to a table, no other transaction can make changes until the first one is complete. Below is an example on how to use locks.



# **Backing Up Your Database**

Backing up a database is an essential part of maintaining data integrity and ensuring that you can recover from hardware failure, data corruption, or other unforeseen issues. Here's a comprehensive backup strategy for our database:

**Frequency of Backups**

The frequency of database backups should be determined by how often the data changes and the importance of the data. For a busy database:

* Full Backups: Should be performed daily during off-peak hours to minimize impact on database performance. This provides a snapshot of the entire database at a point in time.
* Incremental Backups: Can be performed every few hours, depending on the volume of transactions. These backups record only the changes since the last full or incremental backup, making them faster and smaller in size.

**Automating Backups**

Backups can be automated using cron jobs on a Linux server or Task Scheduler on Windows.

**Storage of Backups**

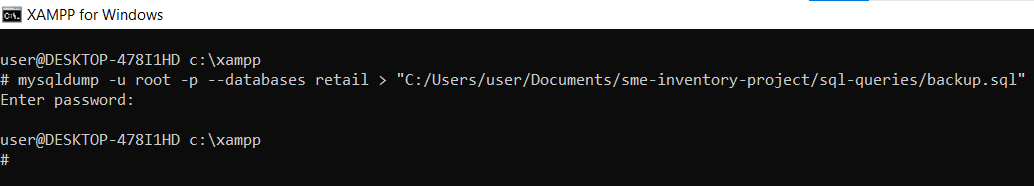
Backups should be stored in a secure, remote location separate from the database server to protect against server failure:

* Local Storage: A secure and encrypted partition on the same network, but not on the same physical server as the database.
* Remote Storage: A cloud storage solution like Amazon S3, Google Cloud Storage, or a dedicated remote backup server.

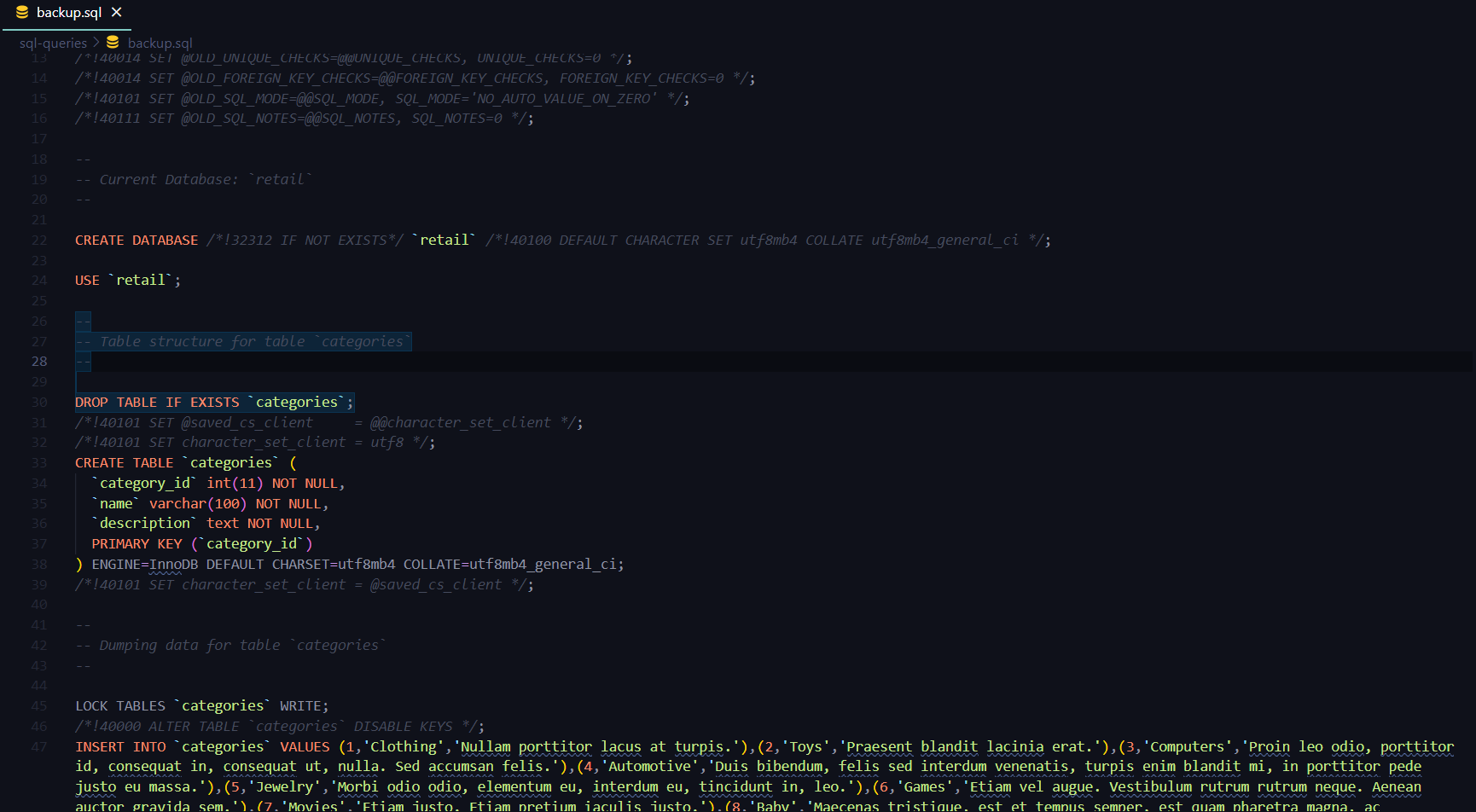
**Security of Backups**

Backup files should be encrypted to prevent unauthorized access, especially when stored off-site or in the cloud. Furthermore, access to backup files should be restricted to authorized personnel only, and backup files should be transmitted over secure channels.

**Backing up our database**

To backup our database we used the command in the screenshot below:  


Here is a portion of the backup files obtained from executing the above command.



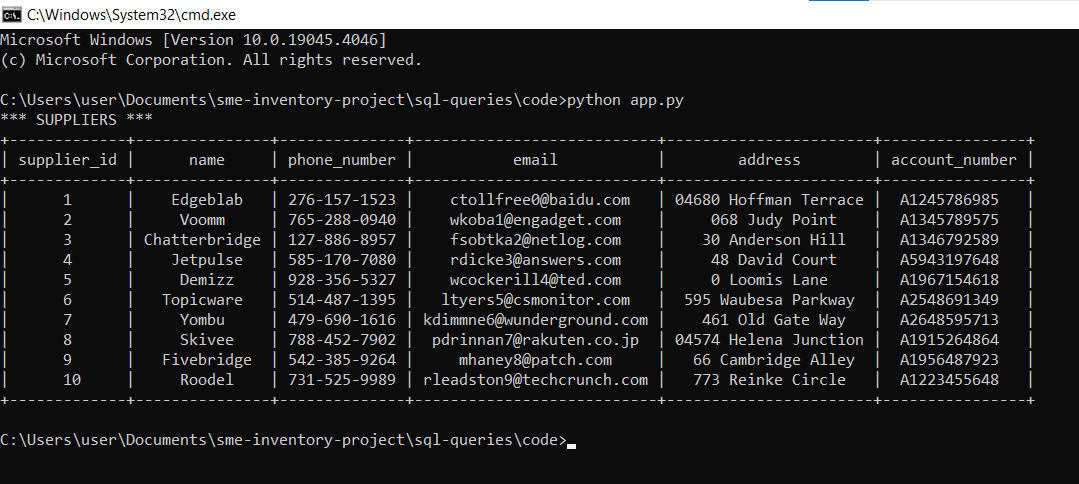
# **Programming**

We created a Python script that connects to our database, retrieves data, and generates a report which is both displayed on the screen and saved to a text file. I used mysql.connector to create a connection between the program and the database and also for output organization I used PrettyTable library. Below is the script that accomplishes this task:



Upon execution of the above Python script the program outputs the data fetched from the database in this case a list of all the suppliers in our database as shown in the screenshot below.

Also the script creates a txt file name report with the same contents as displayed in the screenshot below



# **Suggested Future Work**

**Limitations of the Current Database:**

1. Flexibility: The schema-based structure requires pre-defined schemas and any changes to this schema can be complex and disruptive.
2. Performance: For certain types of queries or data-intensive operations, relational databases can be slower compared to NoSQL databases which are designed for speed.

**Addressing Shortcomings:**

1. Flexibility: Incrementally adopt a service-oriented architecture (SOA) that allows for micro-services to interact with various databases that may be better suited for their specific needs.
2. Performance: Optimize queries, add more indexing, and evaluate stored procedures to ensure they are providing the necessary performance benefits. Use caching solutions where appropriate.

**Migration to Cloud Services:**

Migrating the database to a cloud platform can significantly increase performance and availability. Cloud services offer managed databases that come with benefits such as:

* Automated backups and disaster recovery solutions.
* Dynamic scaling of resources to match load requirements.
* Global distribution to reduce latency by serving data from locations closer to the user.
* Integrated monitoring and performance tuning tools.
* Security: Cloud providers often have robust security measures that might be more sophisticated than what can be managed on-premises.

**Advantages and Disadvantages of NoSQL:**

**Advantages:**

* Schema-less: NoSQL databases allow the storage of unstructured data and can handle a variety of data types.
* Scalability: They are designed to scale out by distributing the data across many servers.
* High Performance: NoSQL databases are optimized for specific data models and access patterns which can lead to faster performance for certain tasks.

**Disadvantages:**

* Complex Transactions: NoSQL databases often lack the ability to perform complex transactions that are ACID-compliant.
* Consistency: Many NoSQL databases sacrifice consistency in favor of availability and partition tolerance.
* Data Redundancy: NoSQL databases may require more storage due to data denormalization.

Improving the current database system should take into account both the immediate needs of the application it supports and the future growth trajectory, ensuring the database architecture can evolve along with the application.

# **Activity Log**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Contributor** | **Activity** | **Description** |
| Jan 14, 2024 | Leelasrinivasraju Sarikonda and Mohan Nadimpalli | Project kickoff | Researched on various database systems and their applications |
| Jan 16, 2024 | Mohan Nadimpalli | Initial Proposal Drafting | Drafted the initial proposal, focusing on project objectives and significance. |
| Jan 18, 2024 | Leelasrinivasraju Sarikonda | Research on Data Sources | Researched potential data sources for the inventory management system. |
| Jan 21, 2024 | Mohan Nadimpalli | Data Files Organization | Organized data files into CSV format for database integration. |
| Jan 24, 2024 | Leelasrinivasraju Sarikonda | Alternative Database Models Exploration | Explored and documented NoSQL and Hierarchical models as alternatives to relational databases. |
| Jan 28, 2024 | Leelasrinivasraju Sarikonda | GitHub Setup | Created the GitHub repository for the project and prepare the README document. |
| Jan 31, 2024 | Leelasrinivasraju Sarikonda | Relational Database Design Process | Identified functional dependencies among the fields provided, designated determinants, and outlined entity sets. Initiated naming conventions for entity sets. |
| Feb 2, 2024 | Mohan Nadimpalli | Relational Database Design Process (Cont.) | Established relationships among entity sets, identifying connectivity and participation for each. |
| Feb 4, 2024 | Leelasrinivasraju Sarikonda | Relational Database Design | Analyzed entity sets for adherence to Normalization. Began ER diagram design in Vertabelo. |
| Feb 7, 2024 | Mohan Nadimpalli | CSV to SQL Python Script | Developed Python Script to convert CSV data files to SQL INSERT queries. |
| Feb 9, 2024 | Leelasrinivasraju Sarikonda | Data Definition Language (DDL) Scripts | Used Vertabelo to generate DDL scripts for database creation. Evaluating the script ensuring foreign keys were linked. Prepared documentation and screenshots for report inclusion. |
| Feb 11, 2024 | Mohan Nadimpalli | Data Manipulation Language Scripts | Wrote SQL commands for various operations including insert, update, delete, and select queries. Prepared documentation and screenshots for report inclusion. |
| Feb 15, 2024 | Leelasrinivasraju Sarikonda | Conceptual Model Creation | Developed the initial conceptual model by identifying essential entity sets and their relationships. |
| Feb 17, 2024 | Mohan Nadimpalli | Logical Model Development | Expanded upon the conceptual model by adding attributes to each entity set and identifying functional dependencies. |
| Feb 19, 2024 | Leelasrinivasraju Sarikonda | Normalization and Logical model ERD creation | Performed normalization analysis and drawing the Entity-Relationship Diagram (ERD) to visually represent the logical model |
| Feb 21, 2024 | Mohan Nadimpalli | Physical Model and SQL Scripting | Finalized the physical model by specifying data types and adding constraints. Compiled and formatted the report with the new template. |
| Feb 22, 2024 | Leelasrinivasraju Sarikonda | Index Creation | Identified and defined indexes for optimizing database queries. |
| Mohan Nadimpalli | Database Security Setup | Outlined database user roles, crafted GRANT SQL statements for appropriate access control. |
| Feb 23, 2024 | Leelasrinivasraju Sarikonda | Implementation of Database Views | Created views combining data from multiple tables for easy access. Provided SQL implementations and screenshots. |
| Mohan Nadimpalli | Transaction Management | Demonstrated transaction with an example showcasing integrity and consistency. |
| Feb 24, 2024 | Mohan Nadimpalli | Locking Mechanisms and Concurrent Access | Demonstrated table locking to prevent data inconsistencies during concurrent transactions, with examples and screenshots illustrating data consistency. |
| Feb 25, 2024 | Leelasrinivasraju Sarikonda | Database Backup Strategy | Developed a backup strategy detailing command execution, automation, storage, and security. |
| Mohan Nadimpalli | Database Programming with Python | Wrote a Python script for database data extraction and presentation. |