

Lords of Data Project

Presentation: ClassicModels

ClassicModels

Home Products Login

Welcome to ClassicModels

Your one-stop shop for premium die-cast models and replicas. From classic cars to vintage aircraft, explore our curated collection of high-quality collectibles.

[Shop All Products](#)

[Login](#)



Our Global Offices

Sydney

Australia

Paris

France

Tokyo

Japan

London

UK

For the BLG317E Database Systems course

Introduction



Lords of Data consists of 5 members:

- **Muhammed Kağan TEMUR (150230093)**
 - Mainly responsible for the `orderdetails` table, as well as the `payments` table.
 - Handled order processing logic & finance analytics.
 - Helped design main website layout.
 - Helped set up Flask routes for the website's general flow.
 - Built the ER diagram.

Payment History			
Date	Transaction ID	Amount	Actions
2005-01-18	GJ597719	\$8,307.28	
2003-11-24	MU817168	\$52,548.49	
2003-08-20	H0576374	\$41,016.75	

Checkout
Review your items and complete your order.

Order Summary

Product	Unit Price	Qty	Subtotal
2002 Suzuki XREO	\$150.62	10	\$1506.20
ATA: B757-300	\$118.65	6	\$711.90

Order Notes
Special instructions for delivery, gift wrap requests, etc.

Payment Details

Subtotal	\$2218.10
Shipping	Free
Total	\$2218.10

Place Order → **Return to Cart**

Order #10386

Order Items

Product	Qty	Price	Total
1900s Vintage Bi-Plane	3	\$56.86	\$220.54
The Titanic	4	\$83.14	\$3741.30
The Queen Mary	3	\$80.44	\$241.20
Pont Yacht	4	\$52.42	\$229.68
1999 Yamaha Speed Boat	1		

Summary

Subtotal (18 items)	\$46,968.52
Shipping	Free
Grand Total	\$46,968.52

Sales Rep Performance vs Office Average

Office	Sales Rep	Rep Revenue	Office Avg
Boston	Stuart Patterson	\$50,969.40	\$49,000.00
Boston	David Bell	\$44,200.31	\$49,000.00
London	Larry Bott	\$73,200.59	\$74,975.35
London	Berry Jones	\$70,486.3.91	\$73,747.35
NYC	George Vanau	\$66,937.70	\$57,794.06
NYC	Foon Yue Tseng	\$48,9312.67	\$57,794.06
Paris	Diane Hernandez	\$19,925.11	\$19,925.11
Paris	Pierre Chastellier	\$56,825.98	\$56,792.32
Paris	Loul Bonduar	\$55,946.75	\$61,752.32
Paris	Martin Gerard	\$39,747.74	\$47,726.57

Introduction



Lords of Data consists of 5 members:

- **Celal GÜNDÖĞDU (150230007)**
 - Mainly responsible for the **orders** table ,as well as the **offices** table.
 - Order history maintaining and managing system.
 - Office unit system and management for employees.
 - Complex analytical queries relating office performance.
 - Search and filter implementations.
 - Complete password authentication system.
 - UI modifications.

Our Global Offices

Sydney Australia <small>Employees: 4</small>	Paris France <small>Employees: 5</small>	Tokyo Japan <small>Employees: 2</small>	London UK <small>Employees: 2</small>
5-11 Wentworth Avenue Floor #2 NSW 2010 +61 2 9264 2451 Edit Delete	43 Rue Joffroy D'abbans 75017 +33 14 723 4404 Edit Delete	4-1 Kioicho 102-8578 +81 33 224 5000 Edit Delete	25 Old Broad Street Level 7 EC2N 1HN +44 20 7877 2041 Edit Delete

CUSTOMER PROFILE
Euro+ Shopping Channel

[Back to Dashboard](#) [Filters](#)

Sort Order	Product Lines	Price Range	Order Status
<input checked="" type="radio"/> Newest First <input type="radio"/> Oldest First <input type="radio"/> Price: High to Low <input type="radio"/> Price: Low to High	<input type="checkbox"/> Classic Cars <input type="checkbox"/> Motorcycles <input type="checkbox"/> Planes <input type="checkbox"/> Ships <input type="checkbox"/> Trains <input type="checkbox"/> Trucks and Buses	<input type="checkbox"/> \$0-1000 <input type="checkbox"/> \$1000-10000 <input type="checkbox"/> \$10000-50000 <input type="checkbox"/> \$50000-100000 <input type="checkbox"/> >100000	<input type="checkbox"/> In Process <input type="checkbox"/> Shipped <input type="checkbox"/> Cancelled <input type="checkbox"/> Resolved

Showing 7 orders

#18424 2020-05-21	6 Items 1939 Cadillac Limousine, 1940 Ford Pickup Truck, 1952 Alpine Renault 1300 "spend zero"	\$29,310.30 IN PROCESS
#18427 2020-05-13	6 Items 1936 Harley Davidson El Knucklehead, 1969 Harley Davidson Ultimate Chopper, 1996 Moto Guzzi 1100i "Customer doesn't like the colors and precision of the models."	\$28,574.90 RESOLVED
#18432 2020-05-03	11 Items 1926 Ford Fire Engine, 1954 Greyhound Scenicruiser, 1957 Chevy Pickup	\$46,895.48 SHIPPED

Page 1 of 4

Corporate Analytics Hub



Introduction



Lords of Data consists of 5 members:

- **Hüseyin Hayri DEDE (150220002)**
 - Mainly responsible for the employees table, as well as the `employee_reports` table.
 - Dashboard page
 - Employee hierarchy
 - Employee reporting system
 - Sales and employee performance analytics

My Subordinates

NAME	ID	EMAIL	JOB TITLE	OFFICE	ACTION
Lou Bondur	1337	lbondur@classicmodelcars.com	Sales Rep	Paris	Fire
Larry Bott	1501	lbott@classicmodelcars.com	Sales Rep	London	Fire
Pamela Castillo	1401	pcastillo@classicmodelcars.com	Sales Rep	Paris	Fire
deneme deneme	1626	deneme@classicmodelcars.com	Sales Rep	London	Fire
Gerard Hernandez	1370	ghernandez@classicmodelcars.com	Sales Rep	Paris	Fire
Barry Jones	1504	bjones@classicmodelcars.com	Sales Rep	London	Fire

Management: Add New Sales Rep

New reps will report to you

First Name _____ Last Name _____ Extension _____ Email _____

Office Assignment
Select an Office... [+ Add Sales Rep](#)

A random password will be generated and shown once.

Sales Analytics

⚠ At Risk Employees (No Sales/Activity)

The following Sales Reps have zero orders.

- Tom King (ID: 1619) - [ting@classicmodelcars.com](#)
- Yoshimi Kato (ID: 1625) - [ykato@classicmodelcars.com](#)
- deneme deneme (ID: 1626) - [deneme@classicmodelcars.com](#)

Employee Performance (Revenue by Product Line)

SALES REP	PRODUCT LINE	TOTAL REVENUE GENERATED
Leslie Jennings	Classic Cars	\$373416.76
Leslie Jennings	Motorcycles	\$119788.86
Leslie Jennings	Planes	\$67956.68
Leslie Jennings	Ships	\$41565.71
Leslie Jennings	Trucks	\$17965.32
Leslie Jennings	Trucks and Buses	\$163280.84
Leslie Jennings	Vintage Cars	\$298556.37
Leslie Thompson	Classic Cars	\$143885.77
Leslie Thompson	Motorcycles	\$43921.71
Leslie Thompson	Planes	\$40675.58

[← Previous](#) Page 1 [Next →](#)

✓ [Submit Weekly Report](#)

Report Content

Report Date: 2025-11-21 21:43:05

Report Summary: I am currently reporting only from G. Hernandez so far. I am still waiting for other sales representatives' reports.

Team Reports (Subordinates)

EMPLOYEE	ID	JOB TITLE	DATE	REPORT
Gerard Hernandez	1370	Sales Rep	2025-11-21 21:32:55	We have arranged a meeting with Alpha Cognac (Customer #242). We talked about their recent orders and agreed on new orders.

✓ [My Recent Reports](#)

Report Date: 2025-11-21 21:43:05

Report Summary: I am currently reporting only from G. Hernandez so far. I am still waiting for other sales representatives' reports.

Introduction

Lords of Data consists of 5 members:

- **Ahmet Said Genç (150220092)**
 - Mainly responsible for the customer table.
 - Login page
 - Feel lucky page
 - Search and sort mechanism for employees' customer table
 - Customers CRUD
 - Product sort logic
 - UI enhancements



The screenshot displays a dark-themed web application interface. At the top, a header bar includes links for Home, Products, Cart, Welcome, and Logout. Below the header are three main sections: 'My Profile', 'Financial Summary', and 'Payment History'. The 'My Profile' section shows basic information for a user named Carine Schmitt, including address details (54, rue Royale, Nantes, France) and contact numbers (40.32.2555). The 'Financial Summary' section provides a breakdown of total orders (\$22,314.36), total payments (\$22,314.36), and a green progress bar for balance. The 'Payment History' section lists two transactions: one from 2018-12-18 (DK324932) and another from 2019-10-19 (HQ386386). The bottom portion of the interface features a 'ClassicModels' header and several account management sections: 'Account Settings' (with profile information and edit profile link), 'Financial Summary' (showing \$0.00 for total orders, total payments, and outstanding balance), 'Security' (with password change and forgot password links), and a red 'DELETE ACCOUNT' button with a warning message about permanent data loss. The final section at the bottom is a 'Welcome Back' login form for 'Customer' or 'Employee', requiring a 'Customer Number' (103), 'Password' (shown as dots), and a 'Login as Customer' button. A 'Forgot your account?' link is also present.

Introduction



Lords of Data consists of 5 members:

- **Erce Baran Keskin (070220716)**
 - Mainly responsible for the products table, as well as the productlines table.
 - Contributed to the Products, Manage Products, and Product Line Report pages.
 - Performed performance analytics per product line.

Our Collections

Explore our premium scale models by category

Classic Cars
Attention car enthusiasts: Make your wildest car ownership dreams come true. Whether you are looking...

[Browse Collection →](#)

Motorcycles
Our motorcycles are state of the art replicas of classic as well as contemporary motorcycle legends ...

[Browse Collection →](#)

Planes
Unique, diecast airplane and helicopter replicas suitable for collections, as well as home, office or...

[Browse Collection →](#)

Ships
The perfect holiday or anniversary gift for executives, clients, friends, and family. These handcar...

[Browse Collection →](#)

Trains
Model trains are a rewarding hobby for enthusiasts of all ages. Whether you're looking for collecti...

[Browse Collection →](#)

Trucks and Buses
The Truck and Bus models are realistic replicas of buses and specialized trucks produced from the ea...

[Browse Collection →](#)

Product Line Performance

Start	End	Status	Run			
01.01.2004	01.01.2005	Shipped				
Product Line	# Products	Units Sold	Revenue	Est. Gross Profit	Gross Margin	Customers
Classic Cars	37	15,424	\$ 1,682,980.21	\$ 671,878.21	39.9%	66
Vintage Cars	24	10,487	\$ 823,927.95	\$ 337,219.36	40.9%	64
Motorcycles	13	5,976	\$ 527,243.84	\$ 222,485.41	42.2%	34
Trucks and Buses	11	4,853	\$ 448,702.69	\$ 176,415.25	39.3%	29
Planes	12	5,439	\$ 438,255.50	\$ 168,722.36	38.5%	31
Ships	9	3,752	\$ 292,595.34	\$ 116,371.77	39.8%	29
Trains	3	1,290	\$ 86,897.46	\$ 30,590.05	35.2%	19

Product Management

Manage Catalog, Inventory, and Pricing

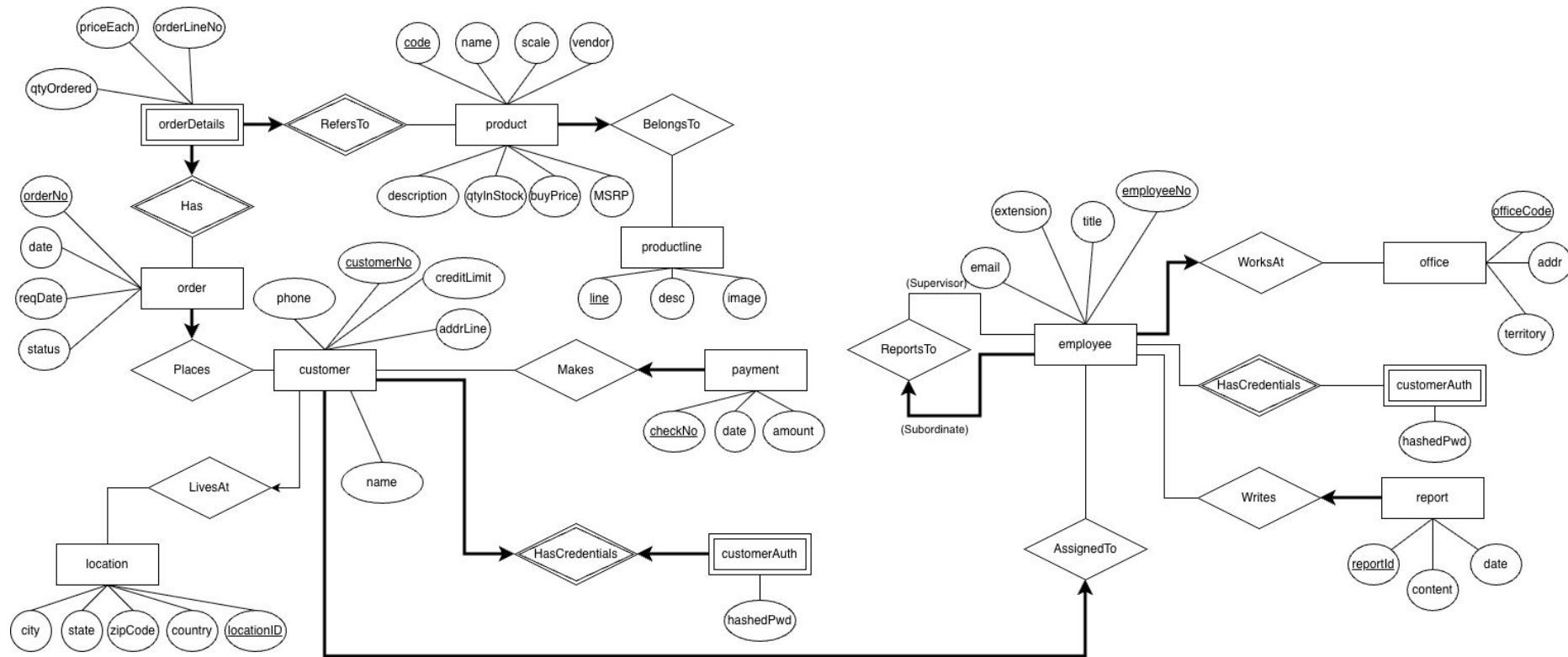
+ Add New Product

Show: 10	All Vendors	All Product Lines	Sort by...					
Code	Name	Line	Scale	Vendor	In Stock	Buy Price	MSRP	Actions
S18_1889	1948 Porsche 356-A Roadster	Classic Cars	1:18	Gearbox Collectibles	8826	\$53.90	\$77.00	<input checked="" type="checkbox"/> <input type="checkbox"/>
S18_3685	1948 Porsche Type 356 Roadster	Classic Cars	1:18	Gearbox Collectibles	8990	\$62.16	\$141.28	<input checked="" type="checkbox"/> <input type="checkbox"/>
S24_2766	1949 Jaguar XK 120	Classic Cars	1:24	Classic Metal Creations	2350	\$47.25	\$90.87	<input checked="" type="checkbox"/> <input type="checkbox"/>
S10_1949	1952 Alpine Renault 1300	Classic Cars	1:10	Classic Metal Creations	7305	\$98.58	\$214.30	<input checked="" type="checkbox"/> <input type="checkbox"/>
S24_2887	1952 Citroen 15CV	Classic Cars	1:24	Exoto Designs	1452	\$72.82	\$117.44	<input checked="" type="checkbox"/> <input type="checkbox"/>
S24_3856	1956 Porsche 356A Coupe	Classic Cars	1:18	Classic Metal Creations	6600	\$98.30	\$140.43	<input checked="" type="checkbox"/> <input type="checkbox"/>
S18_4721	1957 Corvette Convertible	Classic Cars	1:18	Classic Metal Creations	1249	\$69.93	\$148.80	<input checked="" type="checkbox"/> <input type="checkbox"/>
S18_4933	1957 Ford Thunderbird	Classic Cars	1:18	Studio M Art Models	3209	\$34.21	\$71.27	<input checked="" type="checkbox"/> <input type="checkbox"/>
S24_2840	1958 Chevy Corvette Limited Edition	Classic Cars	1:24	Carousel DieCast Legends	2542	\$15.91	\$35.36	<input checked="" type="checkbox"/> <input type="checkbox"/>

Overview of ClassicModels

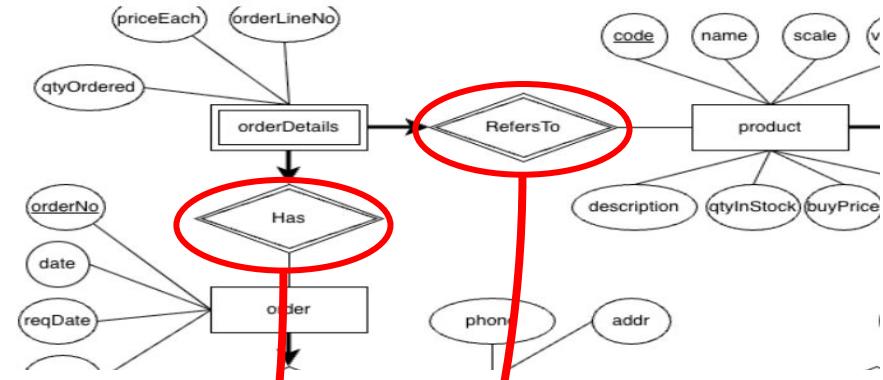
- ClassicModels is a relational database-powered full-stack web application for a retailer of scale models (classic cars, motorcycles, planes).
- The motivation behind its creation was to automate an otherwise poorly managed system. Our application utilises MySQL and Flask to fasten procedures.
- The goals of the project were:
 - To build distinct portals for **Customers**, **Employees**, and **Managers** for security.
 - To create data-driven real-time analytics depending on sales data to make it easier for managers to interpret their data.
 - To enforce business rules at the database level.
 - To remove manual calculation errors.
 - To optimize storage and access.

Full ER Diagram (Chen Notation)



ER-to-Relational Mapping

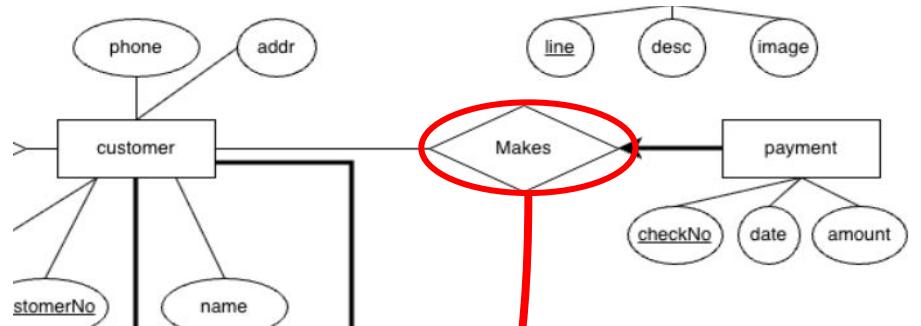
- The orderdetails entity is a **weak entity** that is used to **resolve the many-to-many relationship** between product and order.
- The Has and RefersTo identifying relationships were built into this table using the `orderNumber` and `productCode` FKS respectively.
- When mapping, we used a **surrogate key** (`orderDetailsNumber`) instead of relying on the candidate key alone to simplify operations.
- Integrity constraints were also included to cascade changes in identification data of the order or product in the entry.



```
CREATE TABLE `orderdetail` (
  `orderDetailsNumber` INT          NOT NULL AUTO_INCREMENT,
  `orderNumber`        INT          NOT NULL,
  `productCode`        VARCHAR(15)  CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci,
  `quantityOrdered`   INT          NOT NULL,
  `priceEach`         double       NOT NULL,
  `orderLineNumber`  SMALLINT     NOT NULL,
  PRIMARY KEY (`orderDetailsNumber`),
  UNIQUE KEY `unique_line_item` (`orderNumber`, `productCode`),
  FOREIGN KEY (`orderNumber`) REFERENCES `orders`(`orderNumber`)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  FOREIGN KEY (`productCode`) REFERENCES `products`(`productCode`)
    ON DELETE CASCADE
    ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

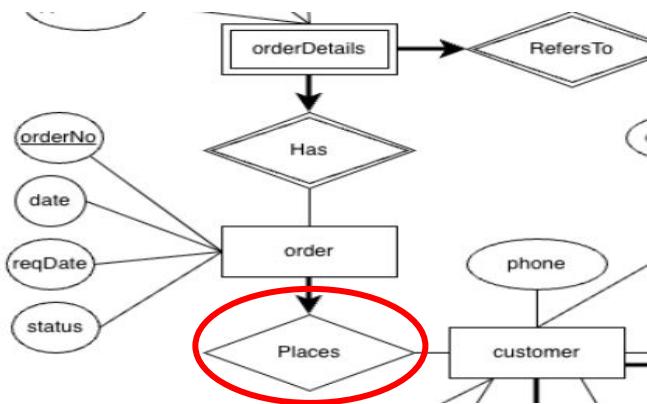
- The **payment** entity and the Makes one-to-many relationship were mapped to the payments table.
- The relationship itself is covered by the **customerNumber** foreign key.
- Referential integrity was enforced using ON DELETE/UPDATE CASCADE.
- Since **checkNumber** is **globally unique**, we marked payments as a **strong entity**.



```
CREATE TABLE payments (
    customerNumber  INT          NOT NULL,
    checkNumber    VARCHAR(50) NOT NULL,
    paymentDate    DATE        NOT NULL,
    amount          DECIMAL(10,2) NOT NULL,
    PRIMARY KEY (checkNumber),
    FOREIGN KEY (customerNumber)
        REFERENCES customers(customerNumber)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

ER-to-Relational Mapping

- The **orders** is a **strong entity** that represents the core purchase transaction in the system.
- Places** is an **one-to-many relationship** between **orders** and **customers**
- orderNumber** serves as the unique identifier derived from the ER model. Foreign Key constraints guarantee that no order exists without a valid customer.
- Referential integrity was enforced using **ON DELETE/UPDATE CASCADE**.
- Nullable attributes like **shippedDate** and **comments** correctly model **optional information** that may not be known at order creation time.

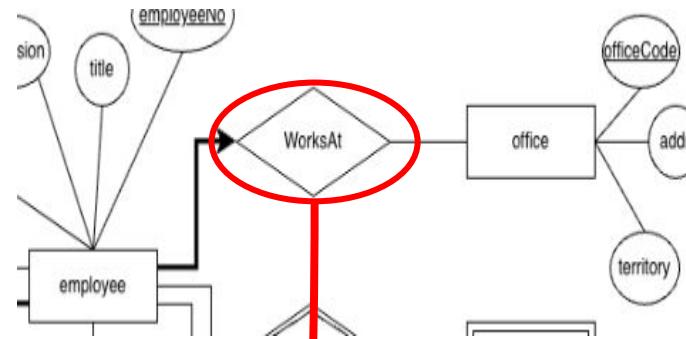


```
CREATE TABLE `orders` (
  `orderNumber` int(11) NOT NULL,
  `orderDate` date NOT NULL,
  `requiredDate` date NOT NULL,
  `shippedDate` date DEFAULT NULL,
  `status` varchar(15) NOT NULL,
  `comments` text DEFAULT NULL,
  `customerNumber` int(11) NOT NULL,
  PRIMARY KEY (`orderNumber`),
  KEY `customerNumber`(`customerNumber`),
  CONSTRAINT `orders_ibfk_1` FOREIGN KEY (`customerNumber`)
    REFERENCES `customers`(`customerNumber`)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
```

ER-to-Relational Mapping

- **offices** is an **independent entity** that represents physical company locations.
- **officeCode** serves as the unique, stable identifier derived from the ER model. Requires no Foreign Keys, ensuring existence without external dependencies.
- **WorksAt** implements the One-to-Many relationship by placing the **officeCode** Foreign Key in the **employees** table

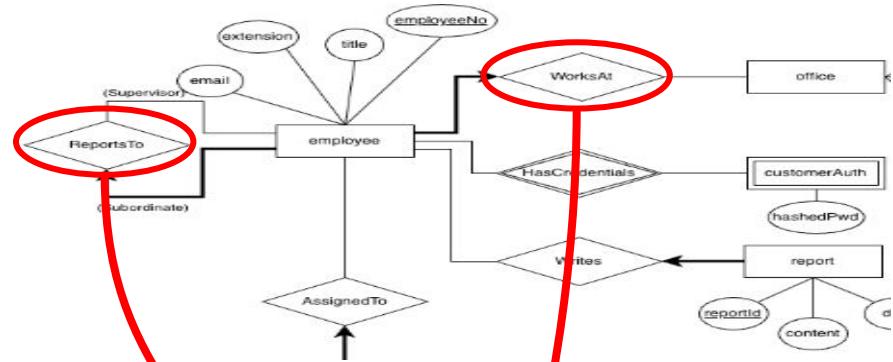
```
CREATE TABLE `offices` (
    `officeCode` varchar(10) NOT NULL,
    `city` varchar(50) NOT NULL,
    `phone` varchar(50) NOT NULL,
    `addressLine1` varchar(50) NOT NULL,
    `addressLine2` varchar(50) DEFAULT NULL,
    `state` varchar(50) DEFAULT NULL,
    `country` varchar(50) NOT NULL,
    `postalCode` varchar(15) NOT NULL,
    `territory` varchar(10) NOT NULL,
    PRIMARY KEY (`officeCode`)
```



```
CREATE TABLE `employees` (
    `employeeNumber` int(11) NOT NULL,
    `lastName` varchar(50) NOT NULL,
    `firstName` varchar(50) NOT NULL,
    `extension` varchar(10) NOT NULL,
    `email` varchar(100) NOT NULL,
    `officeCode` varchar(10) NOT NULL,
    `reportsTo` int(11) DEFAULT NULL,
    `jobTitle` varchar(50) NOT NULL,
    PRIMARY KEY (`employeeNumber`),
    KEY `reportsTo` (`reportsTo`),
    CONSTRAINT `fk_employees_reports_to`
        FOREIGN KEY (`reportsTo`) REFERENCES `employees` (`employeeNumber`),
    CONSTRAINT `fk_employees_office`
        FOREIGN KEY (`officeCode`) REFERENCES `offices` (`officeCode`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

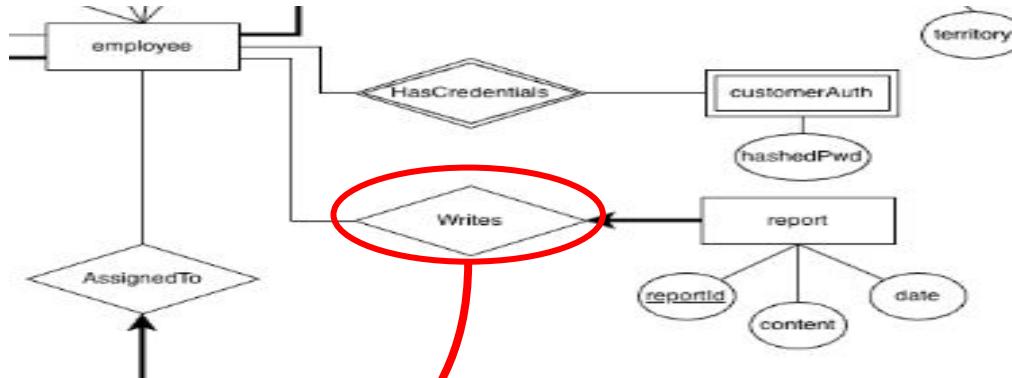
- The WorksAt one-to-many relationship were mapped to the offices table.
- The relationship itself is covered by the `officeCode` foreign key.
- Referential integrity was enforced using ON DELETE SET NULL and ON UPDATE CASCADE.
- The ReportsTo one-to-many relationship were mapped to employees table itself. It forms a self referencing relationship.
- Referential integrity was enforced using ON UPDATE CASCADE.



```
CREATE TABLE `employees` (
  `employeeNumber` int(11) NOT NULL,
  `lastName` varchar(50) NOT NULL,
  `firstName` varchar(50) NOT NULL,
  `extension` varchar(10) NOT NULL,
  `email` varchar(100) NOT NULL,
  `officeCode` varchar(10) NOT NULL,
  `reportsTo` int(11) DEFAULT NULL,
  `jobTitle` varchar(50) NOT NULL,
  PRIMARY KEY (`employeeNumber`),
  KEY `reportsTo` (`reportsTo`),
  CONSTRAINT `fk_employee_reports_to`
    FOREIGN KEY (`reportsTo`) REFERENCES `employees` (`employeeNumber`)
    ON DELETE SET NULL
    ON UPDATE CASCADE,
  CONSTRAINT `fk_employee_office`
    FOREIGN KEY (`officeCode`) REFERENCES `offices` (`officeCode`)
    ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

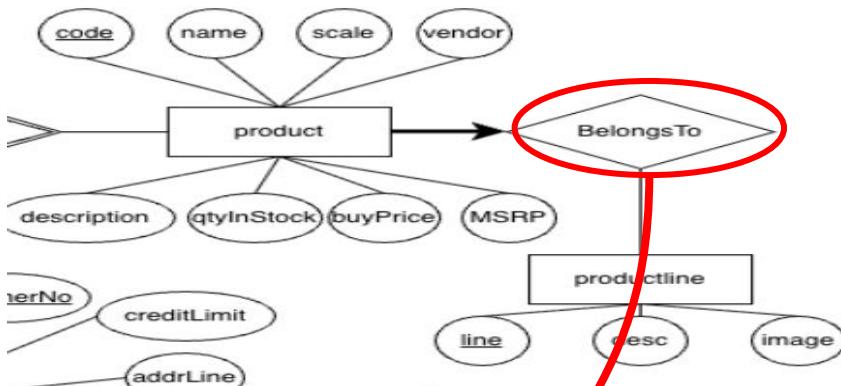
- The Writes one-to-many relationship were mapped to the employees table.
- The relationship itself is covered by the employeeNumber foreign key.
- Referential integrity was enforced using ON DELETE/UPDATE CASCADE .



```
CREATE TABLE IF NOT EXISTS `employee_reports` (
    `reportId` int(11) NOT NULL AUTO_INCREMENT,
    `employeeNumber` int(11) NOT NULL,
    `reportContent` text NOT NULL,
    `reportDate` datetime DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (`reportId`),
    KEY `employeeNumber`(`employeeNumber`),
    FOREIGN KEY (`employeeNumber`) REFERENCES `employees` (`employeeNumber`)
    ON DELETE CASCADE
    ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

- The BelongsTo **one-to-many** relationship was integrated into the employees table via the productLine FK.
- productlines** is a **strong entity** that represents the types of products the retailer sells.



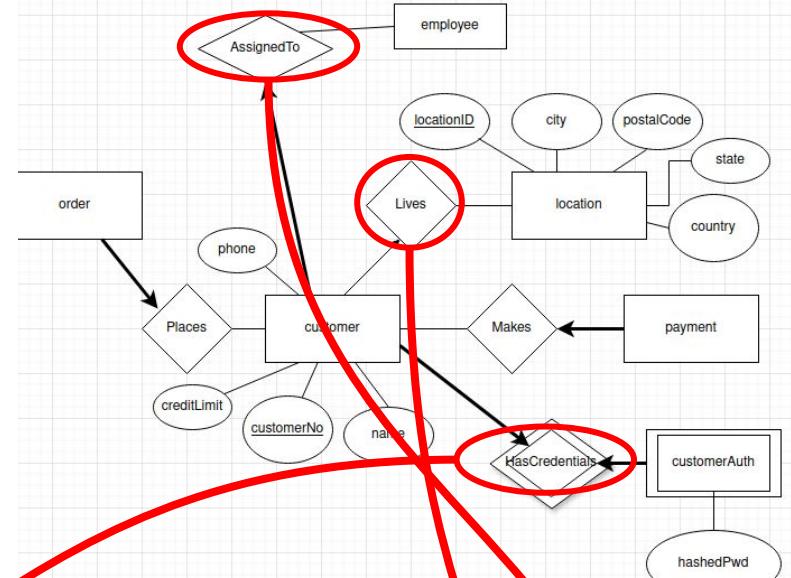
```
CREATE TABLE `productlines` (
  `productLine` varchar(50) NOT NULL,
  `textDescription` varchar(4000) DEFAULT NULL,
  `htmlDescription` mediumtext DEFAULT NULL,
  `image` mediumblob DEFAULT NULL,
  PRIMARY KEY (`productLine`)
);
```

```
CREATE TABLE `products` (
  `productCode` varchar(15) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
  `productName` varchar(70) NOT NULL,
  `productLine` varchar(50) NOT NULL,
  `productScale` varchar(10) NOT NULL,
  `productVendor` varchar(50) NOT NULL,
  `productDescription` text NOT NULL,
  `quantityInStock` smallint(6) NOT NULL,
  `buyPrice` double NOT NULL,
  `MSRP` double NOT NULL,
  PRIMARY KEY (`productCode`),
  KEY `productLine` (`productLine`),
  CONSTRAINT `products_ibfk_1` FOREIGN KEY (`productLine`) REFERENCES `productlines` (`productLine`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

ER-to-Relational Mapping

- The **Lives** one-to-many relationship was mapped by placing the primary key of **Location** as a foreign key in the **Customers** table.
- This enforces referential integrity while allowing multiple customers to share the same location, resulting in a normalized design

```
CREATE TABLE `customer_auth` [
    `customerNumber` int NOT NULL,
    `hashedPassword` varchar(255) NOT NULL,
    PRIMARY KEY (`customerNumber`),
    CONSTRAINT `fk_customer_auth_num` FOREIGN KEY (`customerNumber`)
] ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
/*!40101 SET character_set_client = @saved_cs_client */;
```



```
-- Create the Normalized Customers Table
CREATE TABLE `customers` [
    `customerNumber` int(11) NOT NULL,
    `customerName` varchar(50) NOT NULL,
    `contactLastName` varchar(50) NOT NULL,
    `contactFirstName` varchar(50) NOT NULL,
    `phone` varchar(50) NOT NULL,
    `addressLine1` varchar(50) NOT NULL,
    `addressLine2` varchar(50) DEFAULT NULL,
    `locationID` int(11) DEFAULT NULL,
    `salesRepEmployeeNumber` int(11) DEFAULT NULL,
    `creditLimit` double DEFAULT NULL,
    PRIMARY KEY (`customerNumber`),
    KEY `salesRepEmployeeNumber` (`salesRepEmployeeNumber`),
    KEY `fk_customer_location` (`locationID`),
    CONSTRAINT `customers_ibfk_1` FOREIGN KEY (`salesRepEmployeeNumber`)
        REFERENCES `employees` (`employeeNumber`) ON DELETE SET NULL,
    CONSTRAINT `customers_ibfk_2` FOREIGN KEY (`locationID`)
        REFERENCES `locations` (`locationID`)
```

Normalization

- The orderdetails table is already in BCNF, because:
 - ✓ **1NF:**
 - All columns contain atomic values.
 - All rows are unique.
 - There are no repeating groups.
 - ✓ **2NF:**
 - There are no partial dependencies, this is because it has only one primary key.
 - Even if we considered the candidate key (`orderNumber, productCode`), it is also free of partial dependencies.
 - The table is in 1NF.
 - ✓ **3NF:**
 - There are no transitive dependencies (non-key to non-key FDs).
 - The table is in 2NF.
 - ✓ **BCNF:**
 - Every determinant is a candidate key.
 - The table is in 3NF.

```
CREATE TABLE `orderdetails` (
    `orderDetailsNumber` INT          NOT NULL AUTO_INCREMENT,
    `orderNumber`        INT          NOT NULL,
    `productCode`        VARCHAR(15) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci,
    `quantityOrdered`   INT          NOT NULL,
    `priceEach`         double       NOT NULL,
    `orderLineNumber`  SMALLINT     NOT NULL,
    PRIMARY KEY (`orderDetailsNumber`),
    UNIQUE KEY `unique_line_item` (`orderNumber`, `productCode`),
    FOREIGN KEY (`orderNumber`) REFERENCES `orders`(`orderNumber`)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (`productCode`) REFERENCES `products`(`productCode`)
        ON DELETE CASCADE
        ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

Normalization

- The initial payments table's PK was (`customerNumber`, `checkNumber`). This made it 1NF.

✓ 1NF:

- All columns contain atomic values.
- All rows are unique.
- There are no repeating groups.

✗ 2NF:

- The table is in 1NF.
- checkNumber is already unique on its own, creating partial dependencies.**
- The Fix: checkNumber as the PK.**

- This approach was chosen because `checkNumber` is unique for each payment.

✓ 3NF:

- There are no transitive dependencies. (non-key to non-key FDs).
- The table is in 2NF.

✓ BCNF:

- Every determinant is a candidate key.
- The table is in 3NF.

```
-- Unnormalized version
-- CREATE TABLE payments (
--   customerNumber INT NOT NULL,
--   checkNumber VARCHAR(50) NOT NULL,
--   paymentDate DATE NOT NULL,
--   amount DECIMAL(10,2) NOT NULL,
--   PRIMARY KEY (customerNumber, checkNumber),
--   FOREIGN KEY (customerNumber) REFERENCES customers(customerNumber)
--     ON DELETE CASCADE
--     ON UPDATE CASCADE
-- );
-- Normalized
CREATE TABLE payments (
  customerNumber INT          NOT NULL,
  checkNumber   VARCHAR(50)    NOT NULL,
  paymentDate   DATE          NOT NULL,
  amount        DECIMAL(10,2)  NOT NULL,
  PRIMARY KEY (checkNumber),
  FOREIGN KEY (customerNumber)
    REFERENCES customers(customerNumber)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

Normalization

- The orders table is already in BCNF, because:

✓ **1NF:**

- All columns contain **atomic values**.
- All rows are unique.
- There are no repeating groups.

✓ **2NF:**

- The table is in 1NF.
- There are **no partial dependencies**, because the table has a single-attribute primary key (`orderNumber`).

✓ **3NF:**

- The table is in 2NF.
- There are no transitive dependencies (non-key to non-key FDs).

✓ **BCNF:**

- Every determinant is a candidate key.
- The table is in 3NF.

```
--Normalized (BCNF)
CREATE TABLE `orders` (
    `orderNumber` int(11) NOT NULL,
    `orderDate` date NOT NULL,
    `requiredDate` date NOT NULL,
    `shippedDate` date DEFAULT NULL,
    `status` varchar(15) NOT NULL,
    `comments` text DEFAULT NULL,
    `customerNumber` int(11) NOT NULL,
    PRIMARY KEY (`orderNumber`),
    KEY `customerNumber` (`customerNumber`),
    CONSTRAINT `orders_ibfk_1`
        FOREIGN KEY (`customerNumber`)
            REFERENCES `customers` (`customerNumber`)
                ON DELETE RESTRICT
                ON UPDATE CASCADE
);
```

Normalization

- The offices table is already in BCNF, because:

✓ 1NF:

- All columns (`city`, `state`, `country` etc.) contain **atomic values**.
- All rows are unique.
- There are no repeating groups.

✓ 2NF:

- The table is in 1NF.
- There are **no partial dependencies**, because the table has a single-attribute primary key (`officeCode`).

✓ 3NF:

- The table is in 2NF.
- There are **no transitive dependencies**. All non-key attributes (phone, territory, location details) rely *only* on the `officeCode`.

✓ BCNF:

- Every determinant is a candidate key.
- The table is in 3NF.

```
-- Normalized (BCNF)
CREATE TABLE offices (
    officeCode      VARCHAR(10)      NOT NULL,
    city            VARCHAR(50)       NOT NULL,
    phone           VARCHAR(50)       NOT NULL,
    addressLine1   VARCHAR(50)       NOT NULL,
    addressLine2   VARCHAR(50)       DEFAULT NULL,
    'state'         VARCHAR(50)       DEFAULT NULL,
    country         VARCHAR(50)       NOT NULL,
    postalCode     VARCHAR(15)       NOT NULL,
    territory       VARCHAR(10)       NOT NULL,
    PRIMARY KEY (officeCode)
);
```

Normalization

- The employees table is already in BCNF, because:

✓ **1NF:**

- All columns contain atomic values.
- All rows are unique.
- There are no repeating groups.

✓ **2NF:**

- There are no partial dependencies, this is because it has only one primary key.
- The table is in 1NF.

✓ **3NF:**

- There are no transitive dependencies (non-key to non-key FDs).
- The table is in 2NF.

✓ **BCNF:**

- Every determinant is a candidate key.
- The table is in 3NF.

```
CREATE TABLE `employees` (
  `employeeNumber` int(11) NOT NULL,
  `lastName` varchar(50) NOT NULL,
  `firstName` varchar(50) NOT NULL,
  `extension` varchar(10) NOT NULL,
  `email` varchar(100) NOT NULL,
  `officeCode` varchar(10) NOT NULL,
  `reportsTo` int(11) DEFAULT NULL,
  `jobTitle` varchar(50) NOT NULL,
  PRIMARY KEY (`employeeNumber`),
  KEY `reportsTo` (`reportsTo`),
  CONSTRAINT `fk_employees_reports_to`
    FOREIGN KEY (`reportsTo`) REFERENCES `employees` (`employeeNumber`)
    ON DELETE SET NULL
    ON UPDATE CASCADE,
  CONSTRAINT `fk_employees_office`
    FOREIGN KEY (`officeCode`) REFERENCES `offices` (`officeCode`)
    ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

Normalization

- The products table is already in BCNF, because:

✓ 1NF:

- All columns contain atomic values.
- All rows are unique.
- There are no repeating groups.

✓ 2NF:

- There are no partial dependencies, because the table has a single-attribute primary key (`productCode`).
- All non-key attributes are fully functionally dependent on `productCode`.
- The table is in 1NF.

✓ 3NF:

- There are no transitive dependencies (non-key to non-key FDs).
- Descriptive attributes depend directly on `productCode`.
- The table is in 2NF.

✓ BCNF:

- Every determinant is a candidate key.
- The table is in 3NF.

```
5 CREATE TABLE `products` (
6   `productCode` varchar(15) CHARACTER SET
7     utf8mb4 COLLATE utf8mb4_general_ci NOT NULL,
8   `productName` varchar(70) NOT NULL,
9   `productLine` varchar(50) NOT NULL,
10  `productScale` varchar(10) NOT NULL,
11  `productVendor` varchar(50) NOT NULL,
12  `productDescription` text NOT NULL,
13  `quantityInStock` smallint(6) NOT NULL,
14  `buyPrice` double NOT NULL,
15  `MSRP` double NOT NULL,
16  PRIMARY KEY (`productCode`),
17  KEY `productLine` (`productLine`),
18  CONSTRAINT `products_ibfk_1` FOREIGN KEY (`productLine`)
19    REFERENCES `productlines` (`productLine`)
20 ) ENGINE=InnoDB DEFAULT
21 CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

Normalization

- The productlines table is already in BCNF, because:

✓ 1NF:

- All columns contain atomic values.
- All rows are unique.
- There are no repeating groups.

✓ 2NF:

- There are no partial dependencies, because the table has a single-attribute primary key (`productLine`).
- All non-key attributes are fully functionally dependent on `productLine`.
- The table is in 1NF.

✓ 3NF:

- There are no transitive dependencies (non-key to non-key FDs).
- Descriptive attributes depend directly on `productLine`.
- The table is in 2NF.

✓ BCNF:

- Every determinant is a candidate key.
- The table is in 3NF.

```
CREATE TABLE `productlines` (
  `productLine` varchar(50) NOT NULL,
  `textDescription` varchar(4000) DEFAULT NULL,
  `htmlDescription` mediumtext DEFAULT NULL,
  `image` mediumblob DEFAULT NULL,
  PRIMARY KEY (`productLine`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_general_ci;
```

Normalization

✓ 1NF:

- All columns contain atomic values (e.g., one phone number per cell).
- All rows are unique via `customerNumber`.
- There are no repeating groups of data.

✓ 2NF:

- The table is in 1NF.
- Since there is a single-column Primary Key (`customerNumber`), there are **no partial functional dependencies**. Every non-key attribute depends on the whole key.

✓ 3NF:

- The table is in 2NF.
- All non-key attributes (e.g., `customerName`, `phone`) are now functionally dependent **only** on the Primary Key (`customerNumber`) and nothing else.
- The geographic attributes live in a `locations` table, referenced by `locationID`.

```
CREATE TABLE `locations` (
  `locationID` int(11) NOT NULL AUTO_INCREMENT,
  `city` varchar(50) NOT NULL,
  `state` varchar(50) DEFAULT NULL,
  `postalCode` varchar(15) DEFAULT NULL,
  `country` varchar(50) NOT NULL,
  PRIMARY KEY (`locationID`),
  UNIQUE KEY `unique_geo` (`city`, `state`, `postalCode`, `country`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

-- Create the Normalized Customers Table
CREATE TABLE `customers` (
  `customerNumber` int(11) NOT NULL,
  `customerName` varchar(50) NOT NULL,
  `contactLastName` varchar(50) NOT NULL,
  `contactFirstName` varchar(50) NOT NULL,
  `phone` varchar(50) NOT NULL,
  `addressLine1` varchar(50) NOT NULL,
  `addressLine2` varchar(50) DEFAULT NULL,
  `locationID` int(11) DEFAULT NULL,
  `salesRepEmployeeNumber` int(11) DEFAULT NULL,
  `creditLimit` double DEFAULT NULL,
  PRIMARY KEY (`customerNumber`),
  KEY `salesRepEmployeeNumber` (`salesRepEmployeeNumber`),
  KEY `fk_customer_location` (`locationID`),
  CONSTRAINT `customers_ibfk_1` FOREIGN KEY (`salesRepEmployeeNumber`)
    REFERENCES `employees` (`employeeNumber`) ON DELETE SET NULL,
  CONSTRAINT `customers_ibfk_2` FOREIGN KEY (`locationID`)
    REFERENCES `locations` (`locationID`)
```

Dataset Description

- ❖ **What does the data include?**
 - The dataset is a **synthetic dataset** used to simulate business data for a **retailer of scale models** of cars, ships, etc.
 - **Typical relational tables:** customers, products, product lines, orders, order details, payments, employees, and offices which describe sales operations, products, and organizational structure.
- ❖ **What is the source?**
 - The dataset comes from the ClassicModels sample database, originally provided by MySQLTutorial. It can be downloaded directly from places like the MySQL sample DB documentation.
- ❖ **How large is it?**
 - The entire database is about 500 KB in size.
 - It consists of **8 tables** with a total of approximately **3,864 rows and about 59 columns** across the schema.
- ❖ **Did you perform any parsing or scraping?**
 - No, we did not perform any **automated** parsing or scraping. **All images used were manually sourced** from copyright-free repositories and **included as static assets**.
- ❖ **Did you use synthetic data?**
 - The ClassicModels dataset is an educational dataset, and all of its data is synthetic.
 - However, we did not inject any extra data into the database other than testing its functionality.

Features

Our Global Offices

The screenshot displays a grid of eight cards, each representing a global office. Each card includes the office name, location, employee count, address, phone number, and edit/delete buttons.

- Sydney**, Australia (Employees: 4)
5-11 Wentworth Avenue
Floor #2
NSW 2010
+61 2 9264 2451
- Paris**, France (Employees: 5)
43 Rue Jouffroy D'abbans
75017
+33 14 723 4404
- Tokyo**, Japan (Employees: 2)
4-1 Kioicho
102-8578
+81 33 224 5000
- London**, UK (Employees: 2)
25 Old Broad Street
Level 7
EC2N 1HN
+44 20 7877 2041
- Boston**, USA (Employees: 2)
1550 Court Place
Suite 102
02107
+1 215 837 0825
- NYC**, USA (Employees: 2)
523 East 53rd Street
apt. 5A
10022
+1 212 555 3000
- San Francisco**, USA (Employees: 6)
100 Market Street
Suite 300
94080
+1 650 219 4782

Action Denied! You have an outstanding balance of \$9570.00. Please pay first.

Account Settings

The screenshot shows the 'Profile Information' section of the account settings. It includes fields for Full Name, Account Role, Customer Number, Phone Number, and Billing Address.

Profile Information	
Full Name	Palle Ibsen
Customer Number	#227
Phone Number	86 21 3555
Account Role	Customer
Billing Address	Snagsløget 45 Århus, Denmark

Financial Summary

Join ClassicModels

Create your account to start collecting

First Name

John

Last Name

Doe

Company / Customer Name

e.g. Acme Corp.

Phone Number

+1 555 123 4567

Address Line 1

Street address, P.O. box

City

Country

Create Password

Min 8 characters

Create Account

Already have an account? [Login here](#)

Features

All Offices	All Categories	Revenue	Global Category Avg.	Performance
Boston	Classic Cars	\$5,968.81	\$18,439.82	▼ Below Avg
Boston	Vintage Cars	\$2,879.04	\$9,632.62	▼ Below Avg
Boston	Motorcycles	\$8,533.76	\$14,195.27	▲ Above Avg
Boston	Trucks and Buses	6	\$103,893.36	\$13,654.85
Boston	Ships	11	\$76,492.78	\$9,764.68
Boston	Planes	8	\$67,814.36	\$14,466.21
Boston	Trains	4	\$17,046.51	\$4,011.34
London	Classic Cars	34	\$644,626.76	\$18,439.82
London	Vintage Cars	24	\$219,077.88	\$9,632.62
London	Motorcycles	10	\$141,552.32	\$14,195.27

Previous [1](#) [2](#) [3](#) [4](#) [5](#) Next

Classic Cars

Sort by

1952 Alpine Renault 1300	1972 Alfa Romeo GTA	1962 Lancia Delta 16V
Scale: 1:10	Scale: 1:10	Scale: 1:10
Buy Price: \$98.58	Buy Price: \$85.68	Buy Price: \$103.42
MSRP: \$214.3	MSRP: \$138.0	MSRP: \$147.74
Stock: 7305 pieces	Stock: 3252 pieces	Stock: 6791 pieces

--	--	--

Checkout

Review your items and complete your order.

Order Summary

PRODUCT	UNIT PRICE	QTY	SUBTOTAL
1950's Chicago Surface Lines Streetcar	\$62.14	1	\$62.14

Order Notes

Special order

Payment Details

Subtotal	\$62.14
Shipping	Free
Total	\$62.14

Payment will be collected after order confirmation.

[Place Order →](#)

[Return to Cart](#)

Order #10426 placed successfully!

Order #10426

In Process

Order Items

PRODUCT	QTY	PRICE	TOTAL
1950's Chicago Surface Lines Streetcar	1	\$62.14	\$62.14

Summary

Subtotal (1 items)	\$62.14
Shipping	Free
Grand Total	\$62.14

Comments & Notes

Special order

[Cancel Order](#)

[Back to My Orders](#)

Features

Employee Dashboard

Welcome, Diane! (Employee #1002)

All Customers (Management View)

Select a customer to manage orders and payments.

		Click to Sort	Apply		
Atelier graphique	Nantes, France	Rep #1020	ID: 103 No Debt Spent: \$22314.36		
Signal Gift Stores	Las Vegas, USA	Rep #1016	ID: 112 No Debt Spent: \$40100.98		
Australian Collectors, Co.	Melbourne, Australia	Rep #1011	ID: 114 No Debt Spent: \$100565.07		
La Rochelle Gifts	Nantes, France	Rep #1070	ID: 119 No Debt Spent: \$16949.68		
Baane Mini Imports	Stavanger, Norway	Rep #1050	ID: 121 No Debt Spent: \$104224.79		
Mini Gifts Distributors Ltd.	San Rafael, USA	Rep #1055	ID: 124 No Debt Spent: \$564198.24		
Total Orders: 89,909.80		Total Payments: \$89,909.80			
Balance: No debt					
Last Update: 2023-09-01					

Account Settings

Profile Information

Full Name	Diane Murphy	Account Role	Employee
Employee ID	#1002	Job Title	President
Email	dmurphy@classicmodelcars.com	Office Code	1

Security

Password
Update your password regularly to keep your account secure.

Change Password

My Profile

Welcome, Palle Ibsen!

Company: Heintze Collectables

Address:
Smagsloget 45
Århus, Nørre 8200
Denmark

Phone:
86 21 3555

Credit Limit:
\$120,800.00

Financial Summary

Total Orders: \$89,909.80

Total Payments: \$89,909.80

Balance:

No debt

Payment History

DATE	TRANSACTION ID	AMOUNT
2004-11-02	NU21326	\$53,745.34

Employee Dashboard

Welcome, Diane! (Employee #1002)

All Customers (Management View)

Select a customer to manage orders and payments.

		Click to Sort	Apply		
Technics Stores Inc.	Burlingame, USA	Rep #1015	ID: 161 No Debt Spent: \$104545.22		
Handji Gifts & Co	Singapore, Singapore	Rep #1012	ID: 166 Debt: \$2326.18 Spent: \$105420.57		
Herkuu Gifts	Bergen, Norway	Rep #1024	ID: 167 No Debt Spent: \$9752.47		
American Souvenirs Inc	New Haven, USA	Rep #1286	ID: 168 No Debt Spent: \$0.00		
Porto Imports Co.	Lisboa, Portugal		ID: 169 No Debt Spent: \$0.00		
Daedalus Designs Imports	Lille, France	Rep #1370	ID: 171 No Debt Spent: \$61781.70		
Total Orders: 89,909.80		Total Payments: \$89,909.80			
Balance: No debt					
Last Update: 2023-09-01					

Complex Queries

- This complex query is a data analysis function that **joins Offices, Employees, Customers, Orders, OrderDetails** to link office data to individual sales transactions.
- The subquery dynamically calculates the average revenue of other employees in the same office for every row, allowing for a fair, location-based performance comparison. It aggregates revenue for all peers in that office first, then calculates the AVG of those totals.
- The inner GROUP BY statement helps to calculate the average revenue by aggregating individual employee revenue. The outer GROUP BY calculates the total for that individual sales representative.
- It also uses COALESCE to handle NULL values (employees with no sales appear as \$0.00).

```
def get_sales_rep_vs_office_average(self):  
    query = """  
        SELECT  
            o.city AS office_city,  
            e.employeeNumber,  
            CONCAT(e.firstName, ' ', e.lastName) AS sales_rep,  
            COALESCE(SUM(od.quantityOrdered * od.priceEach), 0) AS rep_revenue,  
  
        (  
            SELECT AVG(rep_total)  
            FROM (  
                SELECT  
                    COALESCE(SUM(od2.quantityOrdered * od2.priceEach), 0) AS rep_total  
                FROM employees e2  
                LEFT JOIN customers c2  
                    ON e2.employeeNumber = c2.salesRepEmployeeNumber  
                LEFT JOIN orders o2  
                    ON c2.customerNumber = o2.customerNumber  
                LEFT JOIN orderdetails od2  
                    ON o2.orderNumber = od2.orderNumber  
                WHERE e2.officeCode = e.officeCode  
                GROUP BY e2.employeeNumber  
            ) office_rep_totals  
        ) AS office_avg_revenue  
  
        FROM offices o  
        JOIN employees e  
            ON o.officeCode = e.officeCode  
        LEFT JOIN customers c  
            ON e.employeeNumber = c.salesRepEmployeeNumber  
        LEFT JOIN orders ord  
            ON c.customerNumber = ord.customerNumber  
        LEFT JOIN orderdetails od  
            ON ord.orderNumber = od.orderNumber  
  
        GROUP BY o.city, e.employeeNumber, sales_rep  
        ORDER BY office_city, rep_revenue DESC;  
    """
```

Complex Queries

Sales Rep Performance vs Office Average

OFFICE	SALES REP	REP REVENUE	OFFICE AVG	Δ
Boston	Steve Patterson	\$505875.42	\$446269.31	+\$59606.11
Boston	Julie Firrelli	\$386663.20	\$446269.31	-\$59606.11
London	Larry Bott	\$732096.79	\$718475.35	+\$13621.44
London	Barry Jones	\$704853.91	\$718475.35	-\$13621.44
NYC	George Vanauf	\$669377.05	\$578794.86	+\$90582.19
NYC	Foon Yue Tseng	\$488212.67	\$578794.86	-\$90582.19
Paris	Gerard Hernandez	\$1258577.81	\$616752.32	+\$641825.49
Paris	Pamela Castillo	\$868220.55	\$616752.32	+\$251468.23
Paris	Loui Bondur	\$569485.75	\$616752.32	-\$47266.57
Paris	Martin Gerard	\$387477.47	\$616752.32	-\$229274.85

← Previous

Page 1

Next →

Complex Queries

- This complex query examines the offices in a comparative manner in each product type. It **JOINS Offices, Employees, Customers, Orders, OrderDetails , products , productLines**.
- It utilizes a **correlated nested subquery** to calculate global average. For each record, the subquery calculates the *worldwide* average revenue for that category, allowing for performance comparison.
- It relies on **Dynamic SQL generation** to apply user-defined filters and supports **Pagination** to manage large result sets efficiently.
- It uses **COALESCE** functions to handle **NULL** values, ensuring accurate financial calculations even for product lines that haven't generated sales in specific regions yet.

```
def get_ultimate_analysis_paginated(self, limit=10, offset=0, filter_office=None, filter_category=None):
    """
    THE ULTIMATE QUERY (PAGINATED & FILTERED):
    Analyzes Office performance per Product Category vs Global Averages.
    Includes 7-Table Joins, Nested Subqueries, and Dynamic Filtering.
    """

    # Dynamic WHERE clause construction
    where_clauses = ["pl.productLine IS NOT NULL"]
    params = []

    if filter_office and filter_office != 'All':
        where_clauses.append("o.city = %s")
        params.append(filter_office)

    if filter_category and filter_category != 'All':
        where_clauses.append("pl.productline = %s")
        params.append(filter_category)

    where_sql = " AND ".join(where_clauses)

    query = f"""
    SELECT
        o.city AS Office,
        o.territory AS Region,
        pl.productline AS Category,

        COUNT(DISTINCT ord.orderNumber) AS Order_Count,
        COALESCE(SUM(od.quantityOrdered * od.priceEach), 0) AS Total_Revenue,

        -- COMPLEX NESTED SUBQUERY: Global Average Calculation
        (
            SELECT AVG(sub_sum)
            FROM (
                SELECT SUM(od2.quantityOrdered * od2.priceEach) as sub_sum
                FROM orderdetails od2
                JOIN products p2 ON od2.productCode = p2.productCode
                WHERE p2.productline = pl.productLine -- Correlated Reference
                GROUP BY od2.orderNumber
            ) AS global_stats
        ) AS Global_Category_Avg

    FROM offices o
    JOIN employees e
        ON o.officeCode = e.officeCode
    LEFT JOIN customers c
        ON e.employeeNumber = c.salesRepEmployeeNumber
    LEFT JOIN orders ord
        ON c.customerNumber = ord.customerNumber
    LEFT JOIN orderdetails od
        ON ord.orderNumber = od.orderNumber
    LEFT JOIN products p
        ON od.productCode = p.productCode
    LEFT JOIN productlines pl
        ON p.productLine = pl.productLine

    WHERE {where_sql}

    GROUP BY o.officeCode, pl.productLine
    ORDER BY o.city, Total_Revenue DESC
    LIMIT %s OFFSET %s
    """

```

Complex Queries

All Offices		All Categories				
Office Details	Product Category	Order Count	Total Revenue	Global Category Avg.	Performance	
Boston	Classic Cars	19	\$335,968.81	\$18,439.82	▼ Below Avg	
Boston	Vintage Cars	19	\$172,879.04	\$9,612.62	▼ Below Avg	
Boston	Motorcycles	6	\$118,533.76	\$14,195.27	▲ Above Avg	
Boston	Trucks and Buses	6	\$103,893.36	\$13,654.85	▲ Above Avg	
Boston	Ships	11	\$76,402.78	\$9,764.68	▼ Below Avg	
Boston	Planes	8	\$67,814.36	\$14,464.21	▼ Below Avg	
Boston	Trains	4	\$17,046.51	\$4,011.34	▲ Above Avg	
London	Classic Cars	34	\$644,626.76	\$18,439.82	▲ Above Avg	
London	Vintage Cars	24	\$219,077.88	\$9,612.62	▼ Below Avg	
London	Motorcycles	10	\$141,552.32	\$14,195.27	▼ Below Avg	
Previous		1	2	3	4	5
Next						

Complex Queries

- This query generates a comprehensive performance report by merging workforce data with sales metrics in offices manner.
- It joins **5 tables** (`Offices`, `Employees`, `Customers`, `Orders`, `OrderDetails`) using `LEFT JOINS` with preserving data integrity.
- It uses `COUNT (DISTINCT column)` to accurately count employees and customers. This is crucial to prevent the "duplicate counting caused by the multi-table joins."
- A **Scalar Subquery** is used in the `SELECT` clause to dynamically fetch the specific "Manager" or "VP" for each office without complicating the main `GROUP BY` logic.
- It implements a conditional **CASE statement** to calculate the 'Average Order Value', safely handling potential division-by-zero errors for new offices

```
def get ConsolidatedOfficeStats(self):  
    """  
    MERGED & COMPLEX QUERY:  
    Combines Activity Report + Order Stats into one master query.  
    Features: Multi-Join (5 Tables), Left Joins, Count Distinct, Nested Subquery for Manager.  
    """  
  
    query = """  
        SELECT  
            o.officeCode,  
            o.city,  
            o.country,  
            o.territory,  
            COUNT(DISTINCT e.employeeNumber) as active_employees,  
            COUNT(DISTINCT c.customerNumber) as customer_count,  
            COUNT(DISTINCT ord.orderNumber) as total_orders,  
            COALESCE(SUM(od.quantityOrdered * od.priceEach), 0) as total_revenue,  
  
            COALESCE(  
                (SELECT CONCAT(m.firstName, ' ', m.lastName)  
                 FROM employees m  
                 WHERE m.officeCode = o.officeCode  
                 AND (m.jobTitle LIKE '%Manager%' OR m.jobTitle LIKE '%VP%' OR m.jobTitle LIKE '%President%')  
                 LIMIT 1  
                ), 'Regional Lead'  
            ) as manager_name,  
  
            CASE  
                WHEN COUNT(DISTINCT ord.orderNumber) > 0  
                THEN CAST(COALESCE(SUM(od.quantityOrdered * od.priceEach), 0) AS DECIMAL(10,2)) / COUNT(DISTINCT ord.orderNumber)  
                ELSE 0.00  
            END as avg_ticket_size  
  
        FROM offices o  
        LEFT JOIN employees e  
            ON o.officeCode = e.officeCode  
        LEFT JOIN customers c  
            ON e.employeeNumber = c.salesRepEmployeeNumber  
        LEFT JOIN orders ord  
            ON c.customerNumber = ord.customerNumber  
        LEFT JOIN orderdetails od  
            ON ord.orderNumber = od.orderNumber  
  
        GROUP BY o.officeCode, o.city, o.country, o.territory  
        ORDER BY total_revenue DESC;  
    """
```

Complex Queries

Global Office Performance						
	LOCATION	MANAGER	STAFF / CLIENTS	TOTAL ORDERS	REVENUE	AVG. ORDER VALUE
4	Paris	Gerard Bondur	25 / 29	105	\$3,036,696	\$28,921
7	London	Regional Lead	22 / 17	47	\$1,436,951	\$30,573
1	San Francisco	Diane Murphy	26 / 12	48	\$1,429,064	\$29,772
3	NYC	Regional Lead	22 / 15	39	\$1,157,590	\$29,682
6	Sydney	William Patterson	24 / 10	38	\$1,147,176	\$30,189
2	Boston	Regional Lead	22 / 12	36	\$894,026	\$24,834
5	Tokyo	Regional Lead	22 / 5	16	\$457,110	\$28,569
9	Istanbul	Regional Lead	20 / 0	0	\$0	\$0



Complex Queries

- The "Feel Lucky" query is a complex analytical tool designed to provide a personalized, randomized recommendation for customers. It traverses seven different tables to find a connection between a customer's history, their assigned support staff, and a random product recommendation.
- The query prioritizes the city of the **Sales Representative** assigned to the customer (`o.city`). If a customer is not assigned a representative, the system dynamically falls back to a **completely random office city** (`off.city`). This ensures the user always receives a travel suggestion.
- To generate a valid recommendation, the query executes a **7-table join**:
 - Customer → Employee:** Identifies the Sales Rep.
 - Employee → Office:** Finds the Rep's home base.
 - Customer → Order:** Looks at buying history.
 - Order → OrderDetails → Product:** Finds the specific items purchased.

```
def feel_lucky(self):
    query = """
        SELECT
            COALESCE(o.city, off.city) AS city_to_visit,
            p.productName AS product_to_buy,
            p.productCode,
            CAST(FLOOR(RAND() * 100) AS SIGNED) + 1 AS lucky_number
        FROM customers c

        LEFT JOIN employees e
        ON c.salesRepEmployeeNumber = e.employeeNumber

        LEFT JOIN offices o
        ON e.officeCode = o.officeCode

        JOIN orders ord
        ON c.customerNumber = ord.customerNumber

        JOIN orderdetails od
        ON ord.orderNumber = od.orderNumber

        JOIN products p
        ON od.productCode = p.productCode

        JOIN (
            SELECT city
            FROM offices
            ORDER BY RAND()
            LIMIT 1
        ) off
        WHERE ord.orderNumber = (
            SELECT orderNumber
            FROM orders
            ORDER BY RAND()
            LIMIT 1
        )
        GROUP BY
            o.city,
            off.city,
            p.productName,
            p.productCode

        ORDER BY RAND()
        LIMIT 1;
    """

    return self.execute_query(query)
```

Complex Queries

The screenshot shows a dark-themed website for 'ClassicModels'. At the top, there's a blue header bar with the 'ClassicModels' logo on the left and navigation links for 'Home', 'Products', 'Cart (1)', and 'Welcome, Carine' on the right. A central callout box titled 'Your Lucky Pick' contains three items: 'CITY TO VISIT' (Sydney), 'PRODUCT TO BUY' (1968 Dodge Charger), and 'LUCKY NUMBER' (2). The bottom of the page features a copyright notice: '© 2025 ClassicModels'.

ClassicModels

Home Products Cart (1) Welcome, Carine ▾

>Your Lucky Pick

CITY TO VISIT
Sydney

PRODUCT TO BUY
1968 Dodge Charger

LUCKY NUMBER
2

© 2025 ClassicModels

Complex Queries

- This complex query is a product-line–level analytics reporting function.
- It joins `productlines`, `products`, `orderdetails`, and `orders` to **connect product categories to individual sales transactions**.
- It uses `SUM` and `COUNT` aggregations to calculate units sold, revenue, estimated cost, gross profit, gross margin, and distinct customers per product line.
- The inner `GROUP BY` aggregates **transactional data** at the product-line level.
- The outer `LEFT JOIN` ensures that product lines with no qualifying sales are still included in the report.
- `COALESCE` is used to handle `NULL` values so that missing sales data is displayed as zero.

```
def report_productlines():

    sql = """
        SELECT
            pl.productLine,
            COALESCE(s.numProducts, 0) AS numProducts,
            COALESCE(s.unitsSold, 0) AS unitsSold,
            COALESCE(s.revenue, 0) AS revenue,
            COALESCE(s.estCOGS, 0) AS estCOGS,
            COALESCE(s.estGrossProfit, 0) AS estGrossProfit,
            COALESCE(s.estGrossMargin, 0) AS estGrossMargin,
            COALESCE(s.distinctCustomers, 0) AS distinctCustomers
        FROM productlines pl
        LEFT JOIN (
            SELECT
                pl2.productLine,
                COUNT(DISTINCT pr.productCode) AS numProducts,
                SUM(od.quantityOrdered) AS unitsSold,
                SUM(od.quantityOrdered * od.priceEach) AS revenue,
                SUM(od.quantityOrdered * pr.buyPrice) AS estCOGS,
                SUM(od.quantityOrdered * od.priceEach) - SUM(od.quantityOrdered * pr.buyPrice) AS estGrossProfit,
                CASE
                    WHEN SUM(od.quantityOrdered * od.priceEach) = 0 THEN 0
                    ELSE
                        (SUM(od.quantityOrdered * od.priceEach) - SUM(od.quantityOrdered * pr.buyPrice)) / SUM(od.quantityOrdered * od.priceEach)
                END AS estGrossMargin,
                COUNT(DISTINCT o.customerNumber) AS distinctCustomers
            FROM productlines pl2
            JOIN products pr
            ON pr.productLine = pl2.productLine
            JOIN orderdetails od
            ON od.productCode = pr.productCode
            JOIN orders o
            ON o.orderNumber = od.orderNumber
            WHERE o.status = %s
            AND o.orderDate >= %s
            AND o.orderDate < %s
            GROUP BY pl2.productLine
        ) s
        ON s.productLine = pl.productLine
        ORDER BY revenue DESC;
    """
```

Complex Queries

ClassicModels

Home Products Welcome, Diane ▾

Product Line Performance

Start

End

Status

01.01.2004



01.01.2005



Shipped



Run

Product Line	# Products	Units Sold	Revenue	Est. Gross Profit	Gross Margin	Customers
Classic Cars	37	15,424	\$ 1,682,980.21	\$ 671,878.21	39.9%	66
Vintage Cars	24	10,487	\$ 823,927.95	\$ 337,219.36	40.9%	64
Motorcycles	13	5,976	\$ 527,243.84	\$ 222,485.41	42.2%	34
Trucks and Buses	11	4,853	\$ 448,702.69	\$ 176,415.25	39.3%	29
Planes	12	5,439	\$ 438,255.50	\$ 168,722.36	38.5%	31
Ships	9	3,752	\$ 292,595.34	\$ 116,371.77	39.8%	29
Trains	3	1,290	\$ 86,897.46	\$ 30,590.05	35.2%	19

Complex Queries

- This complex query is an employee-performance analytics reporting function.
- It joins `employees`, `customers`, `orders`, `order_details`, and `products` to connect sales representatives to their specific product-line sales transactions.
- It uses `SUM` aggregations to calculate the total revenue generated by each sales representative per product category.
- The `WHERE` clause filters the dataset to focus exclusively on employees holding the 'Sales Rep' job title.
- The `GROUP BY` aggregates transactional data at the individual employee and product-line levels.
- The `LIMIT` and `OFFSET` clauses are utilized to implement pagination.

```
def get_employee_performance_matrix(self, limit=10, offset=0):
    query = """
        SELECT
            e.firstName,
            e.lastName,
            p.productLine,
            SUM(od.quantityOrdered * od.priceEach) as revenue
        FROM employees e
        JOIN customers c ON e.employeeNumber = c.salesRepEmployeeNumber
        JOIN orders o ON c.customerNumber = o.customerNumber
        JOIN orderdetails od ON o.orderNumber = od.orderNumber
        JOIN products p ON od.productCode = p.productCode
        WHERE e.jobTitle = 'Sales Rep'
        GROUP BY e.employeeNumber, e.firstName, e.lastName, p.productLine
        ORDER BY e.employeeNumber
        LIMIT %s OFFSET %s
    """
    return self.execute_query(query, (limit, offset))
```

Complex Queries

Employee Performance (Revenue by Product Line)

SALES REP	PRODUCT LINE	TOTAL REVENUE GENERATED
Leslie Jennings	Classic Cars	\$373416.76
Leslie Jennings	Motorcycles	\$118788.86
Leslie Jennings	Planes	\$67956.68
Leslie Jennings	Ships	\$41565.71
Leslie Jennings	Trains	\$17965.32
Leslie Jennings	Trucks and Buses	\$163280.84
Leslie Jennings	Vintage Cars	\$298556.37
Leslie Thompson	Classic Cars	\$143865.77
Leslie Thompson	Motorcycles	\$43921.71
Leslie Thompson	Planes	\$40675.58

Complex Queries

- This query is an employee productivity and efficiency tracking function.
- It performs a sequence of **LEFT JOINs** starting from the employees table to customers and orders to map the relationship between sales staff and their transaction history.
- It uses the **COUNT** aggregation to calculate both the total number of customers assigned to a representative and their orders.
- The **WHERE** clause filters the dataset to focus exclusively on employees holding the 'Sales Rep' job title.
- The **GROUP BY** aggregates **transactional data** at the individual employee level.
- The **HAVING** clause serves as a critical filter to choose "unproductive" representatives who have zero successful orders (**order_count = 0**).

```
def get_unproductive_employees(self):  
    query = """  
        SELECT  
            e.employeeNumber,  
            e.firstName,  
            e.lastName,  
            e.email,  
            e.jobTitle,  
            COUNT(c.customerNumber) as customer_count,  
            COUNT(o.orderNumber) as order_count  
        FROM employees e  
        LEFT JOIN customers c ON e.employeeNumber = c.salesRepEmployeeNumber  
        LEFT JOIN orders o ON c.customerNumber = o.customerNumber  
        WHERE e.jobTitle = 'Sales Rep'  
        GROUP BY e.employeeNumber, e.firstName, e.lastName, e.email, e.jobTitle  
        HAVING order_count = 0  
    """  
  
    return self.execute_query(query)
```

Complex Queries

Sales Analytics

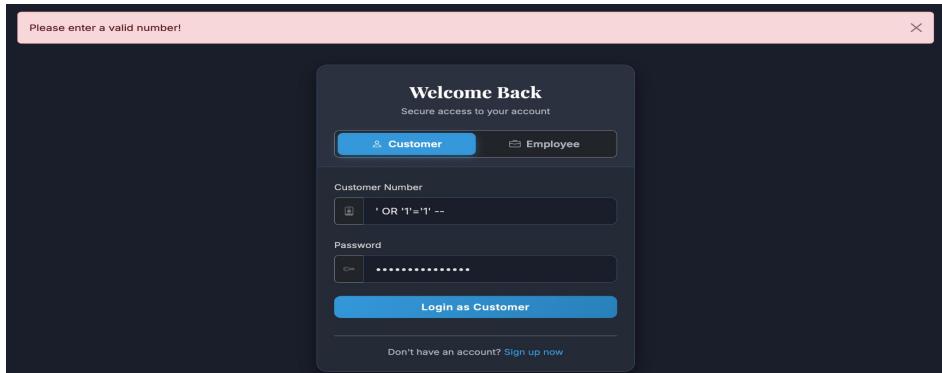
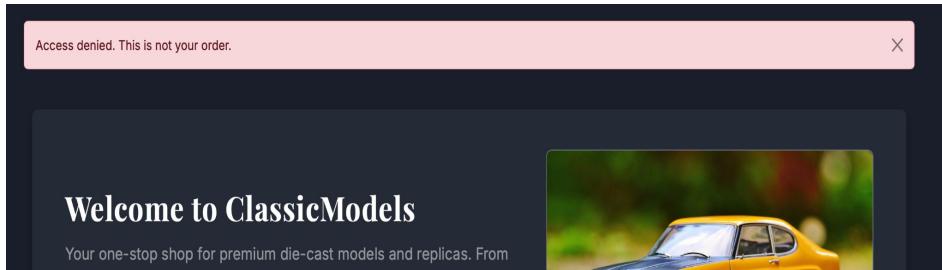
⚠ At Risk Employees (No Sales/Activity)

The following Sales Reps have **zero orders**.

- Tom King (ID: 1619) - tking@classicmodelcars.com
- Yoshimi Kato (ID: 1625) - ykato@classicmodelcars.com

Testing

- The application is functional and secure.
- It **preserves database integrity** as needed.
- The database was tested against **SQL injections**. It handles injections.
- The app was also tested against bad data, and it passed this test as well because it **validates user inputs server-side** (type checks, required fields, etc.) and forbids bad data.
- It enforces role-based access control: **only employees can manage products and only authorized roles can access reports/analytics pages**. We attempted to break into other customers' orders or employee pages, and failed to do so.
- It prevents inconsistent references **using normalized tables**, as well as database integrity.



Questions

If you have any questions, we would love to answer them!



THANK YOU