AI-04018 Sample DB Postgres and Mariadb

Owner	Ⓡ Rigel Arcayan
Tags	Project
Created time	July 11, 2026 10:21 PM

I have a Postgres 18 and Mariadb running in docker and have access via pgadmin and phpmyadmin web interface. I have zero knowledge of these two db. List which sample demo data to download, setup and learn.

Running both Postgres 18 and MariaDB in Docker is a fantastic, isolated environment for diving into SQL. Since you are using **pgAdmin** and **phpMyAdmin**, you can completely skip the command line and handle the setups directly through your browser.

The strategy is to use industry-standard sample datasets. They are clean, well-documented, and perfectly sized for testing queries without sluggishness.

1. For Postgres 18: The "DvdRental" Database

The **dvdrental** dataset is the absolute standard for learning Postgres. It models a fictional movie rental store (customers, inventory, films, actors) and is great for practicing table joins, filtering, and aggregations.

Where to Download

- **Source:** Search for the **Postgres Tutorial DVD Rental sample database** zip file (a simple `.zip` containing a `.tar` file).

How to Setup via pgAdmin

1. **Extract the file:** Unzip the downloaded folder to get the `dvdrental.tar` file.
2. **Create a Blank DB:** Open pgAdmin, right-click **Databases** $\rightarrow$ **Create** $\rightarrow$ **Database...** Name it `dvdrental` and click Save.

3. **Restore the Data:** Right-click your newly created `dvdrental` database and select **Restore...**
4. **Configure Restore:**
    - Set **Format** to *Custom or tar*.
    - In **Filename**, click the browser icon to upload/select your `dvdrental.tar` file.
    - Click the **Restore** button at the bottom right. Refresh your database tree, and you will see all tables populated under `Schemas -> public -> Tables`.

2. For MariaDB: The "ClassicModels" Database

Because MariaDB is fully backward-compatible with MySQL, the classic MySQL **classicmodels** dataset is ideal. It represents an e-commerce retailer selling scale-model cars, containing data for customers, orders, order line items, and employees.

Where to Download

- **Source:** Search for the **MySQL Tutorial ClassicModels sample database** zip file. You are looking for a plain `.sql` text file inside the archive (e.g., `mysqlsampledatabase.sql`).

How to Setup via phpMyAdmin

1. **Create a Blank DB:** Open phpMyAdmin in your browser. Click **New** on the left sidebar. Type `classicmodels` as the database name and click **Create**.
2. **Import the Script:** Select your new `classicmodels` database from the left column, then click the **Import** tab at the top menu bar.
3. **Upload & Run:** Click **Choose File**, select the unzipped `mysqlsampledatabase.sql` file, scroll to the bottom, and click **Import** (or **Go** depending on the exact UI layout). It will execute the text file script and build the tables instantly.

Your First Learning Roadmap

Once the data is in, open the **Query Tool** (in pgAdmin, click the tools icon on top; in phpMyAdmin, click the **SQL** tab) and try these core concepts to see how the syntax compares between the two platforms:

SQL

```
/* 1. Basic Filtering (Works exactly the same in both) */
SELECT * FROM customer WHERE active = 1;

/* 2. Joining Tables (Crucial for relational databases) */
-- Postgres Example: Find who rented what film
SELECT c.first_name, c.last_name, f.title
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id;
```

Would you like a few structured SQL practice problems based specifically on these two datasets to help you practice your first joins and data lookups?

Yes comprehensive and structured SQL practice problems and separate Postgres first then Mariadb

Here is a structured, step-by-step SQL practice curriculum divided into **Core Postgres (dvdrental)** and **Core MariaDB (classicmodels)**.

Each section scales gradually from simple data extraction to advanced relational analysis. Try writing these in your respective web interface Query Tools.

Part 1: Postgres Practice Problems (dvdrental)

The `dvdrental` schema uses snake_case and relies heavily on linking IDs across inventory, rentals, and payments.

🟢 Level 1: Foundations (Filtering & Sorting)

- **Problem 1.1:** Find all active customers. Select their `first_name`, `last_name`, and `email`, sorted alphabetically by their last name.
- **Problem 1.2:** List all films with a rental rate of $4.99 and a replacement cost less than $20.00.
- **Problem 1.3:** Identify all distinct movie ratings (e.g., PG, R, G) available in the `film` table.

🟡 Level 2: Core Analysis (Aggregations & Grouping)

- **Problem 1.4:** Calculate the total number of films available in the inventory, the average rental duration, and the maximum replacement cost.
- **Problem 1.5:** Group the films by their `rating`. Show the rating type and the average rental rate for each rating, sorted from highest average price to lowest.
- **Problem 1.6:** Find the top 5 customers who have spent the most money overall on movie rentals by aggregating the `payment` table.

🔴 Level 3: Advanced Relational (Multi-Table Joins)

- **Problem 1.7:** Write a query to display the first name, last name, movie title, and rental date for every rental made. *(Hint: You will need to join `customer`, `rental`, `inventory`, and `film`).*
- **Problem 1.8:** Figure out which film category (e.g., Action, Sci-Fi, Animation) has generated the most total rental revenue.
- **Problem 1.9:** Identify any customers who currently live in the country of 'United States'. *(Hint: Connect `customer` $\rightarrow$ `address` $\rightarrow$ `city` $\rightarrow$ `country`).*

Part 2: MariaDB Practice Problems (classicmodels)

The `classicmodels` database uses camelCase for some fields and acts as a classic B2B/B2C retail model (orders, products, offices, employees).

🟢 Level 1: Foundations (Filtering & Sorting)

- **Problem 2.1:** List all employees who hold the job title 'Sales Rep', sorted by their `lastName`.
- **Problem 2.2:** Find all products in the 'Classic Cars' product line with a quantity in stock greater than 5,000.
- **Problem 2.3:** List all unique states where customers are located, filtering out any `NULL` or empty values.

🟡 Level 2: Core Analysis (Aggregations & Grouping)

- **Problem 2.4:** Find the total number of orders currently marked with a status of 'Shipped', 'Cancelled', or 'In Process'.
- **Problem 2.5:** Calculate the total dollar amount of payments received per customer ID, but only display customers who have spent more than $100,000 in total. *(Hint: Use a* `HAVING` *clause for the aggregate filter)*.
- **Problem 2.6:** Determine the average profit margin for each product line. Profit margin can be calculated by subtracting `buyPrice` from `MSRP`.

🔴 Level 3: Advanced Relational (Multi-Table Joins)

- **Problem 2.7:** Produce a report showing the order number, order date, product name, quantity ordered, and price each for every item inside order number `10123`.
- **Problem 2.8:** Create a staff directory that displays each sales representative's full name alongside the physical `city` and `country` of the office they are assigned to.
- **Problem 2.9:** Find the name of the customer who placed the single most expensive individual order (based on multiplying `quantityOrdered` by `priceEach` in the order details).

How to Check Your Work

Take your time running these in pgAdmin and phpMyAdmin. When you are ready to review your syntax, see the structural patterns, or if you hit a wall on a specific join, let me know.

Which dataset or level would you like to reveal the solutions for first?

Provide a step by step solution of the the two core practice problems

Here are the step-by-step solutions for all the problems in both databases.

You can copy and paste these directly into the **Query Tool** in pgAdmin for Postgres, or the **SQL** tab in phpMyAdmin for MariaDB.

Part 1: Postgres Solutions (dvdrental)

🟢 Level 1: Foundations

Problem 1.1: Find active customers

SQL

```
SELECT first_name, last_name, email
FROM customer
WHERE active = 1
ORDER BY last_name ASC;
```

- **Why it works:** `WHERE active = 1` filters out inactive accounts, and `ORDER BY ... ASC` sorts them alphabetically from A to Z.

Problem 1.2: Specific rental rates and replacement costs

SQL

```
SELECT title, rental_rate, replacement_cost
FROM film
WHERE rental_rate = 4.99 AND replacement_cost < 20.00;
```

- **Why it works:** The `AND` operator ensures *both* price conditions must be true for a row to show up.

Problem 1.3: Distinct movie ratings

SQL

```
SELECT DISTINCT rating
FROM film;
```

- **Why it works:** `DISTINCT` strips out duplicates, giving you a clean list of unique rating codes (G, PG, R, etc.).

🟡 Level 2: Core Analysis

Problem 1.4: Inventory totals, averages, and maximums

SQL

```
SELECT
    COUNT(*) AS total_films,
```

```
    AVG(rental_duration) AS avg_duration,
    MAX(replacement_cost) AS max_replacement
FROM film;
```

- **Why it works:** `COUNT`, `AVG`, and `MAX` are aggregate functions. `AS` creates readable column headers (aliases) for the calculated results.

Problem 1.5: Average rental rate grouped by rating

SQL

```
SELECT rating, AVG(rental_rate) AS average_rate
FROM film
GROUP BY rating
ORDER BY average_rate DESC;
```

- **Why it works:** `GROUP BY rating` tells SQL to slice the data into groups based on rating before calculating the average. `DESC` sorts from highest price to lowest.

Problem 1.6: Top 5 spending customers

SQL

```
SELECT customer_id, SUM(amount) AS total_spent
FROM payment
GROUP BY customer_id
ORDER BY total_spent DESC
LIMIT 5;
```

- **Why it works:** `SUM(amount)` totals up the cash per customer, and `LIMIT 5` chops off the list after the top 5 rows.

🔴 Level 3: Advanced Relational

Problem 1.7: Who rented what movie (4-Table Join)

SQL

```
SELECT c.first_name, c.last_name, f.title, r.rental_date
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
```

```
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id;
```

- **Why it works:** We use single-letter aliases (`c`, `r`, `i`, `f`) to keep the code clean. The `ON` clauses define the shared primary/foreign keys linking the chain together.

Problem 1.8: Most lucrative film category

SQL

```
SELECT cat.name AS category_name, SUM(p.amount) AS total_revenue
FROM category cat
JOIN film_category fc ON cat.category_id = fc.category_id
JOIN film f ON fc.film_id = f.film_id
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
JOIN payment p ON r.rental_id = p.rental_id
GROUP BY cat.name
ORDER BY total_revenue DESC
LIMIT 1;
```

- **Why it works:** This loops all the way from categories down to individual dollar transactions via a "junction table" (`film_category`), aggregates the sum, and pulls the number one spot.

Problem 1.9: US-based customers

SQL

```
SELECT cust.first_name, cust.last_name, co.country
FROM customer cust
JOIN address a ON cust.address_id = a.address_id
JOIN city ci ON a.city_id = ci.city_id
JOIN country co ON ci.country_id = co.country_id
WHERE co.country = 'United States';
```

- **Why it works:** This traces geographic relationships back to the individual customer profile.

Part 2: MariaDB Solutions (classicmodels)

🟢 Level 1: Foundations

Problem 2.1: Sales Rep directory

SQL

```
SELECT firstName, lastName, jobTitle
FROM employees
WHERE jobTitle = 'Sales Rep'
ORDER BY lastName ASC;
```

Problem 2.2: Classic Cars in stock

SQL

```
SELECT productName, productLine, quantityInStock
FROM products
WHERE productLine = 'Classic Cars' AND quantityInStock > 5000;
```

Problem 2.3: Unique non-null customer states

SQL

```
SELECT DISTINCT state
FROM customers
WHERE state IS NOT NULL AND state <> '';
```

- **Why it works:** `IS NOT NULL` skips missing structural data, and `<>` (not equal to) filters out blank string entries.

🟡 Level 2: Core Analysis

Problem 2.4: Counts of specific order statuses

SQL

```
SELECT status, COUNT(*) AS order_count
FROM orders
WHERE status IN ('Shipped', 'Cancelled', 'In Process')
GROUP BY status;
```

- **Why it works:** `IN` acts like a shorthand container for multiple `OR` conditions.

Problem 2.5: High-value customers (Filter on aggregate)

SQL

```
SELECT customerNumber, SUM(amount) AS totalPayments
FROM payments
GROUP BY customerNumber
HAVING totalPayments > 100000;
```

- **Why it works:** Critical distinction: `WHERE` filters raw tables *before* math happens; `HAVING` filters the finalized groups *after* the math (`SUM`) runs.

Problem 2.6: Profit margins by product line

SQL

```
SELECT productLine, AVG(MSRP - buyPrice) AS avgProfitMargin
FROM products
GROUP BY productLine;
```

- **Why it works:** SQL can execute basic math natively inside your aggregate functions row-by-row before calculating the average.

🔴 Level 3: Advanced Relational

Problem 2.7: Invoice details for Order 10123

SQL

```
SELECT od.orderNumber, o.orderDate, p.productName, od.quantityOrdered, od.priceEach
FROM orderdetails od
JOIN orders o ON od.orderNumber = o.orderNumber
JOIN products p ON od.productCode = p.productCode
WHERE od.orderNumber = 10123;
```

Problem 2.8: Employees and their physical offices

SQL

```
SELECT e.firstName, e.lastName, o.city, o.country
FROM employees e
JOIN offices o ON e.officeCode = o.officeCode;
```

Problem 2.9: Customer name with the single highest value order

SQL

```
SELECT c.customerName, od.orderNumber, SUM(od.quantityOrdered * od.priceEach) AS orderVal
ue
FROM customers c
JOIN orders o ON c.customerNumber = o.customerNumber
JOIN orderdetails od ON o.orderNumber = od.orderNumber
GROUP BY od.orderNumber, c.customerName
ORDER BY orderValue DESC
LIMIT 1;
```

- **Why it works:** It multiplies item quantities by cost, sums up the item lines for each unique order id, groups them clean, and leaves you with the biggest ticket transaction in the system.

Once I mastered the Sql commands, give me a glimpse on how to build web API to execute the queries

Once you are comfortable writing SQL queries, the next step is connecting them to the web. A **Web API** acts as a secure, fast-moving bridge between your database and the outside world (like a mobile app or a web browser).

Instead of typing queries by hand into pgAdmin or phpMyAdmin, a frontend application sends a standard HTTP request (like a web browser loading a page) to your API server, which executes the SQL behind the scenes and returns the data in a clean, universal text format called **JSON**.

Here is a glimpse of how this works structurally using **Node.js** (JavaScript) and **Express**, which is one of the most popular ways to build APIs today.

The Core Blueprint of a Web API

Every relational database API follows the exact same sequence whenever a user clicks a button or loads a data page:

```
[ Frontend App ]  ---> HTTP Request --->  [ Your API Server ]  ---> Runs SQL ---> [ Datab
ase ]
[  (Web/Mobile)  ]  <--- Sends JSON  <---  [  (Node/Python)   ]  <--- Rows Return <--- [
(Postgres/Maria) ]
```

Code Glimpse: A Live API Endpoint

Below is a complete, minimal example of a Node.js API server. It connects to your `dvdrental` Postgres database, sets up a web URL endpoint at `/api/top-customers`, and runs the exact **Problem 1.6** aggregation query you looked at earlier.

JavaScript

```
// 1. Load the required web server and database drivers
const express = require('express');
const { Pool } = require('pg');

const app = express();
const port = 3000;

// 2. Configure the connection to your Docker Postgres instance
const pool = new Pool({
  user: 'postgres',
  host: 'localhost', // or your Docker container IP
  database: 'dvdrental',
  password: 'your_secure_password',
  port: 5432,
});

// 3. Define a web route (URL endpoint) that someone can visit
app.get('/api/top-customers', async (req, res) => {
  try {
    // The exact SQL query you mastered in pgAdmin
    const sqlQuery = `
      SELECT customer_id, SUM(amount) AS total_spent
      FROM payment
      GROUP BY customer_id
      ORDER BY total_spent DESC
      LIMIT 5;
    `;

    // Execute the query against the database pool
    const result = await pool.query(sqlQuery);

    // Send the raw data rows back to the web browser as JSON
    res.json(result.rows);
```

```
  } catch (err) {
    console.error(err.message);
    res.status(500).send('Database connection error');
  }
});

// 4. Start the API server listening for web requests
app.listen(port, () => {
  console.log(`API server running at http://localhost:${port}`);
});
```

What the Output Looks Like

If you were to launch this code script and open your web browser to `http://localhost:3000/api/top-customers`, your API server would capture the request, run the query in Docker, and render a structured payload like this on your screen:

JSON

```
[
  { "customer_id": 148, "total_spent": "211.55" },
  { "customer_id": 526, "total_spent": "208.58" },
  { "customer_id": 178, "total_spent": "194.61" },
  { "customer_id": 137, "total_spent": "194.61" },
  { "customer_id": 144, "total_spent": "189.60" }
]
```

Any modern user interface framework (like React, Vue, or a mobile app SDK) can consume this structural format instantly to draw charts, build administrative dashboards, or render account sheets.

List top topics and books to read about the API you mentioned. I like to read what's inside the API hood on node.js and JSON

To understand what happens deep "under the hood" when Node.js handles a web request, parses JSON, and talks to a database, you need to look past basic syntax and explore how Node.js manages system resources.

Unlike traditional multi-threaded web servers, Node.js runs on a single main thread, using a highly specialized architecture to handle thousands of concurrent database queries without breaking a sweat.

Here are the top engineering concepts you should study, along with the definitive books that break down exactly what happens under the hood.

The Top "Under the Hood" Concepts to Study

1. The Event Loop and Libuv

Node.js is single-threaded, meaning it executes JavaScript code one line at a time. However, it handles database actions asynchronously. When your API runs a query against Postgres, Node.js hands that task over to a C++ library called **libuv**. Libuv manages an internal thread pool and operating system kernels to wait for the database response, leaving the main JavaScript thread completely free to handle other incoming API requests.

2. Stream-Based JSON Parsing

When a client sends a large JSON payload to your API, Node.js doesn't just read it all into memory at once. It reads it as a **Stream**—chunks of raw binary data (buffers) passing through memory. Understanding how Node.js converts these raw buffers into JavaScript objects using V8's internal JSON parser is critical for writing high-performance APIs.

3. Database Connection Pooling

Opening and closing a brand-new connection to Postgres or MariaDB for every single API request is incredibly expensive and slow. Under the hood, APIs use **Connection Pools**—a warm cache of active database connections kept alive in memory. When a request comes in, the API borrows an active connection for a fraction of a millisecond, executes the query, and instantly returns it to the pool.

Core Books for Deep Technical Mastery

The following books avoid generic "how-to" tutorials and focus deeply on internal architecture, performance mechanics, and design patterns.

1. *Node.js Design Patterns* (by Mario Casciaro and Luciano Mammino)

- **Why it’s essential:** This is universally considered the gold standard for understanding Node.js mechanics.
- **What you’ll learn inside:** It features an incredibly detailed breakdown of the **Reactor Pattern**, the inner workings of the **Event Loop**, and how asynchronous control flows work. It also goes deep into how Streams process data chunks under the hood, which is exactly how massive JSON payloads and database records are handled efficiently without exhausting RAM.

2. *Professional Node.js: Building Large-Scale Applications with Express* (by Pedro Teixeira)

- **Why it’s essential:** Focuses heavily on network plumbing and system-level interactions.
- **What you’ll learn inside:** This book explains how Node.js wraps around low-level TCP/IP sockets to handle HTTP traffic. You will learn how network buffers are allocated, how Express routes requests through middleware chains, and how to safely manage data pipelines between network sockets and external relational databases.

3. *High Performance JavaScript* (by Nicholas C. Zakas)

- **Why it’s essential:** To understand the engine that executes your API, you need to understand the V8 engine itself.
- **What you’ll learn inside:** This book teaches you how JavaScript code is optimized by engines under the hood. It covers memory management, execution context, and how data structures like deep JSON objects impact CPU cycles and garbage collection—crucial knowledge for ensuring your API endpoints don't cause memory leaks during heavy database operations.

When you look into these resources, start with the chapters on the **Event Loop** and **Streams**. Mastering those two areas will completely change how you write server-side code and give you a crystal-clear picture of how data flows from a raw Docker network port right into your SQL database.

Where to find a Mariadb northwinds dataset for tutorials

Because MariaDB is fully compatible with MySQL, any standard **MySQL Northwind dataset** will work flawlessly in MariaDB. You can find highly reliable,

community-maintained SQL source scripts on GitHub and GitHub Gists.

Here are the best places to grab the dataset for your tutorials, depending on what variation of the schema you prefer:

1. The Direct SQL Dump (Classic Schema)

If you want a straightforward .sql script that matches the original Microsoft SQL Server schema structures (CamelCase table names like Customers, OrderDetails), you can use a verified GitHub Gist.

- **Source:** ryguyrg / northwind.sql Gist
- **What it contains:** A single raw SQL script that drops the database if it exists, creates the northwind schema, builds all 13 core tables, and populates them with the traditional sample data.

2. The Modernized Schema (Snake_Case)

If you prefer writing tutorials using modern open-source conventions (lowercase with underscores like order_details, customer_id), the **MyWind** project is an excellent choice.

- **Source:** dalers / mywind GitHub Repository
- **What it contains:** It includes separate structure and data files (northwind.sql and northwind-data.sql), an Entity-Relationship Diagram (ERD), and updates standard date constraints to prevent 1997â€“2038 UTC timestamp errors.

3. Multi-Database Sample Project (Docker & Multi-Format)

If you plan to run your tutorials inside Docker containers or need the data in formats other than raw SQL (like JSON flat files), this comprehensive sample repo is perfect.

- **Source:** harryho / db-samples GitHub Repository
- **What it contains:** Ready-to-go Docker Compose configurations alongside raw scripts for MySQL/MariaDB, PostgreSQL, and SQLite.

How to Import It Fast

Once you choose a source and download the .sql file, you can import it into your MariaDB instance using your terminal or command prompt in seconds:

1. **Start your MariaDB server** *Prerequisite*
   Ensure your local or remote MariaDB instance is running.
2. **Create the blank database** *Terminal*
   Log into your MariaDB shell and spin up a target database:
3. **Execute the import script** *CLI Command*
   Run the file directly from your terminal using the redirection operator (<):

Note on compatibility: If your downloaded script already includes CREATE DATABASE IF NOT EXISTS northwind; and USE northwind; at the very top (which the ryguyrg gist does), you can skip Step 2 entirely and just run: mysql -u your_username -p < path/to/northwind.sql.