About the Author

Dr. Hrishikesh (Hrish) Desai
CPA, CFA, EA, CMA, FCA

Associate Professor of Accounting

Director of the Master of Accountancy with Data Analytics Program

Arkansas State University

Contact Information

Linkedin | Twitter | Instagram

SQL Fundamentals: A Practical Guide

Master essential SQL queries through hands-on examples using employee and sales data from Acme Corp.

Why SQL Matters

Universal Language

SQL is the most widely used language for working with databases across industries. Whether you're in accounting, finance, healthcare, tech, or retail, SQL skills are essential for data analysis and decision-making.

Career Essential

Virtually every data role requires SQL proficiency. It's the foundation for data analysts, scientists, engineers, and business intelligence professionals worldwide.

Understanding SQL

Structured

Data is organized in highly structured tables with rows and columns, making it easy to find and retrieve information efficiently.

Query

You ask questions (queries) to retrieve specific data from the database using standardized commands and syntax.

Language

SQL provides a consistent syntax for communicating with databases, regardless of the specific database system you're using.

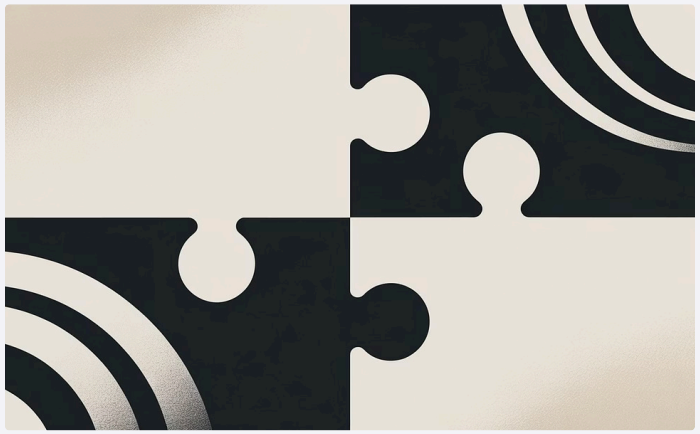
What is a Database?

A database is an organized collection of data stored electronically. Think of it as a digital filing system where information is stored in a structured way that makes it easy to access, manage, and update.

Instead of scattered files and folders, databases use tables to organize information systematically. Each table contains related data, and these tables can connect to each other to form relationships.

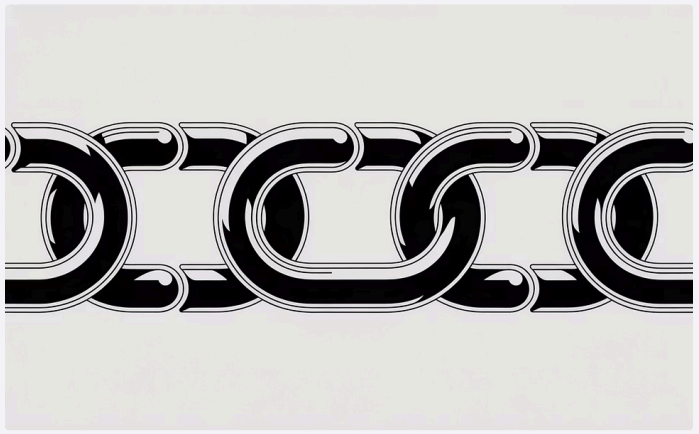
Understanding Table Relationships

Real-world databases rarely store all information in a single table. Instead, data is distributed across multiple related tables to reduce redundancy and maintain data integrity.



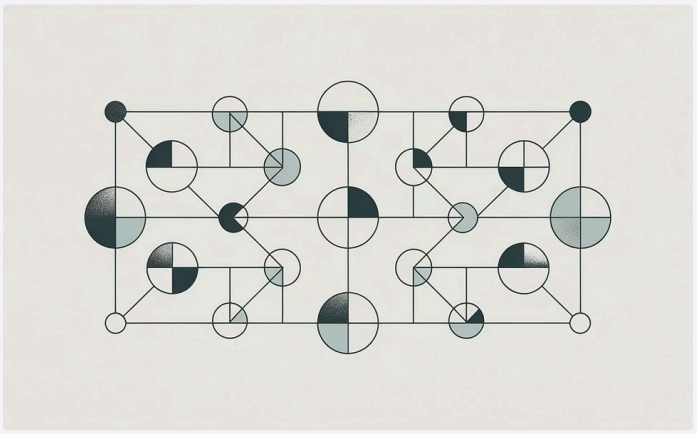
Primary Keys

Unique identifiers in each table that distinguish individual records



Foreign Keys

Columns that reference primary keys in other tables, creating relationships



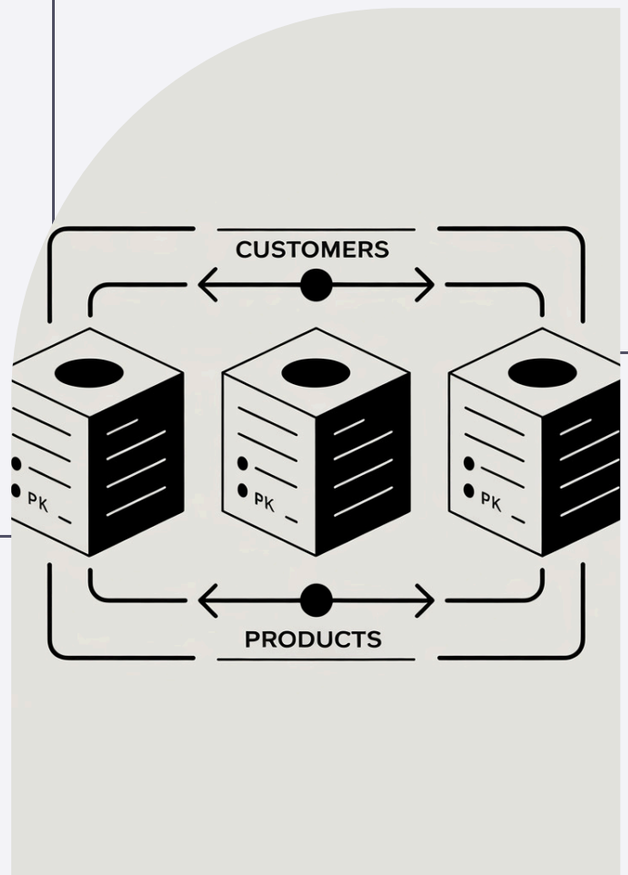
Relationships

Connections between tables that allow you to combine related information

Relational Databases Explained

Customers

customer_id (PK),
customer_name,
email



Orders

order_id (PK),
customer_id (FK),
order_date,
product_id (FK)

Products

product_id (PK),
product_name,
price

Organized Structure

Data is stored in tables (also called relations) with clearly defined columns and rows, similar to spreadsheets but much more powerful.

Connected Tables

Tables can link to each other through common fields, allowing you to combine information from multiple sources seamlessly.

Data Integrity

Relational databases enforce rules to ensure data accuracy, consistency, and reliability across all connected tables.

The Power of Queries

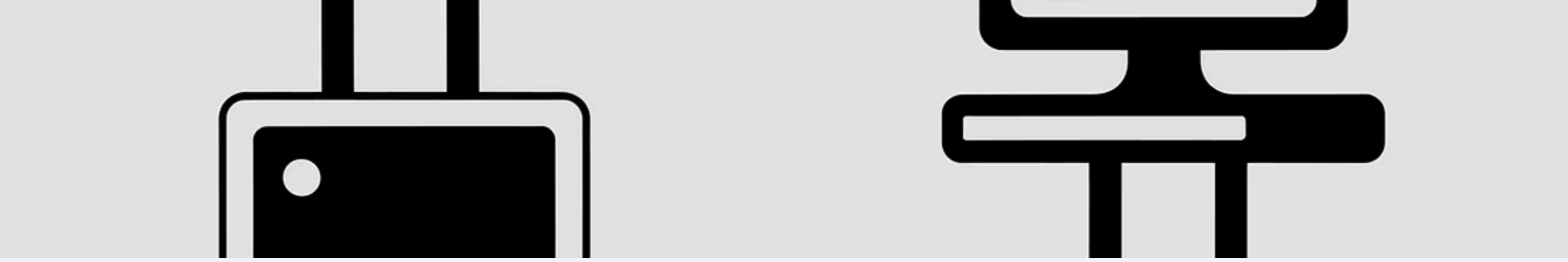
What is a Query?

A query is a request for data from a database. It's how you ask the database to show you specific information based on criteria you define.

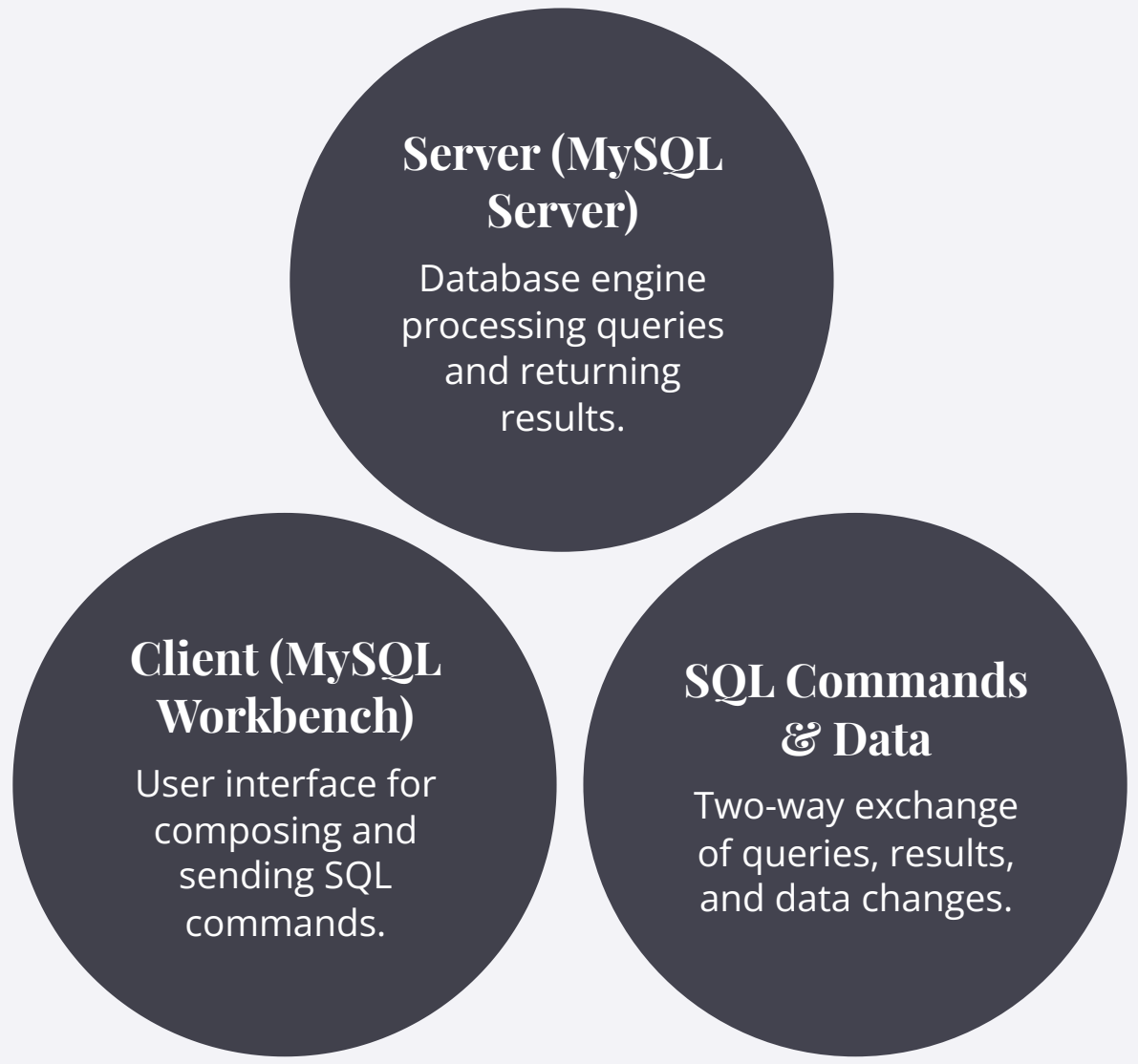
Queries can be simple (show all records) or complex (filter, sort, and combine data from multiple tables).

Setting Up Your Environment

1	2	3
Choose an SQL Editor	**Install Database Server**	**Load Data**
We will use MySQL Workbench. It provides a graphical interface for writing and executing SQL queries.	The editor is just a tool to write queries. You also need a database server (like MySQL) to store and manage your data.	Practice with datasets to build your skills.



Understanding the Client-Server Architecture



To effectively manage and query databases, it's crucial to understand the relationship between the MySQL Server and MySQL Workbench. They operate as a classic client-server system:

- **MySQL Server:** This is the engine (backend). It stores all your data and is responsible for managing, processing, and serving that data when requested.
- **MySQL Workbench:** This acts as the client (frontend). It's the graphical user interface where you write your SQL commands and view the results. It sends your commands to the server and displays the data returned.

Think of the MySQL Server as a vast digital library containing all the books (data). MySQL Workbench is your librarian's desk, where you submit requests for specific books or information. They are two separate programs that communicate with each other, which is why you need both installed for a complete and functional setup.

Installing MySQL Server

Before you can start querying databases, you need to set up the MySQL Server itself. This is the backend engine that makes everything work.

You might be wondering, "Why do I need to install this on my own computer? Can't I just use one online?" That's a great question! For learning and development, installing MySQL Server directly on your computer means it runs locally. This gives you full control over your data and doesn't require an internet connection to practice your skills. Think of it like having your own personal library at home instead of always going to a public one – you have immediate access and complete control over your books. Running it locally is often faster, free, and keeps your practice data private. Once you're comfortable working with a local server, it's much easier to understand and work with cloud-based databases later on.

01

The Database Engine

MySQL Server is the core database engine. It's where your data is actually stored, managed, and processed. Without it, there's no database to interact with.

02

Workbench: Your Interface

Remember, MySQL Workbench is merely a graphical client. It's a tool that allows you to write, execute, and manage your SQL queries and databases, but it doesn't store the data itself.

03

Installation Options

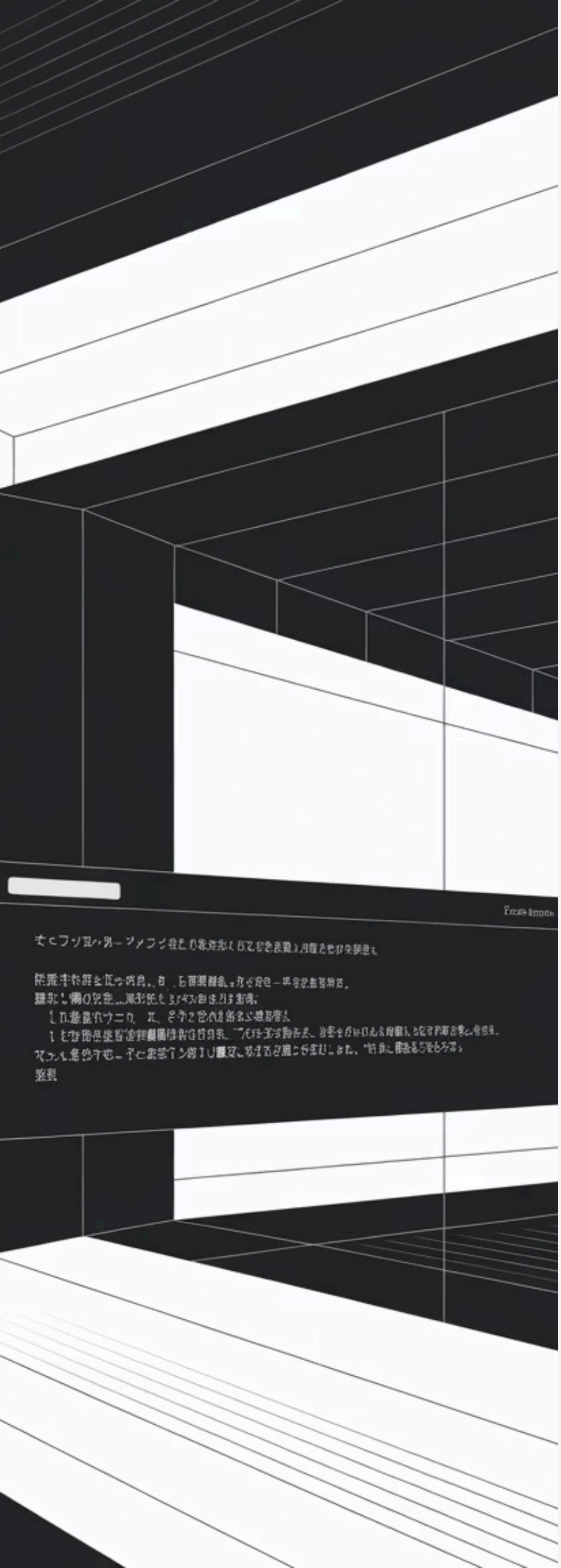
You have two primary ways to install MySQL Server:

- **Direct Download:** Get the official installer directly from the MySQL website (recommended for all operating systems).
- **Homebrew (Mac):** If you're on a Mac, you can use Homebrew, a popular package manager.

04

Homebrew Simplifies (Mac)

For Mac users, Homebrew streamlines the installation and update process, making it quicker and easier to manage your MySQL Server setup.



Installing with Homebrew (Mac)

Homebrew is a free and open-source package manager that simplifies the installation of software on macOS. It makes installing developer tools and other software straightforward.

01

Install Homebrew

First, check if Homebrew is already installed by running this command in your Terminal:

```
brew --version
```

If Homebrew is not installed, open your Terminal and run the following command:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

02

Install MySQL

Once Homebrew is set up, you can easily install the MySQL server with a single command:

```
brew install mysql
```

03

Start MySQL Service

After the installation completes, start the MySQL server as a background service:

```
brew services start mysql
```

==> **Successfully started** `mysql` **(label: homebrew.mxcl.mysql)**

04

Verify Installation

Confirm that MySQL is running and check its version to ensure a successful setup:

```
mysql --version
```

Installing MySQL Workbench

MySQL Workbench is a powerful visual tool. It provides a graphical interface for managing and interacting with your MySQL databases.

01

Official Download

Obtain MySQL Workbench from the official MySQL website. Ensure you select the version compatible with your operating system.

02

Separate Tool

MySQL Workbench is a standalone application, distinct from the MySQL Server itself. You need both for a complete setup.

03

Simple Installation

On macOS, it's typically a drag-and-drop into Applications. For Windows, follow the guided installer steps.

04

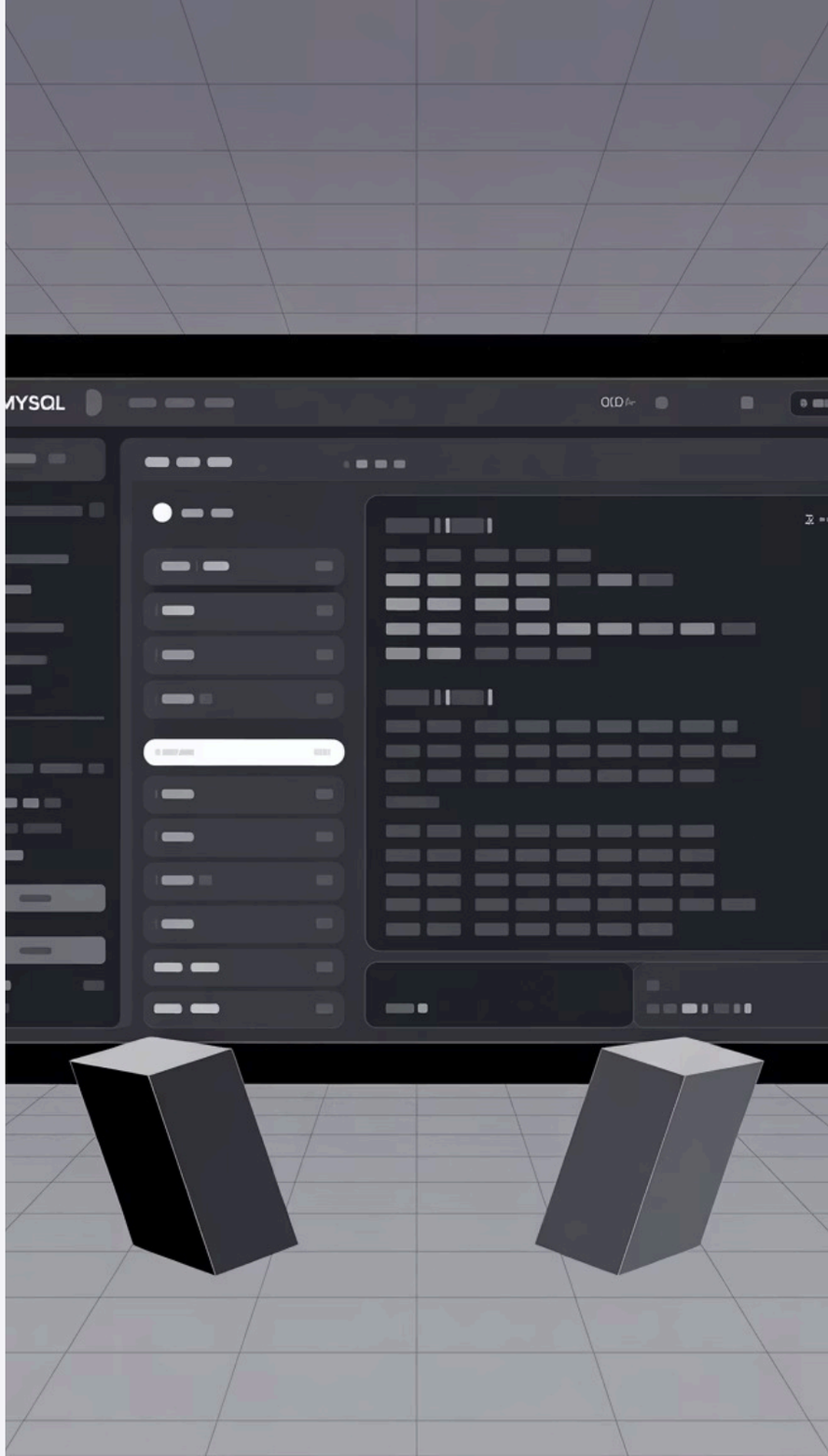
Initial Connection

Upon first launch, you'll be prompted to configure a connection to your running MySQL Server. This links the Workbench to your database.

05

Ready to Use

Once successfully connected, MySQL Workbench is fully operational, allowing you to manage databases and execute queries.



Creating Your First Connection

Connecting MySQL Workbench to your running MySQL Server is the final step to start managing your databases. Follow these steps to establish your first connection profile:

01

Launch Workbench

Open MySQL Workbench. You'll see the main interface, usually with a "MySQL Connections" panel on the left.

02

Add New Connection

Click the `+` icon located next to "MySQL Connections" in the main window to open the "Setup New Connection" dialog.

03

Configure Details

Fill in the connection parameters:

- **Connection Name:** "Local MySQL" (or any descriptive name)
- **Hostname:** `127.0.0.1` or `localhost`
- **Port:** `3306` (the default MySQL port)
- **Username:** `root`

04

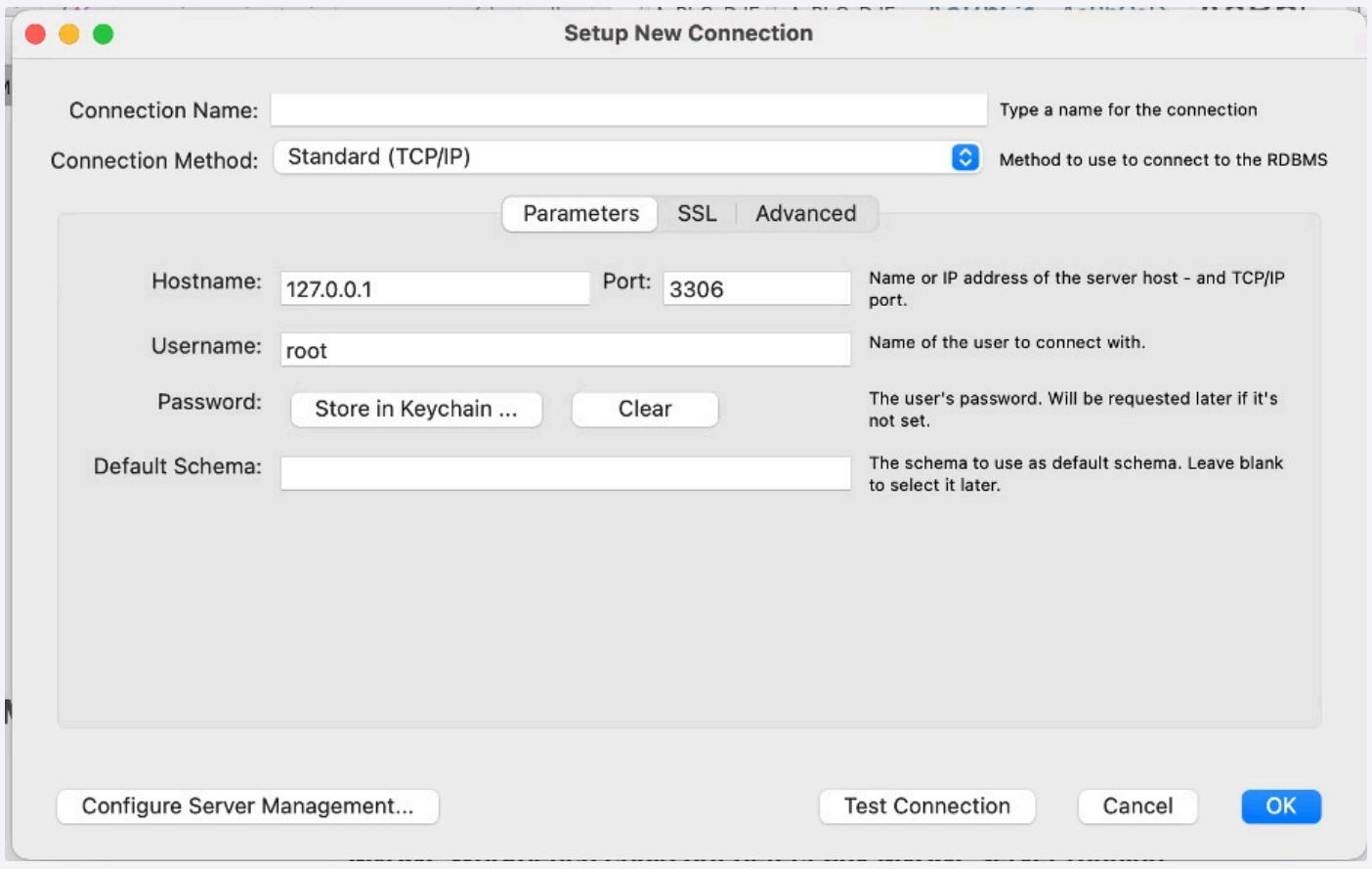
Test & Authenticate

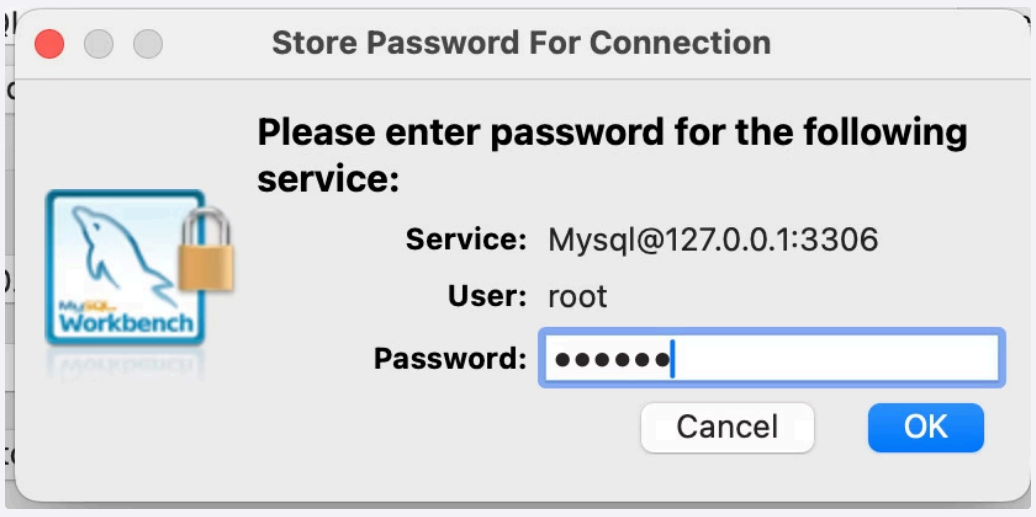
Click "Test Connection". If prompted, enter your MySQL root password. A successful test will confirm communication with the server.

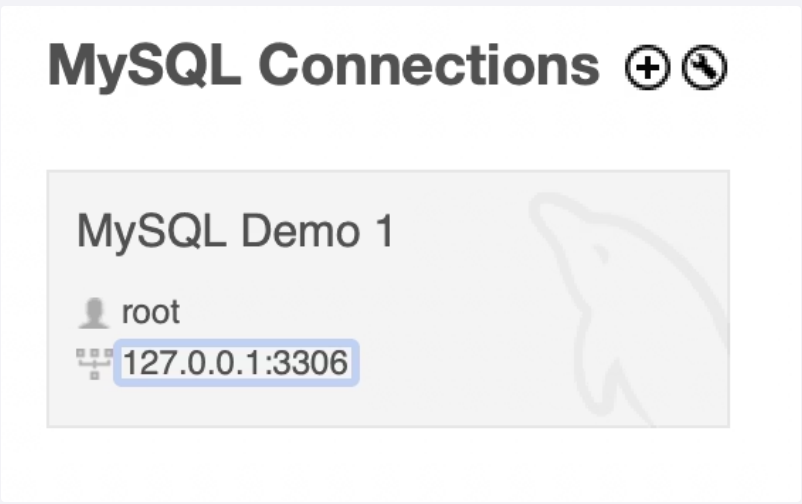
05

Save & Connect

Click "OK" to save the connection. Then, double-click your newly created connection in the "MySQL Connections" panel to open it and start working.







Troubleshooting: Connection Failed

Encountering connection issues with MySQL Workbench? Don't worry, it's a common hurdle. Here are the main reasons your connection might fail and how to resolve them.

MySQL Server Not Running

The most frequent reason for connection failure is an inactive MySQL Server. Workbench needs a running server to connect to.

How to check:

- **Mac:** Open Terminal and run `brew services list`. Look for `mysql` and ensure its status is `started`.
- **Windows:** Search for "Services" (`services.msc`), find the `MySQL` service, and confirm it's `Running`.
- If not running, start the service (e.g., `brew services start mysql` or right-click > Start).

Incorrect Connection Settings

Mismatched details in your Workbench connection profile will prevent access to the server.

- **Hostname:** Ensure it's set to `localhost` or `127.0.0.1`.
- **Port:** Verify the port number is `3306` (this is the default MySQL port).
- **Username:** Confirm the username, usually `root` for local development, and check the password.

Firewall or Port Blocked

Your operating system's firewall or network security settings might be preventing Workbench from communicating with the MySQL Server.

- Temporarily disable your firewall to test if the connection can be established.
- If the connection works, configure your firewall to allow incoming and outgoing connections on port `3306` for MySQL.

Popular SQL Editors

Each editor has unique features, but all allow you to write SQL queries, view results, and manage databases. MySQL Workbench is excellent for beginners due to its intuitive interface and comprehensive documentation.

Understanding Schemas

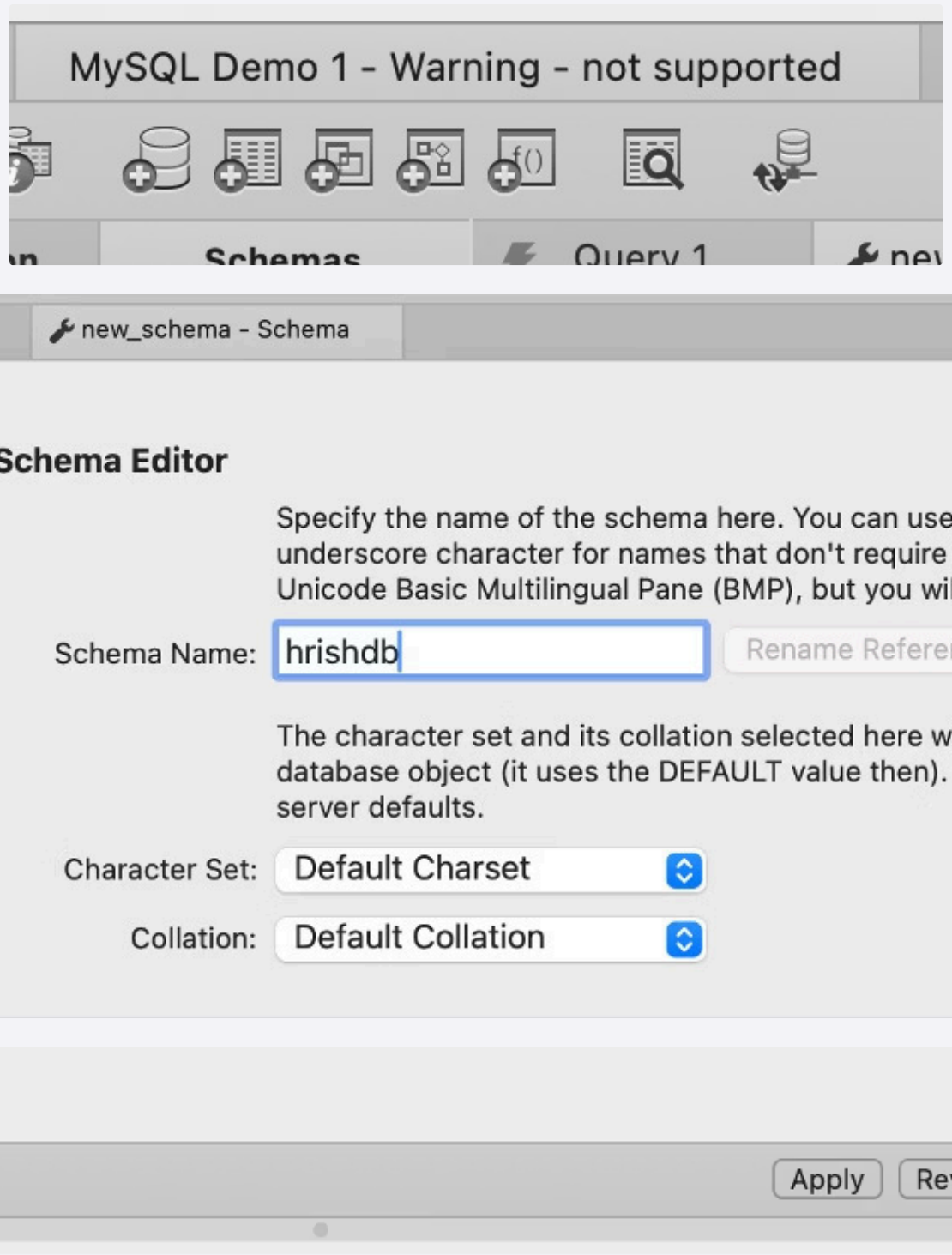
What is a Schema?

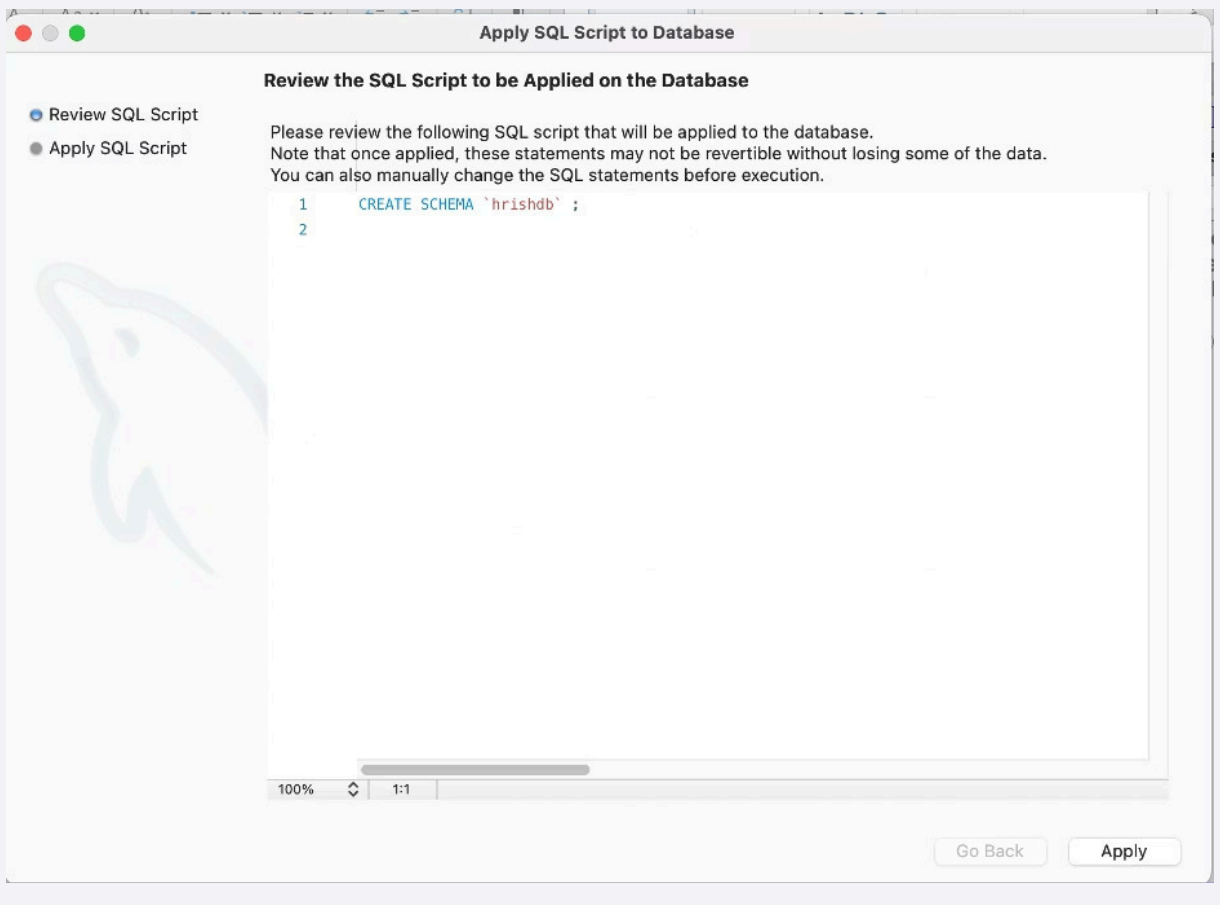
In MySQL, a schema is simply another word for database. It's a container that holds all your tables.

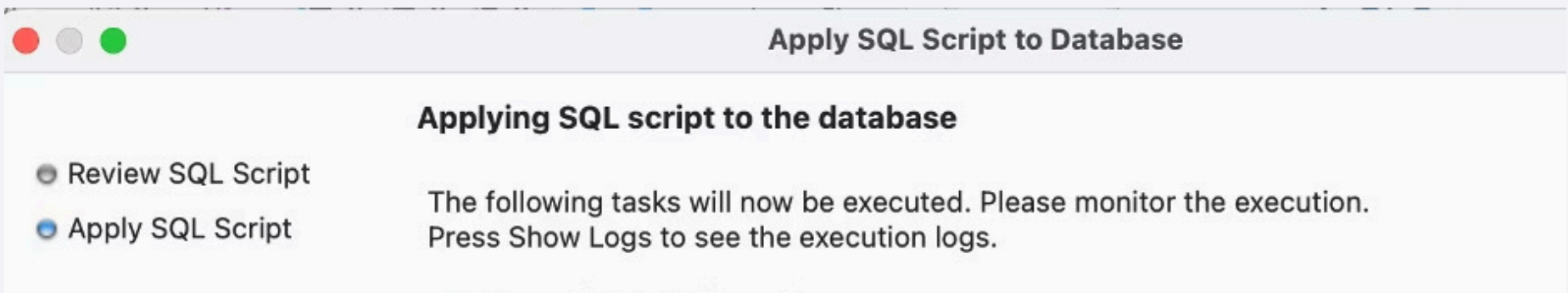
You must create a schema before you can create tables or store any data. Think of it as creating a folder before saving files.

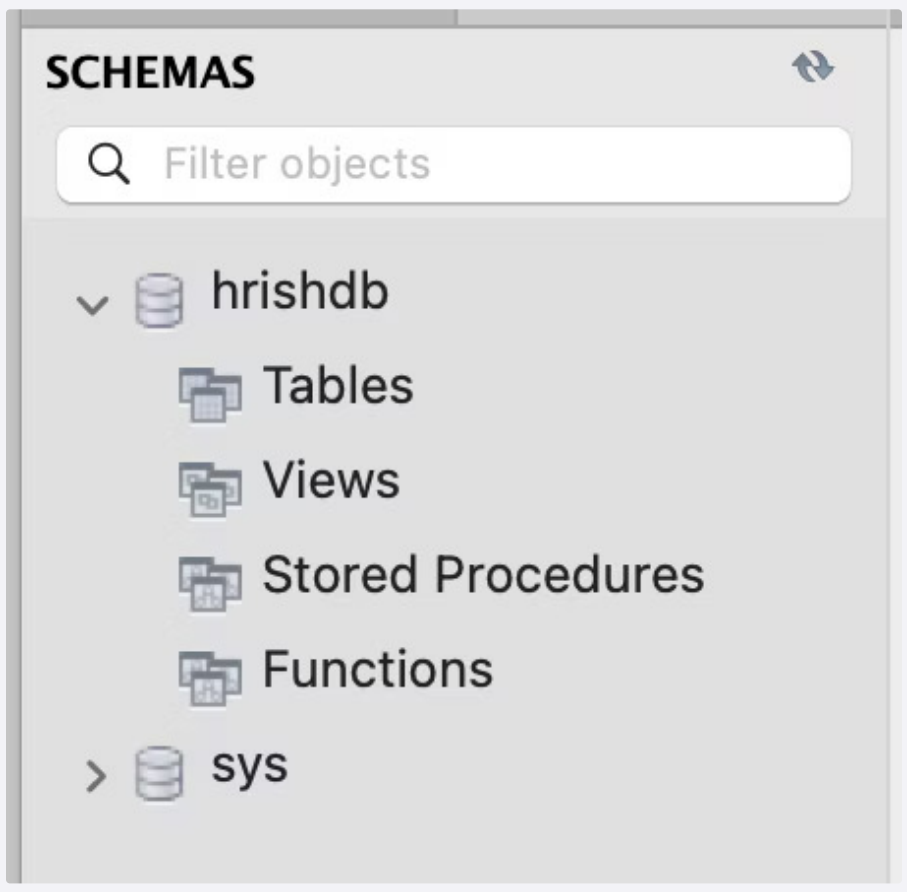
```
CREATE SCHEMA hrishdb;
USE hrishdb;
```

This creates a new schema called "hrishdb" and then sets it as the active database for your queries.









The Big 6 SQL Commands

Command #1: SELECT

View Columns

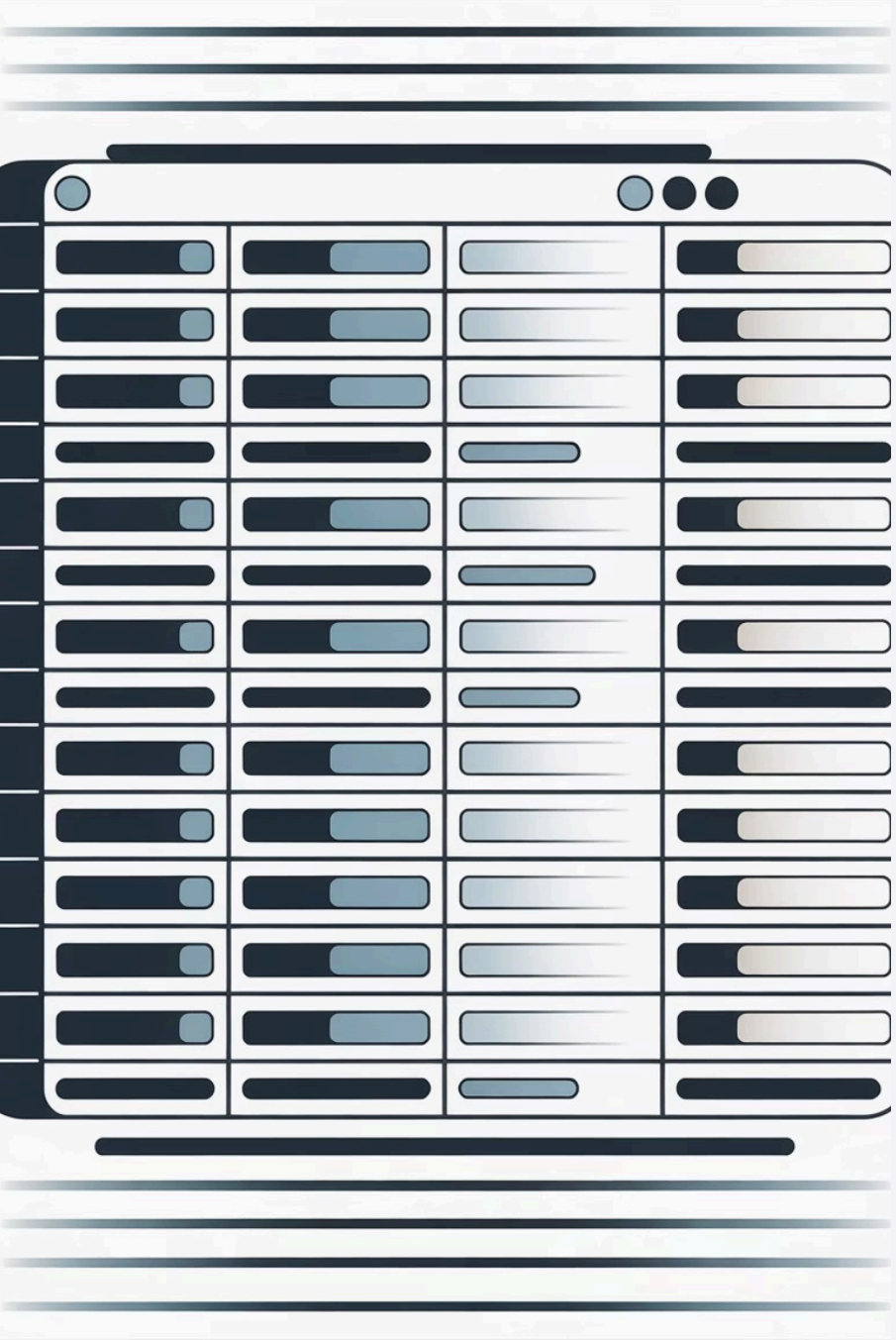
SELECT determines which columns you want to see in your results. Use * for all columns or specify individual column names.

Control Output

You have complete control over what data appears in your results by choosing exactly which columns to display.

```
SELECT first_name, last_name, salary
FROM employees;
```

This query returns only three columns: first name, last name, and salary for all employees in the table.



Command #2: FROM

Specify Your Source

The FROM clause tells SQL which table (or tables) to retrieve data from. Every SELECT statement must have a FROM clause.

You can query a single table or join multiple tables together to combine related information.

```
SELECT *
FROM employees;
```

This retrieves all columns from the employees table. The FROM clause is essential - without it, SQL doesn't know where to find your data.

Command #3: WHERE

The WHERE clause filters your data based on conditions you specify. It's one of the most powerful tools in SQL, allowing you to narrow down results to exactly what you need.

```
SELECT first_name, last_name, department
FROM employees
WHERE salary > 75000;
```

This query returns only employees earning more than $75,000. You can combine multiple conditions using AND, OR, and other logical operators.

WHERE Clause Examples

Single Condition

```
WHERE department = 'Sales'
```

Returns only sales department employees

Multiple Conditions

```
WHERE salary > 70000
AND department = 'Engineering'
```

Returns high-earning engineers

Range Filtering

```
WHERE salary BETWEEN
60000 AND 80000
```

Returns employees in salary range

Command #4: ORDER BY

Sort Your Results

ORDER BY arranges your query results in ascending (ASC) or descending (DESC) order based on one or more columns.

By default, sorting is ascending. Add DESC to reverse the order.

```
SELECT first_name, salary
FROM employees
ORDER BY salary DESC;
```

This shows employees sorted from highest to lowest salary.

Sorting Strategies

1

Single Column

Sort by one field like salary or hire date for straightforward ordering

2

Multiple Columns

Sort by department first, then by salary within each department

3

Mixed Directions

Sort one column ascending and another descending for complex analysis

Command #5: GROUP BY

GROUP BY is a powerful aggregation tool that groups rows sharing common values and allows you to perform calculations on each group. This is essential for summary statistics and analysis.

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

This calculates the average salary for each department. Each row in the result represents one department with its average salary.

Understanding GROUP BY Logic

Identify Groups

Determine what you want each row to represent (e.g., each department)

Choose Aggregation

Decide how to summarize data (AVG, SUM, COUNT, MIN, MAX)

Review Results

Each row shows one group with its calculated summary

Write Query

SELECT the grouping column and aggregated values

Aggregate Functions

AVG()

Calculates the average value of a numeric column across all rows in each group

SUM()

Adds up all values in a numeric column for each group

COUNT()

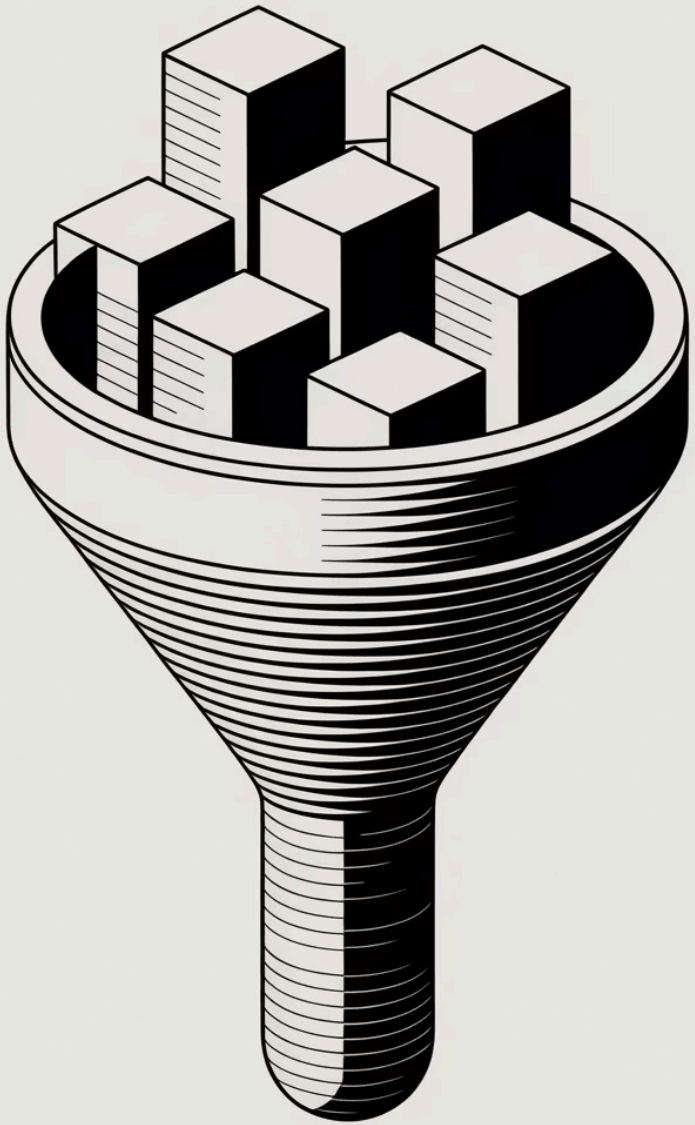
Counts the number of rows in each group or non-null values in a column

MAX()

Finds the maximum value in a column for each group

MIN()

Finds the minimum value in a column for each group



Command #6: HAVING

HAVING filters grouped data after aggregation. While WHERE filters individual rows before grouping, HAVING filters the groups themselves based on aggregate calculations.

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING avg_salary > 80000;
```

This returns only departments where the average salary exceeds $80,000. HAVING always works with GROUP BY.

WHERE vs HAVING

WHERE Clause

- Filters individual rows
- Applied before grouping
- Cannot use aggregate functions
- Filters raw data

```
WHERE salary > 70000
```

HAVING Clause

- Filters grouped results
- Applied after grouping
- Can use aggregate functions
- Filters summarized data

```
HAVING AVG(salary) > 70000
```

The Correct Query Order

SELECT

Choose which columns to display

FROM

Specify which table(s) to query

WHERE

Filter individual rows (optional)

GROUP BY

Group rows for aggregation (optional)

HAVING

Filter grouped results (optional)

ORDER BY

Sort the final results (optional)

This order is mandatory in SQL. You can omit clauses, but those you include must follow this sequence.

Complete Query Example

```
SELECT department, COUNT(*) AS employee_count,
       AVG(salary) AS avg_salary
FROM employees
WHERE hire_date >= '2020-01-01'
GROUP BY department
HAVING employee_count > 5
ORDER BY avg_salary DESC;
```

This comprehensive query finds departments with more than 5 employees hired since 2020, showing the count and average salary for each, sorted by salary from highest to lowest.

Beyond the Big 6: LIMIT

Control Result Size

LIMIT restricts the number of rows returned by your query. This is useful for previewing data, testing queries, or displaying top results.

Always place LIMIT at the very end of your query, after ORDER BY.

```
SELECT first_name, salary
FROM employees
ORDER BY salary DESC
LIMIT 10;
```

Returns only the top 10 highest-paid employees.

COUNT Function

COUNT is an aggregate function that returns the number of rows matching your criteria. It's one of the most frequently used functions in SQL for data analysis.

COUNT(*)

```
SELECT COUNT(*)
FROM employees
WHERE department = 'Sales';
```

Counts all rows in the Sales department

COUNT(column)

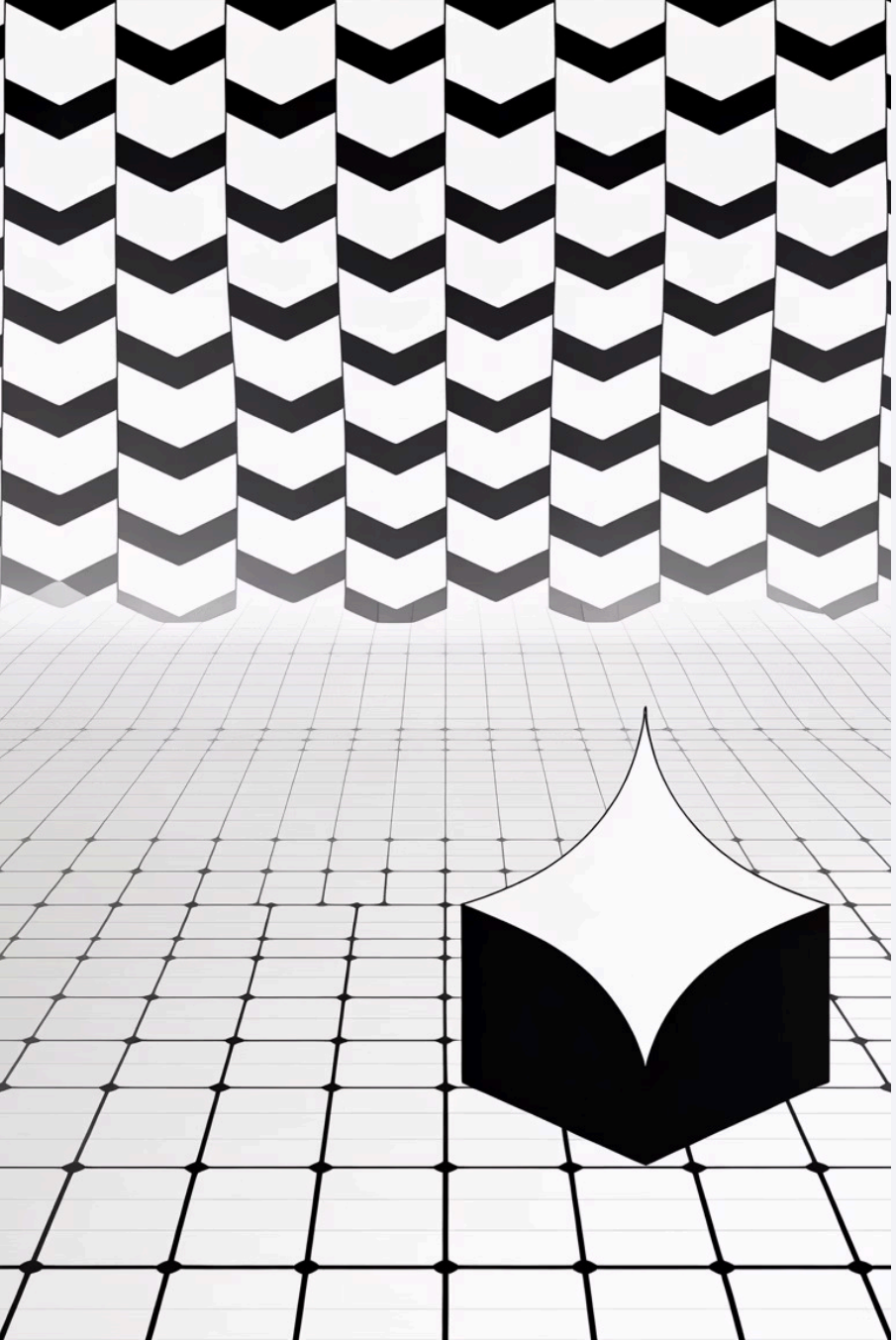
```
SELECT COUNT(email)
FROM employees;
```

Counts non-null email addresses

COUNT with GROUP BY

```
SELECT department,
    COUNT(*) AS total
FROM employees
GROUP BY department;
```

Counts employees per department



DISTINCT Keyword

Remove Duplicates

DISTINCT returns only unique values, eliminating duplicate rows from your results. This is essential when you want to see all possible values without repetition.

```
SELECT DISTINCT department
FROM employees
ORDER BY department;
```

Returns each unique department name only once, sorted alphabetically.

DISTINCT Use Cases

1

Find Unique Values

Discover all distinct categories, departments, or product types in your data

2

Count Unique Items

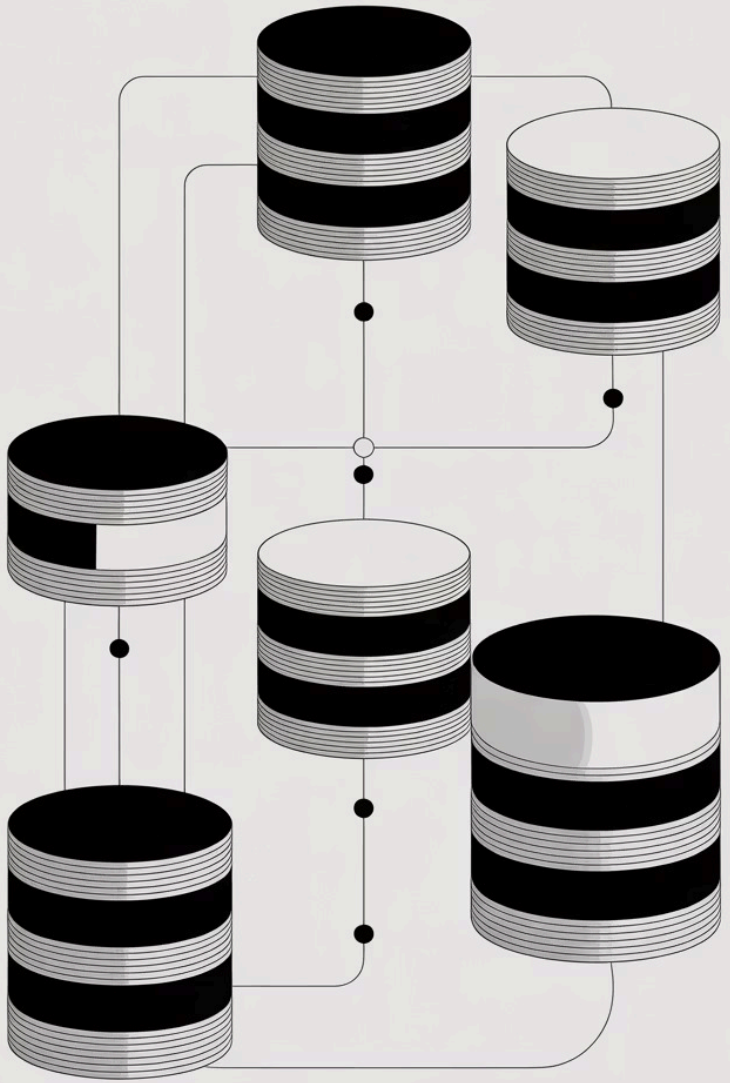
Combine with COUNT to find the number of unique values: COUNT(DISTINCT department)

3

Data Quality Checks

Identify unexpected duplicates or verify data consistency across tables

Working with Multiple Tables



Introduction to JOINs

JOINs combine rows from two or more tables based on related columns. They're essential for working with normalized databases where information is spread across multiple tables.

```
SELECT employees.first_name, employees.last_name,
       departments.dept_name
FROM employees
LEFT JOIN departments
ON employees.dept_id = departments.dept_id;
```

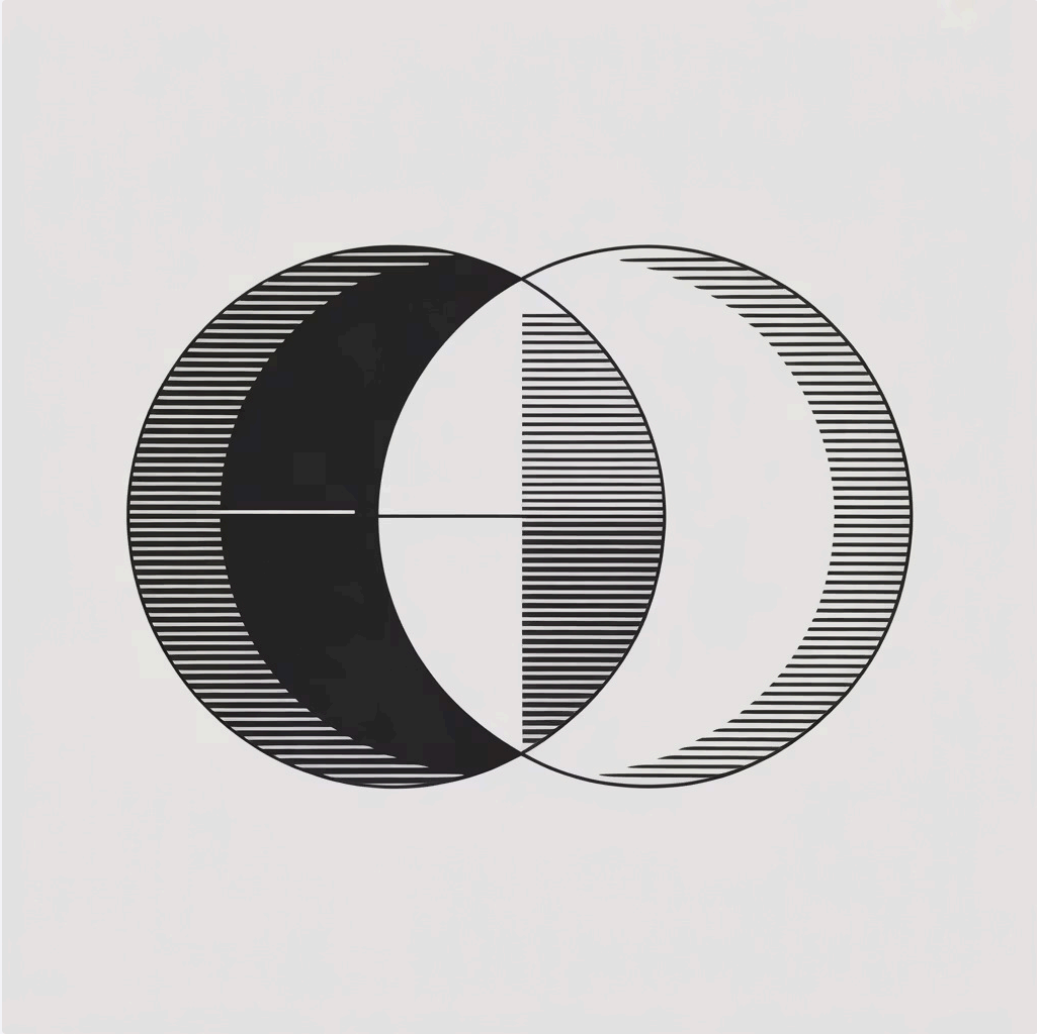
This query combines employee information with their department names by matching department IDs.

LEFT JOIN Explained

How It Works

A LEFT JOIN returns all rows from the left table and matching rows from the right table. If there's no match, NULL values appear for right table columns.

This ensures you keep all records from your primary table even if related data is missing.



JOIN Syntax Breakdown

01

Start with SELECT

Choose columns from both tables, prefixing with table names: table1.column1, table2.column2

02

Specify FROM table

Name your primary (left) table that you want all records from

03

Add LEFT JOIN

Specify the second table you want to join to your primary table

04

Define ON condition

Specify which columns connect the tables: table1.id = table2.foreign_id

Practical JOIN Example

Let's combine employee data with their performance reviews stored in a separate table.

```
SELECT e.employee_id, e.first_name, e.last_name,
       r.review_date, r.rating, r.comments
FROM employees e
LEFT JOIN reviews r
ON e.employee_id = r.employee_id
WHERE r.review_date >= '2024-01-01'
ORDER BY r.rating DESC;
```

This shows all 2024 reviews with employee details, sorted by rating. Table aliases (e, r) make the query more readable.

Types of JOINs

INNER JOIN

Returns only rows with matches in both tables. Most restrictive but ensures complete data.

LEFT JOIN

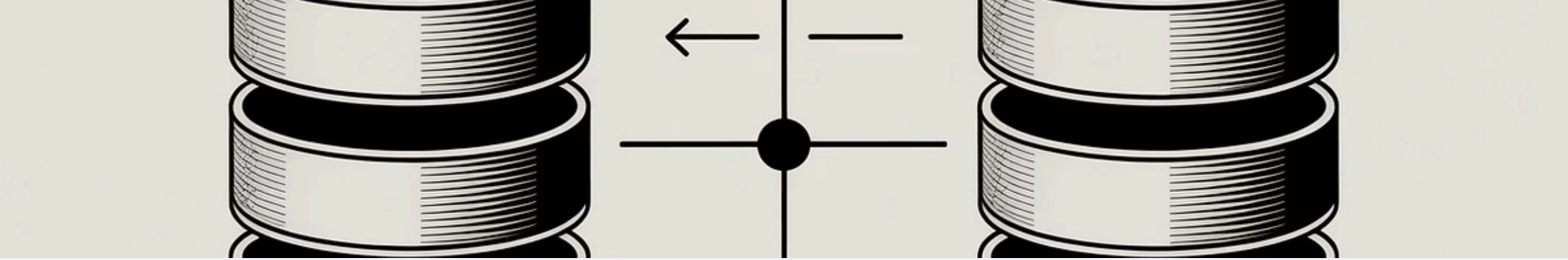
Returns all rows from left table plus matches from right. Most commonly used.

RIGHT JOIN

Returns all rows from right table plus matches from left. Less common than LEFT JOIN.

FULL OUTER JOIN

Returns all rows from both tables, with NULLs where there's no match.



Understanding Our Datasets

We'll work with two interconnected tables that represent a typical business database structure. The **employees** table contains 16 employees at Acme Corp with their names, departments (Sales, Accounting, Marketing, Operations), salaries, employment status, hire dates, and emails.

The **sales_transactions** table records individual sales made during Q1 2024. Each row represents a single sale of a product (laptops, printers, software licenses, etc.) and includes the dollar amount. These tables are linked through employee ID, creating a one-to-many relationship where one employee can have many sales transactions.

Dataset 1: Employees Table Structure

Employee ID

Unique identifier for each employee

Employee Name

Full name of the employee

Department

Sales, Accounting, Marketing, or Operations

Salary

Annual salary amount

Full-Time Status

Yes or No employment type

Hire Date

Date employee was hired

Email

Company email address

Dataset 2: Sales Transactions Structure

Quarter ID
Q1_2024 time period identifier

Product ID
Unique product code

Category
Electronics, Office Supplies, Software, Services

Product Name
Laptop, Printer, CRM License, etc.

Employee ID
Links to employees table

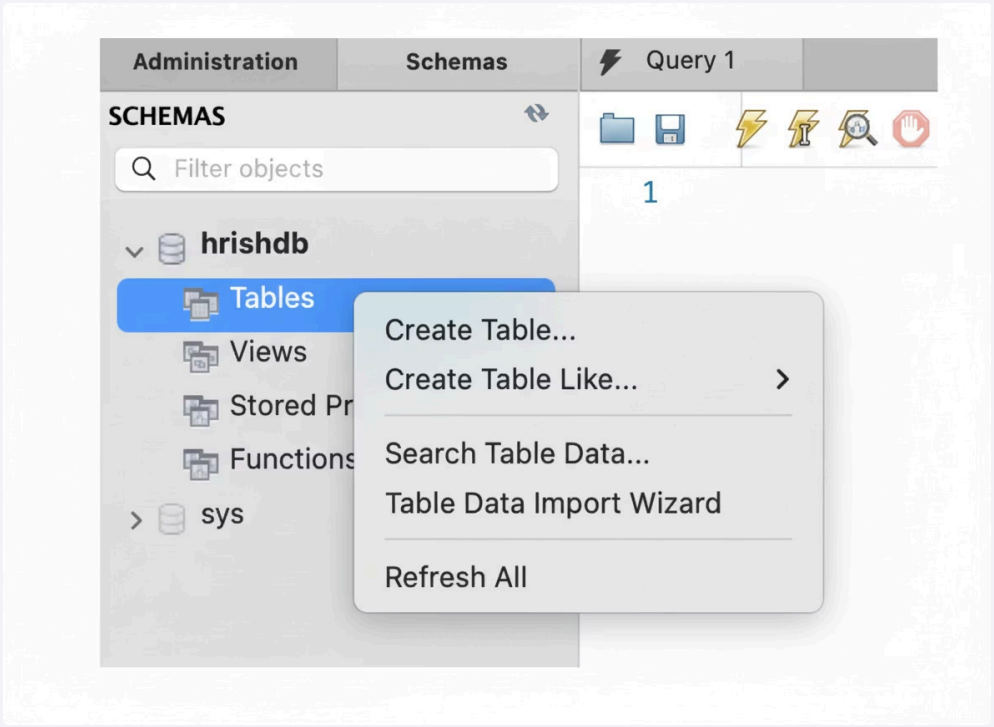
Sale Amount
Dollar value of the sale

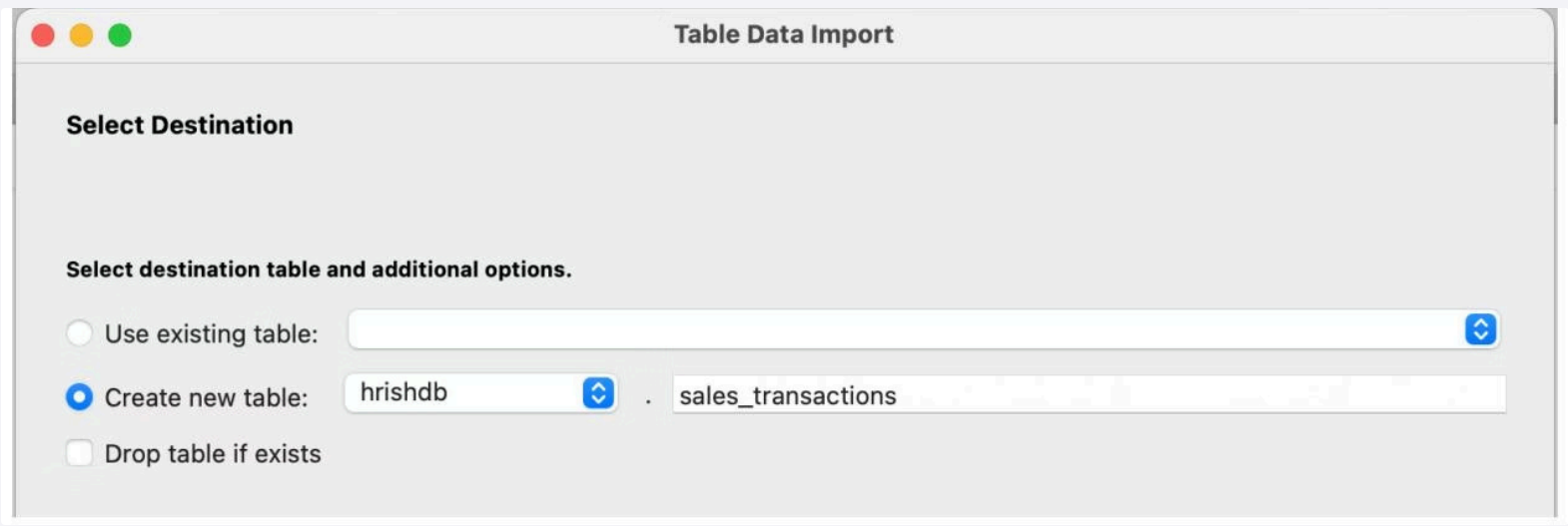
Note: Only Sales department employees (IDs 1, 4, 7, 11, 15) appear in the transactions table, which makes business sense - accountants and operations staff typically don't make direct sales.

Setting Up: Importing the Databases

Before we can query data, we need to import our CSV files using the **Table Data Import Wizard**.

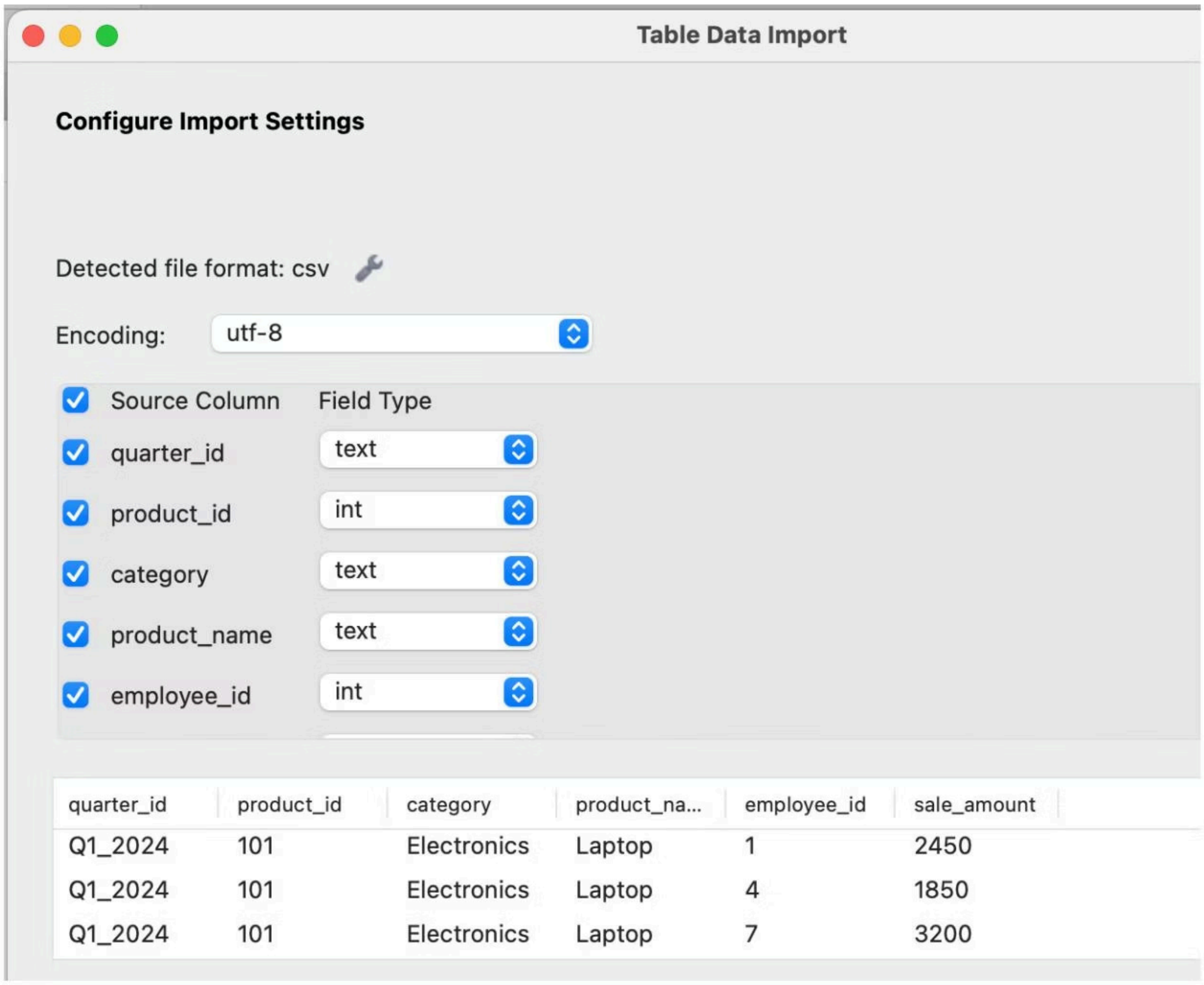
Follow the import wizard by clicking **Next** through each step, then click **Finish** to complete the import. After importing, use the **Refresh** button (top right next to "SCHEMAS") to verify the table appears correctly in your database.





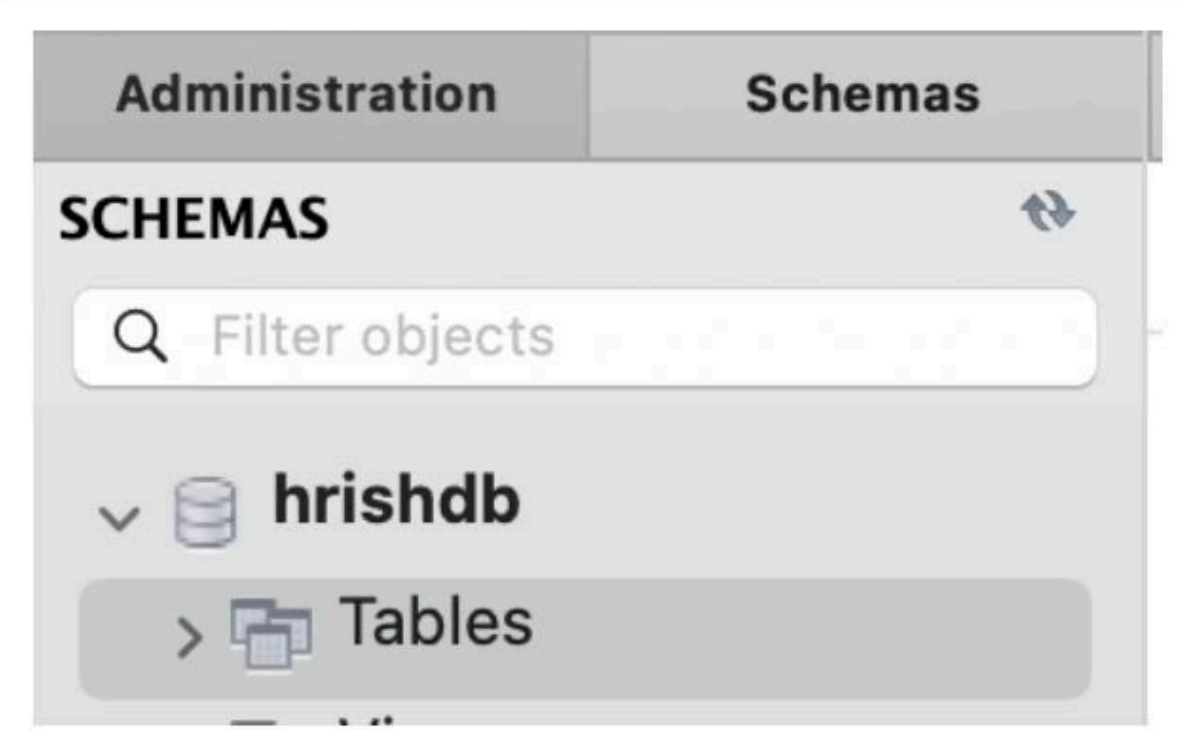
Import Process: Configuration Steps

During the import process, you'll configure how the database reads your CSV file. This includes mapping data types for each field. The Table Data wizard guides you through these settings to ensure your data imports correctly. Pay attention to data type assignments.



Verifying the Import

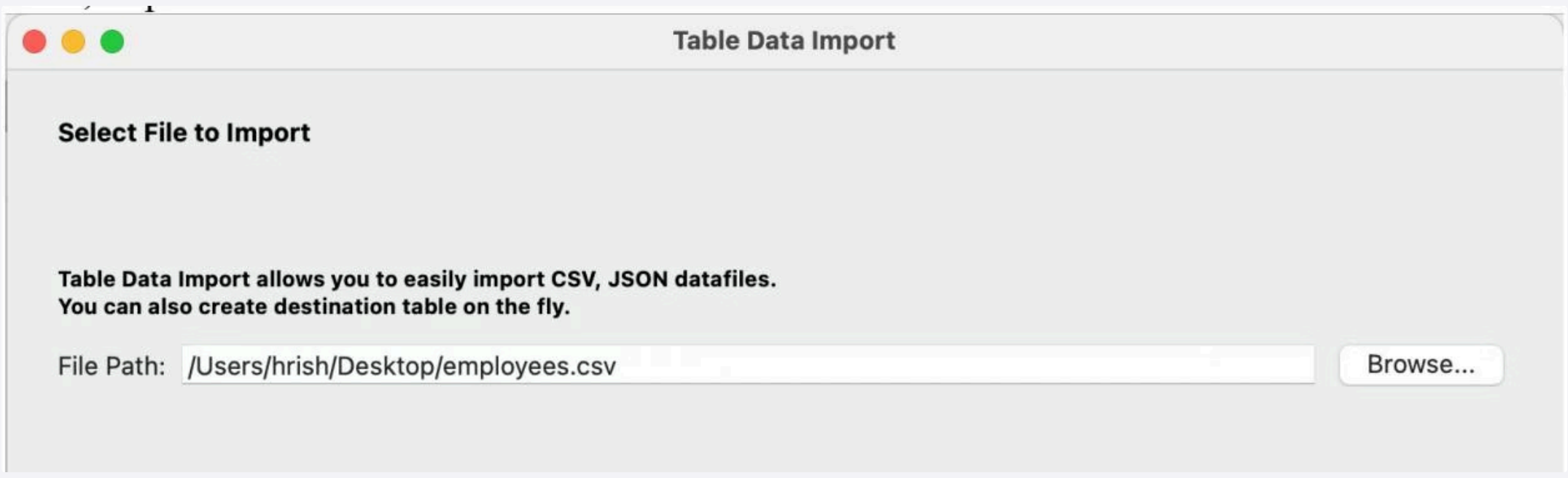
After clicking Finish, the import process completes. Click the **Refresh** button to see your newly imported employees and sales transaction tables.

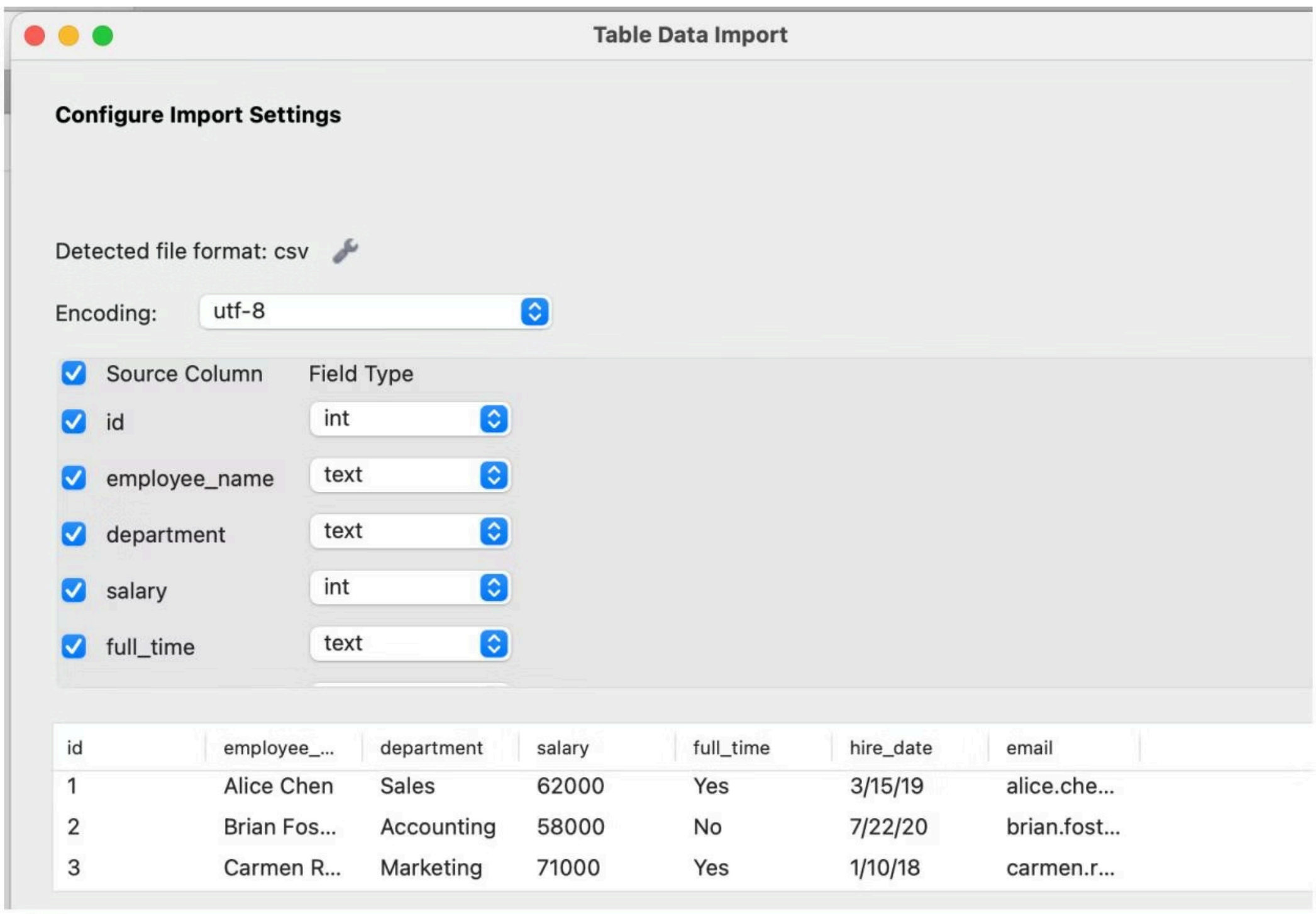


Importing the Employees Database

Now we repeat the import process for our second dataset: sales_transactions.csv. This table contains all the Q1 2024 sales data that we'll later join with our employee information.

Follow the same import wizard steps as before. This table structure is slightly different, containing transaction-specific fields like quarter_id, product_id, category, product_name, employee_id, and sale_amount.





You are now ready to practice SQL

You now have a complete relational database structure to practice SQL commands.

```
-- ================================================================
-- Dataset 1: employees (company employee records)
-- Dataset 2: sales_transactions (sales by employee)
-- ================================================================
```

Query 1: SELECT * - Viewing the Entire Table

The most basic SQL query retrieves all data from a table. The `SELECT *` statement means "select all columns," and `FROM employees` specifies which table to query.

Objective: View the complete employees table with all columns and all rows. This is useful for initial data exploration and understanding the full structure of your dataset.

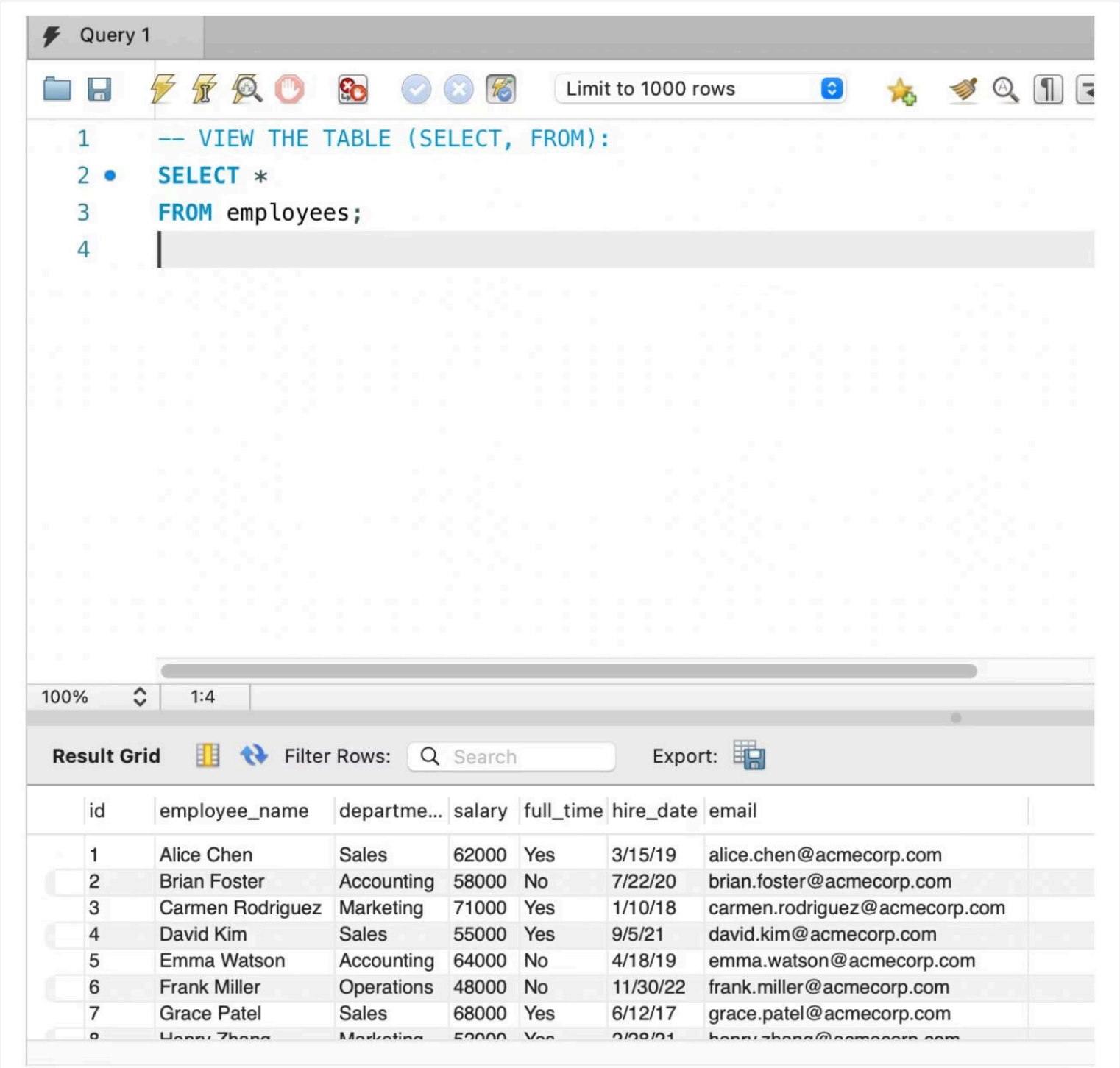
```
-- VIEW THE TABLE (SELECT, FROM):
SELECT *
FROM employees;
```

The asterisk (*) is a wildcard that represents "all columns." This query will return every field for every employee: id, employee_name, department, salary, full_time, hire_date, and email.

Query 1 Results: Complete Employee Data

The query returns all 16 employees with their complete information displayed in a table format. You can see the full range of departments (Sales, Accounting, Marketing, Operations), salary variations, and employment types.

This comprehensive view helps you understand the data structure and identify patterns, such as which departments have more employees or the salary ranges across the organization.



Query 2: SELECT Specific Columns

Instead of retrieving all columns, you can specify exactly which columns you want to see. This makes your results cleaner and more focused on the information you need.

Objective: Display only the employee name, salary, and full-time status. This focused view is more readable and efficient, especially when working with tables that have many columns.

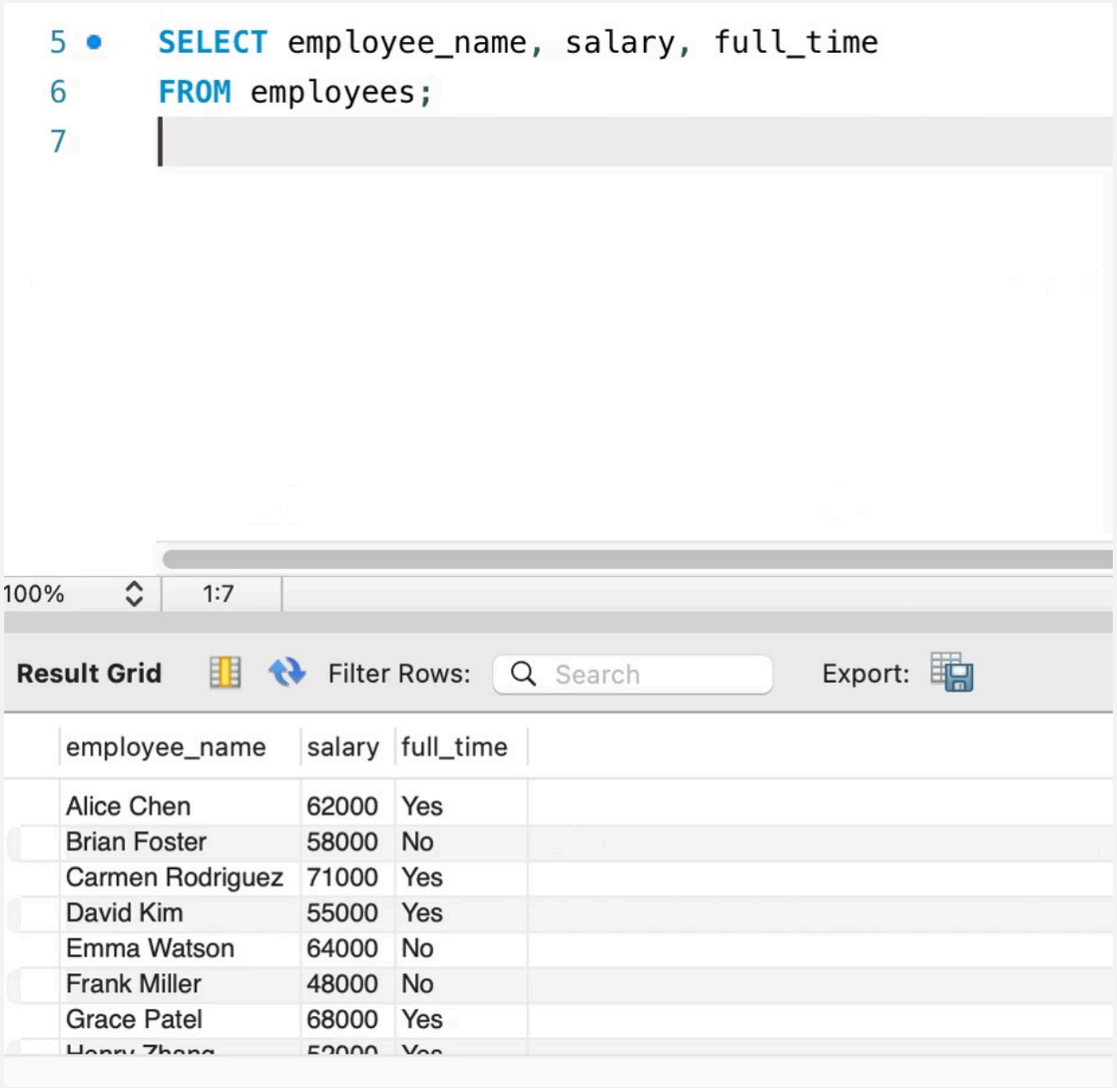
```
-- VIEW THE TABLE (SELECT, FROM):
SELECT employee_name, salary, full_time
FROM employees;
```

By listing specific column names after SELECT, separated by commas, you control exactly what data appears in your results. This is more efficient than SELECT * when you only need certain fields.

Query 2 Results: Focused Column View

The results now show only three columns: employee_name, salary, and full_time status. This streamlined view makes it easier to focus on specific information without the distraction of unnecessary columns.

You can quickly scan the data to see salary levels and employment types for each person. This type of selective querying is fundamental to efficient database work.



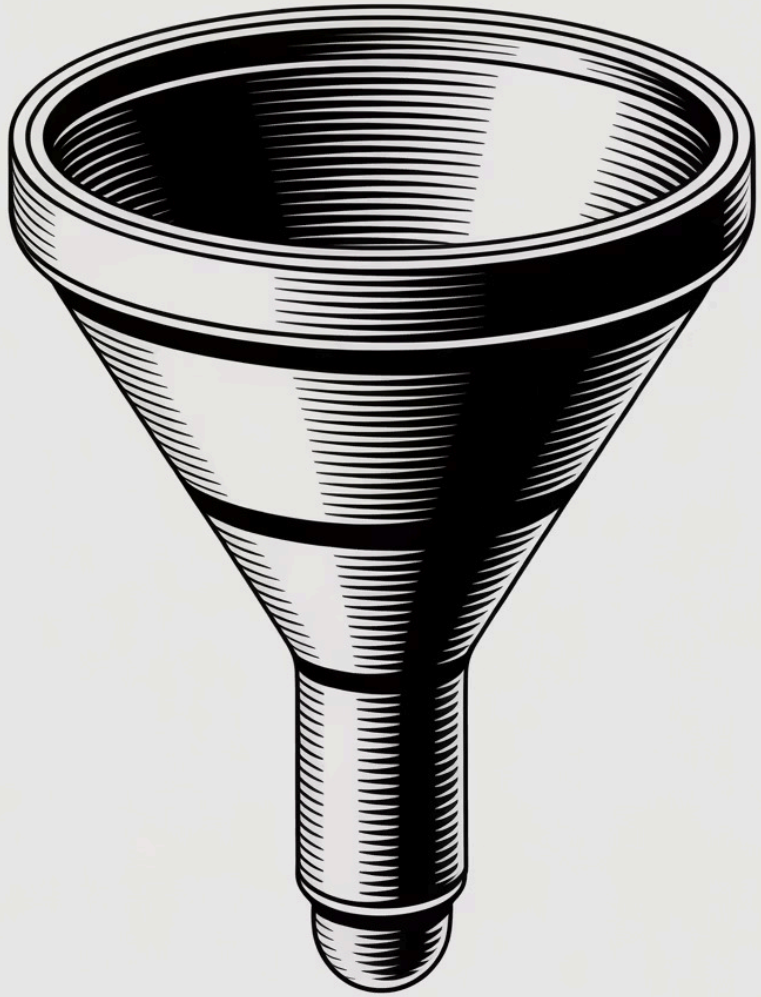
Query 3: WHERE - Filtering Full-Time Employees

The WHERE clause allows you to filter results based on conditions. This is one of the most powerful features of SQL, enabling you to retrieve only the rows that meet specific criteria.

Objective: Show only employees who are full-time. This filters out part-time employees, giving you a focused view of your full-time workforce.

```
-- Show employees who are full-time (WHERE):
SELECT employee_name, salary, full_time
FROM employees
WHERE full_time = 'Yes';
```

The WHERE clause acts as a filter. Only rows where the full_time column equals 'Yes' will be included in the results.

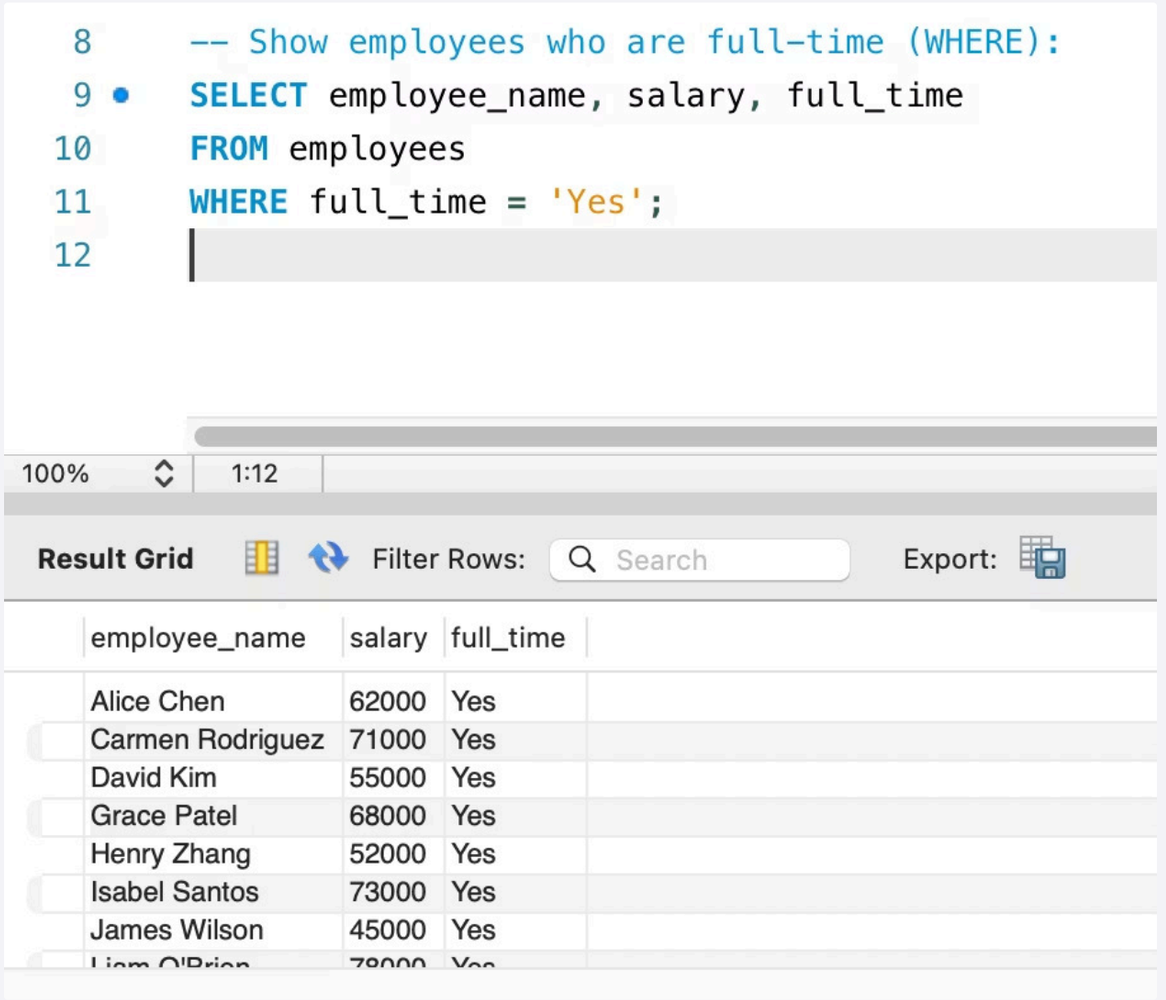


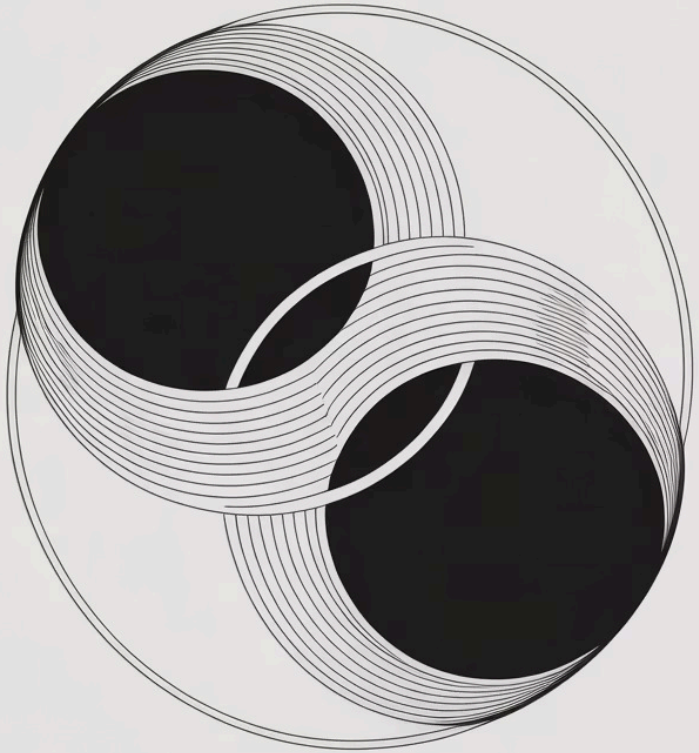
Query 3 Results: Full-Time Employees Only

The WHERE clause successfully filtered the results to show only employees where full_time = 'Yes'.

Notice how the results now exclude part-time employees like those with 'No' in the full_time column. This type of filtering is essential for payroll calculations and workforce planning.

The filtered view makes it easy to focus on your full-time staff and their salary distribution without the distraction of part-time employee data.





Query 4: WHERE with AND - Multiple Conditions

You can combine multiple conditions using the AND operator. This allows for more sophisticated filtering by requiring that all conditions be true for a row to be included.

Objective: Find full-time employees who earn more than $60,000. This helps identify your higher-paid full-time staff, useful for compensation analysis or identifying senior team members.

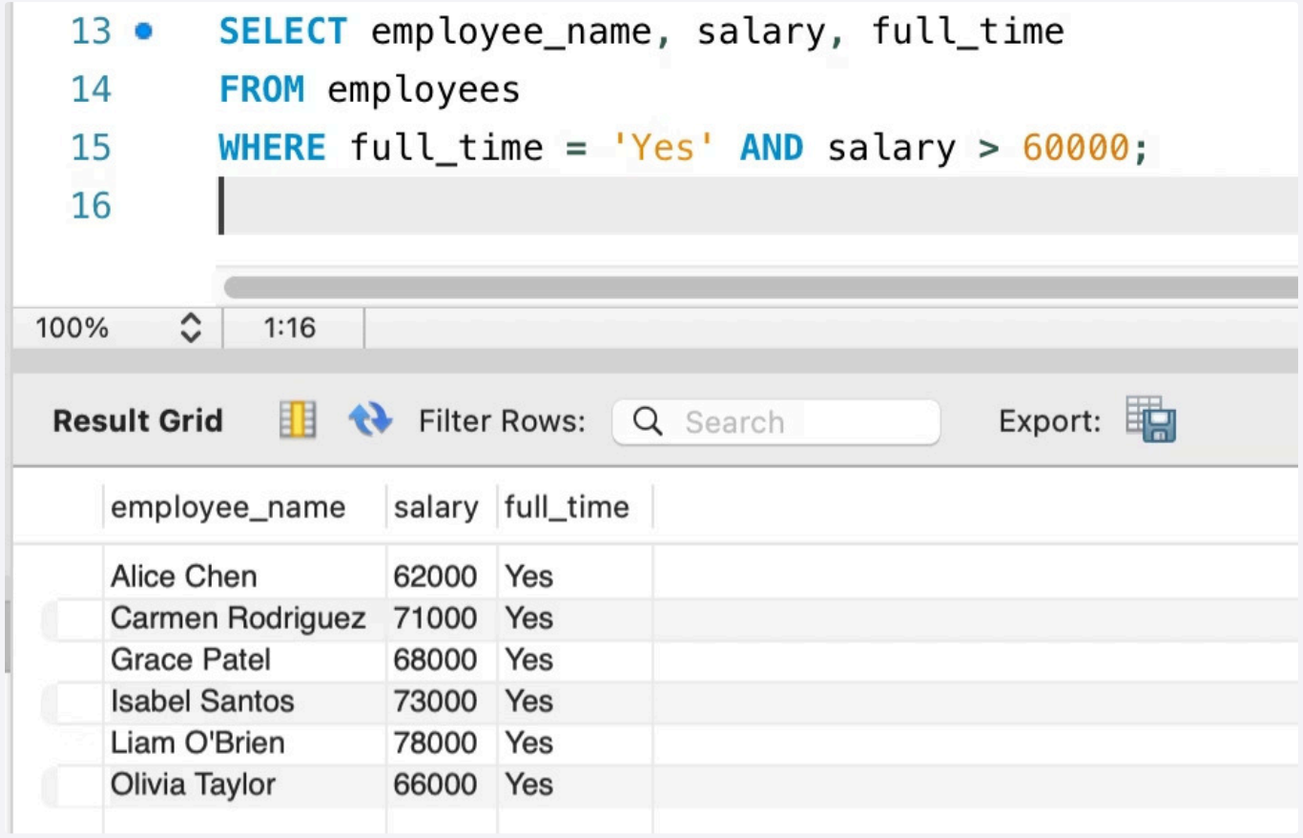
```
SELECT employee_name, salary, full_time
FROM employees
WHERE full_time = 'Yes' AND salary > 60000;
```

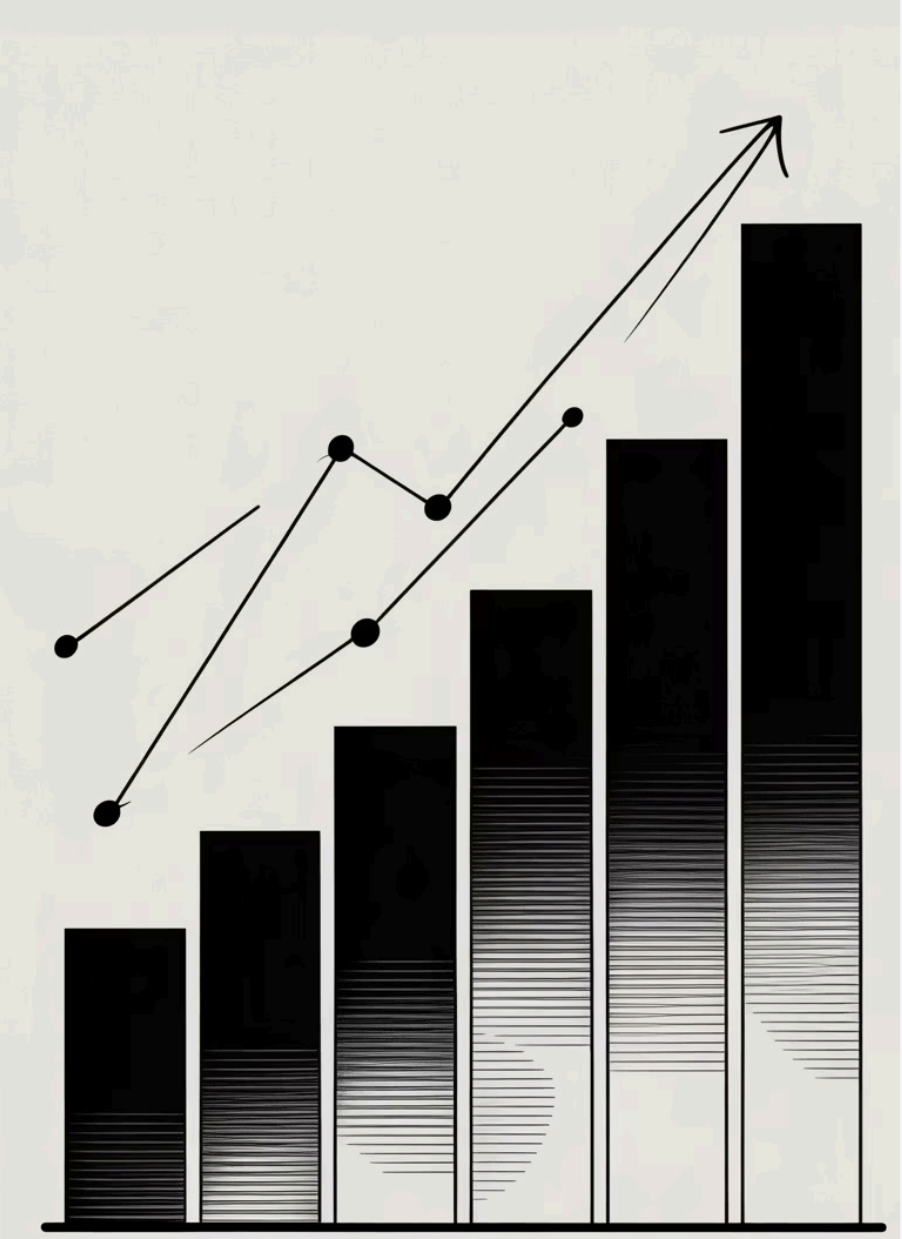
The AND operator means both conditions must be true: the employee must be full-time AND their salary must exceed $60,000.

Query 4 Results: High-Earning Full-Time Employees

The combined WHERE conditions with AND successfully filtered the results to show only employees who are both full-time AND earn more than $60,000. This query returned a smaller subset of employees who meet both criteria. This type of multi-condition filtering is extremely useful for further analysis, such as identifying senior staff members, analyzing compensation tiers, or finding employees eligible for certain benefits or programs.

The AND operator ensures that both conditions must be true - an employee earning $70,000 but working part-time would not appear in these results, nor would a full-time employee earning $50,000.





Query 5: ORDER BY - Sorting Results Ascending

The ORDER BY clause sorts your results based on one or more columns.

Objective: Display full-time employees earning over $60,000, sorted by salary from lowest to highest. This helps you see the salary progression within this group.

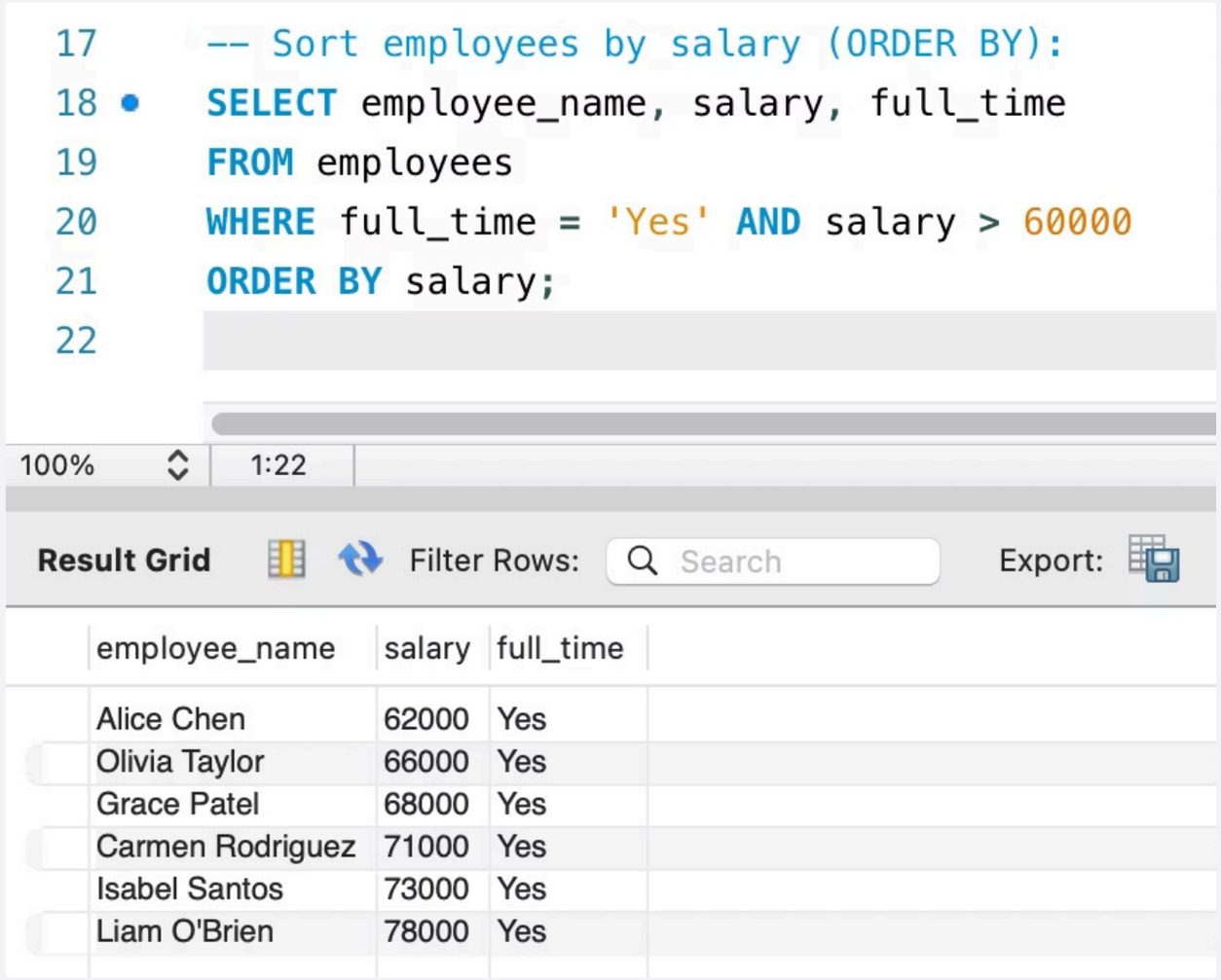
```
-- Sort employees by salary (ORDER BY):
SELECT employee_name, salary, full_time
FROM employees
WHERE full_time = 'Yes' AND salary > 60000
ORDER BY salary;
```

ORDER BY salary arranges the results with the lowest salary first, progressing to the highest. This makes it easy to identify salary ranges and compare compensation levels.

Query 5 Results: Salary Sorted Ascending

The ORDER BY clause successfully sorted the filtered results by salary in ascending order (lowest to highest). This type of sorting is valuable for compensation analysis, identifying salary gaps, and understanding the distribution of wages within specific employee segments.

Notice how the data flows logically from the lowest qualifying salary to the highest, making patterns and outliers immediately visible.



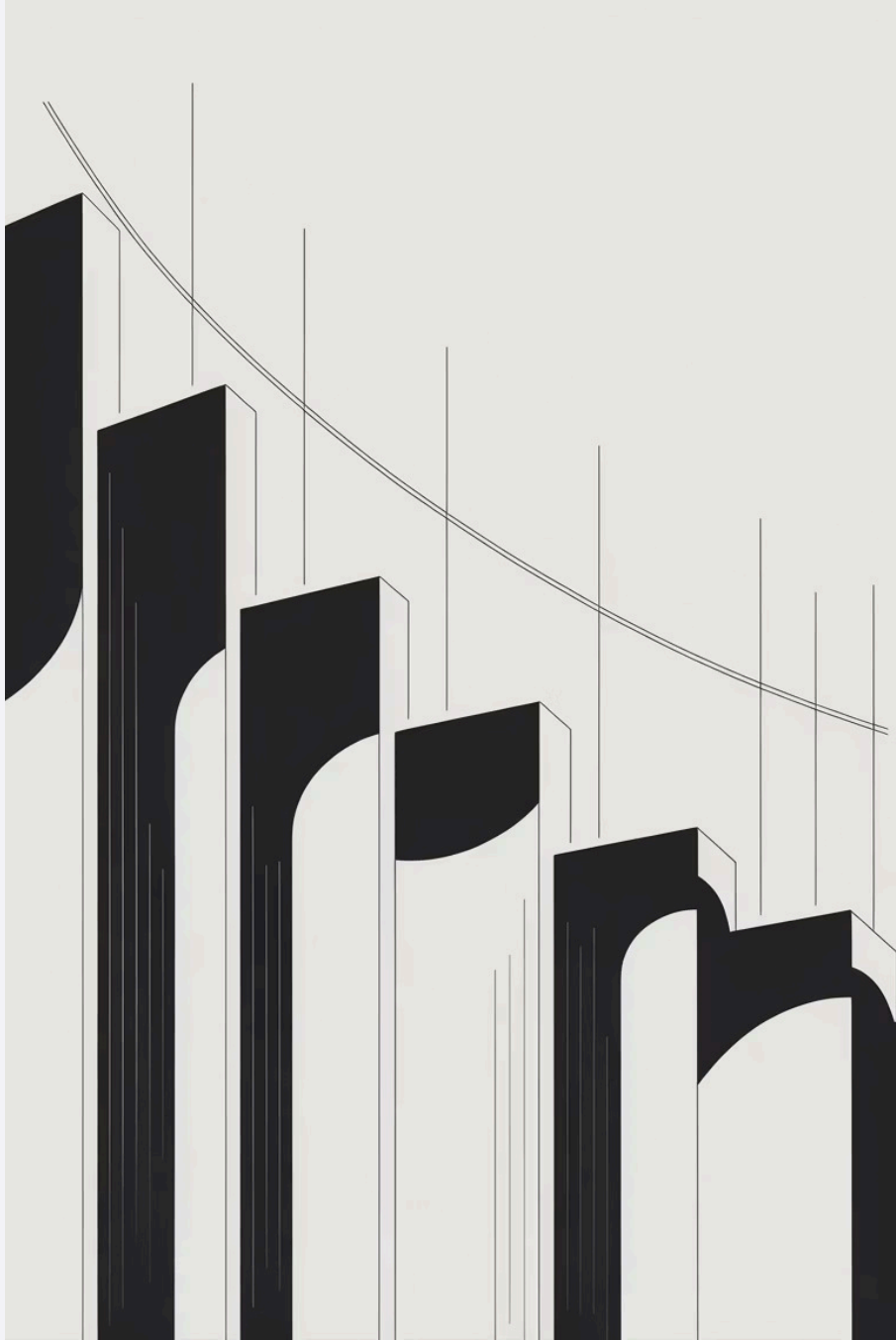
Query 6: ORDER BY DESC - Sorting Descending

Adding DESC (descending) after ORDER BY reverses the sort order, showing highest values first. This is particularly useful when you want to see top performers or highest values immediately.

Objective: Show the same filtered employees but sorted from highest to lowest salary. This quickly identifies your highest-paid full-time employees earning over $60,000.

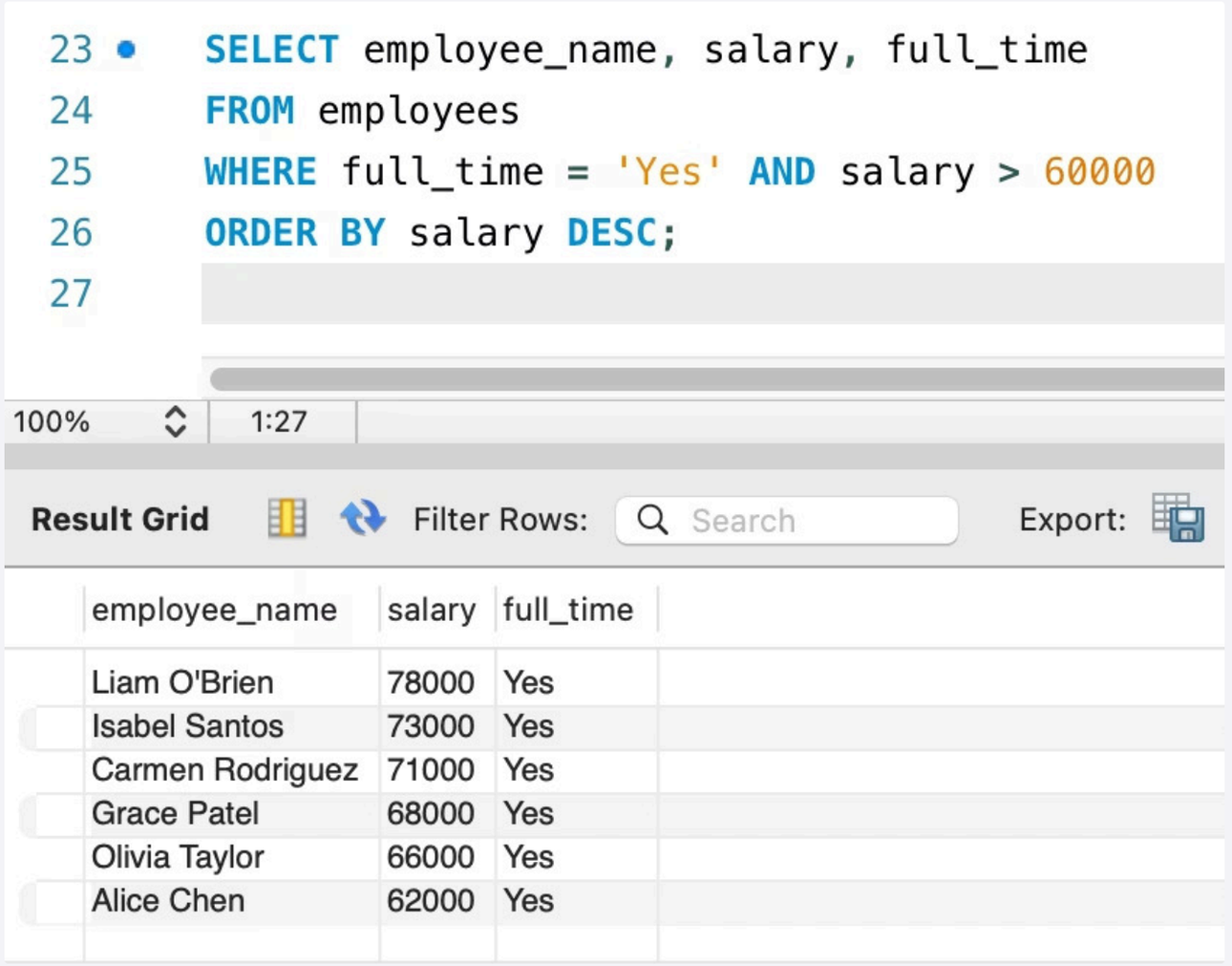
```
SELECT employee_name, salary, full_time
FROM employees
WHERE full_time = 'Yes' AND salary > 60000
ORDER BY salary DESC;
```

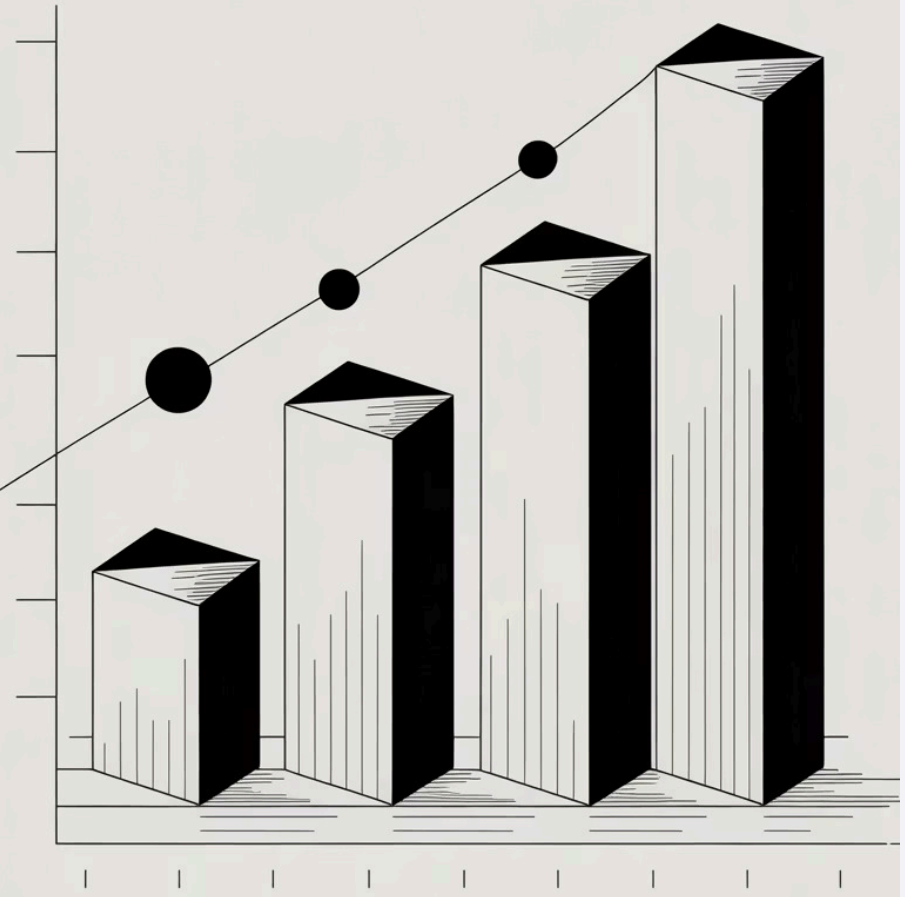
The DESC keyword reverses the sort order. Now the highest salary appears first, making it easy to identify top earners at a glance.



Query 6 Results: Highest Salaries First

The ORDER BY DESC clause successfully sorted the results in descending order, showing the highest-paid employees first. This reverse sorting makes it immediately clear who the top earners are in your filtered dataset. This descending view is particularly useful for budget planning, and identifying employees who might be candidates for leadership roles or retention programs.





Query 7: GROUP BY - Aggregating Data

GROUP BY combines rows that have the same values in specified columns and allows you to perform calculations on each group. This is essential for summary statistics and analysis.

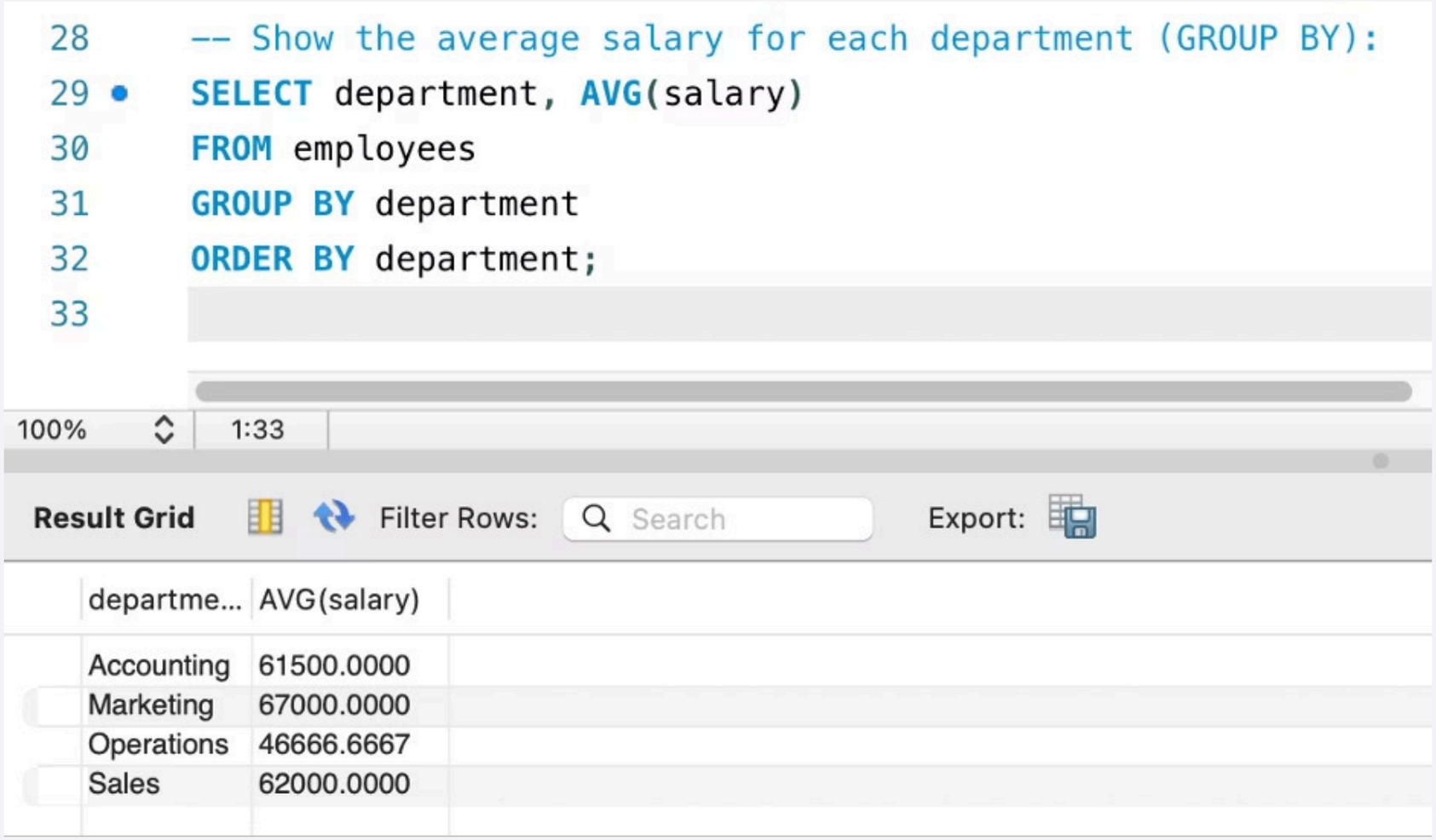
Objective: Calculate the average salary for each department. This provides insight into compensation differences across departments and helps with budgeting and equity analysis.

```
-- Show the average salary for each department (GROUP BY):
SELECT department, AVG(salary)
FROM employees
GROUP BY department
ORDER BY department;
```

GROUP BY department creates separate groups for Sales, Accounting, Marketing, and Operations. AVG(salary) calculates the average salary within each group.

Query 7 Results: Average Salary by Department

The GROUP BY clause successfully grouped employees by department and calculated the average salary for each. This aggregated view provides valuable insights into compensation distribution across the organization. The results show four departments with their respective average salaries: Accounting, Marketing, Operations, and Sales. This type of analysis is essential for budget planning, identifying compensation disparities, and ensuring equitable pay across departments.



Query 8: HAVING - Filtering Aggregated Results

HAVING is like WHERE, but it filters groups after aggregation rather than filtering individual rows before aggregation. Use HAVING with GROUP BY to filter based on aggregate calculations.

Objective: Identify departments where the average salary is below $60,000. This helps identify departments that might need compensation adjustments or have more junior staff.

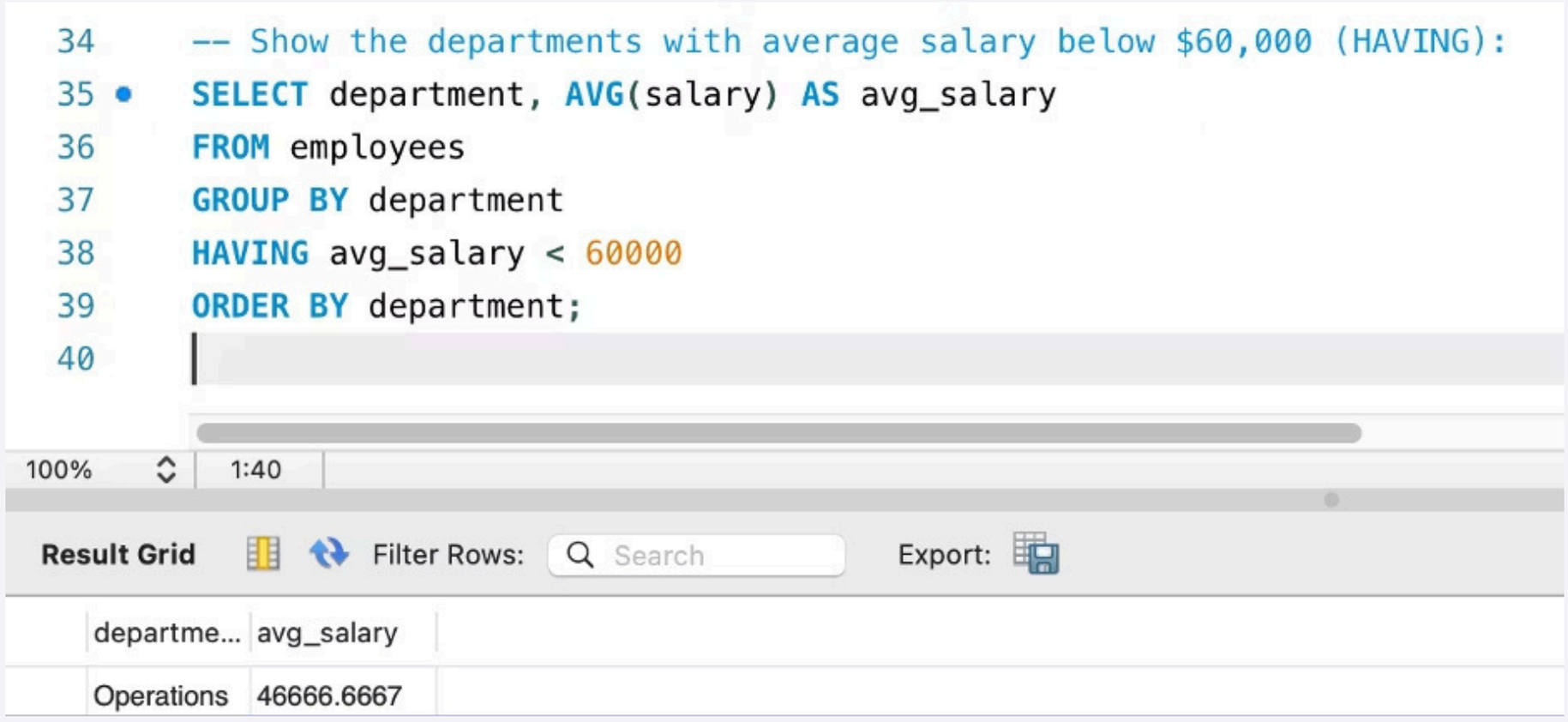
```
-- Show departments with average salary below $60,000 (HAVING):
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING avg_salary < 60000
ORDER BY department;
```

The AS keyword creates an alias (avg_salary) for the calculated average. HAVING filters the grouped results, showing only departments where the average is below $60,000. Note: WHERE cannot be used here because it operates before grouping.

Query 8 Results: Departments Below $60,000 Average

The HAVING clause successfully filtered the grouped results to show only departments where the average salary falls below $60,000. This query identified departments that may need compensation review or have more junior staff compositions. The results show which departments have lower average salaries, making it easy to identify potential pay equity issues or departments with different seniority distributions. Unlike WHERE, which filters individual rows, HAVING filters after the aggregation is complete.

This type of analysis is important for budget allocation, and ensuring competitive compensation across all departments in your organization.



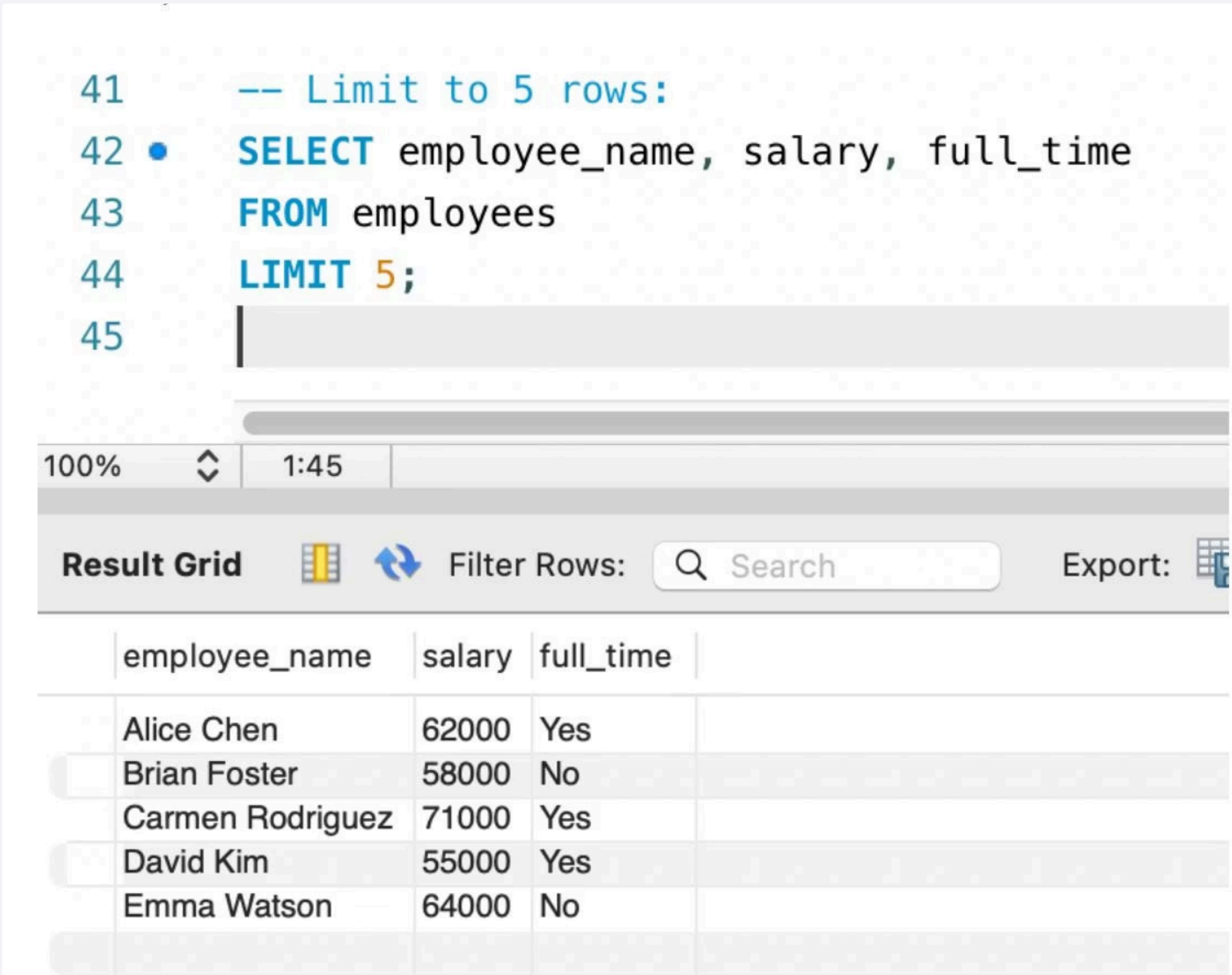
Query 9: LIMIT - Restricting Result Count

LIMIT restricts the number of rows returned by your query. This is useful for previewing data, testing queries, or implementing pagination (dividing large amounts of content into smaller portions) in applications.

Objective: Display only the first 5 employees from the table. This gives you a quick sample of the data without overwhelming your screen with all rows.

```
-- Limit to 5 rows:
SELECT employee_name, salary, full_time
FROM employees
LIMIT 5;
```

LIMIT 5 stops the query after returning 5 rows. This is particularly valuable when working with large datasets where you want to verify your query logic before retrieving thousands or millions of rows.



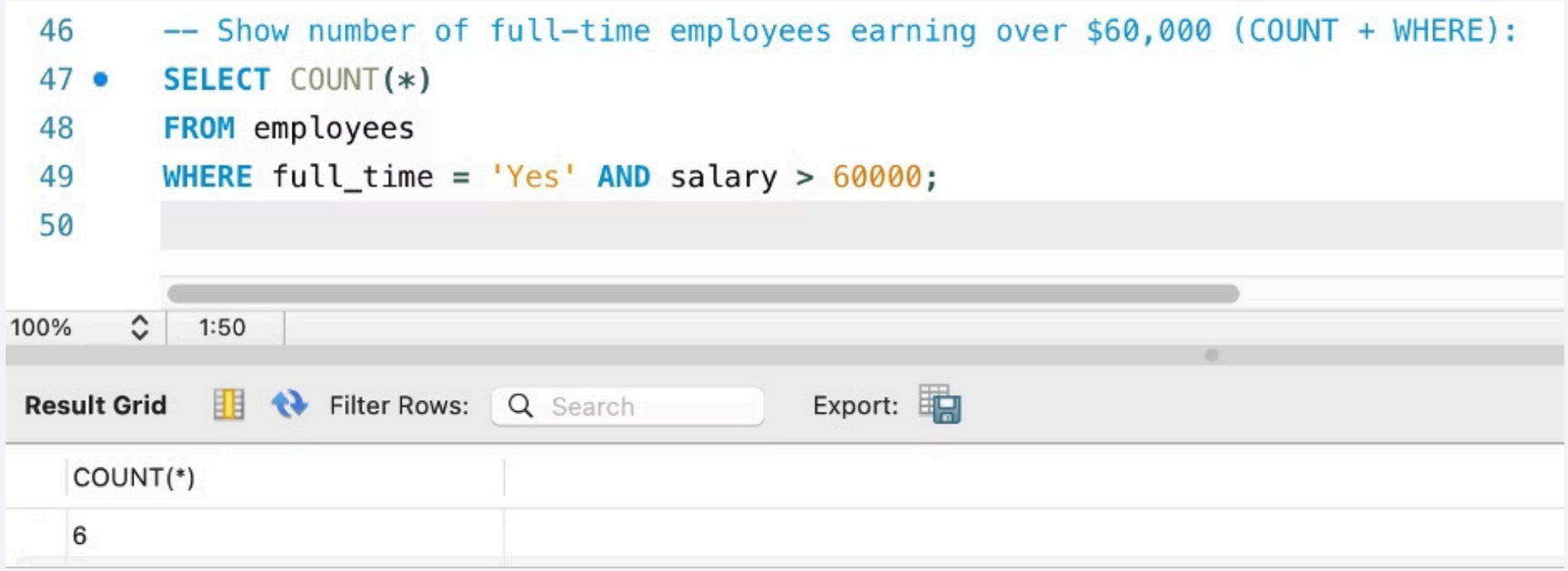
Query 10: COUNT with WHERE - Counting Filtered Rows

COUNT is an aggregate function that returns the number of rows. Combined with WHERE, it tells you how many rows meet specific criteria.

Objective: Count how many full-time employees earn over $60,000. This gives you a single number representing the size of this employee segment, useful for workforce planning and analysis.

```
-- Show number of full-time employees earning over $60,000
-- (COUNT + WHERE):
SELECT COUNT(*)
FROM employees
WHERE full_time = 'Yes' AND salary > 60000;
```

COUNT(*) counts all rows that pass the WHERE filter. The result is a single number, not a list of employees. This is perfect for quick statistics and dashboard metrics.



Query 11: DISTINCT - Finding Unique Values

DISTINCT removes duplicate values from your results, showing only unique values. This is essential for understanding the range of values in a column.

Objective: Display all unique salary values in the employees table. This shows you the distinct salary levels without repetition, helping you understand your compensation structure.

```
-- Show unique salary values (DISTINCT):
SELECT DISTINCT salary
FROM employees
ORDER BY salary;
```

Without DISTINCT, if multiple employees have the same salary, that value would appear multiple times. DISTINCT ensures each salary amount appears only once, giving you a clear picture of your salary tiers.

```
-- Distinct or Unique salary values only:
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC;
```

```
50
51    -- Distinct or Unique salary values only:
52    SELECT DISTINCT salary
53    FROM employees
54    ORDER BY salary DESC;
55
```

100% 1:55

Result Grid Filter Rows: Search Export:

salary
78000
73000
71000
68000
66000
64000
62000

Understanding Table Relationships

Before we join tables, it's important to understand how they're related. The employees and sales_transactions tables are connected through the employee ID field.

The **id** field in the employees table corresponds to the **employee_id** field in sales_transactions. This creates a one-to-many relationship: one employee can have many sales transactions, but each transaction belongs to only one employee.

1

Employees Table

Contains employee details with unique ID

2

Sales Transactions

References employee_id for each sale

This relationship allows us to combine employee information (like name and department) with their sales performance data (products sold and amounts).

Query 12: LEFT JOIN - Combining Tables (Part 1)

JOIN operations combine data from multiple tables based on a related column. A LEFT JOIN returns all rows from the left table and matching rows from the right table.

Objective: Combine employee information with their sales transactions to see which employee made which sales. This creates a comprehensive view linking employee details to their sales performance.

```
-- Show the sales transactions for each employee: LEFT JOIN
-- Table 1
SELECT *
FROM employees;

-- Table 2
SELECT *
FROM sales_transactions;
```

First, we examine both tables separately to understand their structure. The employees table has employee details, while sales_transactions has product and sale information.

Query 12: LEFT JOIN - Combining Tables (Part 2)

Now we join the tables using the relationship between employees.id and sales_transactions.employee_id. The LEFT JOIN ensures all employees appear in the results, even if they have no sales.

```
-- Both tables are connected by employee_id.
-- We want the sales for each employee.
SELECT *
FROM employees LEFT JOIN sales_transactions
ON employees.id = sales_transactions.employee_id;
```

The ON clause specifies how to match rows between tables. LEFT JOIN means all employees appear in results; employees without sales will have NULL values in the sales columns. This is important for complete reporting.

```
56    -- Show the sales transactions for each employee: LEFT JOIN
57    -- Table 1
58 •  SELECT *
59    FROM employees;
60
61    -- Table 2
62 •  SELECT *
63    FROM sales_transactions;
64
65    -- Both tables are connected by employee_id. We want the sales for each employee.
66 •  SELECT *
67    FROM employees LEFT JOIN sales_transactions
68        ON employees.id = sales_transactions.employee_id;
69
```

100% 5:69

Result Grid Filter Rows: Search Export:

id	employee_name	departme...	salary	full_time	hire_date	email	quarter_id	product_...	category
1	Alice Chen	Sales	62000	Yes	3/15/19	alice.chen@acmecorp.com	Q1_2024	402	Services
1	Alice Chen	Sales	62000	Yes	3/15/19	alice.chen@acmecorp.com	Q1_2024	401	Services
1	Alice Chen	Sales	62000	Yes	3/15/19	alice.chen@acmecorp.com	Q1_2024	302	Software
1	Alice Chen	Sales	62000	Yes	3/15/19	alice.chen@acmecorp.com	Q1_2024	301	Software
1	Alice Chen	Sales	62000	Yes	3/15/19	alice.chen@acmecorp.com	Q1_2024	202	Office Supplie
1	Alice Chen	Sales	62000	Yes	3/15/19	alice.chen@acmecorp.com	Q1_2024	201	Office Supplie
1	Alice Chen	Sales	62000	Yes	3/15/19	alice.chen@acmecorp.com	Q1_2024	102	Electronics

Query 13: LEFT JOIN with Specific Columns

When joining tables, SELECT * can produce overwhelming results with many duplicate columns. It's better to specify exactly which columns you need from each table.

Objective: Create a cleaner joined view showing only employee ID, name, product sold, and sale amount. This focused output is more readable and practical for analysis.

```
-- Let's just select specific relevant columns:
SELECT
    employees.id,
    employees.employee_name,
    sales_transactions.product_name,
    sales_transactions.sale_amount
FROM employees LEFT JOIN sales_transactions
ON employees.id = sales_transactions.employee_id;
```

Use table_name.column_name notation to specify which table each column comes from. This prevents ambiguity when both tables have similarly named columns and makes your query more maintainable.

```
70    -- Let's just select specific relevant columns:
71 •  SELECT employees.id, employees.employee_name,
72           sales_transactions.product_name, sales_transactions.sale_amount
73    FROM employees LEFT JOIN sales_transactions
74        ON employees.id = sales_transactions.employee_id;
75
76
```

100% 1:75

Result Grid Filter Rows: Search Export:

id	employee_name	product_name	sale_amount
1	Alice Chen	Maintenance Contract	600
1	Alice Chen	Installation Fee	150
1	Alice Chen	CRM License	850
1	Alice Chen	Accounting Suite	1500
1	Alice Chen	Desk Chair	275
1	Alice Chen	Printer	380
1	Alice Chen	Tablet	890

Key SQL Concepts Covered

Basic Selection

SELECT and FROM for retrieving data from tables

Filtering

WHERE clause for conditional row filtering

Sorting

ORDER BY for arranging results ascending or descending

Aggregation

GROUP BY and aggregate functions like AVG, COUNT

Joining

LEFT JOIN for combining related tables

Result Control

LIMIT, DISTINCT, and HAVING for refining output

SQL Query Building Blocks

01

SELECT

Choose which columns to retrieve (* for all, or list specific columns)

02

FROM

Specify which table(s) to query

03

JOIN

Combine data from multiple related tables

04

WHERE

Filter rows based on conditions

05

GROUP BY

Aggregate rows with same values

06

HAVING

Filter aggregated groups

07

ORDER BY

Sort results ascending or descending

08

LIMIT

Restrict number of rows returned